

Lez5__

October 29, 2023

1 Lezione 5 (Russo Russo)

Nelle reti neurali calcolare il gradiente può essere oneroso.

1.1 Backpropagation

Algoritmo di programmazione dinamica basato su *chain rule* per calcolo del gradiente (ultima volta siamo arrivati con tutte variabili scalari, ma andremo a lavorare anche con vettori e matrici).

Non è conveniente perché abbiamo maggiore complessità (rete a grana più fine), a livello computazionale si hanno molte cose più efficienti che si possono applicare nel caso vettoriale.

1.1.1 Backpropagation per NN

L'algoritmo è più complesso, per semplicità ci limitiamo a vedere una singola istanza di training \bar{x} e le sue *label associate* \bar{t} usate per calcolare la funzione costo $J(\theta)$. Vogliamo calcolare $\nabla_{\theta} J(\theta)$

Forward Pass Si parte dall'input della rete e si calcolano le attivazioni dei vari livelli fino ad arrivare all'obiettivo finale, ossia a calcolare la funzione costo J ; itero su ogni livello (incluso quello di uscita).

Forward pass

```
1  $h^{(0)} \leftarrow x$ 
2 for  $k=1, \dots, L$  do
3    $a^{(k)} \leftarrow W^{(k)} h^{(k-1)} + b^{(k)}$ 
4    $h^{(k)} \leftarrow f^{(k)}(a^{(k)})$ 
5 end
6  $y \leftarrow h^{(L)}$ 
7 return  $J = L(y, t) + \lambda \Omega(\theta)$ 
```

Si inizializza $h^{(0)} = x$ per evitare di distinguere il caso del primo ciclo (x =input).

Abbiamo un *pre-application value* $h^{(k-1)}$ (valore calcolato su quel livello prima di applicare la funzione di attivazione), subito dopo si applica la funzione di attivazione ottenendo il valore di uscita del livello k . L'ultimo $h^{(k)}$ rappresenta il valore di uscita della rete se itero su tutti i livelli.

Backward phase Voglio sapere il gradiente della funzione costo rispetto a tutti i parametri della rete.

È più complesso perché vengono prese scorciatoie per evitare di memorizzare il gradiente di tutte le variabili; ma sapendo come è fatta la rete possiamo ottimizzare l'esecuzione.

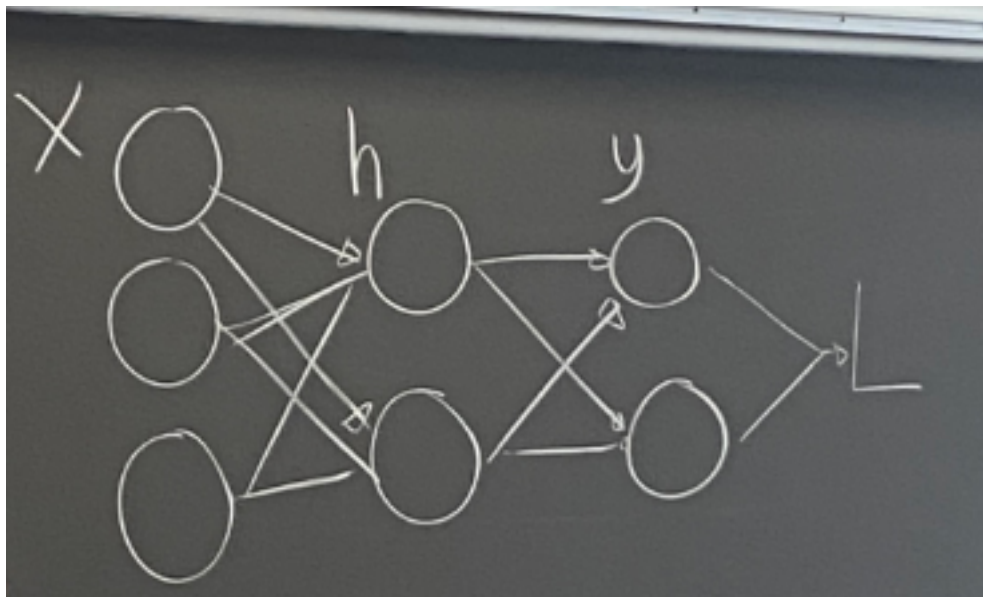
L'obiettivo è calcolare gradiente di J rispetto a tutti i parametri della rete.

Backpropagation for multi-layer NNs

```
1  $\mathbf{g} \leftarrow \nabla_{\mathbf{y}} J = \nabla_{\mathbf{y}} L(\mathbf{y}, \mathbf{t})$ 
2 for  $k=L, \dots, 2, 1$  do
3    $\mathbf{g} \leftarrow \nabla_{\mathbf{a}^{(k)}} J = \mathbf{g} \odot \mathbf{f}^{(k)'}(\mathbf{a}^{(k)})$ 
4    $\nabla_{\mathbf{b}^{(k)}} J = \mathbf{g} + \lambda \nabla_{\mathbf{b}^{(k)}} \Omega(\boldsymbol{\theta})$ 
5    $\nabla_{\mathbf{W}^{(k)}} J = \mathbf{g} \mathbf{h}^{(k-1)T} + \lambda \nabla_{\mathbf{W}^{(k)}} \Omega(\boldsymbol{\theta})$ 
   /* Propagate the gradients */
6    $\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = \mathbf{W}^{(k)T} \mathbf{g}$ 
7 end
8 return  $\nabla_{\mathbf{W}^{(k)}} J, \nabla_{\mathbf{b}^{(k)}} J, \forall k = 1, \dots, L$ 
```

- **Riga 1:** \mathbf{g} memorizzerà tutti i gradienti di cui abbiamo bisogno per continuare con l'iterazione. La prima riga fa una inizializzazione di $\mathbf{g} = \nabla_{\mathbf{y}} J = \nabla_{\mathbf{y}} L(\mathbf{y}, \mathbf{t})$. Posso ignorare il termine di regolarizzazione perché non dipende da \mathbf{y} (devo solo derivare la *Loss Function* rispetto a \mathbf{y}), la derivata dipende dalla Loss Function scelta (es. funzione quadratica si ottiene $\nabla_{\mathbf{y}} L(\mathbf{y} - \mathbf{t})$).
- **Riga 2:** A questo punto l'algoritmo inizia ad iterare operando in modo furbo (rete fluisce in un solo verso), itero sui livelli (parto dall'ultimo e arrivo al primo).

Esempio visivo:



Rete con un livello nascosto (3 input, 1 livello nascosto, 2 uscite). Calcolo la derivata di L rispetto al valore delle uscite. Itero all'indietro.

- **Riga 3:**

Bisogna tenere a mente il ruolo fondamentale di g , la quale rappresenta il gradiente della Loss Function rispetto al livello k considerato sul momento! L'uscita di un livello è:

$h^{(k)} = f^{(k)}(a^{(k)})$ Voglio calcolare il gradiente della funzione costo rispetto al valore di preattivazione $a^{(k)}$.

Basta utilizzare la chain rule. $\frac{\partial L}{\partial a^{(k)}} = \frac{\partial L}{\partial h^{(k)}} * f^{(k)'}(a^{(k)}) = g * f^{(k)'}(a^{(k)})$ dove $*$ indica una moltiplicazione elemento per elemento. Nella riga 3 quindi g ha memorizzato il gradiente della funzione costo rispetto ad $a^{(k)} = W^{(k)}h^{(k-1)} + b^{(k)}$. Nota*: Ho usato la derivata della funzione di attivazione, perché se conosco la funzione di attivazione facilmente ottengo la sua derivata.

Ci si potrebbe chiedere cosa succederebbe se l'ultimo livello usasse la *softmax* (utilizzato in caso di classificazione con n -classi), in questo caso la formula scritta a riga 3 dell'algoritmo non è corretta perché questa assume che qualsiasi uscita sia funzione solo del valore di preattivazione dell'ultimo livello e non di tutti come per la *softmax*. I framework implementano anche la variante per soddisfare questo calcolo.

- **Righe 4 e 5:** Viene usato g per ottenere il gradiente rispetto a b e W . Per fare questo si utilizza anche la funzione costo (perché questa dipende da W e b). $J(\theta) = L(y, \theta) + \lambda \Omega(\theta)$. Quindi, si avrà: $\frac{\partial J}{\partial W^{(k)}} = \frac{\partial L}{\partial W^{(k)}} + \lambda \frac{\partial \Omega(\theta)}{\partial W^{(k)}} = \frac{\partial L}{\partial a^{(k)}} \frac{\partial a^{(k)}}{\partial W^{(k)}} + \nabla_{W^{(k)}} \Omega(\theta) = gh^{(k-1)T} + \lambda \nabla_{W^{(k)}} \Omega(\theta)$.

A livello dimensionale mi torna, g ha un elemento per ciascuno dei neuroni del livello k , h ha un elemento per ogni neurone del livello precedente. La matrice $W^{(k)}$ sono i pesi delle connessioni ed ha colonne pari al numero di neuroni del livello precedente e righe pari al numero del livello corrente. Per b si segue lo stesso ragionamento. Queste informazioni le vado a salvare perché sono le informazioni che mi serve mandare in output dall'algoritmo di *back propagation*.

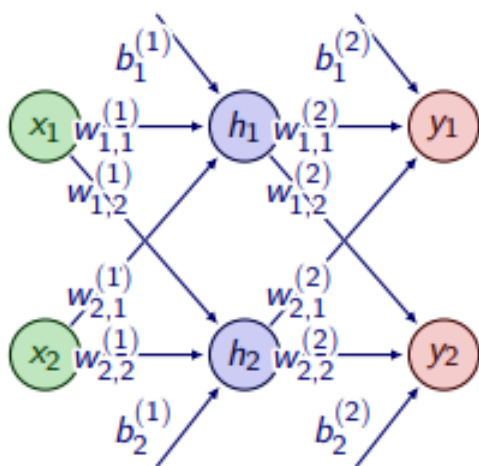
- **Riga 6:** aggiorno g per fare in modo che alla prossima iterazione contenga il gradiente rispetto ad $h^{(k-1)}$ che è quello che mi aspetto nella riga precedente.

A questo punto si continua ad iterare. Mi fermo al primo livello nascosto che contiene gli ultimi parametri. Infine vengono restituiti i valori dei gradienti calcolati.

Recap: Siamo partiti calcolando il gradiente rispetto le uscite, e poi iterando dall'ultimo fino al primo livello nascosto. Siamo passati da h ad a applicando la *chain rule*. La riga 6 propaga all'indietro questi gradienti, e quindi uso g per tale propagazione.

2 Esercizio

Ho una rete neurale NN con input $x = \{0.5, 0.1\}$. Voglio applicare l'algoritmo di backpropagation.



$$W^{(1)} = \begin{bmatrix} 0.15 & 0.25 \\ 0.2 & 0.3 \end{bmatrix}$$

$$W^{(2)} = \begin{bmatrix} 0.4 & 0.5 \\ 0.5 & 0.55 \end{bmatrix}$$

$$b^{(1)} = [0.35, 0.35]$$

$$b^{(2)} = [0.6, 0.6]$$

$$t = [0.01, 0.99]$$

$$\mathcal{L} = \frac{1}{2} \|y - t\|_2^2$$

2.1 Forward phase

Andiamo in avanti, calcoliamo il *pre-activation value* $a^{(1)} = W^{(1)}x + b^{(1)}$, a cui devo però applicare la funzione di attivazione di questo primo livello nascosto, quindi applico la *sigmoid*: $h^{(1)} = \sigma(a^{(1)})$

$$\begin{aligned} a^{(1)} &= W^{(1)}x + b^{(1)} = \\ &= \begin{bmatrix} 0.15 & 0.25 \\ 0.2 & 0.3 \end{bmatrix} \cdot \begin{bmatrix} 0.5 \\ 0.1 \end{bmatrix} + \begin{bmatrix} 0.35 \\ 0.35 \end{bmatrix} = \\ &= \begin{bmatrix} 0.095 \\ 0.155 \end{bmatrix} + \begin{bmatrix} 0.35 \\ 0.35 \end{bmatrix} = \begin{bmatrix} 0.445 \\ 0.505 \end{bmatrix} \end{aligned}$$

$$h^{(1)} = \sigma(a^{(1)}) = \begin{bmatrix} 0.6094 \\ 0.6236 \end{bmatrix}$$

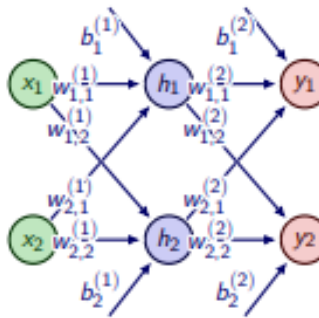
Faccio lo stesso per $a^{(2)}$:

$$\begin{aligned}
\mathbf{a}^{(2)} &= \mathbf{W}^{(2)} \mathbf{h}^{(1)} + \mathbf{b}^{(2)} = \\
&= \begin{bmatrix} 0.4 & 0.5 \\ 0.5 & 0.55 \end{bmatrix} \cdot \begin{bmatrix} 0.6094 \\ 0.6236 \end{bmatrix} + \begin{bmatrix} 0.6 \\ 0.6 \end{bmatrix} = \\
&= \begin{bmatrix} 0.5244 \\ 0.6478 \end{bmatrix} + \begin{bmatrix} 0.6 \\ 0.6 \end{bmatrix} = \begin{bmatrix} 1.1244 \\ 1.2478 \end{bmatrix} \\
\mathbf{y} = \mathbf{h}^{(2)} &= \sigma(\mathbf{a}^{(2)}) = \begin{bmatrix} 0.7548 \\ 0.7770 \end{bmatrix}
\end{aligned}$$

Classificazione binaria risolta male perché i valori di uscita sono 0.7548 e 0.7770. A questo punto calcolo la *funzione di Loss*:

$$\mathcal{L} = \frac{1}{2} \|\mathbf{y} - \mathbf{t}\|_2^2 = \frac{1}{2} \left\| \begin{bmatrix} 0.7448 \\ -0.213 \end{bmatrix} \right\|_2^2 = 0.3$$

2.2 Backward phase



$$\mathbf{g} \leftarrow \nabla_{\mathbf{y}} \mathcal{L} = \mathbf{y} - \mathbf{t} = \begin{bmatrix} 0.7448 \\ -0.213 \end{bmatrix}$$

k=2

$$\begin{aligned}
\mathbf{g} \leftarrow \nabla_{\mathbf{a}^{(2)}} \mathcal{L} &= \mathbf{g} \odot \sigma'(\mathbf{a}^{(2)}) = \\
&= \begin{bmatrix} 0.7448 \\ -0.213 \end{bmatrix} \odot \begin{bmatrix} \sigma(1.1244)(1 - \sigma(1.1244)) \\ \sigma(1.2478)(1 - \sigma(1.2478)) \end{bmatrix} = \\
&= \begin{bmatrix} 0.7448 \\ -0.213 \end{bmatrix} \odot \begin{bmatrix} 0.1851 \\ 0.1733 \end{bmatrix} = \begin{bmatrix} 0.1378 \\ -0.0369 \end{bmatrix} \\
\nabla_{\mathbf{W}^{(2)}} \mathcal{L} &= \mathbf{g} \mathbf{h}^{(1)T} = \begin{bmatrix} 0.1378 \\ -0.0369 \end{bmatrix} \cdot \begin{bmatrix} 0.6094 & 0.6236 \end{bmatrix} = \\
&\quad \begin{bmatrix} 0.0840 & 0.0860 \\ -0.0225 & -0.0230 \end{bmatrix}
\end{aligned}$$

31

Inizializziamo la variabile di appoggio \mathbf{g} rispetto al valore di uscita \mathbf{y} . Poi iteriamo dall'ultimo livello fino al primo.

Derivo la funzione di applicazione e moltiplico per g , facendo un prodotto *elemento per elemento*. Ottengo una matrice, di cui ciascun elemento è la **derivata parziale della Loss rispetto ad uno di quegli elementi**.

Una volta che ho i gradienti posso fare un apprendimento, aggiornare i pesi W in modo opposto al gradiente (sottraggo α per il gradiente). La Loss sarà diminuita, andando avanti diventa sempre più piccola.

2.2.1 Costo computazionale

Partiamo da un grafo computazionale generico. Assumiamo che le operazioni abbiano tutte lo stesso costo. Se ho N nodi, la propagazione in avanti ha un costo proporzionale ad N ossia il numero di variabili in gioco. La propagazione indietro mi porta un'operazione per ogni arco del grafo, in generale quindi $O(N^2)$.

Nel nostro caso la rete feedforward è completamente connessa (L hidden layers con m neuroni ciascuno). Non consideriamo più le operazioni tutte con lo stesso peso (prodotto tra matrici pesa di più). In questo caso in *entrambe le fasi* abbiamo un costo pari a $O(Lm^2)$ che è un piccolo costo rispetto a tutto l'algoritmo di addestramento.

2.3 Generalizing Backpropagation

Nella pratica solitamente si lavora con *mini-batch* di input dove ogni riga rappresenta una istanza del training set. Concettualmente rispetto a prima non cambia niente (posso pensare di linearizzare la matrice come un vettore). Il training funziona allo stesso modo di minibatch: ho un ciclo infinito (finisce solo al raggiungimento di alcuni criteri), calcolo il gradiente ed aggiornare i parametri prendendo sempre mini-batch randomicamente.

Usare mini-batch ha due benefici:

- *a livello computazionale*, il mini-batch più grande permette di parallelizzare a livello hardware alcune operazioni previste dall'algoritmo.
- *a livello di convergenza*, il mini-batch più piccolo permette una convergenza più veloce.

NB: Per scegliere la dimensione del mini-batch solitamente si sceglie una potenza di due perché tutte le dimensioni a livello hardware sono potenze di due. Il parametro ideale dipende anche dall'hardware utilizzato (in particolare GPU). Nella pratica, dato che aumentare troppo la dimensione peggiora la convergenza, bisogna ottimizzare anche questo parametro (di solito iper-parametro ottimizzato alla fine).

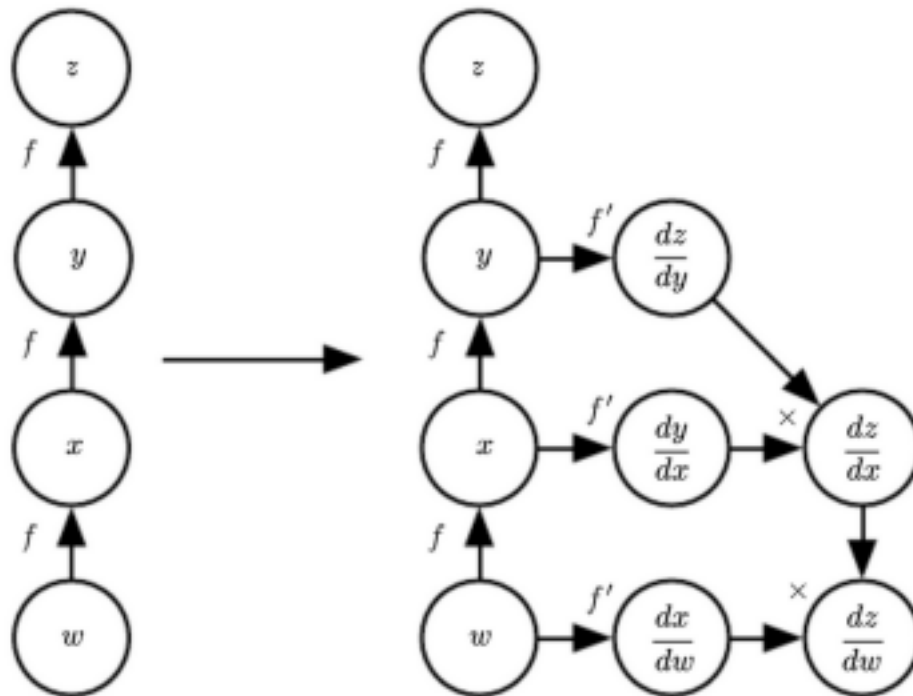
2.3.1 Symbol-to-Number Differentiation

Quando si definisce un Computational graph, questi li definiamo tramite *simboli* che rappresentano le variabili in gioco (maniera simbolica).

L'algoritmo di *Back-Propagation* è stato descritto come un approccio che a partire da valori di input dà in output i valori del gradiente che mi interessa. Questo approccio si chiama **symbol-to-number** differentiation.

Esiste anche l'approccio **symbol-to-symbol**, in cui estendiamo il grafo originale con nuovi nodi che forniscono descrizioni simboliche delle derivate di nostro interesse. Il vantaggio è che se l'algoritmo è efficiente in *forward*, anche la *differenziazione* è efficiente. Tuttavia, sembra che si debbano

conoscere per forza le derivate, altrimenti non si può andare avanti! Possiamo generalizzare ancora di più?



2.4 Generalizzazione maggiore

I framework solitamente permettono di integrare nuove operazioni al loro interno. Tipicamente i framework definiscono una classe astratta **operation** che deve esporre un'interfaccia ben definita, in particolare **backprop()** che definisce la derivata dell'operazione aggiunta, il framework a quel punto sarà in grado di usarla all'interno dell'algoritmo di backpropagation. Ciò ci permette di *inventare* nuove funzioni di attivazione, basta definirla insieme alla sua derivata, e successivamente usarla in un grafo più complesso.