L U I G I S A N T A G A D A

I n t r o d u z i o n e a l l e R e t i N e u r a l i A r t i f i c i a l i

$$\Delta W_{ik} = -\epsilon\left(\sum_{j=1}^{m} \frac{\partial E}{\partial \hat{y}_j} \cdot \frac{\partial \hat{y}_j}{\partial Z_j} \cdot \frac{\partial Z_j}{\partial H_k}\right) \cdot \frac{\partial H_k}{\partial Z_k} \cdot \frac{\partial Z_k}{\partial W_{ik}}$$

Verso la nuova rivoluzione tecnologica

L’autore

Luigi Santagada ha frequentato l’Università degli Studi di Milano, presso la Facoltà di Informatica, nei primi anni ’90, per poi intraprendere un percorso lavorativo che lo ha portato a collaborare principalmente con Microsoft. Durante la sua carriera, ha lavorato con importanti clienti come Vittoria Assicurazioni e Mediobanca, ricoprendo ruoli chiave nell’Application Lifecycle Management (ALM) e nell’ingegnerizzazione del software, sviluppando framework ad hoc e soluzioni di Continuous Integration (CI).

Appassionato di intelligenza artificiale fin dai primi anni ’90, ha progettato sistemi di controllo per il settore automotive per diverse realtà, combinando hardware e software basati su AI. Questi sistemi sono in grado di apprendere e monitorare veicoli e infrastrutture elettriche, rilevando inefficienze prima che si verifichino guasti.

Si definisce con orgoglio un Maker, convinto che l’intelligenza artificiale, unita alla semplicità d’uso dei microcontrollori moderni, possa favorire una nuova ondata di interesse verso l’elettronica e lo sviluppo software applicato a casi reali.

Prefazione

Questo libro si rivolge a tutti coloro che vogliono capire il funzionamento delle reti neurali artificiali per mantenersi al passo con le nuove sfide lavorative legate all'intelligenza artificiale, che sempre più frequentemente permea ogni ambito professionale, dalla produzione industriale all'assistenza sanitaria, fino alla finanza e al marketing. L'evoluzione rapidissima delle tecnologie rende indispensabile per lavoratori, manager e imprenditori conoscere le potenzialità e i limiti di questi strumenti, che non rappresentano più soltanto una prospettiva futura ma una realtà ormai consolidata e in continua espansione. Nel testo troverete spiegazioni semplici e dettagliate, esempi pratici e scenari concreti di applicazione, che vi permetteranno di acquisire familiarità con concetti come il deep learning, il training supervisionato, e la capacità delle reti neurali di apprendere autonomamente dai dati. Oltre a illustrare i principi teorici, il libro affronta aspetti pratici della progettazione e implementazione delle reti, aiutando il lettore a sviluppare competenze immediatamente spendibili nel mondo del lavoro. Che siate programmatori, tecnici, ricercatori o semplicemente appassionati di innovazione, questo libro vi guiderà passo dopo passo nella comprensione di un fenomeno destinato a influenzare profondamente la vostra carriera e il futuro stesso del mercato del lavoro.

Tutto ciò che verrà approfondito sarà prevalentemente concentrato sulle reti neurali basate sul metodo della backpropagation, un approccio supervisionato classico e consolidato che costituisce ancora oggi il cuore pulsante della maggior parte dei modelli moderni di apprendimento automatico e deep learning, come ad esempio le reti convoluzionali (CNN), le reti ricorrenti (RNN) e le reti generative avversarie (GAN). Si tratta quindi di un punto di

partenza ideale per chi desidera avvicinarsi allo studio delle reti neurali artificiali, poiché permette di comprendere in maniera chiara e immediata i concetti fondamentali di apprendimento supervisionato e addestramento tramite correzione degli errori.

Tuttavia, l'originalità di questo libro risiede principalmente nel suo forte orientamento all'applicazione pratica delle reti neurali artificiali sui microcontrollori,con lo scopo finale di permettere al lettore non solo di comprendere teoricamente le reti neurali, ma di essere concretamente in grado di progettare, sviluppare ed eseguire modelli neurali in applicazioni embedded.

In molti sono convinti che i microcontrollori AVR non siano microcontrollori stupidi; spesso è chi li usa ad esserlo. Pertanto non disdegnando l'utilizzo di piattaforme diffuse e facilmente accessibili come Arduino, il testo mantiene comunque un'impostazione altamente versatile e compatibile con una vasta gamma di microcontrollori e microprocessori, inclusi PIC, ARM, ESP32 o PC, adattandosi alle esigenze dei professionisti e degli hobbisti più esigenti.

In sintesi, questo libro non si limita ad essere un semplice manuale introduttivo alle reti neurali, ma costituisce una guida pratica e completa per sviluppare competenze avanzate e concretamente utilizzabili nell'ambito dell'elettronica embedded e del machine learning applicato, colmando il divario esistente tra teoria e realizzazione pratica su dispositivi a basse risorse di calcolo.

Il codice utilizzato nel libro sarà scritto prevalentemente in C o C++, con alcuni riferimenti e confronti con Python. La scelta di C e C++ non è casuale: sono infatti i linguaggi più diffusi e performanti per la programmazione dei microcontrollori, sia AVR che PIC, ARM o ESP32, anche se oggi è estremamente semplice copiare un frammento di codice e incollarlo in strumenti avanzati come ChatGPT per ottenere istantaneamente una conversione in Python o altri linguaggi. Questo permette non solo di adattare

facilmente il codice a piattaforme diverse, ma anche di riutilizzare direttamente il codice scritto per microcontrollori in ambienti desktop, facilitando test, debug e simulazioni preliminari.

Scriveremo il codice in stile "vanilla", cioè senza usare librerie particolari, ma costruendo tutto da zero. In pratica vedremo come implementare direttamente gli algoritmi di apprendimento automatico. L'obiettivo è creare un modello di rete neurale capace di imparare dagli esempi e ripeterne il comportamento. Tutto sarà scritto nel modo più essenziale possibile, basandoci sui fondamenti matematici dell'intelligenza artificiale, in particolare sulle regole che guidano le reti neurali e il processo di backpropagation.

Nello scrivere queste pagine sono consapevole di addentrarmi in un mondo vasto e pieno di insidie, ma credo fermamente che, di questi tempi, sia doveroso provarci.

Ricordo ancora il mio primo libro sulle reti neurali, acquistato intorno al 1997: un trattato chiaro, leggero e capace di stimolare e invogliare anche chi, con conoscenze matematiche un po' arrugginite e ormai condannato alla programmazione ripetitiva del "*metti la cera, togli la cera*", poteva in pochi giorni arrivare a realizzare il suo primo modello capace di apprendere.

Il testo, però, sebbene chiaro, mostrava una certa superficialità sotto un aspetto: non considerava le difficoltà di chi si avvicinava per la prima volta alle reti neurali. Non tanto per la trattazione delle formule in sé, quanto perché mancava un percorso che aiutasse il lettore a rinfrescare le nozioni essenziali prima di affrontare concetti più avanzati. Una guida che suggerisse come riprendere in mano gli strumenti matematici fondamentali avrebbe reso il testo ancora più efficace, permettendo anche a chi non aveva una preparazione solida di comprendere meglio i meccanismi alla base della backpropagation.

Non si pretendeva certo che al lettore venisse spiegato da zero cosa fosse un'espressione del tipo $y = f(x)$, né che il testo si addentrasse in dettagli eccessivi. Tuttavia, sarebbe stato utile includere almeno un decalogo o una serie di indicazioni preliminari per metterlo nella condizione di comprendere appieno i concetti fondamentali delle reti neurali. In fondo, gli elementi essenziali richiesti non sono molti: una comprensione di base delle derivate e delle derivate parziali, la conoscenza della regola della catena per il calcolo delle derivate di funzioni composte, una familiarità con i concetti fondamentali del calcolo differenziale, oltre alla capacità di analizzare la qualità e la coerenza dei dati. È fondamentale anche comprendere il loro ruolo nell'apprendimento, in particolare come queste informazioni vengano utilizzate per ottimizzare i parametri del modello attraverso il processo di backpropagation.

Ho deciso di scrivere questo libro proprio perché, 30 anni fa, i trattati sulle reti neurali artificiali insegnavano come realizzare una rete neurale, mentre oggi la maggior parte dei testi si concentra su come utilizzare l'intelligenza artificiale, e aumenta sempre più la confusione tra deep learning, machine learning, reti neurali e la suddetta intelligenza artificiale. Oggi, infatti, si tende spesso a privilegiare l'uso di librerie preconfezionate piuttosto che spiegare le basi matematiche. Va però riconosciuto che esistono trattati validi che cercano di conciliare entrambe le prospettive, anche se molti di questi risultano destinati a un pubblico di specialisti.

Quasi sempre i testi che trattano di reti neurali adottano Python come linguaggio di riferimento. Ma cosa fare se si desiderasse implementare un modello su un microcontrollore, come un semplice Atmega328 o Arduino? Spesso, nei forum e nei topics dedicati a Python, si nota che molti utenti utilizzano librerie come PyTorch senza avere una chiara comprensione dei meccanismi

sottostanti. Di fronte anche alla minima complicazione, non sanno più come procedere e si ritrovano ad affrontare spiegazioni che sembrano geroglifici dell'antico Egitto ad esempio formule come $W^{T}x = \sum_i \ x_i w_i$.

Sento spesso dire che non esistono libri aggiornati sulla backpropagation e sulle reti neurali, perché questi argomenti sarebbero ormai superati, e che oggi si parla solo di reti generative. Tuttavia, pochi sanno che, dietro le quinte, continuano a operare reti neurali antagoniste basate su processi di forward e retropropagazione dell'errore.

Le reti neurali addestrate tramite backpropagation sono ritenute da alcuni un'architettura "vecchia" rispetto alle più moderne tecniche di intelligenza artificiale generativa. Tuttavia, il loro ruolo rimane fondamentale, in particolare nelle GAN (Generative Adversarial Networks), dove si utilizza la backpropagation per addestrare due reti distinte ma interconnesse: il generatore e il discriminatore.

Le GAN operano in modo diverso rispetto alle reti neurali supervisionate tradizionali, anche se al loro interno è presente un meccanismo di "supervisione" per il discriminatore, che riceve l'etichetta "reale" o "generato". Nelle reti supervisionate classiche, l'obiettivo è minimizzare la differenza tra le previsioni del modello e i valori reali, utilizzando dati etichettati. Nelle GAN, invece, il generatore crea dati artificiali mentre il discriminatore cerca di distinguere tra dati reali e dati generati, aggiornando entrambi i modelli in un processo di addestramento incrociato. Entrambi vengono addestrati contemporaneamente: il generatore cerca di ingannare il discriminatore, mentre il discriminatore si affina nel riconoscere le falsificazioni. Questo processo iterativo, basato sulla backpropagation, permette di migliorare progressivamente la qualità dei dati generati.

Nonostante la backpropagation sia stata introdotta diversi decenni fa, resta un algoritmo di addestramento essenziale per l'ottimizzazione delle reti neurali, incluse quelle generative. La sua capacità di aggiornare i pesi della rete in risposta all'errore rimane il fondamento su cui si basano molte delle architetture moderne di deep learning.

In sintesi, mentre nuove tecnologie emergono nel campo dell'AI, la backpropagation continua ad essere una componente chiave nel training delle reti neurali, inclusi i modelli generativi, dimostrando come un principio consolidato possa ancora guidare le innovazioni più avanzate.

Sommario

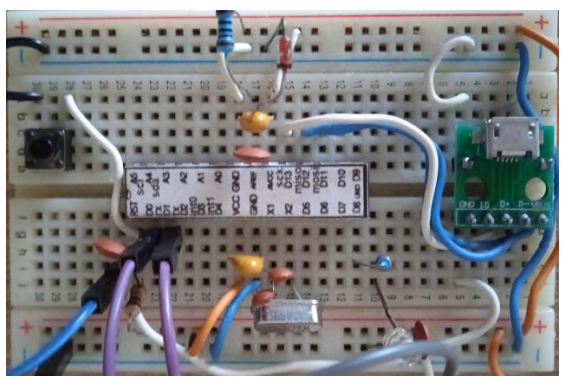
Perchè leggere questo libro

Cercheremo di capire come funziona un modello di rete neurale composto da neuroni artificiali e in che modo possa essere paragonato al cervello biologico. Approfondiremo inoltre il motivo per cui tale modello non sia in grado di eseguire autonomamente un'operazione come 2 + 2, come farebbe un computer, ma possa fornire la risposta corretta solo perché l'ha appresa milioni di volte, proprio come avviene per noi esseri umani.

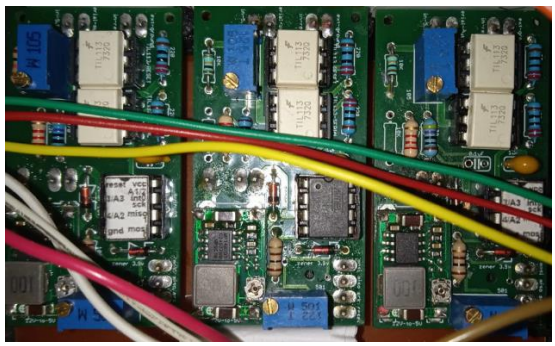
Se chiedeste a ChatGPT di calcolare 108.50 × 45, non riuscirebbe a farlo da solo e se provasse a rispondere potrebbe avvicinarsi molto, ma difficilmente darebbe il risultato corretto.

Ma allora, come fa?

Leggete il libro per scoprirlo...



Dopo avere letto questo libro potrete uploadare il vostro modello di AI su un microcontrollore o direttamente su un Arduino.



Oppure potrete realizzare circuiti più complessi come questi analizzatori di batterie LiFePO4.

Non farò pagine e pagine di citazioni storiche per raccontare da quando sono iniziati i primi studi sulle reti neurali, ma sappiate che esse cominciano intorno al 1940. Cito solamente la legge di Hebb, che fu la prima a formalizzare matematicamente il fenomeno sinaptico. Secondo Hebb, "le cellule che si attivano insieme, si collegano insieme". In termini matematici, questo principio può essere espresso con la seguente formula: $\Delta w_{ij} = \eta \cdot x_i \cdot y_j$ dove:

$\Delta w_{(ij)}$ rappresenta la variazione del peso sinaptico tra il neurone *i* (presinaptico) e il neurone *j* (postsinaptico),

η è il tasso di apprendimento, un parametro che determina la velocità con cui il peso viene aggiornato,

x_i è il livello di attivazione del neurone presinaptico,

y_j è il livello di attivazione del neurone postsinaptico.

Questa formula implica che, quando il neurone *i* si attiva contemporaneamente al neurone *j*, il prodotto $x_i * y_j$ sarà elevato e, di conseguenza, il peso $w_{(ij)}$ aumenterà. Questo rafforzamento sinaptico è alla base dell'apprendimento associativo: le connessioni tra neuroni che tendono a attivarsi insieme diventano più forti, facilitando la trasmissione futura dei segnali.

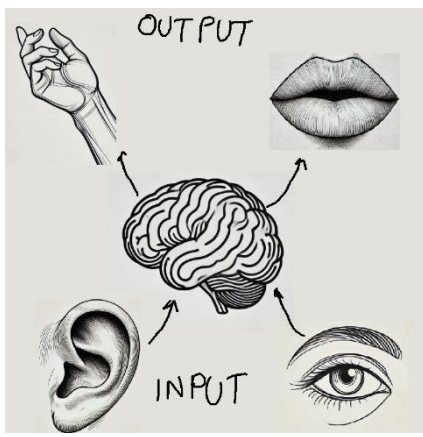
La legge di Hebb ha avuto un impatto fondamentale non solo nelle neuroscienze, ma anche nello sviluppo degli algoritmi di apprendimento nelle reti neurali artificiali, dove regole simili vengono impiegate per aggiornare i pesi in base alle correlazioni presenti nei dati di input.

Solo poi, negli anni ‘80, Rumelhart – insieme a Hinton e Williams – realizzò reti neurali dotate di livelli (o strati) nascosti, che permisero l’applicazione della retropropagazione dell’errore. Questo approccio innovativo ha segnato una svolta fondamentale nell’ambito dell’apprendimento automatico. Grazie all’impiego di strati intermedi, è stato possibile aggiornare iterativamente i pesi in base agli errori commessi durante il processo di apprendimento, dando vita a modelli più complessi ed efficaci.

Mi fermo qui, anche perché con questo libro faremo solo il primo passo verso la conoscenza e la realizzazione di una rete neurale artificiale.

Come funziona.

Le reti neurali artificiali si ispirano alle funzioni di base dei neuroni biologici, ma in modo semplificato, per apprendere modelli dai dati. In un cervello biologico, ci sono neuroni che ricevono segnali (neuroni sensoriali o di input), neuroni che inviano comandi ai muscoli o ad altre parti del corpo (neuroni motori o di output) e neuroni che svolgono entrambe le funzioni (interneuroni). Allo stesso modo, nelle reti neurali artificiali,



alcuni nodi (neuroni artificiali) si occupano principalmente di ricevere dati in ingresso, altri di fornire risultati in uscita e altri ancora possono svolgere entrambe le funzioni, elaborando e trasmettendo informazioni in varie parti della rete.Questo approccio, ispirato alla struttura biologica, consente alle reti neurali artificiali di "apprendere" a riconoscere schemi e relazioni in modo simile (anche se più semplificato) a come fa il cervello umano. Ogni neurone artificiale calcola una funzione sui dati in ingresso e, a seconda del risultato, attiva o meno i neuroni collegati in uscita. L'insieme di questi segnali, passando attraverso diversi strati, produce un output finale. È proprio la ripetizione di questi calcoli e l'aggiustamento iterativo dei parametri interni (i "pesi" delle connessioni) che permettono al sistema di "imparare" dai dati e migliorare le proprie prestazioni nel tempo.

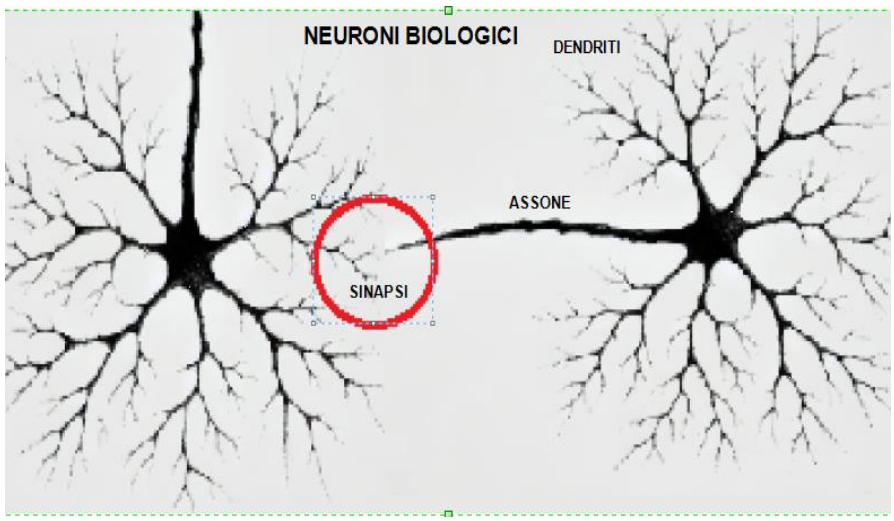
Per comprendere meglio come funziona una rete neurale artificiale, dobbiamo prima fare un parallelo con il funzionamento dei neuroni biologici, per poi vedere come le informazioni vengono trasmesse attraverso i vari componenti della rete. Ogni neurone biologico è composto da diverse parti: il corpo cellulare, l'assone, e le sinapsi.

Il corpo cellulare del neurone riceve segnali attraverso le dendriti, che sono collegamenti ai neuroni vicini; queste informazioni viaggiano poi attraverso l'assone, un lungo "filo" che trasporta il segnale verso altri neuroni o muscoli. Quando il segnale raggiunge la fine dell'assone, va verso la sinapsi, la piccola connessione che separa due neuroni.

Qui, il neurone presinaptico rilascia dei neurotrasmettitori che si legano ai recettori del neurone successivo (neurone postsinaptico), trasmettendo così il segnale.

Questo processo è la base della comunicazione nervosa.

Nelle reti neurali artificiali, il principio di funzionamento è ispirato a questo processo, ma in modo semplificato. Ogni "neurone" nella rete riceve delle informazioni in input (come i segnali nelle dendriti) e, attraverso un processo matematico, decide se inviare o meno un segnale (simile all'attivazione di un potenziale d'azione nel neurone biologico).



L'informazione viaggia attraverso le connessioni tra i neuroni, che possono essere concepite come le sinapsi. La forza di queste connessioni è determinata da un valore numerico chiamato peso. Ogni volta che il segnale viene trasmesso attraverso la rete, il peso delle connessioni cambia riesaminando, o "rafforzando", le connessioni più utili e riducendo quelle meno rilevanti.

L'aggiornamento dei pesi nelle reti neurali artificiali si ispira alla plasticità sinaptica osservata nei sistemi biologici, dove le connessioni tra neuroni si rafforzano o si indeboliscono in base all'attività. Tuttavia, occorre ricordare che la plasticità sinaptica è un meccanismo molto più complesso rispetto agli algoritmi di apprendimento automatico come la backpropagation. Le reti neurali semplificano questo processo in modelli matematici,

traendone ispirazione ma senza riprodurne fedelmente tutte le dinamiche biologiche.

Proprio come nelle reti neurali biologiche, nelle reti neurali artificiali il rafforzamento o il cambiamento dei pesi tra i neuroni (grazie ad algoritmi come la backpropagation e l'aggiornamento dei pesi attraverso la discesa del gradiente) permette al sistema di "apprendere" dai dati, ovvero registrare e ottimizzare le informazioni trasmesse. Inoltre, proprio come nel neurone biologico che conserva e trasmette segnali tramite le sinapsi, le reti neurali artificiali fanno lo stesso, ma attraverso la regolazione continua dei pesi di connessione, affinché l'informazione venga trasmessa efficacemente da un neurone all'altro. Questo processo è alla base del miglioramento della rete e della sua capacità di "preservare" l'informazione durante l'apprendimento. In sintesi, le reti neurali artificiali si ispirano al modo in cui i neuroni biologici comunicano, rafforzando le connessioni richieste per compiti specifici e apprendendo dalla rete stessa attraverso modifiche controllate dei pesi. Nell'immagine si vedono diversi neuroni con i loro prolungamenti, e questa struttura ci aiuta a capire come funziona la trasmissione dell'informazione nel cervello biologico e, per analogia, nelle reti neurali artificiali. Quando un neurone A e un neurone B si attivano contemporaneamente (o con una certa frequenza), la connessione tra loro si rafforza. Questo fenomeno è spesso ricondotto alla legge di Hebb – "neuroni che scaricano insieme si collegano fra loro" – e rappresenta una delle basi teoriche della plasticità sinaptica illustrata in precedenza.

Con il passare del tempo, queste modifiche sinaptiche costituiscono la base dell'apprendimento e della memoria nel cervello biologico.

La funzione di perdita

Da quanto appena esposto, sappiamo che la somma pesata dei valori di attivazione dei neuroni presinaptici (moltiplicati per i rispettivi pesi sinaptici) determina l'attivazione di un neurone postsinaptico.

È fondamentale osservare che, in una rete neurale, gli input (o features) sono variabili indipendenti fissate, mentre gli output (o target) sono variabili dipendenti che non possono essere modificati direttamente.

Di conseguenza, l'unico parametro che possiamo regolare per ottenere un output desiderato, a partire da un dato input, è rappresentato dai pesi sinaptici della rete. Per spiegare meglio questo concetto, consideriamo il seguente esempio. Supponiamo di voler definire la funzione logistica data da

$$y = \frac{1}{1 + e^{-z}}$$

Per trovare *Z*, procediamo come segue:

$$e^{-z} = \frac{1}{y} - 1 \Rightarrow e^{z} = \frac{y}{1 - y} \Rightarrow z = \ln\left(\frac{y}{1 - y}\right)$$

Poiché vale $z = x * w$ dove x è l'ingresso e w è il peso che collega x a y (come mostrato in Figura 4 ,però considerando un solo neurone di ingresso,un peso ed un neurone di output), e poiché y è nota (essendo l'uscita desiderata), l'unico parametro modificabile è il peso w. In questo caso, è facile determinare w una volta noto Z, utilizzando la relazione $w = \frac{z}{x}$.

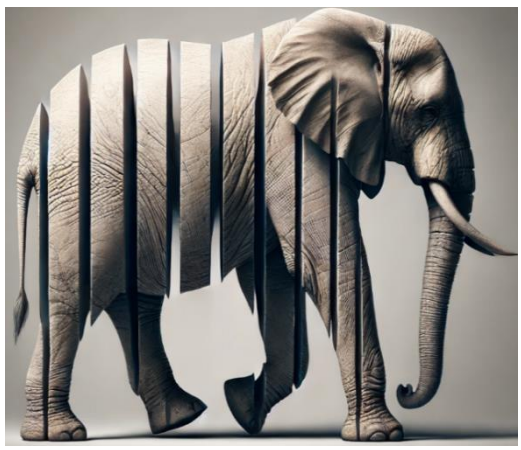
Tuttavia, come possiamo risolvere una situazione in cui avessimo cinque ingressi X_i con relativi pesi W_i collegati a un' uscita Y ?

Prima di proseguire, è necessario ripassare il concetto di derivate, il loro scopo e la loro importanza nello studio delle funzioni. Le derivate, infatti, misurano il tasso di variazione di una funzione rispetto alle variabili indipendenti e sono fondamentali per l'ottimizzazione, come nel caso del calcolo del gradiente nelle reti neurali. Si consiglia quindi di riprendere la lettura solo dopo aver chiarito questo concetto.

Quando si hanno più ingressi e più pesi, la situazione diventa significativamente più complessa. La molteplicità delle interazioni tra ingressi e pesi rende impraticabile una soluzione analitica diretta come nel caso di un solo ingresso e un solo peso. In questo scenario è necessario ricorrere a tecniche di ottimizzazione numerica come il calcolo del gradiente e la discesa del gradiente. Il calcolo del gradiente fornisce la direzione di massima crescita della funzione di perdita, mentre la discesa del gradiente utilizza questa informazione per aggiornare i pesi in modo iterativo, avvicinandosi al minimo della funzione e migliorandone l'accuratezza.

Con più ingressi e più pesi esistono infinite combinazioni di pesi che potrebbero produrre lo stesso valore di Z (la somma pesata degli ingressi). Di conseguenza, non esiste un'unica soluzione, ma un'intera famiglia di soluzioni possibili. In questo caso, è necessario utilizzare il calcolo del gradiente per aggiornare i pesi in modo iterativo, avvicinandosi progressivamente alla soluzione ottimale.

Procediamo per passi: «tagliamo l'elefante a fette»



Se avete ripassato i concetti alla base dello studio della derivata e della discesa del gradiente, possiamo affermare che, poichè non è possibile determinare Y (l'output del modello) utilizzando un solo peso, è necessario studiare il comportamento della funzione sigmoide per identificare il punto in cui essa genera l'errore minimo.

Per capire come la funzione sigmoide varia in funzione di z, dobbiamo calcolare la sua derivata:

$$removed\ from\ demo$$

Dimostrazione: partiamo dalla definizione della sigmoide:

removed from demo

Portiamo il denominatore al numeratore e riscriviamo la sigmoide in questa forma:

$$y = (1 + \mathrm{e}^{-\mathrm{z}})^{-1}$$

Applichiamo quindi la regola della catena per derivare questa funzione composta:

$$removed\ from\ demo$$

Semplificando, otteniamo:

$$removed\ from\ demo$$

Per giungere alla formula iniziale: $\frac{dy}{dz} = y \cdot (1 - y)$ possiamo procedere attraverso i seguenti passaggi e ulteriormente semplificare.

Dalla definizione della funzione sigmoide $y = \frac{1}{1+e^{-z}}$ possiamo ricavare l'espressione di e^{-z} in funzione di y :

$$e^{-z} = \frac{1 - y}{y}$$

Anche il denominatore $(1 + e^{-z})^2$ può essere riscritto in termini di y come segue:

$$removed\ from\ demo$$

1. Sostituendo queste espressioni nella derivata iniziale

$$removed\ from\ demo$$

otteniamo:

$$removed\ from\ demo$$

Abbiamo così dimostrato che: $\frac{dy}{dz} = y \cdot (1 - y)$

Da questa equazione possiamo osservare che la derivata è sempre positiva e tende a 0 (pendenza nulla) mano a mano che Y si avvicina a 0 o a 1. Pertanto, in prossimità di questi valori, non si verificheranno cambiamenti significativi.

A questo punto, possiamo analizzare l'andamento della funzione sigmoide. Sebbene questo esercizio sia propedeutico e importante

per i calcoli successivi, lo studio della sola funzione sigmoide non è sufficiente a risolvere il problema originale. Sarà infatti necessario considerare un modello più completo, composto da più ingressi X, ciascuno con il proprio peso W, e un'unica uscita Y, definita dal sistema.

Per risolvere questo problema, consideriamo la funzione

$$Z_k = \sum_{i=1}^{n} (W_{ik} \cdot X_i)$$

illustrata in Figura 4.

Assegniamo poi un indice k a Z anche se unico, questo ci sarà più familiare poi quando affronteremo i calcoli successivi.

Nel nostro caso, l'uscita Z_k dipende da più ingressi X_i, ciascuno con il proprio peso W_{ik}. Per analizzare come ciascun peso influenzi Z_k, è necessario calcolare separatamente il contributo di ogni singolo peso Wik, usando le derivate parziali.

Iterando l'indice i, determiniamo quindi l'effetto di ciascun peso sull'andamento di Z_k.

La derivata parziale corrispondente è: $removed\ from\ d$emo

dove X_i rappresenta l'ingresso associato al peso W_{ik}.

Come si può intuire, questo procedimento permette di isolare e comprendere l'effetto che ogni singolo peso ha sul comportamento globale della funzione.

Fortunatamente, la matematica offre una soluzione rigorosa per trattare il caso in cui una funzione dipenda da più parametri: si tratta della regola della catena per le derivate parziali, che consente di combinare gli effetti dei singoli parametri e calcolarne l'influenza complessiva.

Finora abbiamo introdotto le derivate parziali $\frac{\partial \hat{y}_i}{\partial z_k}$ e $\frac{\partial z_k}{\partial w_{ik}}$ ma manca ancora un elemento fondamentale.

Abbiamo infatti espresso come varia l'uscita Y rispetto a Z e in che modo Z dipende dai singoli pesi W_{ik}, ma senza confrontare il valore calcolato di Y con il valore osservato (o desiderato), non possiamo sapere come aggiornare ciascun peso.

Per questo motivo, introduciamo il concetto di errore, cioè la differenza tra il valore desiderato (D, ovvero Y target o osservato) e quello ottenuto (Y predetto). Solo così possiamo determinare come modificare i pesi W_{ik} per avvicinare l'uscita effettiva a quella desiderata.

In sintesi, occorre definire una misura di errore e tramite la regola della catena comprendere come ciascun peso debba essere aggiornato per minimizzare questo errore.

A questo punto entra in gioco un nuovo attore in questo scenario: la Y target, ossia la Y desiderata quando Z è composta da n ingressi X moltiplicati per i rispettivi pesi.

Oltre a questo attore, ne compare un altro: l'errore E, definito come

$E = D - \hat{Y}$, dove D è l'uscita desiderata e $\hat{Y}$ è l'uscita predetta dal modello.

In matematica e statistica, questa relazione è alla base della funzione di perdita.

La variabile $\hat{Y}$ rappresenta l'uscita prevista del modello ed è indicata con un "cappello" (circumflex accent) sopra la Y.

$$L(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Questa formula determina l'errore quadratico medio (MSE) di un sistema.

In alternativa, è possibile utilizzare anche il valore assoluto anziché il quadrato. La ragione sarebbe la medesima: se non si usasse il quadrato o il valore assoluto, si rischierebbe che, avendo un errore di 5 in un caso e -5 in un altro, i due errori sommati si annullerebbero, il che falserebbe la misura dell'errore complessivo.

Questa formula permette di ottenere sempre una somma positiva, anche in presenza di errori parziali. In sostanza, gli errori devono annullarsi all'interno delle parentesi, così da non essere considerati nella somma totale.

La scelta tra il quadrato e il valore assoluto dipende da quanto peso si vuole attribuire agli errori più grandi: l'uso del quadrato penalizza maggiormente gli errori grandi rispetto agli errori piccoli.

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}.$$

Inoltre, calcolando la radice, otteniamo il valore di RMSE, ovvero l'errore quadratico medio della funzione di perdita. Un valore di RMSE basso indica che gli errori commessi dal modello sono contenuti, ossia la distanza media tra i valori previsti e quelli reali è ridotta, mentre un RMSE elevato segnala una maggiore variabilità degli errori e, di conseguenza, una minore accuratezza nelle previsioni del modello.

Ciò che dobbiamo ancora comprendere è quale sia l'errore generato dalla funzione e come relazionarlo con i pesi, per poterli poi modificare al fine di ridurre l'errore.

Vediamo pertanto cosa è possibile fare con la funzione di errore modificata utilizzando $\frac{1}{2}$ anziché $\frac{1}{n}$ in modo che si adatti meglio all'esigenza di derivarla.

$$E = \frac{1}{2}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Anche in questo caso, abbiamo una derivata parziale in quanto ci sono più variabili che determinano l'andamento della funzione di errore.

Nell'esempio che stiamo analizzando, con una singola funzione sigmoide, otteniamo un solo valore di output Y (non rilevante per il nostro ragionamento).

La derivata parziale rispetto a $\hat{y}_i$ è:

$$removed\ from\ demo$$

Passaggi:

Applichiamo la regola della derivazione composta.

La derivata di $(y_i - \hat{y}_i)^2$ rispetto a $\hat{y}_i$ è:

$$removed\ from\ demo$$

Il fattore (-1) deriva dalla derivata $-\hat{y}_i$.

Moltiplichiamo per $\frac{1}{2}$:

$$\frac{\partial \mathrm{E_i}}{\partial \hat{\mathrm{y}}_\mathrm{i}} = \frac{1}{2} \cdot 2(\mathrm{y_i} - \hat{\mathrm{y}}_\mathrm{i}) \cdot (-1) = -(\mathrm{y_i} - \hat{\mathrm{y}}_\mathrm{i})$$

e otteniamo :

$$\frac{\partial \mathrm{E}}{\partial \hat{\mathrm{y}}_{\mathrm{i}}} = -(\mathrm{y}_{\mathrm{i}} - \hat{\mathrm{y}}_{\mathrm{i}})$$

Ora possiamo concatenare i seguenti gradienti per determinare l'andamento della funzione $\mathrm{y} = \frac{1}{1+\mathrm{e}^{-\mathrm{z}}}$

$$removed\ from\ demo$$

Concatenando i gradienti derivati tramite la regola della catena, otteniamo un risultato di fondamentale importanza:

$$\frac{\partial \mathrm{E}}{\partial \mathrm{W}_{\mathrm{ij}}} = \frac{\partial \mathrm{E}}{\partial \hat{\mathrm{y}}_{\mathrm{i}}}\ removed\ from\ demo$$

Questo è esattamente ciò che volevamo ottenere: comprendere come l'errore varia in funzione dei pesi, in modo da poterli modificare di conseguenza.

Posso quindi affermare che l'insieme di questi gradienti indica la pendenza e, considerando l'errore, la direzione in cui si sposta la funzione.

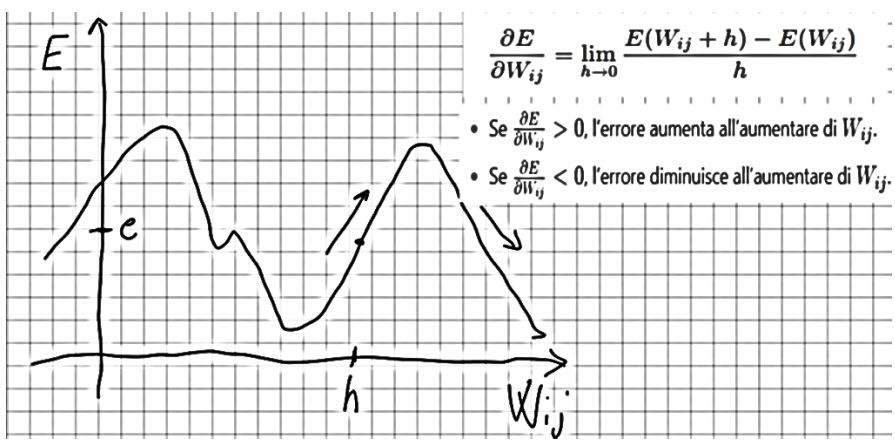


Figura 5

Nella Figura 5 è possibile osservare l'andamento dell'errore in funzione dei pesi W_{ij}.

Chiaramente, per la derivata *h* tende a 0, ma il concetto è analogo a quello del coefficiente angolare: la derivata $\frac{\partial \mathrm{E}}{\partial \mathrm{W_{ij}}}$ rappresenta infatti il coefficiente angolare della retta tangente alla curva $E(W_{ij})$ nel punto W_{ij}.

Se $\frac{\partial \mathrm{E}}{\partial \mathrm{W_{ij}}} > 0$ l'errore aumenta all'aumentare di W_{ij}

Se $\frac{\partial \mathrm{E}}{\partial \mathrm{W_{ij}}} < 0$ l'errore diminuisce all'aumentare di W_{ij}

La derivata determina la direzione in cui la funzione si sposta. Per minimizzare l'errore, è fondamentale introdurre il coefficiente di apprendimento $-\epsilon$, che permette di aggiornare i pesi W_{ij} nella direzione opposta a quella indicata dal gradiente. Se il gradiente è positivo (errore crescente), si riduce W_{ij}; se è negativo (errore decrescente), si aumenta W_{ij}. Questo approccio guida i pesi verso il minimo della funzione di errore, come illustrato in figura 5.

Arriviamo quindi a parlare della modifica del peso tramite il delta peso:

$$\Delta \mathrm{w_{jk}} = -\epsilon \cdot \frac{\partial \mathrm{E}}{\partial \mathrm{w_{jk}}}$$

Tradotto, significa che una piccola quantità "delta" da aggiungere o sottrarre al peso W_{ij} viene calcolata seguendo la direzione opposta al gradiente $\frac{\partial \mathrm{E}}{\partial \mathrm{w_{ij}}}$, moltiplicata per una variabile ϵ (epsilon) che determina l'ampiezza del "passo". Epsilon ϵ, chiamata tasso di apprendimento, regola la velocità con cui i pesi vengono aggiornati per minimizzare l'errore:

Se ϵ è troppo grande, si rischia di saltare il minimo della funzione (superando l'avvallamento ottimale).

Se ϵ è troppo piccola, l'apprendimento diventa estremamente lento.

Per ottimizzare il processo, è cruciale analizzare l'andamento dei gradienti (come si avvicinano a zero, indicando convergenza) e il comportamento dell'errore, sia nella media generale che in singoli casi di addestramento. Questo aspetto verrà approfondito nel seguito della trattazione. Ricordiamo che, al momento, non stiamo trattando reti neurali con backpropagation, ma ci concentriamo su concetti fondamentali isolati.

In sintesi, possiamo affermare che:

$$\Delta w_{ij} = -\epsilon \cdot \frac{\partial E}{\partial w_{ij}} = -\epsilon \cdot \underbrace{\frac{\partial E}{\partial \hat{y}_i}}_{\text{Errore}} \cdot \underbrace{\frac{d\hat{y}_i}{dz}}_{\text{Derivata attivazione}} \cdot \underbrace{\frac{\partial z_i}{\partial W_{ij}}}_{\text{Input } X_j}$$

Dove:

- $\frac{\partial E}{\partial \hat{y}_i} = -(y_i - \hat{y}_i)$ (derivata dell'errore quadratico).
- $\frac{d\hat{y}_i}{dz} = \hat{y}_i(1 - \hat{y}_i)$ (derivata della sigmoide).
- $\frac{\partial z_i}{\partial W_{ij}} = X_j$ (input del neurone).

$$\Delta \mathrm{w_{ij}} = -\epsilon \cdot \frac{\partial \mathrm{E}}{\partial \mathrm{w_{ij}}} = -\epsilon \cdot \frac{\partial \mathrm{E}}{\partial \hat{\mathrm{y}}_\mathrm{i}} \frac{\partial \hat{\mathrm{y}}_\mathrm{i}}{\partial \mathrm{Z_i}} \frac{\partial \mathrm{Z_i}}{\partial \mathrm{W_{ij}}} = -\epsilon \cdot -(\mathrm{y_i} - \hat{\mathrm{y}}_\mathrm{i}) \cdot \hat{\mathrm{y}}_\mathrm{i}(1 - \hat{\mathrm{y}}_\mathrm{i}) \cdot \mathrm{X_j}$$

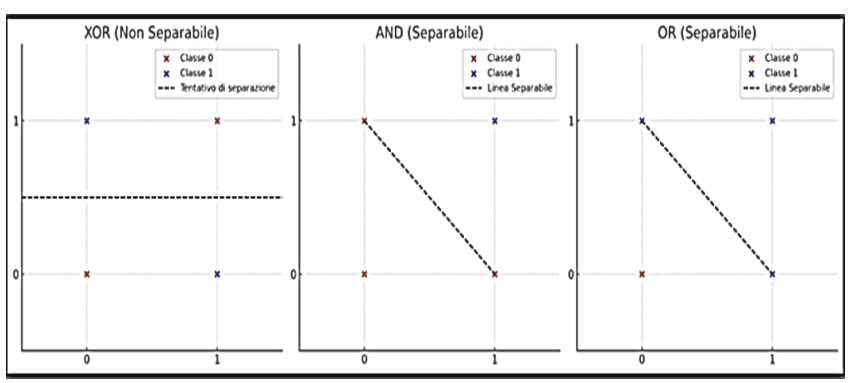
Semplificando ulteriormente:

$$\Delta w_{ij} = \epsilon(y_i - \hat{y}_i) \cdot \hat{y}_i(1 - \hat{y}_i) \cdot x_j$$

Considerando che il modello possiede una singola uscita Y, l'aggiornamento dei pesi mediante la formula viene effettuato iterativamente per ciascun ingresso *j*. In altre parole, mentre l'indice *i* è fisso (poiché abbiamo un'unica unità di output), il ciclo interno si applica a tutti gli ingressi disponibili, aggiornando in maniera indipendente ogni peso W_{ij} in base al contributo dell'ingresso X_j. Il termine $(y_i - \hat{y}_i)$ rappresenta la differenza tra l'uscita desiderata Y e quella predetta $\hat{y}_i$, fornendo un'indicazione diretta dell'errore: un valore che può essere interpretato come la soglia minima accettabile per l'errore. Con questo procedimento, il sistema regola i pesi in modo da minimizzare progressivamente tale differenza, conducendo così l'intera rete verso la soluzione ottimale. Questo esempio, basato su un'unica funzione sigmoide, sottolinea come, in contesti con un singolo output, la fase di aggiornamento si concentri esclusivamente sul perfezionamento di una soluzione, iterando il processo su ogni ingresso.

Abbiamo finora illustrato come, grazie alla regola della catena e all'aggiornamento dei pesi, sia possibile addestrare un singolo neurone (con funzione di attivazione sigmoide) per risolvere problemi di classificazione.

Purtroppo questo modello funziona solo se i dati sono linearmente separabili. In altre parole, la nostra singola sigmoide è in grado di



apprendere funzioni come AND o OR, ma non riesce a gestire correttamente problemi non lineari come XOR. Nell'immagine successiva vedremo chiaramente che, per XOR, non esiste una retta (o iperpiano) che separi correttamente i dati. Questa constatazione ci porta a capire che, per affrontare funzioni non linearmente separabili, occorre introdurre almeno un livello nascosto: è qui che nascono le reti neurali multistrato, capaci di rappresentare e apprendere relazioni molto più complesse.

Figura 6

Osservando l'immagine che confronta le funzioni XOR, AND e OR, emerge chiaramente come un'unica retta (o iperpiano in spazi di dimensioni superiori) sia sufficiente a separare correttamente i punti di AND e OR, ma non riesca a farlo per l'XOR. Quest'ultimo, infatti, presenta una disposizione dei punti (classe 0 e classe 1) che non può essere separata da un singolo confine lineare. Di conseguenza, un singolo neurone con attivazione sigmoide non riesce a rappresentare questa complessità: la sua funzione decisionale è limitata a forme lineari e non può "curvare" lo spazio dei dati in modo da catturare le interazioni non lineari presenti nell'XOR. Per superare questo ostacolo, è necessario ricorrere a reti neurali più profonde o a modelli più sofisticati, in grado di apprendere decisioni non lineari e quindi di gestire funzioni più complesse.

Backpropagation Vanilla code

Le idee hanno bisogno di sedimentarsi

.

Se siete arrivati fin qui e non è tutto chiaro, è assolutamente normale, dopotutto, dalle prime scoperte negli anni ‘40 e ‘50 dei modelli ispirati alle reti neurali, passando per gli anni ‘60 e ‘70, definiti il "periodo buio" dell’intelligenza artificiale, si dovette aspettare fino agli anni ‘80 per scoprire che la backpropagation era la soluzione ideale per risolvere problemi linearmente non separabili, una ragione c’è, quindi non abbattetevi, e dato che siamo in tema di reti neurali, dormiteci sopra!

Sì, avete capito bene: le idee hanno bisogno di tempo per sedimentarsi. Riflettete, rimuginateci un po’ e poi dormiteci su; l’ippocampo farà il resto. Durante il sonno, infatti, l’ippocampo rielabora le informazioni acquisite durante il giorno e aiuta a consolidarle nella memoria a lungo termine.

Anche quando non ero a conoscenza di questo processo, avevo già intuito qualcosa: se prima di addormentarmi ripensavo attentamente agli esercizi di trial in bicicletta che non ero riuscito a eseguire durante il giorno, in particolare quelli che richiedevano equilibrio e movimenti precisi, il giorno dopo questi mi riuscivano quasi magicamente. Quasi sempre mi riuscivano per pochi minuti prima di perderli nuovamente, ma evidentemente qualcosa era cambiato a livello psicofisico, e bastava registrare quel cambiamento avvenuto per compiere un avanzamento.

Ora è il momento di scrivere un po’ di codice, per capire come analizzare i dati acquisiti e monitorare il processo di apprendimento. Sarà utile osservare l’effetto del numero di neuroni nascosti, il valore del coefficiente di apprendimento (epsilon) e, soprattutto, analizzare i gradienti e gli errori della rete

su ogni singolo esempio. Questo ci permetterà di individuare eventuali picchi anomali causati da dati errati, che potrebbero far aumentare eccessivamente l'errore complessivo per epoca.

Sarà necessario approfondire un po' di matematica: la varianza, le tecniche di normalizzazione dei dati, e altri strumenti utili per interpretare correttamente ciò che accade durante l'apprendimento.

Avevo inizialmente pensato di proporre un esempio in cui una rete neurale imparasse a giocare a tris, ma credo sia più interessante e utile concentrarsi su come una rete possa apprendere da dati reali e poi fare previsioni su funzioni non lineari.

Una delle qualità più affascinanti delle reti neurali, come accennato in precedenza, è la loro capacità di apprendere i dati creando una sorta di ricordo fotografico, un'immagine mentale delle informazioni che permette di riconoscere e generalizzare i pattern appresi.

Faccio questa premessa perché ritengo che, prima di buttarsi a scrivere il codice ,che non è altro che una serie di chiamate che racchiudono le regole matematiche fin qui descritte, e non ha molto del classico codice a cui siamo normalmente abituati, sia fondamentale chiarire un concetto importante: non tutti i problemi che una rete neurale deve risolvere sono uguali. Ognuno va affrontato con intelligenza, preparazione matematica, intuito ed esperienza.

Per far sì che la rete apprenda, dobbiamo ragionare come se stessimo insegnando qualcosa a un essere pensante. Il modo in cui presentiamo i dati, il significato che hanno, e la loro coerenza interna sono fondamentali quanto l'architettura stessa della rete.

Faccio un esempio un po' borderline, ma su cui vale la pena riflettere.

Immaginiamo una macchina che debba imparare a comandare dei dispositivi in base all'orario. Supponiamo che un certo dispositivo debba accendersi e spegnersi con un comportamento ripetitivo tra le 23:59 e le 00:10. Se passassimo alla rete neurale gli input temporali in formato numerico crescente fino alle 24:00, e poi ricominciassimo da 00:00, secondo voi quanto sarebbe facile per la rete apprendere quella continuità?

Pensate a un sistema di controllo per una ventola: i pesi della rete dovrebbero adattarsi a un comportamento che, da un punto di vista numerico, presenta un "salto" improvviso da 23.59 a 00:00, anche se in realtà il comportamento in uscita rimane stabile.

Se dessimo questo compito a un bambino, probabilmente comprenderebbe meglio la regolarità se usassimo un orologio analogico a muro invece che una sveglia digitale, proprio perché l'apprendimento visivo sarebbe più graduale e ciclico.

Infatti, è buona pratica , quando si gestiscono ingressi ciclici come il tempo, gli angoli o le stagioni , rappresentare tali dati utilizzando le funzioni seno e coseno, così da mantenere la continuità ciclica. Questo aiuta il sistema ad evitare salti numerici che non corrispondono a vere discontinuità nel comportamento osservato, e rende molto più facile l'apprendimento da parte della rete.

Consideriamo un sistema di propulsione elettrica nel quale è stata implementata una sezione circuitale denominata *Battery Analyzer*. Durante il funzionamento, mentre il veicolo percorre tracciati caratterizzati da differenti condizioni di guida, vengono acquisiti e salvati su una scheda SD i seguenti dati: l'energia erogata in wattora (Wh), la corrente in ampere (A) e sei misurazioni di tensione, ciascuna corrispondente a una delle batterie collegate in serie.

Successivamente, i dati raccolti vengono trasmessi a una rete neurale che riceve in input due variabili: la energia, espressa in

wattora (Wh), e corrente, misurata in ampere (A). La rete produce in output sei valori, ciascuno dei quali corrisponde alla tensione attesa di una specifica batteria collegata in serie.

In tal modo, il sistema impara ad associare specifici livelli di consumo energetico e di intensità di corrente ai valori di tensione caratteristici di ciascuna batteria. Ad esempio, la rete neurale potrebbe apprendere che, in condizioni di batteria ancora carica (ad esempio poco dopo l'avvio del sistema), un consumo di 100 Wh combinato con una corrente di 30 A corrisponde a una tensione di circa 12,5 V sulla prima batteria; mentre, mantenendo invariata la corrente di 30 A, un consumo di 890 Wh porta a una riduzione della tensione a circa 11,0 V.

Chiaramente, a causa delle differenze intrinseche di produzione e dei vari gradi di usura, le tensioni delle sei batterie non saranno mai perfettamente omogenee. Di conseguenza, la rete neurale interpreterà queste informazioni come una sequenza con lievi variazioni, risultando in un andamento non del tutto lineare.

La rete neurale, invece di limitarsi a riprodurre esattamente un singolo valore, apprende i pattern e le relazioni che collegano i dati in ingresso (espressi in wattora e ampere) ai valori attesi in uscita (ovvero, le tensioni delle batterie). In questo modo, il modello sviluppa una comprensione generale delle dinamiche del sistema, consentendogli di generalizzare la conoscenza anche in presenza di piccole variazioni o rumore nei dati.

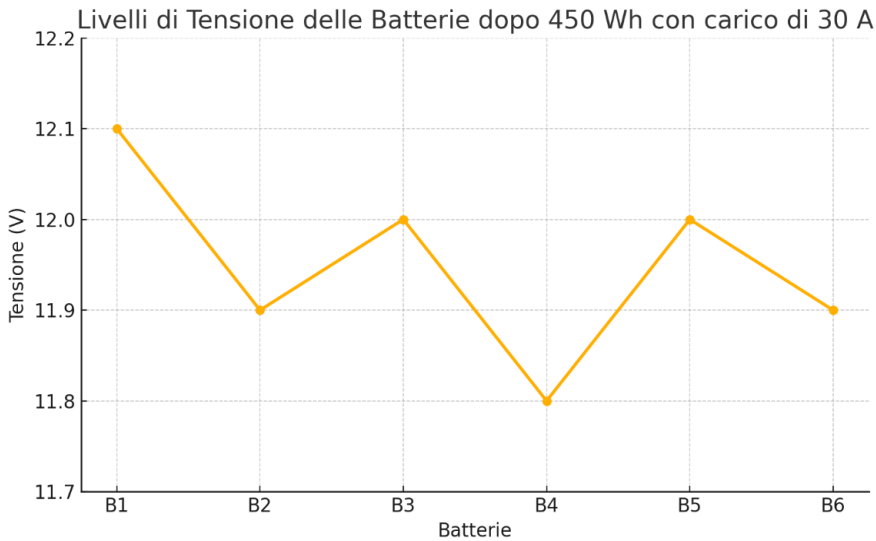


Figura 7

In pratica, questo significa che la rete sarà in grado di:

Generalizzare, ovvero fornire un valore molto vicino a quello reale anche per situazioni nuove e non presenti esattamente nei dati di addestramento.

Identificare anomalie significative rispetto alle condizioni normali apprese (es. tensioni troppo basse o troppo alte rispetto a quelle attese per un certo consumo energetico o di corrente).

Interpretare il comportamento globale delle batterie, piuttosto che memorizzare semplicemente valori puntuali.

Dopo aver completato la fase di addestramento, durante la quale la rete neurale ha imparato a riconoscere i livelli di tensione delle batterie nelle diverse condizioni di utilizzo (combinazioni di energia consumata e corrente istantanea), si passa alla fase operativa.
In questa fase, la rete verrà utilizzata in tempo reale mentre il veicolo è in funzione, confrontando costantemente i valori predetti (ossia quelli considerati normali, sulla base dell'esperienza

acquisita durante l’addestramento) con i valori realmente misurati sul campo.

Se la rete riscontrasse una differenza significativa tra i valori previsti e quelli effettivamente misurati, ad esempio una batteria che presenta tensioni anomale rispetto a quanto atteso in base al carico e allo stato energetico, sarà possibile rilevare tempestivamente un potenziale problema, identificando malfunzionamenti, anomalie o eventuali condizioni di degrado.

Come posso essere sicuro che le informazioni apprese dalla rete neurale siano effettivamente correlate ai dati forniti dal sistema Batteries Analyzer? Sebbene, in linea teorica, possano emergere numerosi scenari, spesso richiedenti soluzioni di codice estremamente specifiche, è fondamentale adottare metodologie di validazione che garantiscano che il modello generalizzi correttamente, evitando così di essere eccessivamente adattato a situazioni particolari.

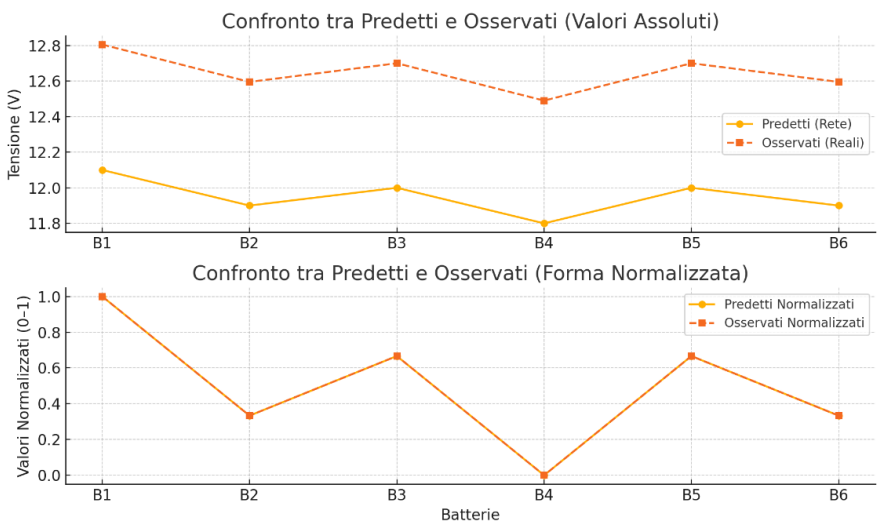


Figura : 8

Bisogna cercare una soluzione che non richieda logica codificata e che “riconosca” autonomamente la somiglianza tra il grafico prodotto dalla previsione della rete e quello mostrato dalla sezione circuitale.

Come dicevo, la rete apprende come se stesse osservando un’immagine, e quindi acquisisce il dato mantenendo una certa proporzione rispetto all’ingresso fornito. Quello che intendo far capire, in un caso di apprendimento come questo, è che se una rete neurale funziona bene, lo dobbiamo valutare principalmente dalla forma e non dalla sostanza.Certo, i dati devono aggirarsi attorno ai livelli osservati, ma non è di fondamentale importanza che coincidano perfettamente con quelli predetti, né tantomeno è necessario applicare fattori moltiplicativi sui dati per portarli esattamente al livello osservato.

La cosa più importante è che i due grafici abbiano una correlazione e un andamento proporzionale. Per valutare la correlazione tra i valori osservati e quelli predetti, la correlazione di Pearson è comunemente utilizzata per misurare il grado di relazione lineare tra due vettori. Questa tecnica funziona bene quando si presume che la relazione sia effettivamente lineare, ossia quando, al variare di un valore, l’altro cambia in maniera proporzionale. Tuttavia, nel contesto del machine learning e in situazioni reali, come ad esempio il monitoraggio dei livelli di batteria di un dispositivo, i dati possono non rispettare sempre questa linearità.

Se i segnali o i vettori presentano comportamenti non lineari, per esempio, in caso di degradazione della batteria che modifica il pattern appreso, la correlazione di Pearson potrebbe non essere la metrica più adatta. In tali casi, è utile considerare anche altre tecniche:

Correlazione di Spearman: Questa misura valuta la relazione tra le variabili basandosi sui ranghi anziché sui valori grezzi, risultando più robusta in presenza di relazioni non lineari o outlier.

Coefficiente di Kendall: Un'altra alternativa che si concentra sull'ordine dei dati, offrendo una valutazione della concordanza tra le variabili.

Metriche di errore: Per avere una visione più completa, si usano misure come l'errore quadratico medio (MSE), l'errore assoluto medio (MAE) e il coefficiente di determinazione (R^2), che permettono di quantificare direttamente la differenza tra il valore osservato e quello predetto.

In sintesi, mentre la correlazione di Pearson è una tecnica valida e diffusa per analizzare relazioni lineari, in contesti complessi come il monitoraggio dei livelli di batteria, dove i dati possono variare in modo non lineare, può essere utile integrare o sostituire questa metrica con altre tecniche più adatte al contesto. In seguito, illustrerò il metodo che ho adottato, basato su una combinazione dell'errore quadratico medio, una normalizzazione nell'intervallo [0,1], l'analisi della varianza e ulteriori tecniche, per ottenere un confronto più accurato dei due vettori.

Il codice riportato nel libro è soltanto dimostrativo e non avrebbe senso copiarlo direttamente, perché molto probabilmente non avrete a disposizione i dispositivi elettronici necessari per testarlo. Per questo motivo, alla fine del libro sarà fornito un link su GitHub da cui potrete scaricare il codice completo insieme ai file CSV contenenti il dataset dei dati rilevati; in questo modo potrete procedere con la fase di addestramento e simulare anche la fase di predicting, utilizzando i dati forniti nel file proprio come se fossero acquisiti direttamente dal dispositivo reale.

Come già anticipato, il codice di una rete neurale artificiale, con livelli di ingresso, nascosto e uscita, è abbastanza standard e varia

solitamente in base alla funzione di trasferimento adottata, a seconda della specifica problematica.

Pertanto, è estremamente semplice riutilizzare la stessa struttura per risolvere diversi problemi. Non ho utilizzato la programmazione orientata agli oggetti, la dependency injection, l'ereditarietà o altri pattern particolari, poiché non intendo complicare eccessivamente l'apprendimento. Di seguito, il codice per la realizzazione di una rete neurale artificiale con funzione di attivazione sigmoide:

Le dichiarazioni:

```
float watts_hour_training[training_samples]{};

//neuroni di output training
float
battery_out_training[training_samples][numberOf_Y]{};
```

Le funzioni.

Inizializza le variabili per il delta da cui generare i pesi random.

```
double xavier_init(double n_x, double n_y) {
	return sqrt(6.0) / sqrt(n_x + n_y);
}
```

Genera i pesi random iniziali.

```
double random_value = distribution(generator);
return random_value;
}
```

Inizializza la rete neurale

```
void init() {
```

```
	}
	for (int i = 0; i < numberOf_Y; i++) {
		cout << "output_bias[" << i << "]" << "=" <<
output_bias[i] << "-BIAS" << "\n";
	}
	//cout << "output elements initialization:\n\n";
	//-----------------------------------  console
output values
	for (int i = 0; i < numberOf_Y; i++) {
		//y[i] = 0.00f;
		cout << "y[" << i << "]=" << y[i] << "\n";
	}
	//-----------------------------------  console W1
values
	cout << "W1 elements initialization:\n\n";
	for (int i = 0; i < numberOf_X; i++) {
		for (int k = 0; k < numberOf_H; k++) {
			W1[i][k] =
get_random_number_from_xavier();
			cout << "W1[" << i << "]" << "[" << k
<< "]" << "=" << W1[i][k] << "\n";
		}
	}
	//-----------------------------------  console W2
values
	for (int k = 0; k < numberOf_H; k++) {
		for (int j = 0; j < numberOf_Y; j++) {
	W2[k][j] = get_random_number_from_xavier();
		cout << "W2[" << k << "]" << "[" << j << "]"
<< "=" << W2[k][j] << "\n";
		}
	}
```

```
}
```

La funzione di trasferimento Signoidale

```
	float sigmoid_activation(float Z) {
			return 1.00f / (1.00f + exp((Z * -1)));
}
```

La funzione che esegue e calcola gli output.

```
	void forward() {
	for (int k = 0; k < (numberOf_H); k++) {
		float Zk = 0.00f;
		for (int i = 0; i < numberOf_X; i++) {
			Zk += (W1[i][k] * x[i]);		}
		//insert X bias
		Zk += hidden_bias[k];
		h[k] = sigmoid_activation(Zk);
	}
	for (int j = 0; j < numberOf_Y; j++) {
		float Zj = 0.00f;
		for (int k = 0; k < numberOf_H; k++) {
			Zj += (W2[k][j] * h[k]);
		}
		//insert H bias
		Zj += output_bias[j];
		y[j] = sigmoid_activation(Zj);}
}
```

La funzione che bilancia i pesi e retropropaga l'errore.

```
            // Aggiornamento del bias del livello di
uscita
            output_bias[j] += _epsilon * delta;
      }
      for (int k = 0; k < numberOf_H; k++) {
            delta = err_H[k] * h[k] * (1.00f - h[k]);
            for (int i = 0; i < numberOf_X; i++) {
                  W1[i][k] += (_epsilon * delta * x[i]);
            }
            hidden_bias[k] += _epsilon * delta;
      }
}
```

La funzione che esegue l'apprendimento.

```
void apprendi() {
            _err_epoca = 0.00f;
            average_err_rete = 0.00f;
            varianza_err_rete = 0.00f;
            for (unsigned long p = 0; p <
training_samples; p++) {
                  x[0] = log(amps_training[p] + 1.0f) /
10.0f;
                  x[1] = log(watts_hour_training[p] +
1.0f) / 10.0f;
                  for (int i = 0; i < numberOf_Y; i++) {
                        d[i]                          =
battery_out_training[p][i] / 10.00f;
                  }
                  forward();
```

```
                    back_propagate();
                    if (_err_rete > _err_epoca) {
                          _err_epoca = _err_rete;
                          max_error_file_index_line     =
((p) * 8) + 1;
                    }
                    listOfErr_rete[p] = _err_rete;
                    average_err_rete += _err_rete;
      }
      _epoca_index = _epoca_index + 1;
            average_err_rete /= training_samples;
            for   (unsigned   long   p   =   0;   p   <
training_samples; p++) {
            varianza_err_rete+=pow(listOfErr_rete[p]-
average_err_rete , 2);
            }
            varianza_err_rete /= training_samples;
            deviazione_std = sqrt(varianza_err_rete);
            cout_counter++;
            is_on_wtrite_file = false;
            if (cout_counter == 10000) {
            std::cout << "\nepoca:" << _epoca_index <<
                  "\nerr_epoca=" << _err_epoca <<
                  "\nmin.         err_epoca="         <<
_err_epoca_min_value <<
                  "\nlast  time  write  on  file  =  "  <<
timeStr <<
                  "\nvarianza di errore di rete = " <<
varianza_err_rete <<
```

```
                    "\nmedia di errore di rete = " <<
average_err_rete <<
                    "\ndeviazione standard errore di rete
= " << deviazione_std <<
                    "\nmax err_epoca is on sample line = "
<< max_error_file_index_line <<
                    "\npercentage dev.standard err_rete /
media err_rete = "
                                                <<
(deviazione_std / average_err_rete) * 100 << "%" <<
                           "\nepsilon="  <<  _epsilon  <<
"\n";
                    cout_counter = 0;
                    if (_err_epoca_min_value > _err_epoca)
{
                           is_on_wtrite_file = true;
                           _err_epoca_min_value          =
_err_epoca;
                    }
             }
             if (is_on_wtrite_file) {
                    setTime();
                                 write_weights_on_file();
                      }
       } while (_err_epoca > _err_amm);
       auto end = std::chrono::system_clock::now();
       std::chrono::duration<double> elapsed_seconds = end
- start;
       double sample_time = elapsed_seconds.count();
```

```
	std::cout << "learning time : " << (int)(sample_time
/ 60) << " minutes.\n";
#ifdef __linux__
	getchar();
#elif _WIN32
	_getch();
#endif
}
```

Legge i pesi dal file HEX.

```
void read_weights_from_file(){
	std::ifstream in(_relative_files_path + "/" +
"model.hex", std::ios_base::binary);
	if (in.good()){
		for (int i = 0; i < numberOf_X; i++){
			for (int k = 0; k < numberOf_H; k++){
				in.read((char*)&W1[i][k],
sizeof(float));
			}
		}
		for (int j = 0; j < numberOf_Y; j++){
			for (int k = 0; k < numberOf_H; k++){
				in.read((char*)&W2[k][j],
sizeof(float));
			}
		}
		for (int k = 0; k < numberOf_H; k++){
			in.read((char*)&hidden_bias[k],
sizeof(float));
```

```
		}
		for (int j = 0; j < numberOf_Y; j++){
			in.read((char*)&output_bias[j],
sizeof(float));
		}









		in.read((char*)&_err_epoca_min_value,
sizeof(float));
		in.read((char*)&_epsilon, sizeof(float));
	}
}
```

Scrive i pesi sul file HEX.

```
void write_weights_on_file() {
	std::ofstream fw(_relative_files_path + "/" +
"model.hex", std::ios_base::binary);
	if (fw.good()) {
		for (int i = 0; i < numberOf_X; i++) {
for (int k = 0; k < numberOf_H; k++) {
```

```
fw.write((char*)&W1[i][k], sizeof(float));
                    }
             }
             for (int j = 0; j < numberOf_Y; j++) {
                    for (int k = 0; k < numberOf_H; k++)
{
                          fw.write((char*)&W2[k][j],
sizeof(float));
                    }
             }
             for (int k = 0; k < numberOf_H; k++) {
                    fw.write((char*)&hidden_bias[k],
sizeof(float));
             }
             for (int j = 0; j < numberOf_Y; j++) {
                    fw.write((char*)&output_bias[j],
sizeof(float));
             }
             fw.write((char*)&_err_epoca_min_value,
sizeof(float));
             fw.write((char*)&_epsilon, sizeof(float));
             //fw.write((char*)&x[numberOf_X - 1],
sizeof(float));
             //fw.write((char*)&h[numberOf_H - 1],
sizeof(float));
             fw.close();
             //cout << "\nFile closed... \n\n";
      }
      else
```

```
            cout << "Problem with opening file";
}
```

Queste righe di codice rappresentano le istruzioni essenziali per realizzare sia l'apprendimento sia l'esecuzione di una rete neurale artificiale in ambito embedded. Solitamente, questo processo si svolge utilizzando un sistema elettronico che acquisisce tutte le misure necessarie per addestrare la rete neurale destinata al microcontrollore. Al termine della fase di addestramento viene generato un file in formato HEX contenente i pesi ottimizzati della rete neurale, grazie alla funzione write_weights_on_file() che si occupa di salvare tali valori. Questo file costituisce il modello definitivo che verrà successivamente utilizzato per la fase di inferenza. Infatti, caricando sul microcontrollore sia il codice relativo alla funzione forward() sia quello relativo alla funzione di attivazione sigmoidale (sigmoid_activation(float Z)) e trasferendo tramite il protocollo ISP il file HEX contenente i pesi direttamente nella memoria EEPROM, si ottiene così un dispositivo in grado di mettere in pratica quanto appreso durante la fase di addestramento.

Ovviamente, tutto ciò che stiamo imparando offre grandi potenzialità in termini di consumi e costi. Basti pensare che un sistema del genere, grazie alla modalità sleep, può funzionare per mesi, o addirittura anni, alimentato da una semplice batteria da 2000 mAh. Tale capacità corrisponde, più o meno, a quella di una batteria di uno smartphone economico, che garantisce circa una decina di ore di utilizzo continuo. Questo evidenzia quanto l'intelligenza artificiale, integrata in sistemi embedded, possa essere non solo potente ma anche estremamente efficiente. Inoltre, imparare a lavorare con sistemi semplici ed efficienti non preclude affatto la possibilità, in futuro, di adottare soluzioni più complesse e più dispendiose in termini di energia. Al contrario, il vero

problema sarebbe partire da sistemi pesanti, ignorando le ottimizzazioni possibili.

È assolutamente possibile utilizzare piattaforme più evolute, come l'RP2040 o altri processori ARM32, magari con MicroPython. Tuttavia, partire da una base solida scritta in C puro e "vanilla" consente di avere il pieno controllo del sistema e di scegliere gli strumenti più adeguati in termini di costi e consumi energetici.

Ho osservato, ad esempio, l'uso di Raspberry Pi per rilevare semplicemente il passaggio di persone tramite sensori PIR: una soluzione che considero uno spreco di risorse. Quando mi fu richiesto di realizzare un sistema simile per un capannone in campagna, privo di alimentazione fissa e destinato a effettuare una chiamata telefonica in caso di intrusione, optai per un ATiny85 abbinato a un sensore PIR. Il microcontrollore consumava pochissimi microampere, mentre il PIR ne richiedeva ancora meno. Il tutto era gestito mediante interrupt, sleep mode e un transistor, utilizzato per alimentare il modulo SIM solo al momento della chiamata. Con una semplice batteria da 3000 mAh, il sistema è rimasto attivo per un anno intero senza problemi.

Il codice riportato nel libro è volutamente incompleto e mostra soltanto le funzioni più importanti, così da poter essere facilmente riutilizzate in diverse situazioni, evitando invece di includere il solito esempio completo che mostra solo l'esercizio appreso dalla rete e che di fatto non ha alcuna utilità pratica, soprattutto perché oggi, con tutte le risorse disponibili online e con l'aiuto dei chatbot, chiunque può ottenere esempi dimostrativi di codice in pochissimi minuti.

In questo codice non ho incluso la funzione che carica gli array per l'addestramento, perché i dati da usare variano sempre a seconda dell'applicazione, e ho scelto anche di non riportare la funzione che esegue il compito di predicting, in quanto troppo specifica per esigenze particolari. Tuttavia, alla fine del libro

troverete un link da cui potrete scaricare il codice completo, anche se vi garantisco che, in questa materia, il codice è davvero l'ultima cosa su cui focalizzarsi: ciò che conta davvero è comprendere a fondo le regole matematiche che esso segue.

Partiamo con la funzione di main(), che permette di scegliere se eseguire una fase di apprendimento o una di esecuzione, impostando come predefinita la modalità di apprendimento. La prima richiesta che il programma pone è il valore di epsilon, il tasso di apprendimento. Questo parametro è fondamentale perché regola la velocità con cui i pesi della rete vengono aggiornati durante l'addestramento. In altre parole, controlla l'entità del "passo" compiuto dall'algoritmo per ridurre l'errore.

Quando si utilizzano funzioni di attivazione come la sigmoide o la ReLU, è comune lavorare con valori di epsilon compresi tra 0 e 1. Di solito si inizia con un valore relativamente alto, come 0.9, per ottenere un apprendimento rapido nelle fasi iniziali. Tuttavia, un tasso di apprendimento troppo elevato può portare a problemi nel processo di ottimizzazione. Il gradiente di una funzione in un punto è il vettore che indica la direzione di massima crescita locale; invertendone il verso, si ottiene il cammino di massima discesa verso il minimo della funzione di costo, proprio come seguire un sentiero ripido che scende lungo una valle. Se il passo è troppo grande, l'algoritmo può "saltare" il minimo, passando dall'altra parte e dovendo tornare indietro, causando un'oscillazione intorno al punto ottimale senza mai stabilizzarsi. Questo fenomeno, noto come overshooting, può compromettere la capacità della rete di convergere in maniera stabile.

Per ottenere il massimo beneficio, è consigliabile iniziare con un valore elevato per una rapida approssimazione iniziale, per poi ridurlo gradualmente man mano che la rete si avvicina alla soluzione ottimale. Questa strategia, che si traduce in un decadimento del tasso di apprendimento, aiuta a stabilizzare il

processo di training, consentendo all'algoritmo di affinare la stima dei pesi in modo più preciso. La scelta del valore ottimale di epsilon dipende dalla specifica architettura della rete e dal problema da risolvere, motivo per cui è sempre utile sperimentare e monitorare attentamente i risultati durante l'apprendimento.

Con questo approccio, si garantisce che la rete impari in modo efficiente, bilanciando la rapidità iniziale con una convergenza più accurata in seguito, in linea con le regole matematiche che governano il processo di ottimizzazione.

Di seguito partirà la funzione apprendi(), la quale si occupa di caricare i dati di training ed eseguire la funzione forward(). Quest'ultima calcola gli output in funzione delle somme pesate e delle attivazioni sia degli ingressi che dei livelli nascosti. Subito dopo, mediante la backpropagation, i pesi vengono aggiornati per ridurre l'errore.

È importante sottolineare che, come già evidenziato, sono coinvolte due variabili fondamentali: l'errore di rete e l'errore di epoca. Per l'errore di rete vengono calcolate la media, la varianza e la deviazione standard, i cui valori vengono poi confrontati con quelli dell'errore di epoca. Questo confronto ci consente di monitorare l'efficacia del processo di apprendimento e di valutare l'andamento del training.

Di seguito i dati di output della console che ci indicano il comportamento durante l'apprendimento.

epoca:550000

err_epoca=0.0353785

min. err_epoca=0.0341305

last time write on file =12:07:51

varianza di errore di rete = 3.01107e-05

media di errore di rete = 0.00976924

deviazione standard errore di rete = 0.00548732

max err_epoca is on sample line = 457

percentage dev.standard err_rete / media err_rete = 56.1694%

epsilon=1e-05

Figura 9

Prima di commentarli, indico un metodo molto semplice ma efficace per ripulire i dati acquisiti. È consigliabile salvare i dati provenienti da un processo di acquisizione in un file CSV. Questo formato è semplice e può essere elaborato facilmente tramite Excel.

Utilizzo un modulo dal costo di pochi centesimi, che si collega al microcontrollore e registra i dati necessari all'addestramento.

I dati, per vari motivi, possono contenere valori errati che potrebbero far pensare che l'errore di epoca, ovvero il maggiore errore rilevato durante l'apprendimento, rappresenti l'errore reale

della rete. In realtà, questo errore potrebbe essere causato da un singolo caso anomalo verificatosi durante l'apprendimento.

Per evitare questo problema, un primo consiglio utile è aprire il file CSV in Excel ed eseguire una procedura di pulizia dei dati. Poiché non tutte le acquisizioni sono identiche, non è sempre possibile scrivere un codice universale che ripulisca automaticamente i dati errati in qualsiasi circostanza. In questi casi, un controllo manuale può aiutare a individuare e rimuovere eventuali anomalie nei dati.

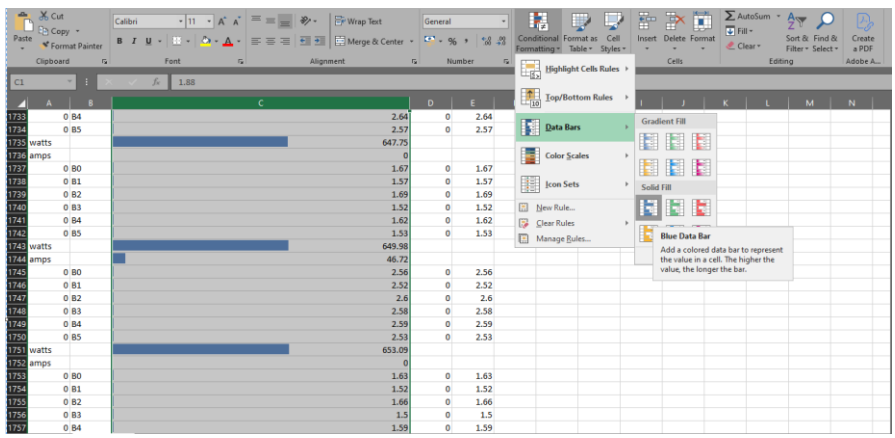


Figura 10

Selezionate la colonna che desiderate monitorare e utilizzate lo strumento indicato nella Figura 10. In questo modo potrete vedere immediatamente se ci sono dati che superano i limiti attesi e rimuoverli prima della fase di apprendimento.

Una volta eliminati i dati che non rientrano nei parametri, potete eseguire il vostro codice e monitorare i risultati, come mostrato nella Figura 9.

Tornando ai dati mostrati in console durante l'apprendimento, notiamo che il rapporto tra la deviazione standard (0.00549) e la media dell'errore di rete (0.00977) è di circa il 56,2%. Un

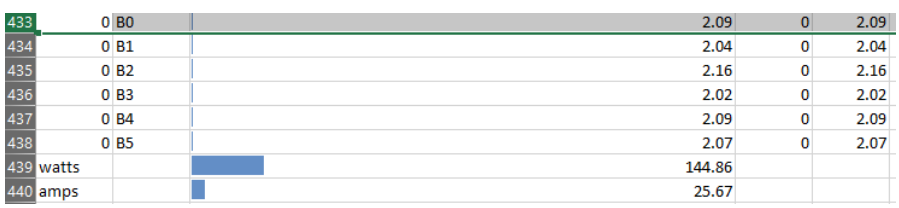
coefficiente di variazione così elevato indica che gli errori si discostano in modo significativo dalla media.

La varianza, invece, è pari a 3.01×10^{-5}, un valore molto piccolo: ciò significa che gli "scarti al quadrato" degli errori sono contenuti, ossia gli errori non si disperdono troppo attorno al loro valore medio. Possiamo anche osservare che il massimo errore della rete è di 0.0353785, mentre la media è di 0.00976924, e questo ci conferma che qualche campione altera il risultato.

Infatti, si osserva che il campione con il massimo errore dell'epoca si verifica alla linea 457, suggerendo che quel particolare dato potrebbe aver influenzato in modo rilevante il calcolo dell'errore durante l'apprendimento.

457	0	B0		2.01	0	2.01
458	0	B1		1.94	0	1.94
459	0	B2		2.06	0	2.06
460	0	B3		1.89	0	1.89
461	0	B4		1.97	0	1.97
462	0	B5		1.95	0	1.95
463	watts			154.23		
464	amps			27.84		

Effettivamente, da una prima analisi, la situazione appare regolare. Tuttavia, esaminando più attentamente altri casi con potenza intorno ai 150 watt e un assorbimento di circa 30 ampere, rilevo che la media della tensione delle batterie si attesta intorno ai 2 volt.



433	0	B0		2.09	0	2.09
434	0	B1		2.04	0	2.04
435	0	B2		2.16	0	2.16
436	0	B3		2.02	0	2.02
437	0	B4		2.09	0	2.09
438	0	B5		2.07	0	2.07
439	watts			144.86		
440	amps			25.67		

Pertanto, al fine di ridurre ulteriormente l'errore complessivo, sarebbe opportuno escludere questo specifico campione dall'apprendimento.
In ogni caso, un errore massimo pari a 0,030 volt su un range compreso tra 0 e 3,5 volt, a fronte di una media dell'errore di circa 0,010 volt, può ritenersi accettabile.

Questo soprattutto perché, come già evidenziato, ciò che importa maggiormente è la relazione tra i due vettori coinvolti: i dati osservati e quelli predetti.

A questo punto, non resta che illustrare come manipolare i due vettori al fine di determinare se essi siano effettivamente simili e correlati tra loro.

L’esecuzione su microcontrollore

Abbiamo ora un file HEX, ovvero il nostro modello, che contiene tutti i pesi e le sinapsi necessari per prevedere un risultato in base agli ingressi forniti. Questa fase, che in origine dovrebbe riguardare esclusivamente il microcontrollore, può essere eseguita anche su qualsiasi piattaforma in grado di compilare ed eseguire il codice. A tal fine ho implementato la funzione predict(), che esegue il lavoro senza aggiornare i pesi, ovvero dopo l’addestramento.

È fondamentale comprendere come sono stati normalizzati i dati in ingresso e quelli in uscita. Come detto in precedenza, tutti i dati devono essere normalizzati in un intervallo compreso tra 0 e 1, per cui sia i watt sia gli ampere devono essere opportunamente convertiti. Il problema, tuttavia, è che una semplice divisione rischierebbe di penalizzare eccessivamente i watts. Ad esempio, se gli ampere, il cui range va da 0 a 50, venissero divisi per 100, mentre i watt, che variano da 0 a 2000, venissero divisi per 10000, si creerebbe uno squilibrio. Se, ad esempio, gli ampere fossero 50, e i watt fossero 10, il sistema avrebbe difficoltà a cogliere i rapporti, poiché valori come 0.50 e 0.001 risulterebbero troppo sproporzionati.

Per ovviare a questo problema ho utilizzato il logaritmo naturale: ln(50) è circa 3.91 e ln(10) circa 2.30; dividendo questi risultati per 10 si ottengono valori molto più bilanciati. Lo stesso procedimento va applicato ai dati di output, che devono essere divisi per 10, considerando che i volt, nel mio caso, variano da 0 a 4. Poiché tutto ciò è già stato gestito durante l’addestramento, il modello si aspetta di ricevere dati normalizzati, e i risultati andranno successivamente moltiplicati per 10 per essere denormalizzati.

In sintesi, la funzione predict() esegue quanto appreso dalla rete.

```
void predict() {
	float normalized_observed_output[numberOf_Y] = {
0.00 };
	float normalized_predicted_output[numberOf_Y] = {
0.00 };
	int sampleIndex = 0;
	while (true) {
		std::cout << "\nInsert file sample line
(Ctrl+C to esc):\n";
		std::cin >> sampleIndex;
		get_sample_for_test(sampleIndex);
		normalizeArray(observed_data,
normalized_observed_output, numberOf_Y);
		x[0] = log(x[0] + 1.00f) / 10.00f;
		x[1] = log(x[1] + 1.00f) / 10.00f;
		forward();
		for (int i = 0; i < 6; i++) {
			y[i] = y[i] * 10.00f;
		}
		normalizeArray(y,
normalized_predicted_output, numberOf_Y);
		float mse = mean_square_error(observed_data,
y, numberOf_Y);
		float overall_mean=
overallMean(normalized_observed_output,
		normalized_predicted_output, numberOf_Y);
```

```
 float percentage = calculateErrorPercentage(mse,
overall_mean);

             float varianza =
calculateVariance(normalized_observed_output,
numberOf_Y);

             std::cout << "percentage = :" << percentage
<< "%\n";

             std::cout << "varianza = :" << varianza <<
"\n";

             // Stampa dei risultati

             std::cout << "\n x[0] = " << exp(x[0] *
10.00f) << " x[1] = " << exp(x[1] * 10.00f) << "\n"

                    << "\n y[0] = " << y[0]

                    << "\n y[1] = " << y[1]

                    << "\n y[2] = " << y[2]

                    << "\n y[3] = " << y[3]

                    << "\n y[4] = " << y[4]

                    << "\n y[5] = " << y[5];

      }
}
```

Prima di eseguire la funzione predict(), se state lavorando con un microcontrollore, dovrete leggere i pesi salvati nella EEPROM; se invece lavorate su PC, potrete richiamare direttamente la funzione che li legge dal file.

Successivamente, è necessario acquisire i dati reali in un determinato istante, ovvero i wattora, gli ampere e i livelli di tensione delle sei batterie. In alternativa, come nella funzione di esempio, potete simulare la lettura da un file che contiene i dati

utilizzati per l’addestramento. Questi valori dovranno poi essere caricati nell’array observed_data e nelle variabili x[0] e x[1].

Se l’addestramento del modello è stato eseguito correttamente, il rapporto percentuale tra l’errore quadratico medio dei due vettori (quello predetto e quello osservato) e la media degli stessi valori normalizzati sarà compreso tra il 20% e il 30%. Questo risultato indica una buona somiglianza nella forma dei due vettori. Infine, grazie all’errore quadratico medio e alla varianza, sarà possibile individuare eventuali errori eccessivi in specifici punti o anomalie significative nei livelli delle batterie rispetto alla media prevista.

Calcolo delta per pesi tra livello nascosto e livello di output.

$$\Delta W_{kj} = \text{removed from demo}$$

- ϵ è il tasso di apprendimento, valore tra 0 e 1.
- $E = removed\ from\ demo$
- $\hat{y}_j = \frac{1}{1+e^{-z_j}} \Longrightarrow \frac{\partial \hat{y}_j}{\partial Z_j} = \hat{y}_j \cdot (1 - \hat{y}_j)$
- $Z_j = \text{removed from demo}$

Calcolo delta per pesi tra livello di input e livello di nascosto.

$$\text{removed from demo}$$

Le prime due derivate parziali sono già state risolte risolte, mentre le altre seguono:

- $Z_j = \sum_{k=1}^{n}$
- $Z_k = \sum_{i=1}^{n}$
- $\frac{\partial H_k}{\partial Z_k}$

Formule per analisi dei dati.

- Min-Max Normalization utilizzata per normalizzare gli array tra valori 0-1

$$x_i^{norm} = \frac{x_i - \min(X)}{\max(X) - \min(X)}$$

- Formula del Mean Squared Error tra due vettori $X^{(1)}$ *e* $X^{(2)}$:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n} (x_i^{(1)} - x_i^{(2)})^2$$

- Root mean squared error (RMSE)

$$\text{RMSE} = \sqrt{\text{MSE}}$$

- Varianza : $\text{Var}(X) = \frac{1}{n}\sum_{i=1}^{n} (x_i - \bar{x})^2$

- Deviazione standard : $\sqrt{\text{Var}(X)}$

Link e risorse

GitHub Repository :

removed from demo version.

Code::Blocks :

https://www.codeblocks.org/downloads/binaries/#imagesoslinux48pnglogo-linux-32-and-64-bit

AvrDudess :

https://blog.zakkemble.net/avrdudess-a-gui-for-avrdude/

Visual Studio Community :

https://visualstudio.microsoft.com/it/vs/community/

Visual Micro :

https://www.visualmicro.com/

Arduino IDE 1.8.X :

https://www.arduino.cc/en/software/

Editor Binario per Linux e per Windows utile per analisi del file HEX:

Per ricevere aggiornamenti, video di approfondimento riguardo l'utilizzo di microprocessori o altro scrivere a LSGSoftware@hotmail.com.