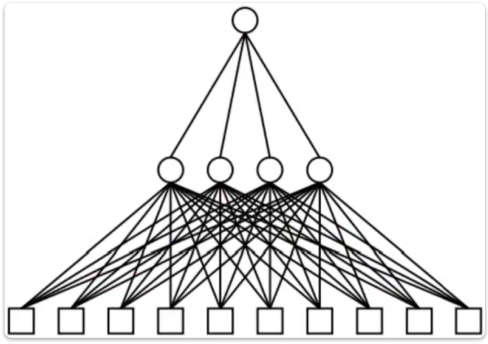
Lezione09

Table of contents

Multi-Layer Perceptrons (MLPs)



Nel momento in cui aggiungiamo almeno 1 livello al nostro percettrone, questo prenderà il nome di Multi-Layer Perceptron (MLP). Di livelli possiamo aggiungerne quanti ne vogliamo, tenendo conto che la rete è sempre una feed-forward; è proprio da questo fatto che la parola *deep learning* emerge. I MLPs:

- sono le reti più utilizzate al giorno d'oggi, perché le SLPs con un solo livello non aggiungono complessità necessaria per calcolare risultati con buona accuratezza;
- hanno almeno 1 *layer nascosto*;
- il numero di neuroni per ogni strato nascosto viene fissato a priori.

(Osserviamo l'immagine) La nomenclatura della rete così segue:

- a_k sarebbe il finto layer contenente i valori d'input, dove non viene fatto alcun calcolo;
- $W_{k,j}$ sono i pesi che collegano gli input al primo layer nascosto;
- a_j sono le unità nascoste, calcolate dagli input precedenti.

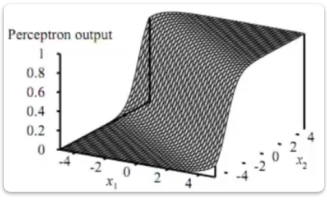
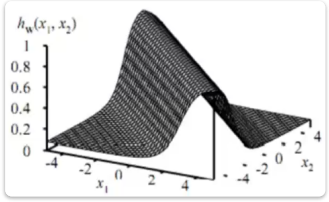
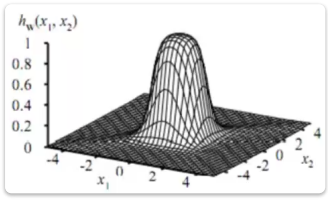
Per un MLP, si parla di *profondità* della rete quando ci chiediamo quanti strati questa abbia; si parla di *larghezza* del singolo layer quando ci si chiede quanti neuroni questo abbia. Ogni strato può avere un numero diverso di neuroni tranne che per quello d'ingresso e quello di uscita, fissati dalla natura del problema. Notare come strutture triangolari compaiano nella schematizzazione della rete: in realtà non è necessario che sia così e non è nemmeno sbagliato se pensiamo di allargare un livello rispetto a uno precedente.

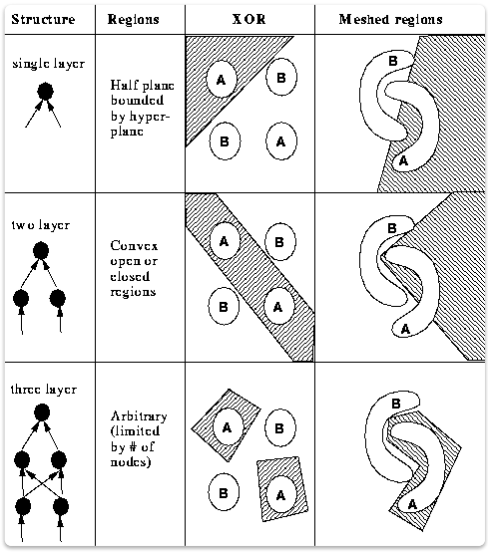
Costruire una rete a multi strati è decisamente più tassante che costruirne una con un solo livello, proprio perché il numero di pesi aumenta considerevolmente; la geometria che tenta di considerare i pesi nello spazio, diventa più difficile da usare; l'*errore* che si accumula per ogni layer nascosto computato, inciderà sul risultato finale. L'algoritmo che useremo per diminuire questo errore, non sarà altro che quello di discesa del gradiente.

Il numero di neuroni usato per la computazione, non necessariamente rimarrà fisso fino alla fine: nel momento in cui pesi si dichiareranno a zero (o molto vicini), la rete potrà decidere se eliminare o meno i neuroni che si vanno a rendere non necessari.

Comprendiamo come un MLP sia quindi intrinsecamente migliore di un SLP.
Se mettiamo anche un solo strato intermedio, la rete funzionerà meglio e anzi, più avanti, impareremo un

teorema che enuncia "un qualsiasi MLP a strato nascosto singolo, sarà sempre sufficiente per approssimare con accuratezza arbitraria una funzione continua".

		
Un SLP ouput $\mathcal{B} = \{0, 1\}$	Un MLP con strato in più	Un MLP ancora più preciso



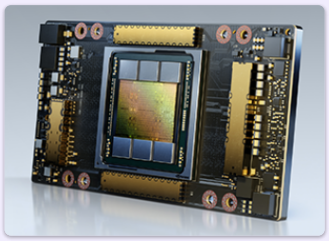
(Osserviamo la tabella) Vediamo la valutazione della funzione `XOR`.

- Usando un SLP, abbiamo imparato che l'approssimazione della funzione non è affatto buona; l'ouput non riusciamo a costruirlo se non con una classificazione con errore 1/4. La situazione peggiora se consideriamo una geometria degli input come sopra, dove A e B sono tra loro concavi (sbagliamo sempre almeno uno dei due).
- Usando un MLP possiamo magari aiutarci aggiungendo uno strato intermedio, un percettrone a 2 livelli. Ipotizziamo di mettere 2 bit input, 2 bit intermedi, 1 bit output. La nostra rete sarà capace di cose più interessanti: i nostri input A sono classificati insieme, come lo sono anche gli input B se girato l'asse che li prende in considerazione; almeno uno dei due è classificato correttamente.
- Siccome però questo non ci basta, poniamoci le domande:
  - allarghiamo la rete?
  - allunghiamo la rete?

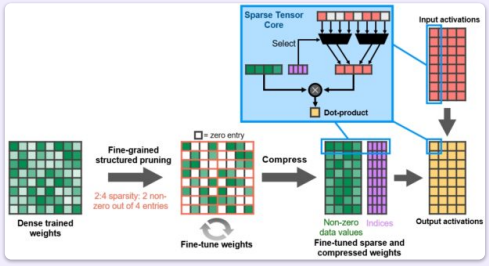
  Potremmo allargare(cosa che vedremo più avanti), me per rimanere coerenti con l'esempio, facciamo un invece un allungamento, aggiungendo uno strato intermedio in più. Con 2 strati intermedi di 2 bit input, 2 neuroni per ogni layer nascosto e 1 output, la nostra rete sarà capace di approssimare in modo esatto.

Nella teoria, una MLP a *3 strati*, permette un'approssimazione esatta, fissata una tolleranza arbitraria, di una funzione non continua. Proprio per la proprietà dei pesi nulli visti sopra, avere questo numero di strati non necessariamente porta a prestazioni peggiori, visto che i neuroni i cui pesi sono prossimi allo zero o molto piccoli, possono essere eliminati.

Le GPU in campo AI



NVIDIA A100 Tensore Core GPU



L'architettura dei Tensore Core

Grazie ai calcoli paralleli e all'uso delle shaders, e rese molto popolari dai recenti trend, questi tipi di GPU nascono per il solo scopo di essere le migliori in campo nell'ambito del deep learning e dell'intelligenza artificiale in genere.

The Back Propagation Algorithm

Una volta capito che dobbiamo fissare una topologia della rete (numero di strati e numero di nodi), una volta capito quali pesi vanno impostati a 0 e deciso l'unico spazio di convoluzione su cui vogliamo addestrare, ci serve un algoritmo di addestramento.

L'algoritmo che viene sempre usato per questa tipologia di reti è sempre e solo uno ed è quello di back propagation. Prende questo nome perché strutturato in 2 fasi:

1. la *fase di propagazione in avanti*, che non è nient'altro che il calcolo dell'uscita;
2. la *fase di propagazione all'indietro*, applicazione della correzione ai pesi data dall'algoritmo di discesa del gradiente, applicando l'algoritmo un livello alla volta.

Per come nasce la formula che corregge i pesi sulla base dell'errore finale, calcolato in funzione di quello proferito dal supervisore, possiamo fare una correzione dei pesi un livello alla volta.

```
// Back-Propagation algorithm
procedure BackPropagation
        Initialize weights to small random values in (-1,1)
        do
                Initialize the cumulative error to zero
                for each pattern in the training set do
                        call ForwardPropagate
                        call BackPropagate
                        Update teh cumulative error
                end
        while (number of epochs < max number of epochs) and
                (cumulative error > accepted tolerance)
end
```

- Prima di tutto vengono inizializzati i pesi a un valore casuale (`Initialize weights`), tipicamente abbastanza basso in valore assoluto per far sì che se già in prossimità dello zero, questo impieghi poco ad arrivarci.
- Viene inizializzato l'errore cumulativo (`Initialize the cumulative error`) sul training set a 0.
- Per ogni pattern del training set (`for each`)
  - viene calcolata l'uscita del training set (`ForwardPropagate`);
  - presa l'uscita della rete e calcolato l'errore rispetto all'uscita aspettata proferita dall'oracolo, che poi va ad aggiornare i pesi all'indietro verso l'input (`BackPropagate`);

    - aggiornato l'errore cumulativo e passiamo al prossimo vettore/pattern (`Update cumulative error`).
- Arrivati alla fine del training set, ricominciamo, finché l'errore cumulativo compiuto non scende sotto la tolleranza che ammettiamo (`cumulative error > accepted tolerance`) e finché non abbiamo raggiunto il numero di epoche che abbiamo deciso essere il massimo (`number of epochs < max number of epochs`).

Esempio: una rete passa al massimo 1K volte il training set; se con 1K passate del training set si otterrà un errore cumulativo complessivo abbastanza basso, verrà semplicemente segnalato l'errore comunque elevato. Viceversa, se l'errore compiuto è più basso, la rete si fermerà segnalando di aver trovato la sua forma ottimale.

Come mai a ogni ciclo compiuto, l'errore viene annullato?

```
Initialize cumulative error to zero
```

Viene fatto perché l'errore cumulativo viene calcolato sul training set.
L'errore cumulativo ci dice quant'è l'errore che andiamo a compiere su tutti i pattern del training set.

Prendiamo in esame l'esempio dello **XOR**: se abbiamo i 4 bit, uno di questi sarà sbagliato producendo un errore cumulativo di 1/4 (25%). Non vogliamo che questo errore venga moltiplicato per il numero di volte con cui facciamo il ciclo `while` esterno, piuttosto vogliamo dire all'utente che su tutto il training set stiamo sbagliando questa quantità.

```
// Back-propagate the error through the network
procedure BackPropagate
        for each neuron in the ouput layer do
                compute the actual error
        end

        for each hidden layer do
                for each neuron in the current layer do
                        1. Compure the fraction of the error for the neuron
                        2. Update weights using the computed fraction of the error
                end
        end
end
```

Update Rule

$$W_{i,j} \leftarrow W_{i,j} + \alpha \cdot a_j \cdot \Delta_i \qquad \Delta_i = \mathtt{Err}_i \cdot g'(\mathtt{In}_i)$$

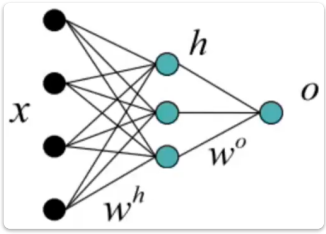
Quello che viene fatto è applicare la tecnica di discesa del gradiente ai pesi $W_{j,i}$ che collegano lo strato nascosto con l'uscita, fatto perché conosciamo l'errore che andiamo a compiere sulla stessa. Una volta calcolata la variazione dei pesi utilizzando l'errore effettivo calcolato (Δ_i), la stessa formula l'applichiamo per aggiornare i pesi $W_{k,j}$

$$\Delta_j = g'(\mathtt{In}_j) \sum_i W_{j,i} \Delta_i \qquad W_{k,j} \leftarrow W_{k,j} + \alpha \cdot a_k \cdot \Delta_j$$

Il nuovo delta non è più l'errore effettivo, siccome sugli strati intermedi l'oracolo non ha modo di conoscerlo, ma quello dipeso dagli errori compiuti (per semplificare le formule, **Err** è diventata la delta moltiplicata per derivata della funzione d'attivazione $g'(\mathtt{In}_j)$.

Se la funzione d'attivazione è la funzione logistica, possiamo sfruttare il fatto che la derivata della funzione logistica è parente stretta della funzione logistica.
Le formule si semplificano: spesso nei testi si trovano formule di aggiornamento dei pesi nelle ipotesi:

1. funzione con unico strato intermedio;
2. funzione di attivazione logistica.



(Osserviamo lo schema) Le seguenti formule descrivono la regole di aggiornamento usando la funzione logistica come funzione di attivazione:

- Strato di uscita

$$\Delta w_j^o = \alpha \delta^o h_j \qquad \delta^o = (y - o) \cdot o \cdot (1 - o)$$

  La variazione dei pesi con apice ^{o} con indice j-esimo, ovvero la correzione che facciamo su uno di questi pesi, dipende dal tasso d'addestramento α moltiplicato per il delta δ calcolato sull'errore o, moltiplicati per l'output sul j-esimo neurone h.

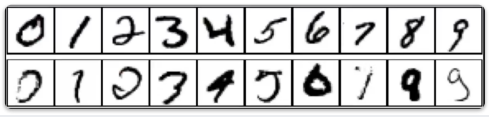
- Strato d'ingresso

$$\Delta w_{ij}^h = \alpha \delta_j^h x_i \qquad \delta_j^h = \delta^o \cdot w_j^o \cdot h_j \cdot (1 - h_j)$$

  La formula è la stessa, siccome conosciamo già i risultati, calcolati con la formula degli strati in uscita. I valori h vengono sostituiti da x_i input dello strato intermedio. La variazione dei pesi andrà ad applicarsi una volta fissata l'uscita su cui stiamo lavorando, ai pesi che collegano l'i-esimo valore di input con l'h-esimo valore di output.

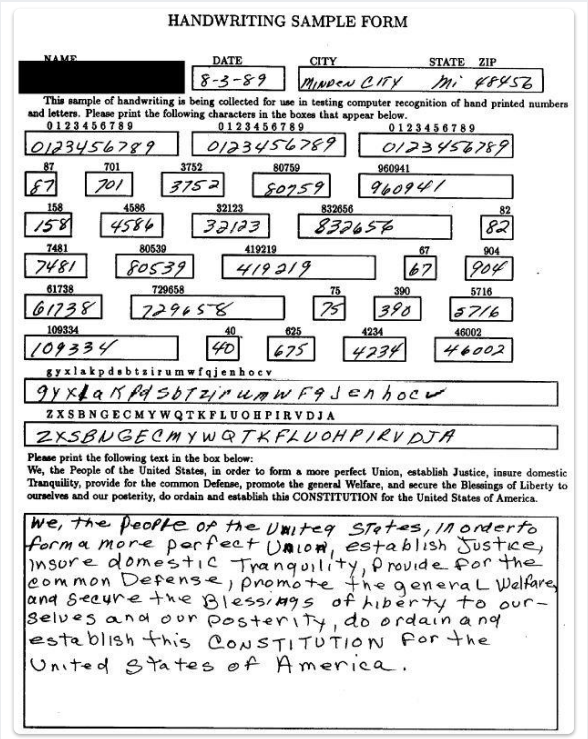
Handwritten Character Recognition

Una volta costruite le formule di aggiornamento pesi, possiamo creare una MLP che riesca a identificare dei caratteri, per esempio i numeri.



Una serie di numeri ben/mal formati

All'inizio degli anni '70, il National Institute for Science and Technology (NIST), avviò la costruzione di un training set di caratteri manoscritti pre-classificati, per l'addestramento supervisionato di una rete (https://www.nist.gov/itl/products-and-services/emnist-dataset).



Uno dei campioni compilati

Il dataset dei numeri ModifiedNIST (MNIST) ha un formato preciso: ogni cifra è contenuta in una casella di 28×28 pixel, 60'000 campioni e 10'000 per la validazione.

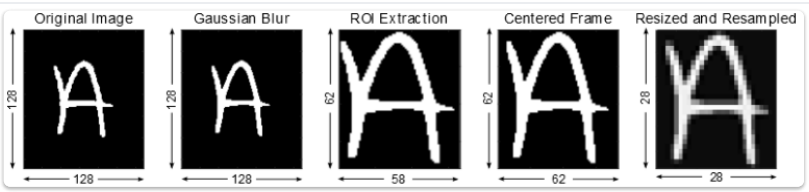


Diagramma del processo di conversione del dataset

Originariamente, le cifre campione avevano un'ampiezza di 128×128; per diminuire la dimensione binaria e quindi alleggerire il peso complessivo del set, un "Gaussian blur" e una "ROI Extraction" sono state eseguite insieme a un centramento.

L'efficacia riportata della rete neurale il cui solo scopo era solo quello di riconoscere le cifre, era doppia rispetto a test effettuati usando esseri umani: la rete era precisa il doppio rispetto all'uomo nel riconoscere i numeri. In percentuale di errore, è stato calcolato uno complessivo su 400-300-10 unità del 1.6%, mentre del 2.4% sulle 3 più vicine cifre, testamento della precisione.

Errori quindi potevano comunque avvenire: nel momento in cui un numero sia scritto in modo ambiguo (perché somiglia più a uno piuttosto che un altro), la rete ha difficoltà nell'identificare la cifra correttamente; i bit associati a una cifra si accendono quando questi dovrebbero invece stare spenti. L'intervento dell'oracolo, che corregge la rete dicendo che una cifra è piuttosto un'altra, riduce l'errore della rete.

23/03/2023