Lez4_

October 29, 2023

1 Lezione 4 (Russo Russo)

2 Training reti neurali

Addestrare rete neurale è la stessa cosa di addestrare qualsiasi altro modello, si ha l'obiettivo di ridurre la funzione costo calcolando i parametri che la minimizzano (solitamente versione stocastica della discesa del gradiente). I parametri del modello (indicati con vettore θ) include matrici W di tutti i livelli che i vettori di bias, proprio con loro lavoriamo per ridurre la funzione costo.

2.1 Funzione costo

Addestarre modello parametrico si vuole minimizzare funzione costo. Abbiamo la **Loss Function** (errore modello nelle predizioni), supponiamo di avere task di regressione con una loss pari ad un errore quadratico medio (non scelta obbligata) $\mathcal{L}(\theta) = \frac{1}{2}(t-y)^2$, la **Cost Function** potrebbe comprendere anche un termine di regolarizzazione: $J(\theta) = \sum_{i=1}^{N} \frac{1}{2}(t^{(i)} - y^{(i)})^2 + \lambda\Omega(\theta)$.

Qui calcoliamo l'errore quadratico medio su tutto il training set e l'errore di regolarizzazione. λ è un iperparametro reale che dice il peso da dare alla funzione di regolarizzazione.

Una rete neurale si può addestrare con discesa stocastica del gradiente (**GSD**) che prevede più epoche di addestramento (epoca = ciclo che scorrono il training set). In ogni ciclo si prendono esempi dal training set e per ognuno si calcola la funzione costo (in particolare il gradiente della funzione in base ai parametri del modello, gradiente= come cambiarebbe la funzione costo al variare dei parametri).

Aggiorno θ a questo punto spostandomi nella direzione opposta al gradiente di un certo passo dipendente da α.

Problemi: - Nelle reti neurali abbiamo inserito non linearità (se non ci fosse non cambierebbe niente dagli altri modelli), sono funzioni di costo non convesse quindi non si ha garanzia che il gradiente porti a raggiungere un minimo globale (si può rimanere intrappolati in un minimo locale della funzione di loss). (Spesso minimi locali in reti neurali hanno minimi vicini tra loro, quindi non è un grande problema). - (Problema trattato oggi) nella rete neurale *calcolo gradiente non è tanto banale*, il problema è come calcolo in maniera efficiente il gradiente (per un solo componente è facile, per tutti diventa lungo).

2.2 Backpropagation

Per risolvere il secondo problema si usa l'algoritmo di Backpropagation che permette di calcolare in modo effiente il gradiente di una funzione. Non include tecniche per aggiornare i pesi, quindi

non è un algoritmo di addrestramento. In particolare è interessante perché viene applicato alle reti neurali.

2.2.1 Algoritmo

L'algoritmo si compone di due fasi:

- **forward fase**: ho un'istanza di training set, la dò alla rete e calcolo le varie funzioni di attivazione, output e funzione costo. Mi "muovo" in avanti, quindi dalle varie attivazioni dell'input verso l'uscita. L'output finale è usato per calcolare la funzione costo.
- **backward fase**: calcolo il gradiente in base ai parametri calcolati attraversando la rete neurale all'indietro, ovvero partendo dall'output.

Questo viene fatto tramite programmazione dinamica perché dovrò riutilizzare varie volte lo stesso risultato e la chain rule.

Forward propagation Ho un vettore $\bar{x}$ di input e attraverso la rete in avanti. Per prima cosa si calcola la variabile z di preattivazione del livello $z = W^{(1)}x + b^{(1)}$ e calcolo poi le h ($\phi(z)$). Infine, mi trovo le $y = W^{(2)}h + b^{(2)}$. (Tutto questo come già visto precedentemente, avendo y posso calcolare anche la loss). Qui abbiamo supposto di avere un unico livello nascosto h e una variabile temporanea z, rappresentante la variabiale di pre-attivazione del livello.

2.2.2 Chain rule

Date due funzioni $f : R \to R$ e $g : R \to R$; posso calcolare la derivata della funzione composta come: $\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$.

Il cambiamento di x porterà a un cambiamento di g che a sua volta porterà un cambiamento di f.

Se associo $y = g(x)$ e $z = f(y) \to \frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$.

Nell'algoritmo bisogna considerare il caso in cui si hanno n funzioni $g_i : R \to R$ e una sola funzione $f : R^n \to R$ e voglio calcolare: $\frac{\partial f(g_1(x),\ldots,g_n(x))}{\partial x} =$

$\sum_{k=1}^{n} \frac{\partial f(g_1(x),\ldots,g_n(x))}{\partial g_k(x)} \frac{\partial g_k(x)}{\partial x}$.

Ogni piccola variazione della x provoca una piccola variazione della g che provoca quindi un piccola variazione di f.

(Se aiuta posso vedere le $g : R \to R^n$ come un unico vettore e scriverlo come il caso precedente).

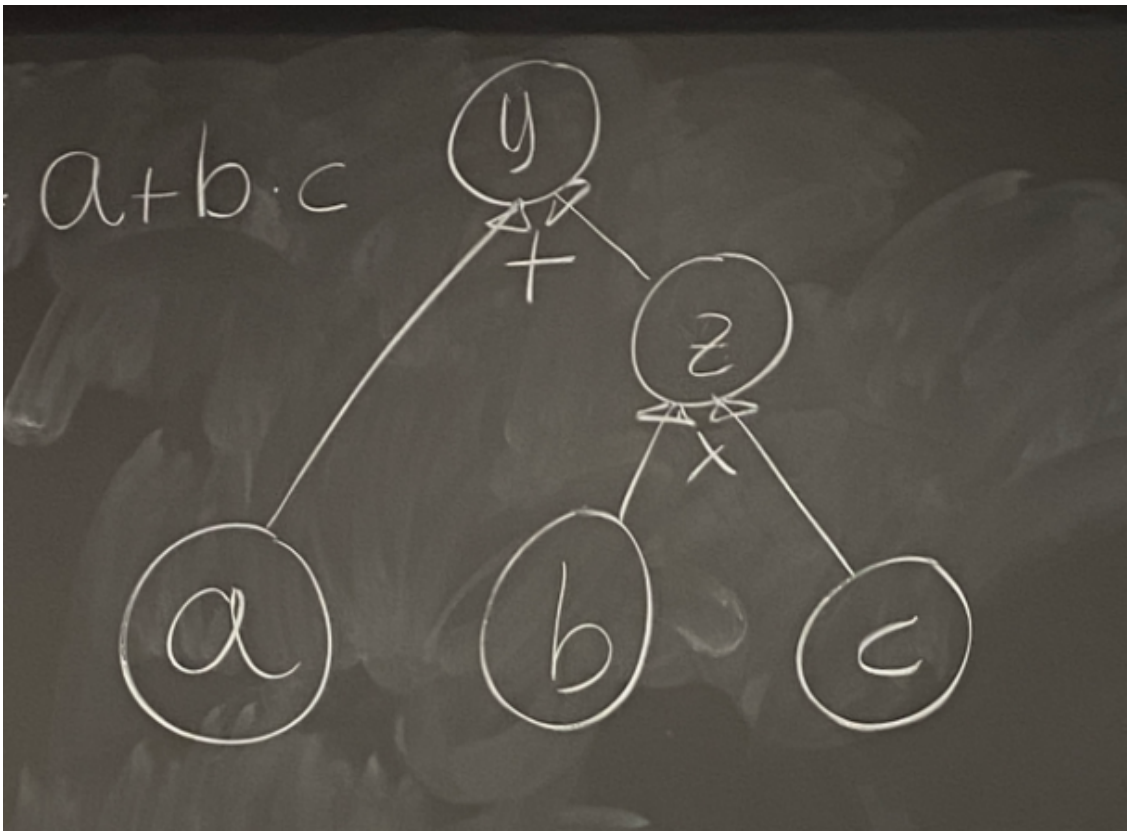
Ulteriore complicazione g potrebbe rappresentare il livello nascosto h, quindi mi interessa anche il caso in cui $g : R^m \to R^n$ e $f : R^n \to R$. in questo caso si avrà .... Si complica la notazione in termini vettoriali. $\nabla_x f(g(x)) = (\frac{dg}{dx})^T \nabla_g f(g(x))$ dove $\frac{dg}{dx}$ è la matrice Jacobiana di g(x). Ogni riga contine il gradiente di una g_i rispetto alla variabile x: $\frac{dg}{dx} = \begin{bmatrix} \frac{dg_1}{dx_1} & \frac{dg_1}{dx_2} & \cdots & \frac{dg_1}{dx_m} \\ \frac{dg_2}{dx_1} & \frac{dg_2}{dx_2} & \cdots & \frac{dg_2}{dx_m} \\ \vdots & \ddots & \ddots & \vdots \\ \frac{dg_m}{dx_1} & \frac{dg_m}{dx_2} & \cdots & \frac{dg_m}{dx_m} \end{bmatrix}$.

Se invece di avere vettori avessimo *tensori*(spazio n-dimensionale) non cambierebbe nulla! Posso trasformare i tensori in matrici di parametri.

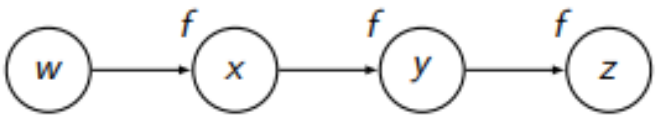
Computational graph - Grafo computazionale Caratterizzato da:

- Ogni nodo indica una variabile.
- Le operazioni sono funzioni di una o più variabili che da come risultato un singolo valore.

Quando *interconnetto* due nodi x e y (ovvero parliamo di *arco (x,y)*) sto indicando che y è il risultato di una certa operazione applicata ad y. Il fatto che questo sia un grafo mi permette di visitarlo ed eseguire diverse operazioni. È interessante perché permette di calcolare l'uscita dei livelli delle reti neurali (per arrivare a costruire poi la rappresentazione di tutta la rete neurale). Può essere utilizzata per calcolare la *chain rule*. (da esempio slide si avrà) $\frac{dz}{dw} = f'(f(f(w))f'(f(w))f'(w)$. Ci sono termini che vengono calcolati più volte, qui entra in gioco la programmazione dinamica (mi salvo i valori non mi serve calcolarli di nuovo). ##### Esempio Sia $y = a + b \cdot c$, il grafo corrispondente è:



Nb: z è figlio di b e c.



$$z = f(y) = f(f(x)) = f(f(f(w)))$$

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} =$$
$$= f'(y)f'(x)f'(w) =$$
$$= f'(f(f(w)))f'(f(w))f'(w)$$

Applying the rule, we often need to evaluate subexpressions more than once. We either store them or re-compute them

Esempio 2
L'idea è che, se devo derivare in funzione di un qualcosa, e non lo so fare, cerco di trasformare l'operazione di derivata in un'altra operazione di derivata che però so eseguire. Nella figura, non so derivare z rispetto w, ma so farlo rispetto y, e quindi riscrivo la derivata seguendo questo ragionamento.

2.3 Backpropagation con variabili scalari

Considero un grafo computazionale generico con n nodi. Le prime $k < n$ variabili rappresentano un input, e assumo un solo output (l'ultimo). L'obiettivo è calcolare il *gradiente* dell'output rispetto ad ogni input. Assumo che ogni nodo u_i è associato ad una certa funzione $f^{(i)}$ (per ogni $i > k$, nodi intermedi). A partire dagli input calcolo tutti i valori delle varibili del grafo.

Assunzione: i nodi sono organizzati secondo un ordinamento topologico (ordinamento dei nodi tale che se esiste un arco da b a z allora nell'ordinamento b viene prima di z, poi in mezzo potrebbe anche esserci altro!).

Si inizializzano le variabili di input con i valori di input nel primo ciclo, poi si scorrono i vari nodi e si applicano le operazioni previste dando un valore a tutte le variabili. Se riprendiamo l'esempio sulla lavagna del prof, avremmo le variabili a, b, c e l'output y.

- Each node $u^{(i)}$ is associated with an operation $f^{(i)}$

$$u^{(i)} = f^{(i)}(\mathcal{A}^{(i)}), \quad \mathcal{A}^{(i)} = \{u^{(j)} | j \in Parent(u^{(i)})\}$$

Forward pass (with scalar variables only)

```
1 for i = 1, ..., k do
2 |   u^(i) ← x_i                    /* Input values */
3 end
4 for i = k + 1, ..., n do
5 |   u^(i) ← f^(i)(A^(i))
6 end
7 return u^(n)
```

$\mathcal{A}$ è l'insieme dei nodi interconnessi ad un generico nodou_i.

Per velocizzare le operazioni, si usa una **lookup table** `grad` per memorizzare le derivata di $u^{(n)}$ rispetto alle varie variabili ($grad[u^{(i)}] = \frac{dg}{dx}$). Inizializzo $grad[u^{(n)}] = 1$, ovvero la derivata di una variabile rispetto *se stessa*. Scorrendo poi la rete al **contrario** calcolo i vari $grad[u^{(i)}]$, applicando la *chain rule*. Una volta fatto questo restituisco il gradiente di $u^{(n)}$ rispetto a tutte le variabili di ingresso.

- We use a lookup table `grad` to store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ for every $u^{(i)}$
- For convenience, let $children(j) = \{i | j \in Parent(u^{(i)})\}$

Backward propagation (with scalar variables only)

1 Run forward pass (previous slide) to evaluate variables
2 Initialize empty lookup table `grad`
3 $\texttt{grad}[u^{(n)}] \leftarrow 1$
4 **for** j=n-1,n-2,...,1 **do**
  /* Apply chain rule */
5 $\texttt{grad}[u^{(j)}] \leftarrow \sum_{i \in children(j)} \texttt{grad}[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$
6 **end**
7 **return** $\{grad[u^{(i)}], i = 1, ..., k\}$

2.3.1 Backpropagation nelle reti neurali

Assumo *singola istanza* di cui voglio calcolare predizione, errore e gradiente. Voglio calcolare $\nabla_\theta J(\theta)$ gradiente di J rispetto a tutti i parametri della rete θ.

Forward pass

Definisco una certa $h^{(0)}$ per poi scorrere sui vari livelli (ordinati secondo un certo criterio) per calcolare $a^{(k)} \leftarrow W^{(k)} h^{(k-1)} + b^{(k)}$ funzione di preattivazione, di conseguenza calcolo anche $h^{(k)} \leftarrow f^{(k)}(a^{(k)})$.
Alla fine calcolo $y \leftarrow h^{(L)}$ per poi restituire la funzione di costo $J = L(y, t) + \lambda \cdot Omega(\theta)$

Forward pass

1 $h^{(0)} \leftarrow x$
2 **for** k=1,..,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** $J = L(y, t) + \lambda \Omega(\theta)$

[ ]: