NOTES
Foundations of Deep Learning

Matteo Suardi
930935
A.Y. 2024-2025

Sommario

Il presente documento fornisce una panoramica esaustiva dei fondamenti del deep learning, illustrando sia gli aspetti teorici sia quelli applicativi nel contesto dell'intelligenza artificiale. Vengono introdotti i concetti chiave dell'AI e del machine learning, evidenziando l'evoluzione dal paradigma tradizionale alla crescente adozione di modelli basati sul deep learning. Particolare attenzione è dedicata all'importanza delle rappresentazioni dei dati, alla definizione di funzioni di mappatura, nonché alla gestione dei dati in differenti forme (visivi, temporali, testuali) e ai problemi connessi quali la *curse of dimensionality*.

Il documento approfondisce gli strumenti matematici alla base del deep learning, con un focus particolare sull'algebra lineare, comprensiva di vettori, matrici, tensori e operazioni fondamentali (trasposizione, moltiplicazione, norme) e le tecniche di decomposizione (eigendecomposition, SVD, PCA) per la riduzione della dimensionalità e l'analisi dei dati. Inoltre, vengono esposti i principi alla base dell'apprendimento supervisionato, con particolare riguardo alla minimizzazione della funzione di costo (loss) tramite l'ottimizzazione, illustrando modelli di regressione (lineare e polinomiale) e l'importanza della regolarizzazione per prevenire l'overfitting.

Un'ampia sezione è dedicata alle reti neurali artificiali, analizzandone la struttura feedforward, il ruolo delle funzioni di attivazione e la metodologia di backpropagation per il calcolo dei gradienti e l'aggiornamento iterativo dei pesi. Vengono inoltre discussi i problemi di ottimizzazione tipici del deep learning, come il vanishing gradient, e le relative tecniche di mitigazione (inizializzazione dei pesi, funzioni di attivazione non saturanti, batch normalization, architetture residuali). Infine, il documento sintetizza la "ricetta" di base per la costruzione e l'ottimizzazione dei modelli di deep learning, evidenziando l'importanza dell'interazione tra dati, funzione di costo, procedura di ottimizzazione e struttura del modello.

Indice

1 Introduzione

1.1 Intelligenza artificiale (AI)

L'**intelligenza artificiale (AI)** è un ramo dell'informatica che si occupa di creare programmi o macchine in grado di svolgere compiti che, se eseguiti da un essere umano, richiedeerebbero intelligenza. Ad esempio: riconoscere un immagine, capire un discorso umano, prendere decisioni o risolvere problemi complessi. L'AI è diventata molto importante nella vita di tutti i giorni e ha anche un forte impatto etico (ad esempio sulla privacy o sicurezza).
Il termine "intelligenza artificiale" è stato coniato da **John McCarthy** nel 1956. Lui la definiva come *"la scienza e l'ingegneria che si occupa di creare* ***macchine intelligenti****.*
Alan Turing, famoso matematico, è considerato uno dei precursori dell'AI. Nel 1950, propose il un metodo (il **Test di Turing**) per capire se una macchina potesse essere considerata "intelligente": se in una conversazione scritta la macchina riesce a farsi passare per un umano, allora è da considerarsi intelligente. Turing non affermava che questo fosse l'unico modo per considerare una macchina intelligente, ma che fosse una prima idea per valutarla.
L'AI si divide in diversi rami, ognuno specializzato in un modo diverso di "pensare" o "apprendere":

- **Logic AI**: si basa sulla logica, cioè sull'uso di regole formali per programmare il ragionamento.
- **Pattern recognition**: fa confronti tra ciò che "vede" e certi schemi o modelli precedentemente conosciuti. Un esempio comune è il riconoscimento facciale o il riconoscimento della scrittura a mano.
- **Common sense knowledge and reasoning (conoscenza e ragionamento di buon senso)**: serve a far capire a una macchina cose che per noi umani sono ovvie (ad esempio: se piove è meglio portare un ombrello).
- **Learning from experience (apprendere dall'esperienza)**: qui entrano in gioco le **reti neurali** e metodi che fanno sì che la macchina "impari" dai dati, un po' come facciamo noi esseri umani.
- **Ontology**: descrive le conoscenze in modo strutturato e formale (ad esempio, come catalogare e descrivere tutti i tipi di animali, piante, oggetti e come sono in relazione tra loro).
- **Heuristics**: un'euristica è una scorciatoia o regola pratica per avvicinarsi alla soluzione di un problema, anche se non sempre è la soluzione perfetta o formale.
- **Genetic programming**: si ispira ai processi di evoluzione biologica e genetica (come mutazioni o incroci) per "evolvere" programmi che risolvono problemi.

Esempi pratici

- **Assistenti vocali**: riconoscono la nostra voce (pattern recognition), interpretano la nostra richiesta (common reasoning e logica) e cercano di darci la risposta migliore (heuristics, database di conoscenze, ecc.).

- **Raccomandazioni su YouTube o Netflix**: il sistema impara dai nostri gusti (learning from experience) per proporci video o film che potrebbero piacerci.

In generale, quando pensiamo all'AI, immaginiamoci software o macchine che automatizzano compiti che richiedono "intelligenza" (anche se, per ora, si tratta di un'intelligenza diversa da quella umana). L'AI è molto usata in tanti settori, dalla medicina (diagnosi di malattie grazie a reti neurali) fino all'automotive (auto a guida autonoma).

1.2 Dall'AI al deep learning

1.2.1 Machine learning

In passato, l'idea alla base dell'AI era quella di programmare manualmente le regole che il computer doveva seguire per risolvere problemi. Queste regole erano espresse in forma matematica e logica. Ad esempio, se avessi voluto creare un programma che riconoscesse se una luce fosse accesa o spenta, potevi scrivere delle regole precise per decidere in base ad alcuni valori.

Tuttavia, molti problemi del mondo reale sono troppo complessi per essere descritti completamente con regole fisse. Per questo motivo, si è capito che i computer devono essere in grado di **imparare** dalle esperienze, cioè dai dati, anziché basarsi solo su regole scritte a mano. Questa capacità di apprendere dai dati è ciò che oggi chiamiamo **machine learning**. Gli algoritmi di ML più semplici sono la **regressione lineare** e la **regressione logistica**.

Regressione lineare La regressione lineare cerca di stabilire una **relazione lineare** tra due variabili:

- x: variabile indipendente (il "fattore" o "input" da cui si parte, ad esempio l'altezza di una persona).
- y: variabile dipendente (ciò che vogliamo prevedere, ad esempio il peso della persona).

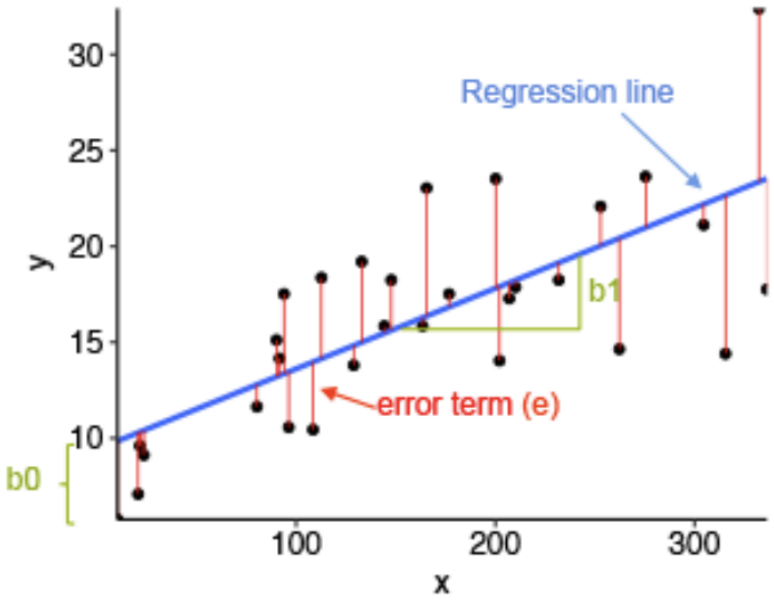
La formula della regressione lineare è:

$$y = \beta_0 + \beta_1 x + \epsilon,$$

in cui:

- β_0 è l'**intercetta**, ovvero il valore di y quando x è 0. Immaginiamolo come il punto da cui parte la retta.
- β_1 è il **coefficiente** che indica quanto y aumenta (o diminuisce) per ogni unità in più di x. Equivale alla **pendenza** della retta.
- ϵ rappresenta l'**errore** o le variazioni non spiegate dal modello; è tutto ciò che non si riesce a prevedere perfettamente.

Esempio pratico: immaginiamo di voler prevedere il prezoz di una casa (y) in base alla sua superficie in metri quadrati (x). La regressione lineare cercherà di tracciare una retta che "passi" tra i punti dati (le case con il loro prezzo e la loro superficie) per poter stimare il prezzo anche di case nuove (dati sconosciuti).



Regressione logistica La regressione logistica si usa quando la variabile da prevedere è **categorica binaria**, cioè ha solo due possibili risultati (sì/no, vero/falso, maschio/-femmina, ecc.).

La formula tipica della regressione logistica usa la funzione sigmoide:

$$p = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}},$$

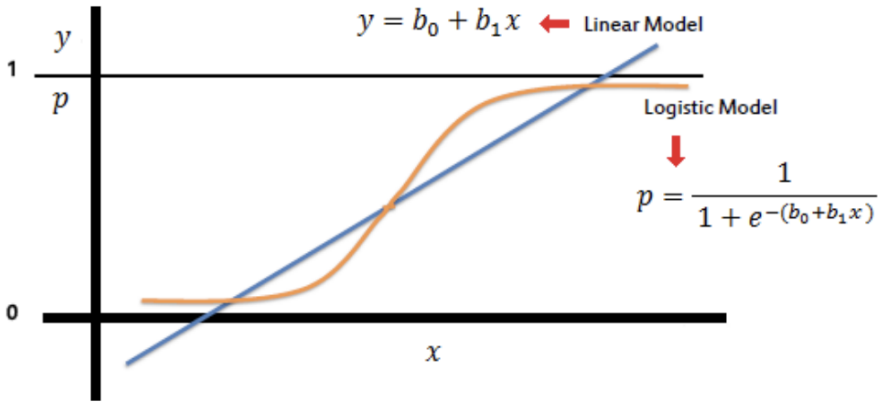
in cui:

- p rappresenta la probabilità che l'evento (ad esempio, "la persona compra il prodotto") si verifichi, ed è sempre compresa tra 0 e 1.
- β_0 e β_1 hanno un ruolo simile a quelli della regressione lineare: β_0 è l'intercetta e β_1 è il coefficiente che indica l'influenza di x.
- e è il numero di Nepero e la base del logaritmo naturale (circa 2.71828).
- La funzione **sigmoide** trasforma il risultato (che può essere qualsiasi numero) in una probabilità compresa tra 0 e 1.

Esempio pratico: consideriamo un sistema che deve decidere se una email è spam o no. Qui, x potrebbe rappresentare alcune caratteristiche della email (come la frequenza di certe parole) e p è la probabilità che l'email sia spam. Se p è superiore ad una certa soglia (ad esempio, 0.5), il sistema classifica la mail come spam.

Funzione di mappatura e importanza delle caratteristiche (features)

- **Concetto di mapping**: si parla di **funzione di mappatura (mapping function)** f, scritta come $x ß f(x)$, per indicare che l'algoritmo impara una funzione che trasforma l'input (x) nell'output desiderato (y). In altre parole, dato un insieme di dati in ingresso, il modello cerca di capire come questi dati siano collegati al risultato che vogliamo prevedere.

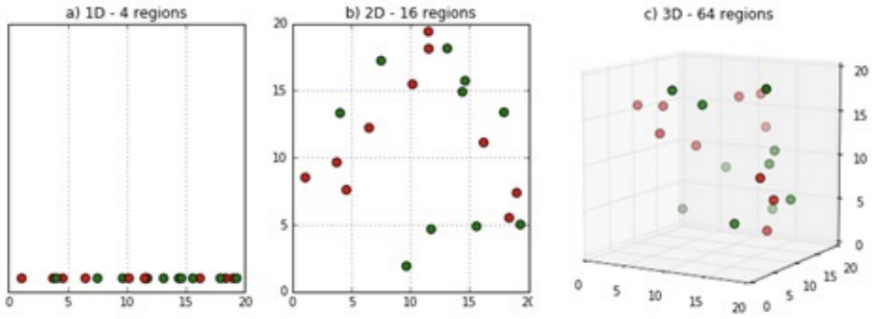


- **Features**: le informazioni (o caratteristiche) che descrivono i dati sono chiamate **features**. La qualità delle previsioni dipende fortemente da come sono rappresentati questi dati. Se scegliamo bene le features, il modello potrà apprendere molto meglio.

Curse of dimensionality Quando si aumenta il numero di features (cioè le dimensioni dei dati), il numero di dati necessari per "imparare" bene la relaizone cresce in maniera esponenziale. Immaginiamo di dover descrivere un'immagine in cui ogni pixel è una caratteristica: un'immagine ad alta risoluzione può avere centinaia di migliaia di pixel e, per questo, sarebbe nececssario un numero enorme di immagini per allenare correttamente il modello. Per far fronte a questo problema, si usano tecniche di **riduzione della dimensionalità** per ridurre il numero di features, proiettando i dat in uno spazio a dimensioni inferiori. Ad esempio:

- Per le immagini si possono usare filtri specifici, come i **Gabor filters**, per estrarre solo le caratteristiche più importanti.
- Oppure, si possono usare istogrammi per differenziare tipi di immagini (ad esempio, distinguere tra una foto di pasta e una di caffè).

Ridurre la dimensionalità può aiutare il modello a gestire meglio i dati, ma potrebbe comportare una perdita di informazioni. Quindi, è importante bilanciare la riduzione con la necessità di mantenere quante più informazioni possibili.

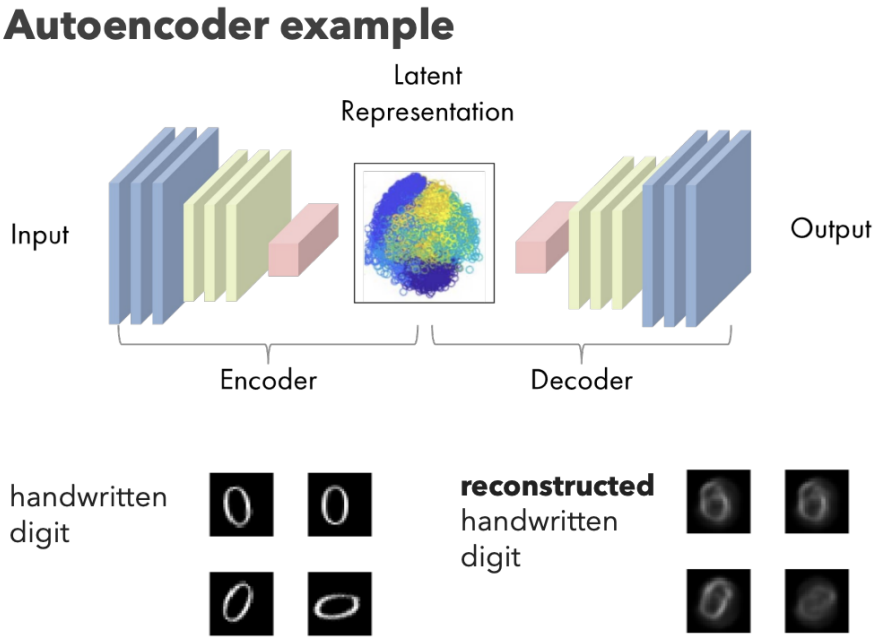


Representation learning e Autoencoder Spesso non è semplice capire quali siano le features migliori da estrarre per un determinato problema. La soluzione è far sì che il modello impari non solo a mappare gli input agli output, ma anche a creare autonomamente una **rappresentazione** (cioè, a estrarre le caratteristiche migliori dai dati). Questo processo è chiamato **representation learning**.

Un esempio di algoritmo di RL sono gli **autoencoder**. Un autoencoder è composto da due parti:

- **Encoder**: è una funzione che trasforma i dati originali in una rappresentazione più compatta e spesso più utile per il modello.
- **Decoder**: cerca di ricostruire i dati originali partendo dalla rappresentazione compattta. L'idea è che, se il modello riesce a ricostruire bene i dati, allora la rappresentazione interna deve contenere le informazioni più importanti.

Esempio pratico: gli autoencoder possono essere usati per ridurre il rumore nelle immagini: si "impara" una rappresentazione pulita e poi si ricostruisce l'immagine eliminando le imperfezioni. Oppure, possono essere usati per comprimere dati in formati più piccoli, mantenendo le informazioni essenziali.



Fattori di variazione I fattori di variazione sono elementi o caratteristiche che spiegano la **variabilità** nei dati. Ad esempio, in un'immagine di un volto umano, fattori come la forma degli occhi, del naso e della bocca sono elementi che aiutano a identificare e differenziare i volti. Lo scopo è separare questi fattori per capire meglio come sono composti i dati e per facilitare l'apprendimento del modello.

Riassunto pratico Immaginiamo di voler insegnare ad un computer a riconoscere se una foto rappresenta un cane o un gatto:

1. **Step 1**: il computer impara dalla statistica (tramite algoritmi come la regressione) come le caratteristiche (dimensioni, forme, colori) delle immagini si collegano alla presenza di un cane o di un gatto.

2. **Step 2**: se le immagini hanno tanti dettagli (migliaia di pixel), potremmo ridurre la dimensionalità usando tecniche specifiche, così il modello lavora su meno dati ma più significativi.

3. **Step 3**: con tecniche di representation learning, il computer può imparare autonomamente quali caratteristiche (features) sono le più importanti per distinguere tra un cane ed un gatto, magari attraverso un autoencoder che comprime e poi ricostruisce l'immagine.

Questa metodologia rende possibile affrontare problemi complessi senza dover specificare ogni singola regola manualmente, affidandosi invece alla capacità dei computer di apprendere dai dati.

1.2.2 Deep learning

Il deep learning è una branca del machine learning che permette alle macchine di imparare concetti complessi a partire da concetti molto più semplici. In altre parole, il deep learning costruisce una "grearchia" di rappresentazioni: partendo dai dati grezzi, il modello impara prima le caratteristiche semplici e, via via, ne combina altre per formare concetti più astratti e complessi.

Composizione di funzioni Immaginiamo di voler calcolare un risultato seguendo più passaggi. Ogni passaggio è una funzione che prende un input e restituisce un output. Nel deep learning, queste funzioni sono "impilate" una sull'altra. Formalmente, possiamo rappresentare il modello come:

$$f(x) = f_n(f_{n-1}(...f_2(f_1(x))...)),$$

questo significa che il risultato finale $f(x)$ si ottiene applicando una serie di funzioni (i cosiddetti *layer*) una dopo l'altra. Ogni funzione trasforma l'input, rendendo la rappresentazione dei dati più adatta a risolvere il problema specifico.

Percettrone Un esempio pratico di una funzione semplice usata nel deep learning è il **percettrone**, chiamato anche "neurone artificiale". Esso prende più input, li moltiplica per dei pesi, somma i risultati, aggiunge un **bias** e applica una funzione di attivazione per ottenere l'output. La formula è:

$$y = f(w_1x_1 + w_2x_2 + ... + w_nx_n + b),$$

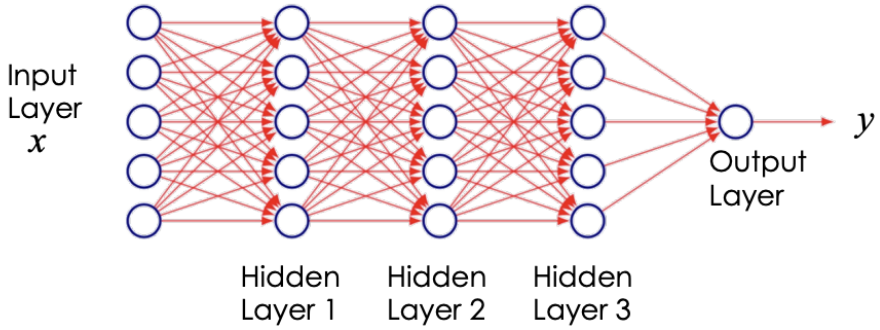
dove:

- $x_1, x_2, ..., x_n$ sono gli input (ad esempio, i pixel di un'immagine),
- $w_1, w_2, ..., w_n$ sono i pesi associati a ciascun input,
- b è il bias (un termine aggiuntivo che aiuta a migliorare la flessibilità del modello),

- f è la funzione di attivazione (ad esempio, la funzione ReLU o la sigmoide) che trasforma la somma in un output, spesso per introdurre la non-linearità nel modello.

Esempio pratico: pensiamo ad un sistema che deve riconoscere se in una foto è presente un gatto. All'inizio, il percettrone potrebbe semplicemente rilevare se ci sono dei bordi o delle curve nell'immagine. In uno strato successivo, altri neuroni combinano queste informazioni per riconoscere forme più complesse come occhi o orecchie, e così via fino a "costruire" la rappresentazione di un gatto.

Reti neurali profonde (DNN) e Multilayer perceptron (MLP) Un **multilayer perceptron (MLP)** è un tipo di rete neurale profonda che consiste in molti strati di percettroni. Ogni strato riceve in input l'output dello strato precedente applica la sua trasformazione. Questo permette al modello di apprendere rappresentazioni molto complesse e astratte.



1.3 Evoluzione dell'AI

1) Approccio tradizionale nel machine learning Nel metodo tradizionale, per far funzionare un algoritmo di ML, si seguono vari passaggi:

- **Pre-elaborazione dei dati**: i dati grezzi (cioè non ancora organizzati) vengono "ripuliti" e trasformati in informazioni utili.
- **Estrazione delle caratteristiche (feature engineering)**: da questi dati si estraggono le "features", ovvero gli aspetti o le informazioni più rilevanti. *Ad esempio, se vogliamo analizzare immagini, potremmo estrarre caratteristiche come il colore medio o il bordo degli oggetti.*
- **Apprendimento e previsione**: queste features vengono poi usate per addestrare un algoritmo che impara a riconoscere schemi e relazioni. Una volta addestrato, il modello può fare previsioni su dati nuovi.

2) Deep learning Il deep learning rappresenta un'evoluzione in cui molti passaggi del processo radizionale vengono automatizzati:

- **Apprendimento gerarchico**: i modelli di deep learning, grazie alle reti neurali artificiali, sono in grado di apprendere rappresentazioni dei dati a più livelli. *In altre parole, il modello costruisce concetti complessi a partire da concetti più semplici, senza bisogno di estrarre manualmente le features.*

- **Input diretti**: i dati grezzi possono essere immessi direttamente nella rete, che si occupa di scopreire da sola le caratteristiche più rilevanti per risolvere il problema.
- **Sfide**: anche se questo approccio elimina il passaggio manuale di feature engineering, rimane difficile trovare l'architettura giusta e impostare i parametri (detti *hyperparameters*) ottimali.

3) Pipeline ibride Sempre più spesso si usano pipeline che combinano tecniche tradizionali e deep learning. In un ***esempio pratico*** con audio:

- Si può trasformare un suono in un'immagine chiamata **spettrogramma**, che rappresenta visivamente l'andamento delle frequenze.
- Lo spettrogramma viene poi processato da una rete neurale profonda (come una rete convolluzionale) che estrae le caratteristiche e fa la previsione, combinando il vantaggio della rappresentazione manuale con la capacità di apprendere da dati grezzi.

4) Dall'AI ristretta (ANI) all'Intelligenza Artificiale Generale (AGI)

- **AI ristretta (ANI)**: per molti decenni, l'obiettivo era creare sistemi intelligenti per compiti specifici e ben delimitati (come il riconoscimento facciale o la traduzione automatica).
- **Obiettivo originario**: l'idea iniziale dell'AI era costruire "macchine pensanti", cioè sistemi che possano avere un'intelligenza simile a quella umana.
- **Artificial General Intelligence (AGI)**: oggi c'è una crescente spinta verso il raggiungimento di un'AI capace di svolgere molti compiti diversi, avvicinandosi a quella "intelligenza generale" che imiti il ragionamento umano.

In sintesi, mentre il machine learning tradizionale richiede molti passaggi manuali per preparare i dati, il deep learning automatizza gran parte di questo processo grazie alle sue reti neurali che apprendono da sole le rappresentazioni utili. Le pipeline ibride e i progressi verso l'AGI rappresentano il futuro, mirando a creare sistemi sempre più versatili e capaci di affrontare compiti complessi in ambienti reali.

1.4 Definizioni di "deep learning"

1. Una classe di tecniche di machine learning che sfruttano numerosi strati di elaborazione non lineare delle informazioni per l'estrazione e la trasformazione delle caratteristiche in modalità supervisionata o non supervisionata, e per l'analisi di pattern e la classificazione.
2. Un sotto-campo all'interno del machine learning basato su algoritmi per apprendere molteplici livelli di rappresentazione al fine di modellare relazioni complesse tra i dati. Le caratteristiche e i concetti di livello superiore sono quindi definiti in termini di quelli di livello inferiore, e tale gerarchia di caratteristiche è chiamata un'architettura profonda. La maggior parte di questi modelli si basa sull'apprendimento non supervisionato delle rappresentazioni.

3. Un sotto-campo del machine learning che si fonda sull'apprendimento di diversi livelli di rappresentazioni, corrispondenti a una gerarchia di caratteristiche, fattori o concetti, dove i concetti di livello superiore sono definiti a partire da quelli di livello inferiore, e gli stessi concetti di livello inferiore possono contribuire a definire molti concetti di livello superiore. Il deep learning è parte di una famiglia più ampia di metodi di machine learning basati sull'apprendimento delle rappresentazioni.

4. Il deep learning è un insieme di algoritmi del machine learning che cercano di apprendere su più livelli, corrispondenti a differenti livelli di astrazione. Solitamente utilizza reti neurali artificiali. I livelli in questi modelli statistici appresi corrispondono a distinti livelli concettuali, dove i concetti di livello superiore sono definiti a partire da quelli di livello inferiore, e gli stessi concetti di livello inferiore possono contribuire a definire numerosi concetti di livello superiore.

5. Il deep learning è una nuova area di ricerca nel machine learning, introdotta con l'obiettivo di avvicinare il machine learning a uno dei suoi obiettivi originari: l'intelligenza artificiale. Il deep learning riguarda l'apprendimento di molteplici livelli di rappresentazione e astrazione che aiutano a dare un senso a dati come immagini, suoni e testi.

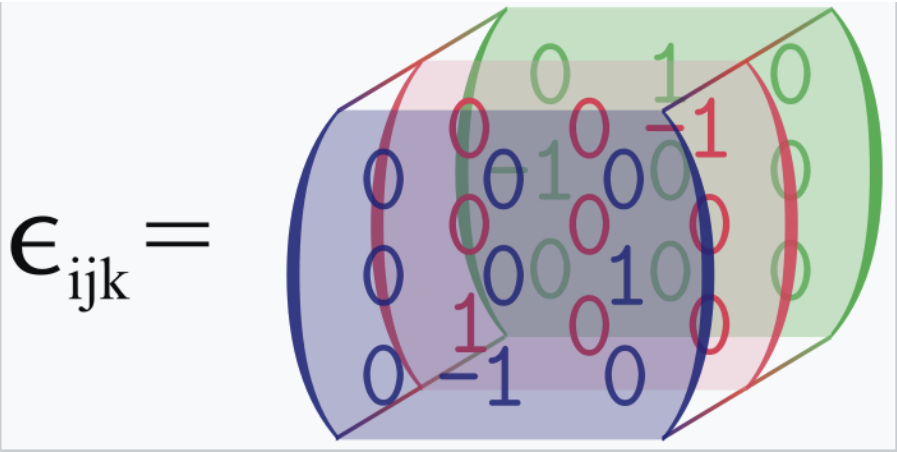
6. Il deep learning è un sottoinsieme del machine learning che coinvolge l'uso di reti neurali artificiali per permettere ai computer di apprendere e prendere decisioni basate sui dati. Queste reti neurali sono composte da molteplici strati di nodi o neuroni interconnessi che collaborano per elaborare e analizzare i dati in ingresso. Il termine "deep" nel deep learning si riferisce al numero di strati presenti in queste reti neurali. Le reti neurali profonde possono avere vari strati, ciascuno dei quali esegue un tipo diverso di calcolo, permettendo loro di apprendere modelli complessi e relazioni tra i dati. Ciò le rende particolarmente efficaci in compiti come il riconoscimento di immagini, l'elaborazione del linguaggio naturale e il riconoscimento vocale.

7. Il deep learning è un tipo di machine learning che utilizza reti neurali artificiali per analizzare e prendere decisioni basate sui dati. Implica la costruzione e l'addestramento di modelli complessi con molteplici strati di neuroni interconnessi, capaci di apprendere e riconoscere pattern nei dati. Questo consente di applicare il deep learning a compiti quali il riconoscimento di immagini e suoni, l'elaborazione del linguaggio naturale e molti altri.

2 Algebra lineare

2.1 Basi di algebra lineare

L'algebra lineare è un ramo della matematica che studia i vettori, le matrici e le altre strutture correlate. È molto importante nel deep learning perché ci fornisce gli strumenti per rappresentare e manipolare dati complessi, come immaini o segnali audio, che spesso sono organizzati in strutture a più dimensioni (dette **tensori**). Le componenti più comuni



dell'algebra lineare sono:

Scalare Uno scalare è semplicemente un **numero singolo**. Può essere un intero come (3 o -2), un numero reale (come 4.5 o -0.7) o un altro tipo di numero.

- **Notazione**: si usano lettere in corsivo e minuscole (ad esempio a, n, x).
- **Esempio pratico**: immaginiamo la temperatura di una stanza. Se diciamo che è di 22°C, quel 22 è uno scalare.
- **Utilizzo nel deep learning**: gli scalari possono rappresentare valori singoli, come il peso di una connsessione in una rete neurale.

Vettore Un vettore è una **lista ordinata di numeri**. Può essere pensato come una freccia nello spazio, che ha una direzione, una punta (verso) e una lunghezza (modulo).

- **Notazione**: un vettore si indica con una lettera in grassetto o con una linea sopra, per esempio $\overline{x}$, e si scrive come:

$$\overline{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

  Se contiene n elementi, diciamo che appartiene allo spazio campionario $\mathbb{R}^n$.
- **Esempio pratico**: pensiamo a un punto su una mappa. Se usiamo due numeri per indicare la latitudine e la longitudine, questi due numeri formano un vettore in $\mathbb{R}^2$.

- **Utilizzo nel deep learning**: i vettori sono usati per rappresentare i dati. Ad esempio, una singola immagine in bianco e nero può essere "appiattita" in un vettore dove ogni numero rappresenta l'intensità di un pixel.

Importanza dell'algebra lineare nel deep learning

- **Rappresentazione dei dati**: nei modelli di deep learning, i dati di input (come immagini, testi o suoni) vengono spesso organizzati in strutture chiamate matrici o tensori, che sono generalmente un insieme di vettori.
- **Operazioni matematiche**: operazioni come la motliplicazione di matrici, la trasposizione e l'inversione sono cruciali per calcolare come i dati si trasformano all'interno di una rete neurale.
- **Ottimizzazione**: durante l'allenamento di un modello, si calcolano gradiente e aggiornamenti dei pesi (che sono scalari e vettori) per migliorare le prestazioni del modello.

Esempio pratico: immaginiamo di voler riconoscere i volti in una foto. L'immagine è trasformata in una matrice di pixel (un insieme di vettori). Attraverso operazioni di algebra lineare, la rete neurale combina e trasforma questi vettori per estrarre le caratteristiche principali (come occhi, naso e bocca) e riconoscere il volto.

2.1.1 Matrici

Matrici e tensori

- Una **matrice** è una tabella bidimensionale di numeri, dove ogni numero si trova in una pposizione identificata da due indici: uno per la riga e uno per la colonna.
  - **Notazione**: se abbiamo una matrice A con m righe e n colonne, la scriviamo come $\mathbf{A} \in \mathbb{R}^{m \times n}$ se contiene numeri reali.
  - **Esempio pratico**: immaginiamo una tabella Excel con i dati. Ogni cella contiene un numero e la posizione della cella è data dalla sua riga e dalla sua colonna.
- Un **tensore** è una generalizzazione delle matrici, in cui:
  - **0 dimensioni**: è uno scalare (un singolo numero).
  - **1 dimensione**: è un vettore (una lista ordinata di numeri).
  - **2 dimensioni**: è una matrice.
  - **3 o più dimensioni**: si parla di tensori, utili per rappresentare dati complessi come immagini a colori (altezza * larghezza * canali). Ad esempio, un'immagine a colori (512*512 pixel con 3 canali per RGB) può essere rappresentata come un tensore 3D.

Operazioni con le matrici

Trasposizione di una matrice La trasposizione di una matrice consiste nel "riflettere" la matrice lungo la diagonale principale (quella che va dall'angolo in alto a sinistra a quello in basso a destra).

- **Notazione**: se A è una matrice, la sua trasposta si indica con A^T e l'elemento in posizione (i, j) di A^T è uguale all'elemento in posizione (j, i) di A.
- ***Esempio pratico***: se $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, allora $A^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$

Prodotto tra matrici Il prodotto di due matrici A e B è possibile quando il numero di **colonne** di A è uguale al numero di **righe** di B. Il risultato è una nuova matrice C in cui ogni elemento $C_{i,j}$ è dato dalla somma dei prodotti degli elementi della i-esima riga di A per quelli della j-esima colonna di B

$$C_{i,j} = \sum_k A_{i,k} \times B_{k,j}$$

Ad esempio, se A è una matrice 2×3 e B è una matrice 3×2, il prodotto $C = AB$ sarà una matrice 2×2.

Somma di matrici e operazioni element-wise

- **Somma di matrici**: si possono sommare due matrici se hanno le stesse dimensioni, sommando gli elementi corrispondenti:

$$C_{i,j} = A_{i,j} + B_{i,j}$$

- **Somma di uno scalare a una matrice**: si aggiunge lo scalare a ogni elemento della matrice.
- **Broadcasting (somma di matrice e vettore)**: quando si somma un vettore ad una matrice, il vettore viene "copiato" lungo le righe (o colonne) e aggiunto elemento per elemento. Ad esempio: se A è una matrice di dimensione 3×3 e b è un vettore di 3 elementi, l'operazione $C = A + b$ aggiunge ciascun elemento di b a ogni riga di A.

Matrice identità e inversione

Matrice identità È una matrice quadrata in cui tutti gli elementi della diagonale principale sono 1 e tutti gli altri sono 0.

- **Proprietà**: moltiplicare una matrice per la matrice identità non cambia la matrice: $I_n \cdot x = x$ per ogni vettore x.
- ***Esempio pratico***: la matrice identità 3×3 è

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Inversione di una matrice Una matrice A è invertibile se esiste un'altra matrice A^{-1} tale che:

$$A^{-1}A = I$$

La matrice deve essere **quadrata** e avere un determinante diverso da zero (vedasi più avanti). Ad esempio, se $A = \begin{bmatrix} 4 & 7 \\ 2 & 6 \end{bmatrix}$, esiste A^{-1} tale che $A^{-1}A = I_2$.

Norme e misura delle dimensioni

Norma di un vettore La norma di un vettore misura la sua "lunghezza" o "grandezza".

- **Proprietà fondamentali**:
  - $||x|| = 0$ se e solo se x è il vettore zero.
  - La norma soddisfa la **disuguaglianza triangolare**: $||x + y|| \leq ||x|| + ||y||$.
  - Per ogni scalare α, $||\alpha x|| = |\alpha|||x||$.
- **Tipi di norme**:
  - **L2 (Euclidea)**:
    $$||x||_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$
    Esempio: lunghezza di una freccia nello spazio.
  - **L1**:
    $$||x||_1 = |x_1| + |x_2| + \cdots + |x_n|$$
    Esempio: la somma delle distanze lungo ogni asse.
  - **Max-norm**:
    $$||x||_\infty = \max\{|x_1|, |x_2|, \ldots, |x_n|\}$$
    Esempio: il massimo spostamento lungo uno degli assi.

Norma di Frobenius (per le matrici) Misura la "dimensione" di una matrice calcolando la radice quadrata della somma dei quadrati di tutti i suoi elementi:

$$||A||_F = \sqrt{\sum_{i,j} A_{i,j}^2}$$

Esempio pratico: se abbiamo una piccola immagine rappresentata da una matrice, la norma di Frobenius ci dà una misura complessiva dell'"intensità" dei pixel.

Operazioni avanzate: ortogonalità e traccia

Ortogonalità Due vettori sono **ortogonali** se il loro prodotto scalare (*dot product*) è zero, cioè se sono perpendicolari.

Se, inoltre, hanno norma pari a 1, si dicono **ortonormali**.

Esempio pratico: in una mappa, la direzione Nord e la direzione Est sono ortogonali.

Traccia (trace) La traccia è la somma degli elementi lungo la diagonale principale di una matrice.

- **Proprietà**: la traccia è invariante rispetto a certe permutazioni (ad esempio, $Tr(ABC) = Tr(BCA)$).

Esempio pratico: se abbiamo una matrice $A = \begin{bmatrix} 5 & 2 \\ 1 & 3 \end{bmatrix}$, la traccia è $5 + 3 = 8$.

Spazio vettoriale e concetti di base

Spazio vettoriale Uno spazio vettoriale è un insieme di vettori in cui si possono eseguire due operazioni:

1. **Addizione vettoriale**: sommare due vettori.
2. **Moltiplicazione per uno scalare**: moltiplicare un vettore per un numero.

Devono valere **proprietà** come l'associatività, l'esistenza di un vettore zero, l'esistenza dell'inverso (il "negativo") e la sua distributività.

Esempio pratico: lo spazio vettoriale $\mathbb{R}^2$ è formato da tutte le coppie di numeri (x, y) che possono rappresentare punti sul piano.

Determinante e invertibilità delle matrici

Determinante Il determinante è un numero associato a una matrice quadrata che fornisce informazioni importanti, come l'invertibiltà della matrice. Per una matrice 2×2:

$$\det(A) = ad - bc \iff A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Per matrici più grandi si usa l'espansione di Laplace[1], che richiede di calcolare "sottomatrici" o "cofattori".

Se $\det(A) = 0$, la matrice è **singolare** (non invertibile).

Invertibilità Una matrice A è invertibile se esiste un'altra matrice A^{-1} tale che:

$$A^{-1}A = I$$

Le matrici non quadrate, o quelle con righe/colonne dipendenti (che portano a $\det(A) = 0$), non possono essere invertite nel senso classico.

[1] https://it.wikipedia.org/wiki/Teorema_di_Laplace

Sistemi di equazioni lineari Un sistema di equazioni lineari ha la forma:

$$Ax = b$$

dove A è una matrice dei coefficienti, x è il vettore delle incognite e b è il vettore dei termini noti.

Un sistema può avere:

- **Una soluzione unica**: se il sistema è determinato.
- **Infinite soluzioni**: se il sistema è indeterminato.
- **Nessuna soluzione**: se il sistema è impossibile.

Esempio pratico: nell'addestramento di una rete neurale, si cerca di minimizzare un errore che spesso si riduce a risolvere un sistema di equazioni.

Rango di una matrice, dipendenza lineare e span

Rango Il rango di una matrice è il numero di righe (o colonne) linearmente indipendenti. Aiuta a capire se un sistema di equazioni ha soluzioni (e, nel caso, quante ne ha).

Esempio pratico: se il rango di A è inferiore al numero di variabili, il sistema $Ax = b$ potrebbe avere infinite soluzioni.

Dipendenza lineare e span Un insieme di vettori è **linearmente indipendente** se nessuno di essi può essere ottenuto come combinazione lineare degli altri.

Lo spazio generato si chiama **span**, ed è l'insieme di tutti i vettori che si possono ottenere come combinazione lineare di un certo insieme di vettori.

Per il sistema $Ax = b$, la soluzione esiste se b appartiene allo span dalle colonne di A.

Matrici e vettori speciali

- **Vettore unitario**: è un vettore che ha norma 1. Si usa per indicare una direzione senza considerare la lunghezza.
- **Matrice simmetrica**: una matrice è simmetrica se $A = A^T$; in altre parole, è speculare rispetto alla diagonale.
- **Matrice ortogonale**: una matrice quadrata A è ortogonale se i suoi vettori riga (o colonna) sono ortonormali, cioè $A^T A = I$.

  Esempio pratico: le trasformazioni di rotazione nel piano sono rappresentate da matrici ortogonali.
- **Matrice diagonale**: è una matrice in cui gli elementi non nulli sono solo sulla diagonale principale.
  - **Efficienza**: moliplicare un vettore per una matrice diagonale equivale a scalare ogni elemento del vettore per il corrispondente elemento diagonale.
  - **Inversione**: la matrice diagonale è invertibile se tutti i valori sulla diagonale sono non nulli, e l'inversa si ottiene semplicemente prendendo il reciproco di ciascun elemento diagonale.

2.2 Decomposizione degli autovalori (Eigendecomposition)

La decomposizione degli autovalori è un metodo dell'algebra lineare che ci permette di "smontare" una matrice quadrata nei suoi componenti fondamentali: gli **autovettori** e gli **autovalori**. Questo processo rivela proprietà della matrice che non sono evidenti osservando semplicemente i suoi elementi.

Autovettori e autovalori Un **autovettore** è un vettore nullo v tale che, quando viene moltiplicato per la matrice A, il suo solo effetto è quello di scalarlo. In altre parole, la direzione di v non cambia, solo la sua lunghezza viene modificata. La relazione matematica è:

$$Av = \lambda v$$

dove λ è l'autovalore corrispondente a v.
Un **autovalore** è il fattore di scala per l'autovettore v. Se λ è grande, il vettore viene allungato; se λ è piccolo (o negativo), il vettore viene compresso o invertito di direzione

Esempio pratico Immaginiamo di avere una gomma da masticare che, se schiacciata in una certa direzione, cambia solo la sua lunghezza lungo quella direzione ma non la direzione stessa. Gli autovettori sono come le "direzioni preferenziali" della gomma, e gli autovalori indica di quanto la gomma venga compressa o allungata in quelle direzioni.

Decomposizione della matrice Per una matrice **diagonalizzabile**, possiamo scrivere:

$$A = V \,\mathrm{diag}(\lambda)\, V^{-1}$$

dove:

- V: una matrice le cui colonne sono gli autovettori di A.
- $\mathrm{diag}(\lambda)$: una matrice diagonale che ha gli autovalori lungo la diagonale.
- V^{-1}: l'inversa della matrice V.

Nel caso particolare di **matrici simmetriche** (cioè $A = A^T$), la decomposizione assume una forma molto bella e ordinata:

$$A = Q\Lambda Q^T$$

dove:

- Q: matrice ortogonale, cioè le sue colonne (autovettori) sono ortonormali (hanno lunghezza 1 e sono perpendicolari tra loro).
- Λ: matrice diagonale con gli autovalori.

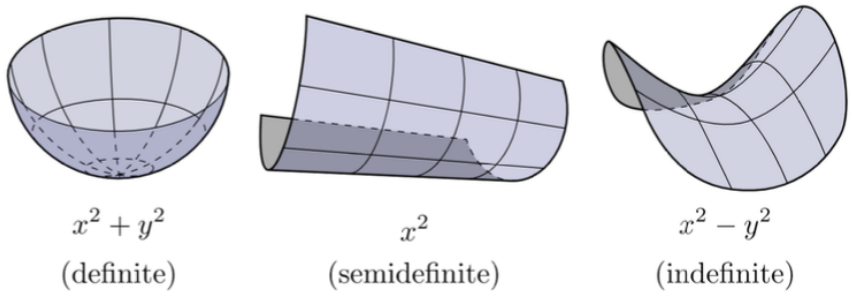
Proprietà e utilità

- **Unicità**: la decomposizione non è sempre unica. Se due (o più) autovettori condividono lo stesso autovalore, possiamo scegliere diversi insiemi di autovettori ortogonali all'interno dello stesso spazio. Per convenzione, spesso si ordinano gli autovalori in ordine decrescente per rendere la decomposizione più "unica" se tutti gli autovalori sono distinti.

- **Singolarità**: se almeno uno degli autovalori è zero, la matrice è singolare (non invertibile).
- **Ottimizzazione**: consideriamo una funzione quadratica:

$$f(x) = x^T Ax \quad \text{con} \quad \|x\|_2 = 1$$

  Se x è un autovettore, allora $f(x)$ assume il valore dell'autovalore corrispondente. Il valore massimo e minimo di $f(x)$ (sotto il vincolo $\|x\| = 1$) corrispondono rispettivamente al massimo e al minimo autovalore di A.
- **Matrici positive**:
  - **Matrice positiva definita**: se tutti gli autovalori sono positivi, per ogni x diverso da zero si ha $x^T Ax > 0$. Queste matrici sono invertibili e il loro inverso è ancora positivo definito.
  - **Matrice positiva semidefinita**: se tutti gli autovalori sono positivi o nulli (ossia, non negativi), si ha $x^T Ax \geq 0$ per ogni x.
- **Matrice indefinita**: se alcuni autovalori sono positivi e altri negativi, la matrice è detta indefinita.



Esempio pratico In molti algoritmi di machine learning e deep learning, la decomposizione agli autovalori viene usata per analizzare e ridurre la dimensionalità dei dati, oppure per ottimizzare funzioni quadratiche (ad esempio, nella tecnica della PCA).

2.3 Decomposizione in valori singolari (Singular Value Decomposition, SVD)

La **Singular Value Decomposition (SVD)** è un metodo potente e generale pe r"smontare" la matrice in componenti più semplici, ed è particolarmente utile quando la matrice non è quadrata (cioè, il numero di righe non è uguale al numero di colonne).

SVD Per una matrice non quadrata A di dimensioni $m \times n$, la SVD permette di descriverla come:

$$A = UDV^T$$

dove:

- U: è una matrice ortogonale di dimensioni $m \times m$. Le sue colonne sono chiamate **left-singular vectors** (vettori singolari sinistri).
- D: è una matrice diagonale (con elementi non nulli solo sulla diagonale) di dimensioni $m \times n$. Gli elementi sulla diagonale, chiamati **singular values**, sono numeri che rappresentano l'importanza o la "forza" di certe direzioni nella matrice. Anche se non necessariamente quadrata, i suoi valori sono ben definiti.
- V: è una matrice ortogonale di dimensioni $n \times n$. Le sue colonne sono i **right-singular vectors**.

Interpretazione attraverso l'eigendecomposition La SVD si collega all'eigendecomposition di due matrici:

- Le colonne di U sono gli autovettori della matrice AA^T (di dimensione $m \times m$).
- Le colonne di V sono gli autovettori della matrice A^TA (di dimensione $(n \times n)$.
- I valori singolari di A sono le radici quadrate degli autovalori di A^TA (ed equivalenti a quelli di AA^T se non sono nulli).

Pseudoinversa e soluzioni di sistemi lineari In molti casi, soprattutto in deep learning e altre applicazioni, potremmo dover risolvere equazioni del tipo:

$$Ax = b$$

Quando la matrice A **non è quadrata** o non è invertibile (ad esempio, se ha righe o colonne ridondanti), non è possibile usare l'inversione tradizionale. In questi casi si usa la **pseudoinversa di Moore-Penrose** A^+, definita attraverso la SVD:

$$A^+ = VD^+U^T$$

dove D^+ è ottenuta prendendo il reciproco dei valori singolari non nulli nella matrice D e trasponendola.

La pseudoinversa ci permette di:

- **Ottenere una soluzione unica**: se esiste una sola soluzione, A^+ resitituisce esattamente quella soluzione.
- **Se ci sono molte soluzioni**: nel caso in cui A abbia più colonne che righe (sistema indeterminato), la pseudoinversa dà la soluzione che ha la **minima norma euclidea** (cioè, la soluzione più "piccola" in termini di lunghezza del vettore).
- **Se non esiste una soluzione esatta**: se A ha più righe che colonne (sistema sovradeterminato), la pseudoinversa fornisce la soluzione che minimizza l'errore $\|Ax - b\|_2$.

Esempio pratico Immagina di avere un dataset di immagini. Ogni immagine può essere rappresentata come una matrice (o un tensore) di pixel. La SVD può essere usata per:

- **Comprimere l'immagine**: separando le informazioni più importanti (i valori singolari più grandi) da quelle meno importanti. Questo è utile per ridurre la dimensione dei dati mantenendo la qualità dell'immagine.
- **Riduzione del rumore**: filtrando le componenti che hanno valori singolari molto bassi, che spesso corrispondono a rumore o dettagli irrilevanti.

In deep learning, queste tecniche sono utili anche per analizzare e comprendere meglio le strutture dei dati e per risolvere problemi di ottimizzazione.

2.4 Principal Components Analysis (PCA)

La PCA è un algoritmo di machine learning che permette di **ridurre la dimensionalità dei dati**, cioè di comprimere un insieme di dati con tante caratteristiche (o dimensioni) in uno spazio di dimensioni inferiori, mantenendo però le informazioni più importanti.

Obiettivo della PCA Immaginiamo di avere m punti nello spazio $\mathbb{R}^n$ (cioè, ogni punto ha n caratteristiche). Con la PCA, vogliamo trasformare questi punti in un nuovo spazio a dimensione l (con $l < n$) in modo da **risparmiare spazio e risorse** (perché conserviamo meno informazioni) e contemporaneamente **mantenere l'essenza dei dati** (le nuove coordinate, chiamate "componenti principali", catturano la maggior parte della variazione e dell'informazione importante presente nei dati).

Funzioni di codifica e decodifica La PCA definisce due funzioni:

- **Codifica**: $f(x) = c$, trasforma il dato origniale $x \in \mathbb{R}^n$ in un codice $c \in \mathbb{R}^l$.
- **Decodifica**: $g(c) \approx x$, cerca di ricostruire il dato origniale a partire dal codice c.

Un modo semplice di decodificare è usare una matrice D (di dimensioni $n \times l$):

$$g(c) = Dc$$

Per minimizzare l'errore, si cerca di ottenere x il più vicino possibile a $g(c)$ misurando la distanza (ad esempio, con la norma L2).

Ottimizzazione e soluzione Si definisce l'errore di ricostruzione come la differenza tra il dato originale x e la sua ricostruzione Dc. Usando il calcolo vettoriale, si arriva a:

$$c^* = D^T x \quad \text{quindi} \quad f(x) = D^T x$$

e la ricostruzione completa diventa:

$$r(x) = g(f(x)) = DD^T x$$

L'obiettivo è scegliere la matrice D in modo che l'errore complessivo (misurato ad esempio con la norma di Frobenius) sia minimo:

$$D^* = \arg\min_D \left\| X - XDD^T \right\|_F$$

dove X è la matrice che contiene tutti i dati.

Dopo vari passaggi matematici, si scopre che la soluzione ottimale è data dagli **autovettori** della matrice di covarianza $X^T X$ che corrispondono agli **autovalori** più grandi. Questi autovettori diventano le colonne della matrice D.

Significato nella pratica

- **Prima componente principale**: è la direzione (linea attraverso l'origine) lungo la quale i dati variano di più. Proiettare i dati su questa linea massimizza la varianza (cioè, cattura quanta informazione c'è).
- **Riduzione dimensionale**: dopo aver calcolato tutte le componenti (cioè, tutti gli autovettori ordinati per varianza), scegliamo di mantenere solo le prime l componenti. Questo significa che riduciamo la dimensione dei dati da n a l, sperando di perdere il meno possibile dell'informazione essenziale.

Esempio pratico Immaginiamo di avere un dataset di immagini, ognuna rappresentata da migliaia di pixel (alte dimensioni). Utilizzando la PCA, proiettiamo le immagini in uno spazio a dimensione ridotta (ad esempio, 50 componenti invece di migliaia di pixel). Questo permette di visualizzare e analizzare i dati in modo più semplice, riducendo il rumore e migliorando la velocità degli algoritmi di machine learning.

3 Machine Learning basics

Quasi tutti gli algoritmi di deep learning possono essere descritti come ingredienti particolari di una ricetta piuttosto semplice:

- La combinazione di specifici **dataset**
- Una **funzione di costo**
- Una procedura di **ottimizzazione**
- Un modello (ANN)

3.1 Focus sui dati

Quando lavoriamo con i dati in machine learning, li rappresentiamo tipicamente come **vettori** e li organizziamo in un **dataset**.

Rappresentazione dei dati come vettori

- **Vettore**: un vettore è una lista ordinata di numeri. Ad esempio, se consideriamo le caratteristiche di una casa (metratura, numero di stanze, età), possiamo rappresentarla in un vettore:

  $$x = [\text{metratura, stanze, età}]$$

  Se il vettore ha d componenti, scriviamo $x \in \mathbb{R}^d$.

- **Dataset**: se raccogliamo m osservazioni (ad esempio, m case diverse), possiamo organizzare i dati in una matrice X dove:

  $$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

  Ogni riga x_i è un vettore che rappresenta un'osservazione.

Mappatura da input a output: la funzione del modello Usiamo una funzione parametrica $f(\Theta)$ che trasforma l'input x in un output y. Questa funzione dipende da parametri Θ che il modello deve imparare dai dati:

$$f(\Theta) : x \longrightarrow y$$

Esempio pratico: se stiamo cercando di predire il prezzo di una casa, il nostro modello imparerà a trasformare le caratteristiche della casa (input) in un prezzo (output).

Apprendimento supervisionato Nell'apprendimento supervisionato, disponiamo di un dataset formato da coppie di dati:

$$D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\}$$

- **Input data space** X: tutti i possibili vettori di X.
- **Output (label) space** Y: i possibili valori che l'output può assumere.

  Se Y è **finito**, si tratta di un problema di **classificazione**. Se Y è continuo, si parla di **regressione**.

Funzione di perdita (loss function) La funzione di perdita $L(\hat{y}, y)$ misura l'errore tra il valore predetto $\hat{y}$ e il reale valore y. Il suo valore può variare in base al problema:

- **Classificazione**: spesso si usa **0/1 loss** (errore 0 se la previsione è corretta, 1 se sbagliata).
- **Regressione**: si usa il **Mean Square Error (MSE)**:

$$L(\hat{y}, y) = (\hat{y} - y)^2$$

DNN: evoluzione del neurone singolo I modelli che utilizziamo sono **deep neural networks (DNN)**, che sono composte da molti strati (o neuroni). Queste reti sono evoluzioni del semplice modello a neurone singolo e permettono di apprendere rappresentazioni sempre più complesse e astratte dai dati.

Divisione del dataset: training, test e validazione Per valutare il modello e assicurarci che impari in modo efficace, dividiamo il dataset in:

- **Training set**: i dati per addestrare il modello.
- **Test set**: i dati usati per valutare la performance del modello su dati non visti durante l'addestramento.
- **Validation set**: talvolta si usa una parte dei dati di training per ottimizzare i parametri del modello.

Tipi di dati in input I modelli di deep learning possono lavorare con vari tipi di dati, ad esempio:

- **Dati visivi**: immagini.
- **Serie temporali**: dati che variano nel tempo (come i dati meteo o i prezzi delle azioni).
- **Dati testuali**: frasi, articoli, ecc.
- **Dati multivariabili**: dati con molte caratteristiche, come le informazioni sui clienti.

3.1.1 Dati visivi

Quando parliamo di dati visivi, ci riferiamo a informazioni che provengono da immagini e video.

Tipi di dati visivi

- **Immagini**
  - **RGB**: sono le immagini più comuni, composte da 3 canali (rosso, verde, blu). Ad esempio, una foto digitale di dimensioni 256x256 pixel viene rappresentata come una matrice con dimensioni 256x256x3.
  - **Immagini con più canali**: in alcuni casi, possono esserci immagini con un numero maggiore di canali (ad esempio, aggiungendo un canale per la profondità o altre informazioni).
- **Video**: un video è essenzialmente una sequenza di immagini (fotogrammi) catturate ad una certa frequenza (frame per second, FPS). Ad esempio, un video full HD (1920x1080) a 30 fps è una successione di immagini ad alta risoluzione.

Gestione dei dati visivi e limitazioni computazionali Le immagini ad alta risoluzione hanno molte informazioni (molti pixel), il che può richiedere molta memoria e potenza di calcolo. Il **downsampling** è il processo di riduzione della risoluzione. Non si tratta solo di ridurre le dimensioni, ma anche di **resampling** dei pixel, ossia creare una nuova versione dell'immagine con meno pixel.

Oltre al downsampling, esistono tecniche come la **parallelizzazione dei dati e dei modelli**, che permettono ddi elaborare immagini grandi suddividendo il lavoro in parti più piccole.

Rappresentare le immagini per il machine learning Una tecnica comune consiste nell'"appiattire" (**flatten**) le immagini, trasformando la matrice in un vettore monodimensionale. Il problema che sorge è che, appiattendo l'immagine, si possono perdere informazioni importanti sui rapporti spaziali tra i pixel, ovvero come i pixel vicini sono collegati tra loro.

L'**approccio tradizionale** consiste nella PCA per ridurre la dimensionalità, ma questa tecnica spesso non riesce a preservare bene le **relazioni locali** tra i pixel. Per far fronte al problema, sono state individuate due tecniche alternative:

- **Receptive fields**: invece di appiattire l'immagine, si possono considerare piccole aree (ad esempio, quadrati o rettangoli) che vengono processate insieme. Questo metodo consente al modello di "vedere" le relazioni tra i pixel vicini e di ridurre il numero di parametri (pesi) necessari.
- **Unità localmente connesse**: tecniche simili alle reti neurali convoluzionali (CNN), ma senza la condivisione dei pesi. Queste tecniche permettono una rappresentazione più precisa delle relazioni locali, migliorando il riconoscimento di oggetti e segmentazione nelle immagini, anche se aumentano la complessità computazionale.

Per immagini che includono ulteriori canali, come le immagini RGB con un canale di profondità (RGB+D), è importante considerare anche le correlazioni locali tra i canali.

Dati video e il fattore temporale Un video è composto da numerosi fotogrammi ripresi in sequenza.

Oltre ai filtri spaziali (che analizzano la disposizione dei pixel in un'immagine), nei video si usano **filtri temporali** per catturare i cambiamenti e le relazioni tra i fotogrammi nel tempo. Ad esempio: in un video di una persona che cammina, i filtri temporali aiutano il modello a capire il movimento e la dinamica dell'azione.

L'aggiunta della dimensione temporale aumenta il numero di pesi e la complessità del modello, richiedendo più risorse di calcolo.

3.1.2 Serie temporali

I dati di serie temporali sono dati che variano nel tempo e vanno letti come sequenze ordinate. Ciò significa che l'ordine dei dati è fondamentale perché i valori attuali dipendono dai valori precedenti.

Tipi di serie temporali

- **Audio**: un segnale audio è una serie temporale in cui ogni campione rappresenta l'intensità del suono in un istante. Un esempio è la registrazione di una canzone, in cui il suono varia in base al tempo.
- **Dati da sensori**: possono essere fondamentalmente di due tipi:
  - **Dati inerziali**: provenienti da accelerometri e giroscopi, che misurano il movimento e l'orientamento.
  - **ECG/EEG**: i segnali elettrocardiografici (ECG) e elettroencefalografici (EEG) monitorano, rispettivamentem l'attività cardiaca e cerebrale.

Sfide nell'elaborazione delle serie temporali

- **Dipendenza temporale**: a differenza delle immagini, dove l'informazione si trova nelle relazioni spaziali (tra pixel vicini), nei dati temporali è fondamentale tenere conto dell'ordine e della dipendenza tra dati successivi. Ad esempio: in una registrazione audio, il suono di una nota dipende da quello precedente. Perdere questa sequenza può alterare la comprensione del brano.
- **Dimensione e complessità**: utilizzare strati completamente connessi (fully connected layers) per ogni punto temporale può portare a modelli troppo grandi e lenti, soprattutto se la sequenza è molto lunga.
- **Estrazione di caratteristiche**: è importante decidere se lavorare sul segnale grezzo o estrarre delle **feature** che riassumono aspetti rilevanti, come frequenza e ampiezza (audio) e modelli di movimento, orientamento, variazioni del battito o dei segnali cerebrali (dati da sensori).

Approcci per gestire i dati di serie temporali Invece di utilizzare il segnale completo, si possono calcolare delle **feature** che catturano le informazioni essenziali. In particolare:

- *Audio*: si può utilizzare una trasformata di Fuorier[2] per ottenere lo spettro delle frequenze.
- *Sensori*: si possono estrarre statistiche come media, varianza o pattern di movimento.

Un altro approccio consiste nell'utilizzo delle **reti neurali ricorrenti (RNN)**. Queste reti sono progettate appositamente per dati sequenziali perché hanno una **memoria interna** che permette loro di "ricordare" informazioni passate. Ad esempio: nel riconoscimento vocale, le RNN possono utilizzare il contesto delle parole precedenti per interpretare correttamente la parola attuale.

Esistono anche modelli che combinano tecniche di estrazione di feature con strati completamente connessi o convoluzionali per gestire sequenze particolarmente lunghe o complesse.

3.1.3 Dati testuali

I dati testuali sono informazioni basate su testo, come documenti scritti o trascrizioni di discorsi. Questo dati sono composti da sequenze di simboli discreti (parole o caratteri) e l'ordine in cui appaiono è fondamentale per il loro significato.

Approcci di rappresentazione del testo

- **Bag-of-Words (BOW)**: questo metodo rappresenta un documento come un vettore di frequenze, contando quante volte appare ogni parola. Il suo vantaggio consiste nell'essere semplice da calcolare, ma di contro ignora l'ordine delle parole, quindi perde informazioni sul contesto e sulla struttura della frase. Ad esempio: immaginiamo di avere due frasi:
  - *Il gatto mangia il topo.*
  - *Il topo mangia il gatto.*

  Con un modello BOW, entrambe potrebbero avere lo stesso vettore, non riconoscendo quindi la differenza nel significato.
- **Continuous Bag-of-Words (CBOW)**: in questo approccio, il modello impara a predire una parola target basandosi sul contesto (le parole vicine). Così facendo, riesce a catturare alcune relazioni semantiche tra le parole, pur non mantenendo l'ordine esatto. Ad esempio, se il contesto è *"Il . . . è sul tetto"*, il modello può dedurre che la parola più probabile sia *"gatto"*.
- **Word Embeddings (es. Word2Vec)**: questi metodi trasformano le parole in vettori densi a bassa dimensione, dove parole con significati simili hanno rappresentazioni simili. Il vantaggio è che non solo catturano le relazioni semantiche, ma grazie a tecniche come le finestre mobili (sliding windows), riescono anche a preservare qualche informazione sull'ordine e sul contesto. Ad esempio: in uno spazio di embedding, le parole "re" e "regina" saranno vicine perché hanno significati correlati, mentre "re" e "casa" saranno più distanti.

[2] https://it.wikipedia.org/wiki/Trasformata_di_Fourier

Considerazioni pratiche Dal momento che l'ordine delle parole è importante per il significato, è necessario scegliere modelli che lo considerino (**sequenzialità**). Ad esempio, modelli basati su reti neurali ricorrenti (RNN) o trasformatori (Transformer) sono progettati per processare sequenze e mantenere il contesto nel tempo.

Anche la **scelta del modello** è importante: ad esempio, un modello **BOW** può funzionare bene per compiti semplici, come la classificazione di documenti in categorie (ad es., spam vs non spam), dove l'ordine non è fondamentale. Mentre **CBOW** o **Word Embeddings** sono più adatti quando è importante cogliere il significato e le relazioni tra le parole, come nell'analisi del sentiment o nella traduzione automatica.

Per quanto riguarda le **risorse computazionali**, I modelli più sofisticati, come Word2Vec, richiedono una quantità maggiore di dati e potenza computazionale, ma offrono una rappresentazione del testo molto più ricca e utile per compiti complessi.

3.2 Funzione di costo

La funzione di costo (o **loss/error function** è una formula matematica che misura quanto le previsioni di un modello si discostano dai valori reali. In altre parole, ci dice "quanto sbagliamo" quando facciamo una previsione.

Ottimizzazione nel deep learning In deep learning, vogliamo trovare i parametri (indichiamoli con θ) che minimizzano l'errore. Questo problema si esprime matematicamente come:

$$\theta^* = \arg\min_{\theta} f(x, \theta)$$

Qui, $f(x, \theta)$ è la funzione di costo che dipende dai dati di input x e dai parametri θ.

Spesso ci concentriamo sul **minimizzare** $f(x, \theta)$. Se invece volessimo massimizzare una funzione, possiamo scrivere il suo negativo, $-f(x, \theta)$.

Esempio con funzioni lineari Consideriamo un semplice modello lineare per fare previsioni. Supponiamo di voler prevedere il prezzo di una casa in base alla sua dimensione. La nostra funzione lineare può essere scritta come:

$$f(x, \theta) = \theta_0 + \theta_1 x$$

Dove:

- x rappresenta la dimensione della casa (**variabile indipendente**)
- θ_0 è l'**intercetta** (il prezzo base quando $x = 0$)
- θ_1 è il coefficiente che determina quanto aumenta il prezzo per ogni unità di aumento della dimensione

Possiamo scrivere questo modello in forma vettoriale:

$$f(x, \theta) = \theta^T x$$

dove x e θ sono vettori.

Per capire quanto il nostro modello si discosta dalla realtà, definiamo una **funzione di costo**, ad esempio il **Mean Squared Error (MSE)**:

$$L(\hat{y}, y) = (\hat{y} - y)^2$$

Dove:

- $\hat{y} = f(x, \theta)$ è il prezzo predetto dal modello
- y è il prezzo reale

L'obiettivo è quello di trovare i parametri θ che minimizzano la somma degli errori quadratici su tutte le osservazioni.

Esempio pratico: immaginiamo di avere un dataset di case. Ogni casa è rappresentata da un vettore x che contiene la dimensione della casa. Abbiamo inoltre il prezzo reale di ciascuna casa, y.

Il nostro compito è quello di:

Predire il prezzo utilizzando il modello lineare:

$$\hat{y} = \theta_0 + \theta_1 x$$

Calcolare l'errore per ogni casa usando il MSE:

$$\text{Errore} = (\hat{y} - y)^2$$

Ottimizzare il modello: troviamo i parametri θ_0 e θ_1 che minimizzano la somma di questi errori. In pratica, il modello "impara" aggiustando θ per ridurre l'errore complessivo.

3.3 Ottimizzazione

L'ottimizzazione consiste nel cercare i parametri θ che permettono al modello di compiere al meglio un compito, minimizzando (o massimizzando) una funzione obiettivo (la **funzione di costo**).

Problema generale di ottimizzazione Matematicamente, cerchiamo di risolvere:

$$\theta^* = \arg\min_{\theta} f(x, \theta)$$

dove $f(x, \theta)$ è la funzione di costo che dipende dai dati x e dai parametri θ.

3.3.1 Ottimizzazione convessa

La **convessità** è una proprietà matematica molto utile in ottimizzazione.

- **Insieme convesso**: un insieme C è convesso se, per ogni coppia di punti x e y in C, ogni punto sulla linea che li connette (cioè $tx + (1-t)y$ per $t \in [0, 1]$) appartiene ancora a C.
- **Funzione convessa**: una funzione $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ è convessa se per ogni x e y nel suo dominio e per ogni $t \in [0, 1]$:

  $$f(tx + (1-t)y) \leq tf(x) + (1-t)f(y)$$

  Questo significa che la linea retta che connette $f(x)$ e $f(y)$ si trova sopra (o sulla) la curva di f
- **Proprietà chiave**: in un problema di ottimizzazione convessa, **ogni minimo locale è anche un minimo globale**. Questo rende l'ottimizzazione molto più semplice e sicura: se troviamo un punto in cui la funzione è minima nel suo intorno, possiamo essere certi che non esiste un punto con un valore più basso altrove.

Esempi di funzioni convesse

1. **Funzioni univariate**
   - **Funzione esponenziale**: e^{ax} è convessa per ogni a su $\mathbb{R}$. La sua curvatura è sempre rivolta verso l'alto.
   - **Funzione potenza**: x^a è convessa per $a \geq 1$ oppure per $a \leq 0$ su $\mathbb{R}^+$ (numeri reali non negativi). Ad esempio: x^2 è convessa, mentre funzioni con a minore di 1 (es. $\sqrt{x}$) potrebbero non esserlo su tutto l'intervallo.
   - **Funzione logaritmica**: $\log x$ è **concava** su $\mathbb{R}^+$, il che è l'opposto della convessità.
2. **Funzione affine**: una funzione del tipo $a^T x + b$ è sia convessa che concava. Ad esempio, una retta come $y = 3x + 2$ è affine e non presenta curvatura.
3. **Funzione quadratica**: una funzione come
   $$\frac{1}{2}x^T Q x + b^T x + c$$
   è convessa se la matrice Q è **positiva semidefinita**. Ad esempio, $f(x) = x^2$ è convessa.
4. **Least squares loss**
5. **Norme**

Gradiente e proprietà delle funzioni convesse Prima di procedere con le tecniche di ottimizzazione, definiamo il **gradiente**. Dato un vettore $x = (x_1, \ldots, x_d)$ e una funzione $f(x)$, il gradiente è il **vettore delle derivate parziali**:

$$\nabla f(x) = \left(\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \ldots, \frac{\partial f(x)}{\partial x_d} \right)$$

Il gradiente indica la **direzione** in cui la funzione cresce più rapidamente. In contesti di ottimizzazione, se $\nabla f(x) = 0$ in un punto x e f è convessa, allora x è un minimo globale. Le funzioni convesse presentano tre **proprietà chiave**:

1. **Caratterizzazione di primo ordine**: se f è differenziabile e convessa, allora per ogni x, y nel dominio di f vale:
   $$f(y) \geq f(x) + \nabla f(x)^T (y - x)$$
   Questa disuguaglianza significa che la retta tangente a f in x è una sottostima globale della funzione.
2. **Caratterizzazione di secondo ordine**: se f è due volte differenziabile, f è convessa se e solo se la sua matrice Hessiana (cioè la matrice delle derivate seconde) è positiva semidefinita per ogni x nel dominio:
   $$\nabla^2 f(x) \succeq 0$$
3. **Jensen's inequality**: se f è convessa e X è una variabile casuale con valori nel dominio di f, allora:
   $$f(\mathbb{E}[X]) \leq \mathbb{E}[f(X)]$$

Esempio di ottimizzazione convessa Consideriamo il **problema dei minimi quadrati** in una regressione lineare. Il modello è:

$$f(x, \theta) = \theta_0 + \theta_1 x$$

e la funzione di costo (MSE) è:

$$L(\theta) = \sum_{i=1}^{m} (\theta_0 + \theta_1 x_i - y_i)^2$$

Questa funzione $L(\theta)$ è un esempio di **funzione quadratica** ed è convessa perché la matrice associata (derivata seconda) è positiva semidefinita. In pratica, ciò significa che se applichiamo un algoritmo (come il gradiente discendente), qualsiasi minimo locale che troviamo sarà anche il minimo globale, garantendo che abbiamo raggiunto la migliore soluzione possibile per i nostri dati.

3.3.2 Ottimizzazione non convessa

Nei problemi non convessi, invece, la funzione di costo può avere **più minimi locali**, cioè dei "buchi" o "valli" in cui il modello potrebbe bloccarsi senza raggiungere il minimo globale.

La maggior parte dei modelli di deep learning ha funzioni di costo non convesse, a causa della complessità dei modelli (molti strati e non linearità). Di conseguenza, gli algoritmi di ottimizzazione (come lo stochastic gradient descent) possono non garantire di trovare il minimo globale, ma nella pratica riescono comunque a trovare soluzioni che funzionano molto bene.

Tecniche avanzate come il momentum, l'ADAM e altri adattamenti del gradiente discendente aiutano a superare alcuni dei problemi legati alla non convessità.

Esempio pratico Supponiamo di voler addestrare una rete neurale per riconoscere immagini di gatti e cani.

Definiamo una **funzione di costo** che misura l'errore tra le previsioni della rete (ad esempio, "gatto" o "cane") e le etichette reali.

Venendo all'**ottimizzazione**:

- **Se la funzione fosse convessa**: ogni punto in cui l'errore diminuisce nel suo intorno sarebbe il minimo globale, rendendo l'ottimizzazione più semplice e sicura.
- **Nel caso reale (non convesso)**: la funzione di costo ha molte "valli" e "picchi". Utilizziamo algoritmi come lo **stochastic gradient descent** che, pur non garantendo il minimo globale, riescono a trovare una configurazione dei parametri che riduce molto l'errore.

3.3.3 Gradient descent

Nel deep learning, l'ottimizzazione è il processo attraverso cui cerchiamo i parametri θ che permettono al modello di svolgere al meglio il proprio compito, minimizzando la funzione di costo. Nei problemi di apprendimento supervisionato, lavoriamo con coppie di dati

e etichette (x_i, y_i) e il nostro modello è una funzione parametrica $f(x, \theta)$ che trasforma l'input x in una previsione $\hat{y}$. L'obiettivo è trovare i parametri ottimali θ^* tali che:

$$\theta^* = \arg\min_{\theta} L(f(x, \theta), y)$$

Questa funzione di costo, che misura l'errore tra le previsioni del modello e i dati reali, spesso risulta essere un problema di ottimizzazione non convesso. Anche se la non convessità implica la presenza di molteplici minimi locali, nella pratica utilizziamo algoritmi di ottimizzazione — basati in parte su principi della convessità — per trovare soluzioni utili.

L'intuizione dietro il gradient descent Il gradient descent è un algoritmo iterativo che minimizza una funzione spostandosi ripetutamente nella direzione del **gradiente negativo** (la direzionein cui la funzione decresce più rapidamente). L'idea alla base è molto semplice:

- **Gradiente**: per una funzione $f(\theta)$, il gradiente $\nabla f(\theta)$ è un vettore che contiene le derivate parziali rispetto a ciascun parametro e indica la direzione di massima pendenza in salita.
- **Regola di aggiornamento (update rule)**: per migliorare la soluzione, aggiorniamo i parametri θ secondo la regola:

  $$\theta_{i+1} = \theta_i - \alpha \nabla f(\theta_i)$$

  Dove:

  - θ_i è il valore corrente dei parametri,
  - α è il **learning rate** (un fattore positivo che determina la dimensione del passo)
  - $\nabla f(\theta_i)$ è il gradiente calcolato nel punto θ_i.

Se il gradiente in θ_i è grande, significa che c'è molta pendenza e possiamo fare un passo significativo per ridurre il valore della funzione. Se il gradiente è piccolo (vicino a zero), siamo vicini a un minimo (locale o globale).

Convergenza e scelta del learning rate In **funzioni convesse**, ogni minimo locale è anche un minimo globale. Quindi, seguendo il gradiente negativo, il gradient descent convergerà al punto di minimo migliore. In **funzioni non convesse**, invece, il gradient descent potrebbe convergere a un minimo locale, e il risultato finale può dipendere dai valori iniziali e dalla scelta del learning rate.

È fondamentale scegliere il giusto **learning rate**. Un valore troppo alto può far "saltare" oltre il minimo e persino far divergere l'algoritmo. Un valore troppo basso, invece, rallenta il processo di convergenza.

Per quanto riguarda i **criteri d'arresto**, l'algoritmo può fermarsi quando:

- La variazione della funzione di costo tra iterazioni diventa inferiore a una certa soglia.
- Oppure, dopo un numero massimo prestabilito di iterazioni.
- Anche se, in teoria, il minimo si raggiunge quando $\nabla f(\theta) = 0$, nella pratica monitoriamo il cambiamento nella funzione di costo o il numero di iterazioni.

Varianti del gradient descent A seconda delle dimensioni del dataset, esistono principalmente due varianti:

1. **Batch Gradient Descent**
    - Calcola il gradiente dell'intera funzione di costo usando tutti i dati.
    - **Vantaggio**: offre aggiornamenti precisi
    - **Svantaggio**: è computazionalmente costoso per dataset molto grandi, perché ad ogni interazione si processano tutti gli esempi
2. **Stochastic Gradient Descent (SGD)**
    - Aggiorna i parametri usando un singolo esempio (o un piccolo gruppo, detto mini-batch) per volta.
    - **Vantaggio**: riduce enormemente il carico computazionale per ogni aggiornamento; è molto utile quando il dataset è molto grande.
    - **Svantaggio**: gli aggiornamenti sono più rumorosi (alta varianza) e possono far oscillare la funzione obiettivo. Tuttavia, diminuendo gradualmente il learning rate, l'algoritmo tende a convergere.

***Esempio pratico*: riconoscimento di immagini di gatti e cani** Immagina di addestrare una nrete neurale per distinguere tra immagini di gatti e cani:

- **Dataset**: ogni immagine (input) è associata a un'etichetta (output) che indica "gatto" o "cane".
- **Modello**: la rete neurale $f(x, \theta)$ prende in input l'immagine e restituisce una previsione.
- **Funzione di costo**: potremmo utilizzare una loss come la $0/1$ loss o una funzione di costo più raffinata (ad es. cross-entropy) per misurare l'errore tra la previsione e l'etichetta reale.
- **Ottimizzazione**: utilizziamo gradient descent per aggiornare i parametri:

$$\theta_{i+1} = \theta_i - \alpha \nabla J(\theta_i)$$

    - Se il problema fosse convesso, ogni minimo locale sarebbe globale, garantendo la migliore soluzione.
    - Nel deep learning, la funzione di costo non è convessa; quindi, l'algoritmo potrebbe fermarsi in un minimo locale, ma in pratica questi minimi locali sono spesso sufficienti per ottenere buone prestazioni
- **Scelta della variante**: se il dataset è molto grande (migliaia di immagini), probabilmente si userebbe lo **stochastic gradient descent** o una sua variante (come ADAM) per rendere l'addestramento più efficiente.

Il gradient descent è una tecnica fondamentale nell'ottimizzazione dei modelli di deep learning. Attraverso aggiornamenti iterativi basati sul gradiente, l'algoritmo cerca di ridurre la funzione di costo, migliorando così le prestazioni del modello. Mentre nelle funzioni convesse il metodo garantisce la convergenza al minimo globale, nel caso dei modelli non convessi tipici del deep learning si ottengono soluzioni buone (anche se potrebbero essere minimi locali). Le varianti come il batch gradient descent e lo stochastic gradient descent offrono diversi compromessi in termini di precisione e velocità di calcolo, rendendole strumenti indispensabili nella pratica del machine learning.

3.4 Fondamenti di machine learning

Nel machine learning, l'obiettivo è costruire un modello che, dato un input x, possa prevedere un output y nel modo più accurato possibile.

Dati e funzione del modello Supponiamo di avere n esempi (osservazioni) $(x_i,\ y_i)$ che sono stati raccolti in modo indipendente e identicamente distribuito (i.i.d.) dalla distribuzione $P(x, y)$.

Utilizziamo una funzione parametrica $f(x, \theta)$ che mappa gli input x agli output y. Ad esempio, nel caso di una regressione lineare possiamo definire:

$$f(x, \theta) = \theta_0 + \theta_1 x_1 + \cdots + \theta_d x_d = \theta_x^T$$

dove $\theta = [\theta_0, \theta_1, \ldots, \theta_d]$ sono i parametri (o pesi) del modello.

Rischio e empircal risk **Risk**: il rischio di un modello $f(x, \theta)$ è definito come il valore atteso della funzione di perdita (o costo) che misura l'errore tra la previsione $f(x, \theta)$ e il valore reale y:

$$R(\theta) = E[L(f(x, \theta), y)]$$

L'obiettivo dell'algoritmo di apprendimento è trovare i parametri θ^* che minimizzano questo rischio:

$$\theta^* = \arg\min_{\theta} R(\theta)$$

Empirical risk: poiché la distribuzione $P(x, y)$ è sconosciuta, non possiamo calcolare il rischio esatto.Invece, usiamo il **rischio empirico** che è la media degli errori sul nostro dataset di training:

$$R_{emp}(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(f(x_i, \theta), y_i)$$

Secondo la legge dei grandi numeri, se il nostro training set è rappresentativo della distribuzione $P(x, y)$, il rischio empirico si avvicinerà al vero rischio.

Esempio con funzione lineare e least squares Consideriamo una semplice ipotesi lineare:

$$f(x, \theta) = \theta_0 + \theta_1 x_1 + \cdots + \theta_d x_d = \theta^T x$$

e definiamo perdita usando l'errore quadratico (squared loss):

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} (y_i - f(x_i \theta))^2 = \frac{1}{N} ||y - X\theta||^2$$

Dove X è la matrice dei dati e y il vettore degli output reali.

- **Soluzione in forma chiusa**: per minimizzare $L(\theta)$, poinamo il gradiente uguale a zero:

$$\nabla L(\theta) = -2X^T(y - X\theta) = 0$$

Risolvendo, otteniamo:

$$\theta^* = (X^T X)^{-1} X^T y$$

Qui, $(X^T X)^{-1} X^T$ è la pseudoinversa di Moore-Penrose di X.

Nota: sebbene questa soluzione sia elegante, in pratica è impraticabile per dataset grandi o modelli complessi; per questo motivo si ricorre a metodi di ottimizzazione iterativa (come il gradient descent).

Generalizzazione ed errori

- **Generalizzazione**: il vero obiettivo del machine learning non è solo ottenere un basso errore sul training set, ma far sì che il modello funzioni bene anche sui nuovi dati (test error). La capacità del modello di fare questo si chiama **generalizzazione**.
- **Capacità del modello**: la scelta dello **spazio delle ipotesi** (l'insieme delle funzioni che il modello può apprendere) determina la capacità del modello. Ad esempio, una regressione lineare ha una capacità limitata (solo funzioni lineari), mentre includere polinomi o utilizzare reti neurali aumenta la capacità e il rischio di overfitting.

Esempio pratico Immaginiamo di voler costruire un modello per prevedere di una casa:

- **Input**: caratteristica della casa (dimensione, numero di stanze, età, ecc.) rappresentate come un vettore $x \in \mathbb{R}^d$.
- **Output**: prezzo della casa y.
- **Modello**: una funzione lineare $f(x, \theta) = \theta^T x$.
- **Funzione di perdita**: utilizziamo l'errore quadratico:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \theta^T x_i)^2$$

- **Obiettivo**: troviamo θ^* che minimizza il rischio empirico. Se il modello e i dati sono rappresentativi, questa soluzione minimizzerà anche il vero rischio e il modello generalizzerà bene su nuove case.
- **Sfide**:
  - Se il modello è troppo semplice, potrebbe non catturare tutte le variazioni nel prezzo (underfitting).
  - Se il modello è troppo complesso, potrebbe adattarsi troppo al training set e non funzionare bene su nuovi dati (overfitting).

I fondamenti del machine learning ruotano attorno alla definizione di un modello $f(x, \theta)$, alla minimizzazione di una funzione di costo (rischio) e alla capacità di generalizzare a dati non visti. Attraverso esempi come la regressione lineare e la minimizzazione dell'errore quadratico, comprendiamo come il problema di ottimizzazione (spesso risolto con tecniche iterative) sia al centro dell'addestramento di modelli che, idealmente, dovrebbero avere basso errore sia sul training che sul test set. La scelta della capacità del modello e la gestione dei compromessi tra underfitting e overfitting sono sfide fondamentali in questo campo.

3.4.1 Regressione polinomiale

La **regressione polinomiale** è un'estenzione della regressionelineare che permette di modellare relazioni non lineari tra l'input x e l'output y. Anche se la funzione non è lineare in x, essa è lineare rispetto ai parametri θ. Ad esempio, un modello polinomiale di grado m si scrive:

$$f(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_m x^m$$

Esempio e concetti chiave

- **Modello di terzo ordine**: Immaginiamo di avere dati che seguono una relazione non lineare e scegliamo di usare un polinomio di terzo ordine, cioè:

  $$f(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$$

  Questo modello può catturare curvature e variazioni più complesse rispetto ad un semplice modello lineare.

- **Capacità del modello e complessità**: la complessità di un modello è legata al numero di parametri liberi (i "gradi di libertà"). Un modello complesso ha più parametri e può adattarsi maggiormente ai dati, ma rischia di overfittare. Un modello semplice, invece, potrebbe non essere abbastanza flessibile per catturare la complessità reale dei dati.

Tecniche per trovare il giusto equilibrio Per ottenere un modello che generalizzi bene sui dati nuovi, dobbiamo bilanciare tra underfitting e overfitting. Ecco alcune strategie:

- **Validazione incrociata (cross-validation)**: dividendo il dataset in training e test (o usando k-fold cross validation), possiamo tracciare due curve:
  - **Errore di training**: misura quanto il modello si adatta ai dati di training.
  - **Errore di generalizzazione (test)**: misura quanto il modello performa su dati non visti.

  L'obiettivo è trovare il punto in cui entrambi gli errori sono minimizzati. Questo punto rappresenta l'ottimale capacità del modello.

- **Controllare la complessità del modello**
  - **Restrizione della complessità**: se abbiamo pochi dati, è preferibile usare un modello meno complesso.

  - **Ricerca del modello migliore (model search)**: testare diversi gradi polinmoiali per trovare quello che offre il miglior compromesso tra bias e varianza.

- **Aumento dei dati (data augmentation)**: se possibile, aumentare la quantità di dati reali o sintetici aiuta il modello a generalizzare meglio.

- **Regularization**: tecniche di regolarizzazione (come L1 o L2) aggiungono una penalità alla funzione di costo per evitare che i parametri assumano valori troppo elevati, riducendo il rischio di overfitting.

Riassunto con esempio Immaginiamo di voler prevedere il prezzo di una casa in base a una caratteristica x (ad esempio, la metratura). Se i dati seguono una relazione non lineare, possiamo scegliere un modello polinomiale, ad esempio di terzo grado:

$$f(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$$

- **Se il modello è troppo complesso** (magari scegliendo un polinomio di grado 10), rischiamo di adattarci troppo ai dati di training, includendo anche il rumore. Questo porta ad un modello instabile sui dati nuovi (overfitting).

- **Se il modello è troppo semplice** (ad esempio, solo una retta $f(x) = \theta_0 + \theta_1 x$), non riusciremo a catturare le variazioni reali, causando un errore elevato sia sui dati di training che sui dati di test (underfitting).

Utilizzando la validazione incrociata, osserviamo le curve degli errori e troviamo il grado del polinomio che minimizza entrambi gli errori. Inoltre, potremmo applicare regolarizzazione per evitare che il modello si adatti troppo ai dati di training.

In conclusione: la regressione polinomiale è un potente strumento per modellare relazioni non lineari, ma la chiave è trovare il giusto equilibrio tra complessità e capacità di generalizzazione. Controllare il numero di parametri, utilizzare tecniche di validazione, aumentare i dati e applicare la regolarizzazione sono tutti metodi fondamentali per ottenere un modello efficace che non solo riduce l'errore sul training set, ma funziona bene anche su dati nuovi.

3.4.2 Regolarizzazione

La regularizzazione è una tecnica utilizzata per migliorare la capacità di generalizzazione di un modello, ovvero per ridurre il rischio di overfitting. In pratica, oltre a minimizzare l'errore sui dati di training, introduciamo una penalità sui pesi del modello, incoraggiando l'algoritmo a trovare soluzioni che non dipendano troppo da variazioni specifiche del training set.

Loss function e empirical risk In machine learning, la funzione di costo (o loss function) empirica, per un dataset con N esempi, si esprime come:

$$L(\theta, X, y) = \frac{1}{N} \sum_{i=1}^{N} L(f(x_i, \theta), y_i)$$

Qui $f(x_i, \theta)$ rappresenta la previsione fatta dal modello per l'esempio x_i e $L(f(x_i, \theta), y_i)$ è il costo associato alla differenza tra la previsione e il valore reale y_i.

Introduzione alla regolarizzazione Per controllare la complessità del modello e favorire pesi più piccoli (meno suscettibili di overfitting), modifichiamo il criterio di training aggiungendo un termine di **regolarizzazione**, noto anche come **weight decay**. Ad esempio, nel caso della regressione lineare, l'algoritmo di Ridge Regression aggiunge al costo originale il quadrato della norma dei pesi moltiplicato per un parametro di regolarizzazione λ:

$$\theta^* = \arg\min_{\theta} \left\{ \frac{1}{N} \sum_{i=1}^{N} L(f(x_i, \theta), y_i) + \lambda \sum_{j=1}^{m} \theta_j^2 \right\}$$

- λ: è un valore scelto a priori che controlla l'intensità della penalizzazione. Un λ più alto forza i pesi a rimanere piccoli, mentre un λ più basso permette al modello di adattarsi maggiormente ai dati.
- **Norma** $||\theta||^2$: è la somma dei quadrati dei pesi (in altre parole, la norma L2). Penalizzando pesi grandi, la regolarizzazione evita che il modello "memorizzi" il training set.

Differenza tra empirical loss e regularized empirical loss

- **Empirical loss (senza regolarizzazione)**: minimizzare esclusivamente l'errore sui dati di training può portare a modelli che si adattano troppo ai dettagli specifici del training set, rischiando di performare male su dati nuovi.
- **Regularized empirical loss**: aggiungendo il termine di regolarizzazione, otteniamo:

  $$L_{\text{reg}}(\theta, X, y) = \frac{1}{N} \sum_{i=1}^{N} L(f(x_i, \theta), y_i) + \lambda ||\theta||^2$$

  Questo metodo bilancia tra la riduzione dell'errore sui dati di training ed evita di avere modelli troppo complessi che possono overfittare.

Ricetta generale per i modelli di deep learning Quasi tutti gli algoritmi di deep learning seguono una ricetta semplice che comprende:

- **Dataset**
- **Funzione di costo**: una funzione che misura l'errore e può includere un termine di regolarizzazione (come il weight decay).
- **Procedura di ottimizzazione**: un algoritmo di ottimizzazione (ad esempio gradient descent, mini-batch SGD, Adam, ecc.) con un determinato learning rate.
- **Modello**: la struttura del modello (ad esempio una rete neurale, una CNN, ecc.) che viene addestrato.

Esempio pratico Supponiamo di voler costruire un modello per prevedere il prezzo di una casa tramite regressione lineare. Se il modello è troppo flessibile, rischia di adattarsi troppo ai dati di training (overfitting). Per evitare questo, possiamo utilizzare la regolarizzazione.

- **Modello**:
$$f(x, \theta) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_d x_d$$
- **Empirical loss senza regolarizzazione**:
$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} (y_i - f(x_i, \theta))^2$$
- **Regularized empirical loss (with Ridge regression)**:
$$L_{\text{reg}}(\theta) = \frac{1}{N} \sum_{i=1}^{N} (y_i - f(x_i, \theta))^2 + \lambda \sum_{j=1}^{d} \theta_j^2$$

  Scegliendo un valore appropriato per λ, possiamo controllare la complessità del modello, favorendo soluzioni con pesi più piccoli e, di conseguenza, un modello che generalizza meglio sui dati nuovi.

La regolarizzazione è una parte fondamentale del processo di addestramento in deep learning. Aggiungendo un termine di penalizzazione alla funzione di costo, il modello non solo impara a ridurre l'errore sui dati di training, ma anche a mantenere i pesi entro limiti ragionevoli per prevenire l'overfitting. Questo, insieme al dataset, alla funzione di costo e al metodo di ottimizzazione, compone la "ricetta" di base per costruire modelli di deep learning efficaci.

4 Feed Forward Networks

4.1 Reti neurali artificiali

Le **reti neurali artificiali (ANN)** sono modelli di apprendimento ispirati in parte a come il cervello umano elabora le informazioni. Sebbene il termine "deep learning" oggi vada oltre l'aspetto strettamente neuroscientifico, l'idea di utilizzare numerosi strati di rappresentazioni vettoriali (dette hidden layers, ovvero strati nascosti) rimane centrale.

Struttura di una rete feedforward Una rete feedforward è composta da:

- **Strato di input**: riceve i dati grezzi. Ad esempio, in un problema di riconoscimento delle cifre scritte a mano, ogni pixel dell'immagine può essere un input.
- **Strati nascosti (hidden layers)**: sono strati intermedi in cui il modello apprende rappresentazioni più astratte e complesse dei dati. Questi strati non hanno output "desiderati" specifici, ma sono addestrati tramite l'algoritmo per aiutare la trasformazione dei dati dall'input all'output.
- **Strato di output**: fornisce la risposta finale del modello, ad es. l'etichetta "digit 7" se si tratta di riconoscimento di cifre.

Non-linearità e funzioni di attivazione Una caratteristica fondamentale delle reti neurali è la **non-linearità**.

È necessaria perché se il modello fosse interamente lineare, anche se avessimo più strati, l'output sarebbe comunque una funzione lineare degli input. Le relazioni reali tra input e output sono spesso molto complesse e non lineari.

Per introdurre questa non-linearità, ogni strato nascosto applica una **funzione di attivazione**, come ad esempio la ReLU (Rectified Linear Unit), la sigmoid o la tanh. Queste funzioni determinano quali informazioni (ossia, quali "neuroni") verranno "accese" e trasmesse allo strato successivo.

Ad esempio: immaginiamo un modello per riconoscere se in una foto c'è un gatto oppure un cane.

- L'**input** sono i pixel dell'immagine.
- Il **primo strato nascosto** potrebbe rilevare caratteristiche semplici, come bordi e contorni.
- Un secondo strato nascosto, grazie alla funzione di attivazione, può combinare queste informazioni per riconoscere forme e parti del corpo.
- Lo **strato di output** fornirà una probabilità o una classificazione finale ("gatto" o "cane").

Addestramento e rappresentazione Le reti neurali sono **biologicamente ispirate** dai neuroni del cervello, dove ogni neurone riceve segnali, li elabora e poi trasmette il risultato. Sebbene le ANN non replicano fedelmente il funzionamento del cervello, usano questo concetto di "attivazione" per determinare quali segnali passare oltre.

Poiché il training non fornisce un "obiettivo" per ogni singolo strato nascosto, il modello deve apprendere da solo quali rappresentazioni interne sono utili per trasformare

l'input in output. In questo senso, gli **hidden layers** sono come "scatole nere" che estraggono e combinano le caratteristiche rilevanti dai dati.

Il compito principale della rete è apprendere una funzione $f(x, \theta)$ che mappa l'input x nell'output previsto $\hat{y}$. Grazie agli strati multipli e alle funzioni di attivazione, **questa mappatura diventa non lineare** e molto flessibile, capace di modellare relazioni complesse.

Esempio pratico Immaginiamo di addestrare una rete neurale per riconoscere le cifre scritte a mano (come nel famoso dataset MNIST):

- **Input**: ogni immagine di una cifra è composta, ad esempio, da 28×28 pixel. Questi pixel sono "appiattiti" e inseriti nello strato di input, rappresentando un vettore di 784 valori.
- **Strato nascosto 1**: potrebbe usare la funzione di attivazione ReLU per rilevare bordi e linee nei numeri.
- **Strato nascosto 2**: rielabora queste informazioni per identificare pattern più complessi, ad esempio curve tipiche del numero "3" o la forma angolare del numero "7".
- **Output**: lo strato finale presenta 10 neuroni, ognuno associato a una delle cifre da 0 a 9. La rete assegna la cifra con la probabilità più alta come previsione.

  In questo modo, la rete neurale apprende a riconoscere non solo la semplice presenza di pixel, ma modelli e caratteristiche che compongono il significato della cifra, tutto grazie alla combinazione di strati multipli e funzioni di attivazione non lineari.

Le reti feedforward, o reti neurali artificiali, sono alla base del deep learning. Utilizzano una serie di strati (input, nascosti e output) e introducono non-linearità tramite funzioni di attivazione per apprendere rappresentazioni complesse dai dati. Questa architettura consente al modello di costruire livelli di astrazione, trasformando gli input grezzi in previsioni accurate.

4.1.1 Funzioni di attivazione

Le funzioni di attivazione sono elementi fondamentali nelle reti neurali perché introducono la non-linearità necessaria per modellare relazioni complesse tra input e output. In altre parole, dopo ogni strato (o neurone), la funzione di attivazione trasforma il segnale, mantenendolo all'interno di limiti desiderati e contribuendo alla convergenza dell'algoritmo di addestramento.

Perché serve una funzione di attivazione

1. **Controllo dell'output**: senza una funzione di attivazione, l'output di un neurone potrebbe crescere troppo, specialmente in reti molto profonde con milioni di parametri. Le funzioni di attivazione limitano l'ampiezza dei valori in uscita.
2. **Problema del vanishing gradient**: è importante che la funzione di attivazione non spinga il gradiente verso zero, poiché durante l'addestramento l'aggiornamento dei pesi si basa sul calcolo dei gradienti. Se il gradiente diventa troppo piccolo, l'apprendimento si arresta (vanishing gradient).

3. **Efficienza computazionale**: dal momento che le funzioni di attivazione vengono applicate ad ogni neurone e ad ogni iterazione, devono essere calcolate in modo rapido e a basso costo computazionale.

4. **Differenziabilità**: le reti neurali sono addestrate attraverso algoritmi basati sul gradiente (ad es. backpropagation). Pertanto, la funzione di attivazione deve essere differenziabile (o almeno parzialmente) affinché il calcolo dei gradienti sia possibile.

Tipi di funzione di attivazione Nella pratica, la scelta della funzione di attivazione può dipendere dal tipo di rete e dal problema da risolvere. Di solito, in una rete neurale si utilizza la stessa funzione di attivazione in tutti gli strati nascosti.

Ecco alcune delle funzioni di attivazione più comuni:

1. **Identity**

$$f(x) = x$$

Restituisce semplicemente l'input. Utile in casi particolari dove non si vuole modificare il segnale.

2. **Sigmoid**

$$f(x) = \frac{1}{1 + e^{-x}}$$

Mappa il valore d'ingresso in un intervallo tra 0 e 1, adatto per problemi di classificazione binaria.

Svantaggio: Può causare il problema del gradiente vanescente.

3. **Tanh (ipertangente)**

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Mappa l'input in un intervallo tra -1 e 1, offrendo una media zero, il che può aiutare nel training.

Svantaggio: Anche la tanh può soffrire del gradiente vanescente.

4. **Arctan**

$$f(x) = \arctan(x)$$

Mappa l'input in un intervallo tra $-\frac{\pi}{2}$ e $\frac{\pi}{2}$. Può essere utile in alcuni problemi di regressione.

5. **ReLU (Rectified Linear Unit)**

$$f(x) = \max(0, x)$$

Imposta i valori negativi a zero e lascia invariati quelli positivi. È semplice e computazionalmente efficiente.

Svantaggio: Il problema dei "neuroni morti", dove alcuni neuroni non si attivano mai più.

6. **Leaky ReLU**

$$f(x) = \max(\alpha x, x)$$

dove α è una piccola costante (tipicamente 0.01).

Simile a ReLU, ma consente il passaggio di piccoli valori negativi.

7. **Randomized ReLU**

$$f(x) = \max(0, x + \alpha N(0, 1))$$

dove α è una costante piccola e $N(0, 1)$ è una variabile casuale normale.

Descrizione: Introduce rumore nell'attivazione, migliorando la robustezza con dati rumorosi.

8. **Parametric ReLU (PReLU)**

$$f(x) = \begin{cases} x & \text{se } x \geq 0 \\ \alpha x & \text{se } x < 0 \end{cases}$$

dove α è un parametro appreso durante l'addestramento.

Maggiore flessibilità rispetto a Leaky ReLU.

9. **Binary**

$$f(x) = \begin{cases} 1 & \text{se } x \geq 0 \\ 0 & \text{se } x < 0 \end{cases}$$

Mappa l'input in 0 o 1, utile per classificazione binaria con decisioni nette.

10. **Exponential Linear Unit (ELU)**

$$f(x) = \begin{cases} x & \text{se } x \geq 0 \\ \alpha(e^x - 1) & \text{se } x < 0 \end{cases}$$

Simile alla ReLU, ma per i valori negativi adotta una curva esponenziale.

11. **Softsign**

$$f(x) = \frac{x}{1 + |x|}$$

Output limitato nell'intervallo $(-1, 1)$, utile per gestire input estremi.

12. **Inverse Square Root Unit (ISRU)**

$$f(x) = \frac{x}{\sqrt{1 + \alpha x^2}}$$

Normalizza rispetto alla radice quadrata per ridurre l'impatto di valori elevati.

13. **Inverse Square Root Linear Unit (ISRLU)**

    Simile alla ISRU ma con una transizione più lineare per valori positivi.

    Migliora la stabilità e la continuità dei gradienti.

14. **Square Non-Linearity (SQNL)**

    Funzione mista tra tratti lineari e quadratici, spesso mappa in intervallo $[-1, 1]$.

    Utile in classificazione binaria, con output netto (es. -1 o 1).

15. **Soft Plus**

$$f(x) = \ln(1 + e^x)$$

    Versione liscia della ReLU, derivabile ovunque, adatta a metodi basati sul gradiente.

Considerazioni finali sulla scelta della funzione di attivazione La scelta dell'attivazione è uno degli **iperparametri** del modello:

- **MLP e CNN**: nella maggior parte dei casi si preferisce la **ReLU** o una sua variante (come Leaky ReLU) per la loro efficienza e semplicità.
- **RNN**: per il processamento di sequenze, si usano comunemente funzioni come la **Tanh** e la **Sigmoid**, perché aiutano a mantenere informazioni nel tempo.
- **Personalizzazione**: in alcuni casi, si sperimentano funzioni di attivazione "trainable", dove alcuni parametri all'interno della funzione (come α nella PReLU) vengono appresi durante l'addestramento.

Ad esempio: in una CNN per la classificazione di immagini, ogni strato convoluzionale applica tipicamente la ReLU per introdurre non-linearità in modo semplice ed efficiente. Se il modello dovesse mostrare segnali di "neuroni morti" (cioè, neuroni che non contribuiscono più all'apprendimento perché restano in stato di zero attivazione), si potrebbe passare a una variante come la Leaky ReLU, che permette anche a valori negativi di contribuire con un piccolo segnale.

In conclusione, le funzioni di attivazione sono una componente critica nelle reti neurali, influenzando sia la stabilità che l'efficacia dell'addestramento. Offrono la possibilità di modellare relazioni non lineari e possono essere scelte e adattate in base al tipo di architettura (MLP, CNN, RNN) e al problema specifico. La loro efficienza computazionale, differenziabilità e capacità di mitigare problemi come il gradiente vanescente le rendono indispensabili nella pratica del deep learning.

4.2 Backpropagation

La **backpropagation** è una tecnica fondamentale per addestrare le reti neurali. Serve a calcolare in modo efficiente i gradienti della funzione di costo rispetto a tutti i pesi del modello, permettendo di aggiornare i parametri in modo da ridurre l'errore tra l'output predetto e il target reale.

Contesto generale Nei modelli di deep learning, abbiamo un insieme di esempi di training (x_i, y_i) e un modello parametrico $f(x, \theta)$ che mappa l'input x nell'output $\hat{y}$. L'obiettivo è trovare i parametri θ^* che minimizzano la funzione di costo, ad esempio:

$$\theta^* = \arg\min_{\theta} L(y, f(x, \theta))$$

Perché il problema di ottimizzazione in reti neurali è altamente complesso e spesso non convesso, l'algoritmo di gradient descent (o sue varianti) viene utilizzato per aggiornare i pesi. Tuttavia, per farlo è indispensabile calcolare il gradiente della funzione di costo rispetto a ciascun peso. Qui entra in gioco la backpropagation, che sfrutta il **principio del chain rule** del calcolo differenziale per computare questi gradienti in maniera efficiente.

Motivi per la backpropagation Inizialmente, metodi semplici come la perturbazione casuale di un peso (o di tutte le attivazioni) erano stati tentati per vedere se l'errore diminuiva; tuttavia, questi approcci erano estremamente inefficienti, richiedendo migliaia di passaggi per ogni singolo aggiornamento. La backpropagation, invece, permette di aggiornare tutti i pesi simultaneamente calcolando esattamente la derivata del costo rispetto a ogni parametro, utilizzando la struttura a strati delle reti neurali.

Meccanismo del feedforward Prima di applicare la backpropagation, il dato passa attraverso il network in una fase detta feedforward propagation:

1. **Strato di input**: il dato x viene inserito.
2. **Strati nascosti**: ogni strato applica una trasformazione lineare seguita da una funzione di attivazione. Ad esempio, per un singolo neurone nel livello l (nodo j):

$$z_j^{(l)} = \sum_{i=1}^{m} w_{ji}^{(l)} a_i^{(l-1)} + b_j^{(l)}$$

   dove:

   - $w_{ji}^{(l)}$ sono i pesi che collegano il neurone i del livello precedente a quello j del livello l,
   - $a_i^{(l-1)}$ sono le attivazioni (output) del livello precedente,
   - $b_j^{(l)}$ è il bias.
     L'attivazione del neurone viene quindi calcolata applicando una funzione non lineare σ:

$$a_j^{(l)} = \sigma(z_j^{(l)})$$

   - **Strato di output**: l'output finale $\hat{y}$ viene ottenuto dall'ultimo strato e confrontato con il target y per calcolare il costo L.

Processo di backpropagation Dopo il feedforward, si procede a calcolare il gradiente della funzione di costo rispetto a ogni parametro, muovendosi all'indietro dalla fine della rete fino all'inizio. La procedura si articola in questi passaggi principali:

1. **Calcolo dell'errore all'output**: si compara l'output $\hat{y}$ con il target y per determinare l'errore della rete. Questo errore è espresso attraverso la funzione di costo $L(\hat{y}, y)$.

2. **Propagazione dell'errore agli strati nascosti**: utilizzando la regola della catena (chain rule), si calcola quanto ogni unità nascosta ha contribuito all'errore finale. In altre parole, si determinano le derivate parziali del costo rispetto alle attivazioni di ciascun neurone:
$$\frac{\delta L}{\delta a_j^{(l)}}$$
Queste derivate vengono poi usate per calcolare i gradienti rispetto ai pesi e ai bias.

3. **Aggiornamento dei pesi**: una volta che il gradiente $\nabla_\theta L$ è stato calcolato per ogni parametro, i pesi vengono aggiornati utilizzando un algoritmo di ottimizzazione, ad esempio il gradient descent:
$$\theta_{i+1} = \theta_i - \alpha \nabla_\theta L$$
Dove α è il learning rate.

Esempio in una rete con più strati Considera una rete con due neuroni in input e una struttura composta da più strati (ad esempio, con 3 livelli, dove i livelli intermedi sono nascosti). I parametri del modello sono organizzati come:
$$W = \{W^{(1)}, W^{(2)}, \ldots, W^{(L)}\} \quad \text{e} \quad b = \{b^{(1)}, b^{(2)}, \ldots, b^{(L)}\}$$

Durante il feedforward, ogni livello l calcola:
$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$
$$a^{(l)} = \sigma(z^{(l)})$$

Con la backpropagation, si calcola il gradiente del costo rispetto a $W^{(l)}$ e $b^{(l)}$ per ciascun livello, partendo dall'output e procedendo all'indietro. Questo consente di aggiornare ciascun parametro in modo che la funzione di costo diminuisca progressivamente, affinando la capacità della rete di fare previsioni accurate.

Conclusioni La backpropagation è il meccanismo chiave che rende possibile l'addestramento delle reti neurali. Esso calcola in modo efficiente i gradienti necessari per aggiornare i pesi, sfruttando:

- **Feedforward**: dove i dati vengono trasformati attraverso gli strati fino all'output.
- **Chain rule**: per propagare l'errore all'indietro e determinare l'influenza di ogni peso sul costo totale.
- **Aggiornamento iterativo dei pesi**: che avviene tramite algoritmi come il gradient descent.

Questo processo iterativo consente alla rete di "imparare" dai dati, migliorando progressivamente le previsioni. Sebbene il calcolo del gradiente possa sembrare complesso, la backpropagation lo rende computazionalmente efficiente, anche per modelli di molteplici strati e con milioni di parametri.

4.2.1 Ottimizzazione del gradiente

L'ottimizzazione del gradiente è la procedura mediante la quale aggiorniamo i pesi w e i bias b in una rete neurale per minimizzare una funzione di costo, come ad esempio l'errore quadratico medio (RMSE). Utilizziamo una versione del gradient descent in mini-batch, che aggiorna i pesi usando un sottoinsieme (mini-batch) dei dati ad ogni iterazione.

Definizione della funzione di costo Supponiamo di usare il RMSE (Root Mean Squared Error) come funzione di costo. La funzione di costo per un mini-batch di m campioni è:

$$L(y, \hat{y}) = \frac{1}{2m} \sum_{n=k+1}^{k+m} ||y_n - \hat{y}_n||^2$$

Qui, y_n sono i valori reali e $\hat{y}_n$ sono le previsioni del modello. Lo scopo è trovare il vettore dei pesi w (e bias b) che minimizza L:

$$w = w - \alpha \nabla_w L$$

dove α è il learning rate, che controlla la dimensione del passo.

Aggiornamento dei pesi e dei bias Per aggiornare i pesi in una rete neurale, calcoliamo il gradiente della funzione di costo rispetto a ciascun peso. Nei modelli a più strati, questo viene fatto attraverso la **backpropagation**, che sfrutta la **catena del derivato (chain rule)**.

Per **aggiornare un peso**, consideriamo la regola di aggiornamento per il peso $w_{ij}^{(l)}$ con indice i nel livello precedente e j nel livello corrente l. La derivata della funzione di costo rispetto a $w_{ij}^{(l)}$ si esprime come:

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} \cdot a_i^{(l-1)}$$

dove:

- $a_i^{(l-1)}$ è l'attivazione del neurone i nel livello $l-1$.
- $\delta_j^{(l)}$ è il **gradiente locale** per il neurone j al livello l e si calcola come:

$$\delta_j^{(l)} = \frac{\partial L}{\partial a_j^{(l)}} \cdot \sigma' \left(z_j^{(l)} \right)$$

  Qui, $z_j^{(l)}$ è l'input lineare al neurone, definito da:

$$z_j^{(l)} = \sum_{i=1}^{m} w_{ij}^{(l)} a_i^{(l-1)} + bj^{(l)}$$

  e $\sigma'(z_j^{(l)})$ è la derivata della funzione di attivazione.

La **regola di aggiornamento** diventa quindi:

$$w_{ij}^{(l)} \longleftarrow w_{ij}^{(l)} - \alpha \delta_j^{(l)} a_i^{(l-1)}$$

Il bias $b_j^{(l)}$ viene aggiornato in modo simile, dato che la sua derivata è:

$$\frac{\partial L}{\partial b_j^{(l)}} = \delta_j^{(l)}$$

e quindi la regola di aggiornamento è:

$$b_j^{(l)} \longleftarrow b_j^{(l)} - \alpha \delta_l^{(l)}$$

Ruolo della catena del derivato (chain rule) La **chain rule** in calcolo permette di calcolare la derivata di una funzione composta. Se consideriamo una funzione composta $f(g(x))$, la sua derivata è:

$$(f \circ g)'(x) = f'\left(g(x)\right) \cdot g'(x)$$

Nel contesto delle reti neurali, la funzione di costo L dipende in modo complicato dai pesi perché l'output $\hat{y}$ è il risultato di numerosi strati e funzioni di attivazione. La backpropagation applica ripetutamente la chain rule per "propagare" l'errore dallo strato di output fino ai primi strati, permettendo di ottenere l'espressione del gradiente per ogni peso.

***Esempio pratico*: aggiornamento nell'output layer** Consideriamo un semplice esempio in cui il livello di output (livello L) ha un solo neurone.

- **Output del neurone**:

$$\hat{y} = \sigma(z^{(l)}) \quad \text{dove} \quad z^{(L)} = W^{(L)} a^{(L-1)} + b^{(L)}$$

- **Errore (con RMSE)**: la funzione di costo per un singolo esempio è:

$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

- **Calcolo della derivata**: per il neurone di output, il gradiente locale $\delta^{(L)}$ si calcola come:

$$\delta^{(L)} = \frac{\partial L}{\partial \hat{y}} \cdot \sigma'\left(z^{(L)}\right) = (\hat{y} - y) \cdot \sigma'\left(z^{(L)}\right)$$

- **Aggiornamento del peso**: per un peso che collega un neurone del livello $L-1$ all'output, l'aggiornamento sarà:

$$\frac{\partial L}{\partial w_i^{(L)}} = \delta^{(L)} \cdot a_i^{(L-1)}$$

e il peso viene aggiornato come:

$$w_i^{(L)} \longleftarrow w_i^{(L)} - \alpha \delta^{(L)} a_i^{(L-1)}$$

Questo processo, applicato iterativamente per tutti i pesi e bias della rete, permette al modello di migliorare gradualmente le proprie previsioni.

Conclusioni Riassumendo, la **gradient optimization** nei modelli di deep learning si basa su questi passaggi:

1. **Feedforward**: i dati vengono propagati attraverso la rete per ottenere l'output.
2. **Calcolo dell'errore**: si confronta l'output ottenuto $\hat{y}$ con il target y usando una funzione di costo, come il RMSE.
3. **Backpropagation**: utilizzando la chain rule, si calcolano i gradienti della funzione di costo rispetto a ogni peso e bias.
4. **Aggiornamento**: i parametri vengono aggiornati secondo la regola:

$$\theta \longleftarrow \theta - \alpha \nabla_\theta L$$

   con aggiornamenti specifici per pesi e bias basati sui rispettivi gradienti.

Questo approccio permette di ottimizzare efficacemente la rete, adattando i parametri per migliorare le previsioni, anche in contesti complessi e non convessi come quelli tipici del deep learning.

4.2.2 Local gradient

Nel processo di backpropagation in una rete neurale a più strati, un aspetto cruciale è il calcolo dei **local gradient** $\delta^{(l)}$ in ogni livello l. Tali gradienti forniscono il "peso" che l'errore deve avere per ogni neurone (o nodo) nello strato, consentendo di aggiornare i pesi e i bias.

Local gradient allo strato di output Se lo strato finale L ha attivazione σ e output $a^{(L)}$, la derivata del costo rispetto alle attivazioni finali si indica con $\nabla_a L$. Il **local gradient** in uscita è:

$$\delta^{(L)} = \nabla_a L \odot \sigma' \left(z^{(L)}\right)$$

dove:

- $\nabla_a L$ è il vettore di derivate del costo rispetto alle attivazioni $a^{(L)}$,
- $\sigma'(z^{(L)})$ è la derivata della funzione di attivazione calcolata in corrispondenza di $z^{(L)}$,
- $\odot$ indica il prodotto elemento per elemento (Hadamard product).

Local gradient negli hidden layers Per uno strato nascosto $l (1 \leq l \leq L)$, il local gradient $\delta^{(l)}$ deriva da quello dello strato successivo $\delta^{(l+1)}$. La relazione si basa sulla **chain rule** e sulla struttura della rete:

$$\delta^{(l)} = (w^{(l+1)})^T \delta^{(l+1)} \odot \sigma'(z^{(l)})$$

dove:

- $w^{(l+1)}$ è la matrice dei pesi che collega lo strato l allo strato $l+1$,
- $\delta^{(l+1)}$ è il *local gradient* dello strato successivo,

- $\sigma'(z^{(l)})$ è la derivata della funzione di attivazione al livello l.

Lo sforzo di "correzione" che un neurone allo strato l deve apportare ($\delta^{(l)}$) dipende da:

1. quanto lo lo strato successivo "reclama" aggiustamenti ($\delta^{(l+1)}$),
2. come i pesi $(w^{(l+1)})^T$ propagano questa richiesta indietro, e
3. la pendenza (derivata) della funzione di attivazione in $z^{(l)}$.

Riassunto dei principali formulati

1. **Local gradient output layer**:
$$\delta^{(L)} = \nabla_a L \odot \sigma'(z^{(L)})$$
2. **Local gradient hidden layer** l:
$$\delta^{(l)} = (w^{(l+1)})^T \delta^{(l+1)} \odot \sigma'(z^{(l)})$$
3. **Gradiente rispetto ai pesi**:
$$\frac{\partial L}{\partial w_{ij}^{(l)}} = a_i^{(l-1)}\, \delta_j^{(l)}$$
4. **Gradiente rispetto ai bias**:
$$\frac{\partial L}{\partial b_j^{(l)}} = \delta_j^{(l)}$$

Conclusioni La catena del derivato (**chain rule**) è il fulcro che consente di calcolare i local gradient in maniera efficiente, propagando l'errore indietro dallo strato di output verso gli strati iniziali.

Il local gradient a un livello l (neurone j) deriva dal local gradient dello strato successivo, tenendo conto della pendenza della funzione di attivazione e dei pesi che collegano i due strati.

Una volta calcolati i local gradient, gli aggiornamenti di pesi e bias si ottengono moltiplicando $\delta_j^{(l)}$ per le attivazioni in ingresso ($a_i^{(l-1)}$) o pari a $\delta_j^{(l)}$ nel caso dei bias.

Questa struttura modulare del calcolo del gradiente è ciò che rende possibile il training di reti neurali profonde, anche con milioni di parametri, in modo relativamente efficiente.

4.2.3 Vanishing gradient

Nel training di reti neurali profonde, un ostacolo comune è il **vanishing gradient** (gradiente che si annulla). Questo fenomeno si verifica durante la retropropagazione (backpropagation) e consiste nel fatto che il gradiente della funzione di costo rispetto ai pesi delle prime (o "più basse") layer diventa sempre più piccolo man mano che l'errore viene propagato all'indietro lungo la rete. Di conseguenza, le modifiche apportate ai pesi nei livelli iniziali sono minime, rendendo difficile il loro aggiornamento efficace e rallentando, o addirittura bloccando, la convergenza del modello.

Origine del problema: il ruolo della chain rule Il vanishing gradient deriva dalla **regola della catena** usata per calcolare i gradienti nelle reti neurali. Consideriamo in particolare il calcolo del gradiente rispetto a un peso nei layer $l-1$. Supponiamo che, a livello l, il gradiente locale sia rappresentato da $\delta^{(l)}$ e che la funzione di attivazione abbia una derivata $\sigma'(z^{(l-1)})$. Il gradiente rispetto a un peso $w_{(ij)}^{(l-1)}$ (collegando il neurone i nel livello $l-2$ al neurone j nel livello $l-1$) viene calcolato come:

$$\frac{\partial L}{\partial w_{ij}^{(l-1)}} = \delta_j^{(l-1)} a_i^{(l-2)},$$

dove il termine $\sigma_j^{(l-1)}$ (local gradient per il neurone j nel livello $l-1$) si calcola propagando l'errore dello strato successivo:

$$\delta^{(l-1)} = (w^{(l)})^T \delta^{(l)} \odot \sigma'(z^{)(l-1)}.$$

Qui, $(w^{(l)})^T$ trasmette l'errore dal livello l a quello $l-1$ e $\sigma'(z^{(l-1)})$ è la derivata della funzione di attivazione al livello $l-1$.

Problema: se uno o più termini (come $\sigma'(z^{(l-1)})$) sono piccoli, il prodotto di questi termini lungo molti layer causa una rapida diminuzione del gradiente. Quindi, man mano che si va più indietro nella rete, i gradienti possono diventare così piccoli da non contribuire efficacemente all'aggiornamento dei pesi.

Conseguenze sul training

- **Aggiornamenti inefficaci nei livelli iniziali**: i primi layer (quelli vicini all'input) ricevono gradienti molto piccoli, il che significa che i loro pesi vengono aggiornati di pochissimo. Questo impedisce al modello di apprendere caratteristiche utili dall'input, compromettendo la generalizzazione.
- **Convergenza lenta o stagnazione**: se i gradienti si annullano prima di poter influenzare i pesi dei primi layer, l'algoritmo può impiegare molto tempo per convergere oppure bloccarsi in un punto subottimale.

Tecniche per mitigare il problema Per contrastare il vanishing gradient, sono state sviluppate diverse tecniche:

1. **Inizializzazione dei pesi migliore**: metodi come l'inizializzazione *Xavier* o *He* mirano a impostare i pesi iniziali in modo che le attivazioni non si saturino (cioè non cadano in zone dove le derivate sono vicine a zero).
2. **Funzioni di attivazione non saturanti**: Funzioni come la ReLU e sue varianti (ad esempio, Leaky ReLU o Parametric ReLU) tendono a mantenere gradienti più robusti rispetto a funzioni come il Sigmoid o la Tanh, che saturano per valori molto alti o bassi.
3. **Batch normalization**: normalizza gli input di ogni layer, riducendo la variazione delle attivazioni e contribuendo a mantenere gradienti più stabili.
4. **Architectural innovations**: approcci come l'uso di reti residuali (ResNet) che introducono shortcut (connessioni residue) per facilitare il flusso dei gradienti attraverso molteplici layer.

5. **Pretraining non supervisionato**: tecniche come il pretraining con Restricted Boltzmann Machines (RBM) sono state usate in passato per inizializzare le reti in maniera tale da rendere il training supervisionato più efficace.

Conclusioni Il problema del vanishing gradient rappresenta una sfida significativa nel training di reti neurali profonde, poiché la propagazione del gradiente lungo molti layer può causare aggiornamenti quasi nulli nei livelli iniziali. Attraverso tecniche innovative come l'inizializzazione appropriata, l'adozione di funzioni di attivazione non saturanti, la normalizzazione batch e l'architettura residua, è possibile mitigare questo problema e rendere il training più efficace.

Questo insieme di metodi consente di addestrare reti neurali molto profonde che, nonostante la complessità, riescono a generalizzare bene anche su dati non visti.

4.2.4 Esempio di backpropagation con attivazione identità

Questo esempio illustra come funzionano concretamente i passaggi di backpropagation, aggiornando i pesi e i bias in una piccola rete neurale con funzione di attivazione identità ($\sigma(x) = x$), utilizzando un mini-batch di $m = 3$ campioni e la **loss MSE** (Mean Squared Error).

Struttura della rete e parametri iniziali

- **Rete con 3 layer** (2 nascosti + 1 layer di output), ognuno con 2 neuroni negli strati nascosti e 1 neurone di output finale.
- **Funzione di attivazione**: $\sigma(x) = x \implies \sigma'(x) = 1$.
- **Pesi e bias iniziali**: tutti impostati a 0.5 ($w_{ij}^{(l)} = 0.5, \quad b_j^{(l)} = 0.5$).
- **Funzione di costo (MSE)**:

$$L(y, \hat{y}) = \frac{1}{2m} \sum_{k=1}^{m} ||y_k - \hat{y}_k||^2$$

  .
- **Algoritmo di ottimizzazione**: mini-batch SGD con learning rate $\alpha = 0.01$ e batch size $m = 3$.

Equazioni per le attivazioni:

1. **Layer 1** (2 neuroni):

$$a_1^{(1)} = w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + b_1^{(1)}, \quad a_2^{(2)} = w_{12}^{(2)} x_1 + w_{22}^{(2)} x_2 + b_2^{(2)}$$

2. **Layer 2** (2 neuroni):

$$a_1^{(2)} = w_{11}^{(2)} a_1^{(1)} + w_{21}^{(2)} a_2^{(1)} + b_1^{(2)}, \quad a_2^{(2)} = w_{12}^{(2)} a_1^{(1)} + w_{22}^{(2)} a_2^{(1)} + b_2^{(2)}$$

3. **Layer 3** (uscita, 1 neurone):

$$\hat{y} = w_{11}^{(3)} a_1^{(2)} + w_{21}^{(3)} a_2^{(2)} + b_1^{(3)}$$

Poiché $\sigma(x) = x$, all'atto pratico $a_j^l = z_j^{(l)}$, semplificando i calcoli.

Calcolo delle derivate (local gradient) Dalla funzione di costo MSE e l'attivazione identità:

- **Local gradient allo strato di output (L)**:

$$\delta^{(3)} = (y_k - \hat{y}_k)$$

(Il segno dipende da come scriviamo l'errore; qui assumiamo $\delta^{(3)} = \hat{y}_k - y_k$ o viceversa in base alla derivata; l'importante è l'uso coerente nel calcolo degli aggiornamenti).

- **Local gradient allo strato $l = 2$ (secondo layer nascosto)**:

$$\delta^{(2)} = w^{(3)}\delta^{(3)} \quad \text{(avendo } \sigma'(x) = 1\text{)}$$

Se il neurone di output ha indice 1, e gli indici del layer 2 sono $(1, 2)$, allora

$$\delta_1^{(2)} = w_{11}^{(3)}\delta_1^{(3)}, \quad \delta_2^{(2)} = w_{21}^{(3)}\delta_1^{(3)}$$

- **Local gradient allo strato $l = 1$ (primo layer nascosto)**:

$$\delta^{(1)} = (w^{(2)})^T\delta^{(2)} \quad \text{(sempre con } \sigma'(x) = 1\text{)}$$

In pratica, ciascun neurone nel layer 1 ottiene il suo δ dalla combinazione lineare dei δ del layer 2 e dei pesi $w^{(2)}$.

Aggiornamento dei pesi e dei bias Una volta calcolati i local gradient $\delta_j^{(l)}$ per ciascun neurone, applichiamo le regole di aggiornamento:

1. **Aggiornamento pesi:**

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \alpha \cdot \frac{1}{m}\sum_{k=1}^{m}\left[\frac{\partial L(y_k, \hat{y}_k)}{\partial w_{ij}^{(l)}}\right]$$

Poiché $\frac{\partial L}{\partial w_{ij}^{(l)}} = a_i^{(l-1)}\delta_j^{(l)}$,

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \alpha \cdot \frac{1}{m}\sum_{k=1}^{m}\left[a_i^{(l-1)}(k)\,\delta_j^{(l)}(k)\right]$$

2. **Aggiornamento bias**:

$$b_j^{(l)} \leftarrow b_j^{(l)} - \alpha \cdot \frac{1}{m}\sum_{k=1}^{m}\left[\delta_j^{(l)}(k)\right]$$

Given x_1 and x_2 as input samples (with mini-batch $m = 3$), we have

k	x_1	x_2	$a_1^{(1)}$	$a_2^{(1)}$	$a_1^{(2)}$	$a_2^{(2)}$	$\hat{y}$	y	$\delta_1^{(3)}$	$\delta_1^{(2)}$	$\delta_2^{(2)}$	$\delta_1^{(1)}$	$\delta_2^{(1)}$	$a_1^{(2)}\delta_1^{(3)}$	$a_2^{(2)}\delta_1^{(3)}$	$a_1^{(1)}\delta_1^{(2)}$	$a_1^{(1)}\delta_2^{(2)}$	$a_2^{(1)}\delta_1^{(2)}$	$a_2^{(1)}\delta_2^{(2)}$	$x_1\delta_1^{(1)}$	$x_2\delta_1^{(1)}$	$x_1\delta_2^{(1)}$	$x_2\delta_2^{(1)}$
1	0.1	0.2	0.65	0.65	1.15	1.15	1.65	2	0.35	0.175	0.175	0.175	0.175	0.4025	0.4025	0.114	0.114	0.114	0.114	0.0175	0.0175	0.035	0.035
2	0.3	0.3	0.8	0.8	1.3	1.3	1.8	1.9	0.1	0.05	0.05	0.05	0.05	0.13	0.13	0.04	0.04	0.04	0.04	0.015	0.015	0.015	0.015
3	0.4	0.4	0.9	0.9	1.4	1.4	1.9	1.8	-0.1	-0.05	-0.05	-0.05	-0.05	-0.14	-0.14	-0.045	-0.045	-0.045	-0.045	-0.02	-0.02	-0.02	-0.02
μ						**AVERAGE**			0.117	0.058	0.058	0.058	0.058	0.3925	0.3925	0.037	0.037	0.037	0.037	0.0042	0.0042	0.01	0.01

Now we can proceed with the update.

$$
\begin{aligned}
w_{11}^{(1)} &= 0.5 - 0.01(0.0042) = 0.499... = w_{12}^{(1)} \\
w_{21}^{(1)} &= 0.5 - 0.01(0.01) = 0.499... = w_{22}^{(1)} \\
w_{11}^{(2)} &= 0.5 - 0.01(0.0037) = 0.499.. = w_{12}^{(2)} = w_{21}^{(2)} = w_{22}^{(2)} \\
w_{11}^{(3)} &= 0.5 - 0.01(0.3925) = 0.496... = w_{21}^{(3)} \\
b_1^{(1)} &= 0.5 - 0.01(0.0058) = 0.4994... = b_2^{(1)} \\
b_1^{(2)} &= 0.5 - 0.01(0.0058) = 0.4994... = b_2^{(2)} \\
b_1^{(3)} &= 0.5 - 0.01(1.227) = 0.4988... = b_2^{(3)}
\end{aligned}
$$

Considerazioni sul vanishing gradient Nell'esempio mostrato:

- **Attivazione identità**: tutti i neuroni applicano la funzione $f(x) = x$. Se i pesi sono piccoli (o $\sigma'(x)$ fosse <1), le derivate si moltiplicano e possono ridursi rapidamente passando da un layer all'altro.
- **Effetti**: i layer più vicini all'input possono ricevere gradienti quasi nulli, rallentando di molto l'apprendimento.
- **Rimedi**: uso di funzioni di attivazione non saturanti (ad es. ReLU), inizializzazione dei pesi più attenta (Xavier, He), batch normalization, architetture come ResNet, ecc.

Conclusioni In questo esempio abbiamo visto i passi concreti del backpropagation con mini-batch SGD su 3 campioni e attivazione identità. I pesi, inizialmente a 0.5, vengono aggiornati in base ai gradienti medi calcolati sui 3 esempi, illustrando come i gradienti derivino dalla chain rule. Tuttavia, l'uso di un'attivazione identità in reti profonde può esacerbare il vanishing gradient problem, richiedendo tecniche avanzate per mantenere gradienti significativi nei primi layer. Questo esempio, sebbene semplice, mostra la sequenza di calcoli e gli effetti di un'architettura lineare sul processo di training.