Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Triennale in Matematica

INTRODUZIONE ALLE RETI NEURALI PROFONDE E ALLA DISCESA STOCASTICA DEL GRADIENTE

Relatore:
Chiar.mo Prof.
FABRIZIO LILLO

Presentata da:
ANDREA RINALDI

Sessione Unica
Anno Accademico 2019/2020

Intelligence is not merely the absorption
and accumulation of facts and bits of information,
but the ability to expand that information exponentially,
to encourage its mutation,
to stand on the foundation of accumulated fact
and reach out and up into wider worlds of thought.

Roland Merullo

Introduzione

Gli ultimi decenni sono stati caratterizzati da un grande progresso tecnologico che ha inciso in modo consistente sulla società moderna, entrando a far parte della vita quotidiana e cambiando le abitudini.
In particolare, lo sviluppo delle intelligenze artificiali ha permesso alle macchine di prendere decisioni in modo autonomo, basandosi esclusivamente su conoscenze assunte.

Il *Machine Learning*, o *apprendimento automatico*, è una branca dell'intelligenza artificiale in ampia espansione che permette alle macchine di svolgere specifici compiti alla luce dell'esperienza compiuta e degli errori commessi, senza la necessità di un continuo apporto nella programmazione da parte dell'uomo.
In questo modo è possibile acquisire una maggiore indipendenza ed efficienza.
Infatti, tali macchine sono in grado di raggiungere autonomamente l'obiettivo per le quali sono state programmate mediante l'estrazione di informazioni ricorrenti da un insieme di dati con l'ausilio di metodi del Machine Learning.
Tuttavia, è stato osservato che alcune attività di immediata risoluzione per l'uomo risultano complicate per una macchina se vengono applicati esclusivamente i metodi classici del Machine Learning.

A tale problema il *Deep Learning*, o *apprendimento profondo*, è riuscito a fornire una soluzione, permettendo alla macchina di estrarre informazioni semplici da un insieme di dati per poi combinarle ottenendo informazioni più elaborate. In particolare, il risultato finale viene raggiunto elaborando, in maniera gerarchica, diversi livelli di rappresentazione dei dati corrispondenti a caratteristiche, fattori o concetti.

Questo elaborato analizza il processo di addestramento di una rete neurale, prestando particolare attenzione all'ottimizzazione dei pesi della rete in quanto fondamentale al fine di minimizzare l'errore tra i risultati ottenuti e quelli desiderati.
Tra gli algoritmi più significativi in grado di modificare i pesi dei neuroni della rete vi è l'algoritmo di backpropagation, il cui funzionamento viene descritto nell'elaborato.
Inoltre, viene illustrata l'applicazione di alcuni metodi di ottimizzazione utilizzati per trovare i punti di minimo della funzione costo. Tali metodo vengono confrontati tra loro con l'obiettivo di individuare quelli con caratteristiche di convergenza migliori.

In particolare, nel primo capitolo di questo elaborato vengono presentate alcune tecniche di Machine Learning insieme ai suoi tre differenti sistemi di apprendimento automatico.
Più precisamente, viene esaminato l'apprendimento supervisionato e vengono introdotte le sue categorie di classificazione e regressione.
Infine, vengono brevemente descritti alcuni altri metodi Machine Learning per l'apprendimento supervisionato.

Il secondo capitolo è interamente dedicato alla descrizione del Deep Learning e delle tecniche di cui esso si avvale per permettere alle macchine di apprendere. Contestualmente, viene introdotto il concetto di rete neurale artificiale e della sua architettura.
Infine, questo capitolo si focalizza sul processo di addestramento della rete ed, in particolare, dell'ottimizzazione dei pesi mediante la descrizione della funzione costo e dell'algoritmo di backpropagation.

Il terzo capitolo presenta i tre differenti metodi di discesa del gradiente, batch, minibatch e SGD, utilizzati per determinare i punti di minimo della funzione costo. Dopo aver confrontato tali metodi, il capitolo si conclude con una descrizione di alcuni algoritmi in grado di ottimizzare il metodo SGD, permettendo così una migliore convergenza al punto di ottimo della funzione costo.

Indice

Elenco delle figure

Capitolo 1

Machine Learning

Il Machine Learning è una branca dell'Intelligenza Artificiale che si pone l'obiettivo di far apprendere a macchine e sistemi in modo automatico mediante l'utilizzo di metodi ed algoritmi per la creazione automatica di modelli a partire dai dati.
In particolare, il Machine Learning è un metodo di analisi dati che automatizza la costruzione di modelli analitici, basandosi sull'idea che i sistemi e le macchine possano imparare dai dati, identificare modelli autonomamente e prendere decisioni con un intervento umano ridotto al minimo. Essi, in questo modo, imparano a svolgere determinati compiti perfezionando, tramite l'esperienza, le proprie risposte e funzioni.
Alla base dell'apprendimento automatico ci sono differenti algoritmi che, partendo da nozioni primitive, riescono a prendere specifiche decisioni piuttosto che altre e sono in grado di effettuare azioni apprese nel tempo.

In questo capitolo viene introdotto il funzionamento generale dell'apprendimento automatico e delle tecniche di cui esso si avvale per apprendere.
In particolare, si analizzano l'apprendimento supervisionato e le sue categorie di classificazione e regressione. Infine vengono presi in esame alcuni modelli algoritmici per l'ottimizzazione di problemi che spesso una macchina deve affrontare.

1.1 Apprendimento automatico

Negli ultimi anni social network, dispositivi elettronici, internet ed altre fonti attualmente disponibili stanno generando quantità di dati sempre maggiori che vengono utilizzate dalle aziende per fornire servizi.
Infatti, applicando tecniche di analisi, le aziende riescono ad estrarre informazioni utili sia per rendersi conto del proprio andamento attuale che per predire quello futuro, riuscendo così a prendere decisioni opportune ed evitando rischi come quelli legati agli investimenti.
Esistono diverse tecniche di apprendimento, conosciute con il nome di *tecniche di Data Mining*, che permettono l'applicazione di questo tipo di analisi. A seconda del tipo di algoritmo che viene utilizzato per permettere l'apprendimento alla macchina, ossia a seconda delle modalità con cui la macchina impara ed accumula dati ed informazioni, si possono individuare tre differenti sistemi di apprendimento automatico: *apprendimento supervisionato, apprendimento non supervisionato* ed *apprendimento per rinforzo.*
I tre modelli di apprendimento sono utilizzati indifferentemente a seconda del tipo di problema e dei dati a disposizione, garantendo così sempre la massima performance ed il migliore risultato possibile per la risposta agli stimoli esterni.

- **Apprendimento supervisionato**: tale apprendimento ha l'obiettivo di istruire un sistema, fornendogli una serie di dati ed informazioni che gli consentano di apprendere. Questo avviene mediante la costruzione di un modello che prende in input alcuni dati di addestramento etichettati x a cui sono associati i relativi valori y di uscita. A partire da tali valori si riescono a fare previsioni su dati non disponibili o futuri stimando la probabilità condizionata $Pr[y|x]$.
  Se l'addestramento ha successo, il sistema impara a riconoscere la relazione incognita che lega le variabili input a quelle output, ed è quindi in grado di fare previsioni anche relative ad outputs non noti a priori.

- **Apprendimento non supervisionato o Hebbiano**: tale apprendimento prevede che le informazioni fornite al sistema non siano codificate, ossia il sistema non ha la possibilità di attingere a determinate informazioni in quanto è a conoscenza solo dei dati di input e non anche dei relativi risultati di output. Dovrà quindi essere il

sistema stesso a catalogare tutte le informazioni in proprio possesso, organizzarle ed imparare il loro significato, il loro utilizzo e, soprattutto, il risultato a cui esse portano. L'apprendimento senza supervisione offre maggiore libertà di scelta alla macchina che dovrà organizzare le informazioni in maniera intelligente.

- **Apprendimento a rinforzo**: tale apprendimento rappresenta la tipologia di apprendimento più complessa. Esso prevede che il sistema sia dotata di strumenti in grado di migliorare il proprio apprendimento e, in particolare, di comprendere le caratteristiche dell'ambiente circostante. Al sistema vengono forniti elementi di supporto, quali sensori, telecamere e GPS, che permettono di rilevare quanto avviene nell'ambiente circostante ed effettuare scelte per un migliore adattamento. Questo tipo di apprendimento è tipico delle auto senza pilota che, grazie a un complesso sistema di sensori di supporto, è in grado di percorrere strade cittadine e non, riconoscendo eventuali ostacoli e seguendo le indicazioni stradali.

In questo elaborato viene esaminato l'*apprendimento supervisionato*, tra le cui tecniche è possibile individuare la *regressione* e la *classificazione*.
I contenuti dei paragrafi successivi sono tratti da [1] e [2].

In un tipico scenario si è in possesso di misure di risultato solitamente quantitative o categoriche che si vogliono predire a partire da un insieme di dati di addestramento in cui è possibile osservare caratteristiche relative ad un insieme di oggetti analizzati. A partire da questi dati si costruisce un modello predittivo che permetterà di prevedere risultati relativi ad oggetti inosservati.
Le variabili indipendenti di cui si è in possesso si denotano con il termine *inputs* o *predictors*; esse influiscono su uno o più variabili dipendenti chiamate *outputs*.
L'obiettivo è utilizzare gli inputs per predire i valori degli outputs, che possono essere

- misure *quantitative*, che forniscono la misura di una grandezza e solitamente sono numeri reali
- misure *qualitative*, chiamate anche categoriche o discrete, che assumono valori in un insieme finito $\mathcal{G}$ contenente diverse etichette e quindi specificano una classe di appartenenza.

A partire da questa distinzione nella tipologia degli outputs è possibile individuare due differenti categorie di apprendimento supervisionato: la *regressione*, che predice outputs quantitativi, e la *classificazione*, che predice outputs qualitativi.
Anche le variabili di input possono essere suddivise in qualitative e quantitative, differenziando così i tipi di metodi utilizzati per le previsioni.
Solitamente, le variabili di input vengono indicate con X, gli outputs quantitativi con Y mentre gli outputs qualitativi con G. Se X è un vettore, la sua componente i-esima viene definita con $x^{\{i\}}$ che rappresenta uno scalare.
Dati i valori di un vettore di input $X = (x^{\{1\}}, ..., x^{\{N\}}) \in \mathbb{R}^{\mathrm{N}}$ e di un vettore di output quantitativo $Y = (y_1, ..., y_N) \in \mathbb{R}^{\mathrm{N}}$ (rispettivamente qualitativo $G = (g_1, ..., g_N)$), la predizione ottenuta a partire da essi si indica con $\hat{Y}$ (rispettivamente $\hat{G}$).
Per poter costruire metodi predittivi è quindi necessario essere in possesso di un insieme di dati di addestramento $(x^{\{i\}}, y_i)$ o $(x^{\{i\}}, g_i)$ con $i = 1, ..., N$.

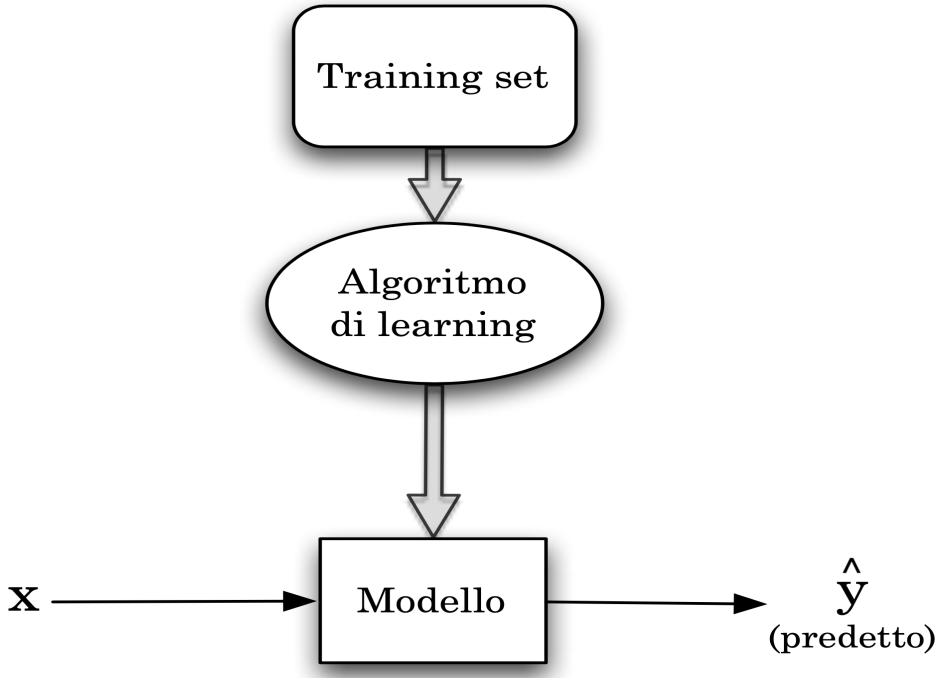


Figura 1.1: Schema generale di apprendimento supervisionato. (Immagine presa da [14])

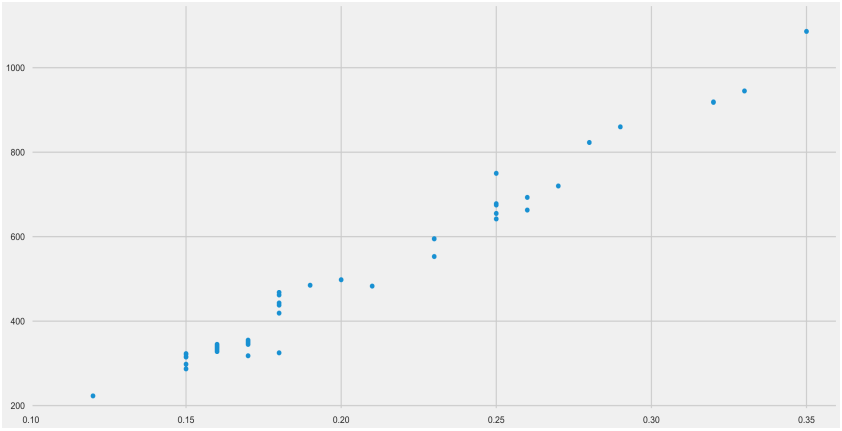
1.2 Regressione

La regressione è una tecnica utilizzata per analizzare una serie di dati che consistono di una variabile dipendente e una o più variabili indipendenti. Essa ha come obiettivo quello di stimare una relazione funzionale esistente tra la variabile dipendente e le variabili indipendenti.
La variabile dipendente nell'equazione di regressione è una funzione delle variabili indipendenti più un termine d'errore. Quest'ultimo è una variabile casuale e rappresenta una variazione non controllabile e imprevedibile nella variabile dipendente.
I parametri presenti nell'equazione sono stimati in modo da descrivere al meglio i dati.
Il metodo più comunemente utilizzato per ottenere le migliori stime è il *metodo dei minimi quadrati* (OLS), sebbene vi siano anche altri metodi.
L'analisi della regressione può essere usata per effettuare previsioni (come dati futuri di una serie temporale) ed inferenze statistiche, per testare ipotesi e per modellare delle relazioni di dipendenza. Questi usi della regressione dipendono fortemente dal fatto che le assunzioni di partenza siano verificate.

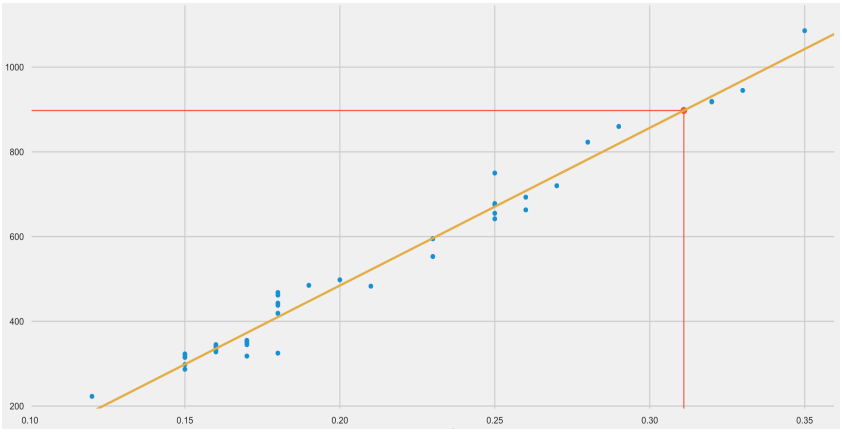
1.2.1 Regressione lineare

La regressione lineare è uno degli approcci più semplici dell'apprendimento supervisionato ed è un utile strumento per predire una risposta quantitativa Y sulla base di un unico predictor X. Essa assume che vi sia approssimativamente una relazione lineare tra le variabili X e Y.
Tale regressione è un modello largamente utilizzato in quanto funge da trampolino di lancio per nuovi approcci di apprendimento che possono essere considerati come generalizzazioni o estensioni della regressione lineare.

Sia $X \in \mathbb{R}^p$ un vettore random di input e sia $Y \in \mathbb{R}$ un vettore random di output quantitativo con distribuzione associata $Pr(X, Y)$.
L'obiettivo della regressione lineare è trovare una funzione $f(X)$ che sia in grado di predire Y a partire dai valori dell'input X, come è possibile osservare in figura 1.2.



(a) Insieme dei dati di addestramento



(b) Retta di regressione lineare

Figura 1.2: Esempio di regressione lineare. (Immagine presa da [14])

Per fare ciò è necessario scegliere una *funzione costo* $L(Y, f(X))$ che possa penalizzare gli errori commessi nella previsione e la più comune e utile è la *funzione di costo quadratica* $L(Y, f(X)) = (Y - f(X))^2$. Questa scelta stabilisce un criterio per la scelta di f, in particolare l'errore di predizione atteso (EPE) è

$$EPE(f) = E(Y - f(X))^2 = \int [y - f(x)]^2 \, Pr(dx, dy). \tag{1.1}$$

Condizionando su X è possibile scrivere EPE come

$$EPE(f) = E_X E_{Y|X}([Y - f(X)]^2 | X). \tag{1.2}$$

È sufficiente minimizzare EPE punto per punto per ottenere

$$f(x) = argmin_c E_{Y|X}([Y-c]^2|X=x). \tag{1.3}$$

La soluzione è quindi

$$f(x) = E(Y|X=x), \tag{1.4}$$

nota anche come *funzione di regressione*.

Il modello di regressione lineare (LRM) o bivariato assume che la funzione di regressione $E(Y|X)$ sia lineare negli inputs $X_1, ..., X_p$. Tale modello è in grado di fornire una adeguata ed interpretabile descrizione di come gli inputs influenzino gli outputs.
Dato un vettore di input $X^{\mathrm{T}} = (X_1, X_2, \ldots, X_{\mathrm{p}})$ si vuole predire un valore reale di output Y. Il modello di regressione lineare ha la forma

$$f(X) = \beta_0 + \sum_{\mathrm{j}=1}^{\mathrm{p}} X_{\mathrm{j}}\beta_{\mathrm{j}} \tag{1.5}$$

in cui β_0 è l'intercetta, chiamata anche *bias*, i β_{j} sono parametri o coefficienti sconosciuti mentre le variabili $\mathrm{X_j}$ possono provenire da diverse fonti quali

- inputs quantitativi
- trasformazioni di inputs quantitativi, come logaritmo o radice quadrata
- esponenziazioni semplici
- interazioni tra variabili.

Solitamente si ha a disposizione un insieme di dati di addestramento $(x^{\{1\}}, y_1) \ldots (x^{\{N\}}, y_{\mathrm{N}})$ da cui si stimano i parametri β.
Ogni $x^{\{i\}} = (x^{\{i1\}}, x^{\{i2\}}, \ldots, x^{\{ip\}})^{\mathrm{T}}$ è un vettore di misurazioni relativo all'osservazione i-esima.
Il metodo più utilizzato per la stima è quello dei *minimi quadrati* in cui si scelgono i coefficienti $\beta = (\beta_0, \beta_1, \ldots, \beta_{\mathrm{p}})^{\mathrm{T}}$ con l'obiettivo di minimizzare la *somma residua di quadrati* (RSS)

$$RSS(\beta) = \sum_{\mathrm{i}=1}^{\mathrm{N}} (y_{\mathrm{i}} - f(x^{\{i\}}))^2 = \sum_{\mathrm{i}=1}^{\mathrm{N}} (y_{\mathrm{i}} - \beta_0 - \sum_{\mathrm{j}=1}^{\mathrm{p}} x^{\{ij\}} \beta_{\mathrm{j}})^2. \tag{1.6}$$

Da un punto di vista statistico, questo criterio è ragionevole se le osservazioni di addestramento $(x^{\{i\}}, y_{\mathrm{i}})$ rappresentano estrazioni casuali indipendenti dalla loro popolazione. Anche se gli $x^{\{i\}}$ non sono stati assegnati in modo casuale, il criterio è ancora valido se gli y_{i} rimangono indipendenti, dati gli inputs $x^{\{i\}}$.
Si noti che (1.6) non fa alcuna supposizione sulla validità del modello (1.5) ma semplicemente trova il migliore modello lineare relativo a tali dati.

Per minimizzare (1.6) si procede nel modo seguente: si denota con $\mathbf{X}$ la matrice Nx$(p+1)$ le cui righe sono costituite da vettori di input (con 1 nella prima posizione) e, similmente, si pone $\mathbf{y}$ l'N-vettore degli outputs dell'insieme di addestramento.
Con queste notazioni è possibile scrivere la somma residua di quadrati come

$$RSS(\beta) = (\mathbf{y} - \mathbf{X}\beta)^T (\mathbf{y} - \mathbf{X}\beta). \tag{1.7}$$

Differenziando rispetto a β si ottiene

$$\frac{\partial RSS}{\partial \beta} = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) \tag{1.8}$$

$$\frac{\partial^2 RSS}{\partial \beta \partial \beta^T} = 2\mathbf{X}^T\mathbf{X}. \tag{1.9}$$

Assumendo che $\mathbf{X}$ abbia rango massimo e che quindi $\mathbf{X}^T\mathbf{X}$ sia definita positiva, si pone la prima derivata uguale a zero

$$\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) = 0, \tag{1.10}$$

ottenendo così l'unica soluzione

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}. \tag{1.11}$$

I valori predetti relativi ad un vettore di input $x^{\{0\}}$ sono quindi dati da

$$\hat{f}(x^{\{0\}}) = \left(\frac{1}{x^{\{0\}}}\right)^T \hat{\beta} \tag{1.12}$$

mentre i valori calcolati utilizzando il modello relativo agli inputs di addestramento sono

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\beta} = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}, \tag{1.13}$$

dove $\hat{y}_i = \hat{f}(x^{\{i\}})$.

I vettori colonna $x^{\{0\}}, ..., x^{\{p\}}$ della matrice $\mathbf{X}$ (con $x^{\{0\}} \equiv 1$) generano un sottospazio di $\mathbb{R}^N$ chiamato anche spazio delle colonne di $\mathbf{X}$.
$RSS(\beta) = ||\mathbf{y} - \mathbf{X}\beta||$ viene minimizzato scegliendo $\hat{\beta}$ in modo che il vettore residuo $\mathbf{y} - \hat{\mathbf{y}}$ sia ortogonale a tale sottospazio.
La matrice $\mathbf{H} = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$, detta matrice "cappuccio", compie tale proiezione ortogonale e per questo motivo è chiamata anche matrice di proiezione.

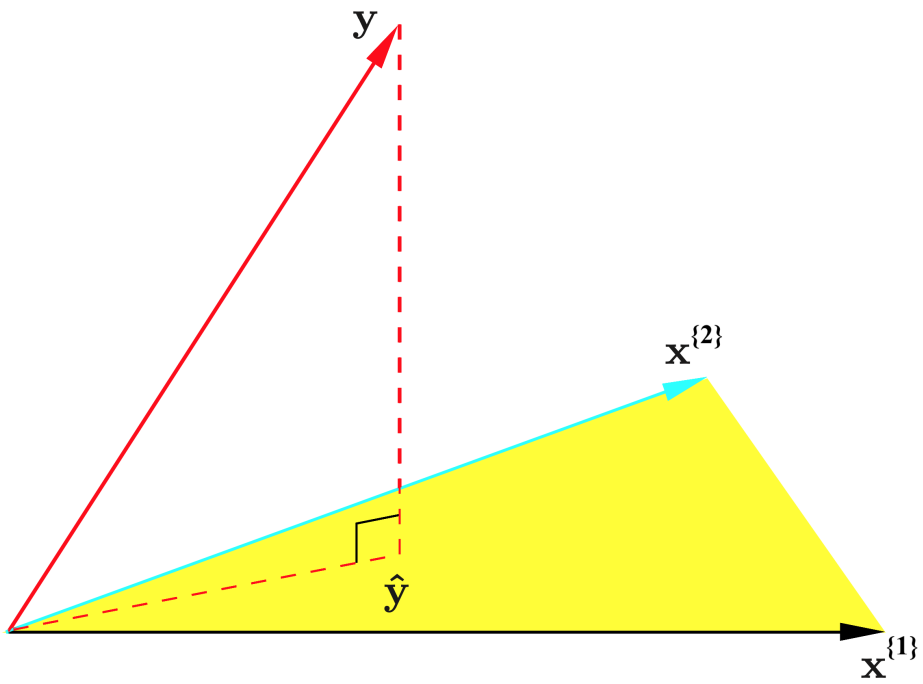


Figura 1.3: Rappresentazione geometrica della stima dei minimi quadrati. Il vettore $\mathbf{y}$ viene proiettato ortogonalmente sull'iperpiano generato dai vettori di input $x^{\{1\}}$ e $x^{\{2\}}$. Il vettore $\hat{\mathbf{y}}$ rappresenta il vettore predetto dai minimi quadrati. (Immagine presa da [1])

Può accadere che le colonne di $\mathbf{X}$ non siano linearmente indipendenti e che quindi $\mathbf{X}$ non abbia rango massimo. In tal caso la matrice $\mathbf{X}^T\mathbf{X}$ è singolare e i coefficienti $\hat{\beta}$ non sono definiti in modo unico.
Nonostante ciò, i valori $\hat{\mathbf{y}} = \mathbf{X}\hat{\beta}$ continuano ad essere le proiezioni di $\mathbf{y}$ sullo spazio delle colonne di $\mathbf{X}$ con la differenza che vi è più di un modo per esprimere tale proiezione in termine dei vettori colonna di $\mathbf{X}$.
È possibile risolvere la non unicità della rappresentazione ricodificando e rimuovendo le colonne ridondanti in $\mathbf{X}$.

Ci sono due principali motivazioni per cui spesso non si è soddisfatti delle stime ottenute utilizzando i minimi quadrati:

1. l'*accuratezza della previsione*: le stime compiute dai minimi quadrati hanno spesso un basso bias ma una grande varianza. Il bias rappresenta quanto la media della statistica si discosta dal valore esatto, ossia è la differenza tra la previsione media del modello ed il valore corretto che si vuole prevedere. La varianza, invece, rappresenta quanto i campioni si discostano dal valor medio, ossia la variabilità della previsione. Tale accuratezza può alcune volte essere migliorata riducendo o ponendo alcuni coefficienti uguali a zero. Questo comporta, da una parte, un peggioramento dei valori dei biases e, dall'altra, una riduzione della varianza dei valori predetti, migliorando così l'accuratezza generale della previsione fatta

2. l'*interpretazione*: dato un grande numero di predictors, si vorrebbe determinare un più piccolo sottoinsieme che mostri effetti più forti. Per poter fare ciò è necessario sacrificare alcuni dei più piccoli dettagli.

La regressione lineare non è appropriata per trattare risposte qualitative in quanto le stime ottenute potrebbero non rientrare nell'intervallo [0, 1] e quindi non riuscirebbero ad essere interpretate come misure di probabilità.

1.3 Classificazione

Il modello di regressione lineare assume che la variabile di risposta Y sia quantitativa mentre, in molti casi, la variabile di risposta risulta essere qualitativa.
Per questo motivo sorge la necessità di introdurre un nuovo metodo, chiamato classificazione, in grado di predire risposte qualitative. Il nome classificazione deriva dal fatto che predire una risposta qualitativa a partire da un'osservazione significa classificare tale osservazione, ossia essa viene assegnata ad una categoria o classe.
Così come nella regressione, anche nella classificazione si ha un insieme di addestramento $(x^{\{1\}}, y_1), ..., (x^{\{n\}}, y_n)$ che viene utilizzato per costruire un classificatore.
Quest'ultimo deve essere in grado di funzionare bene non solo con i dati di addestramento ma anche con nuovi dati osservati.

Nella classificazione, la predizione ottenuta $G(x)$ prende valori da un insieme discreto $\mathcal{G}$. Per questo motivo è sempre possibile dividere lo spazio di input in una collezione di regioni etichettate in base alla classificazione. I bordi di queste regioni posso essere irregolari o lisci a seconda del tipo di funzione di predizione che si ha.

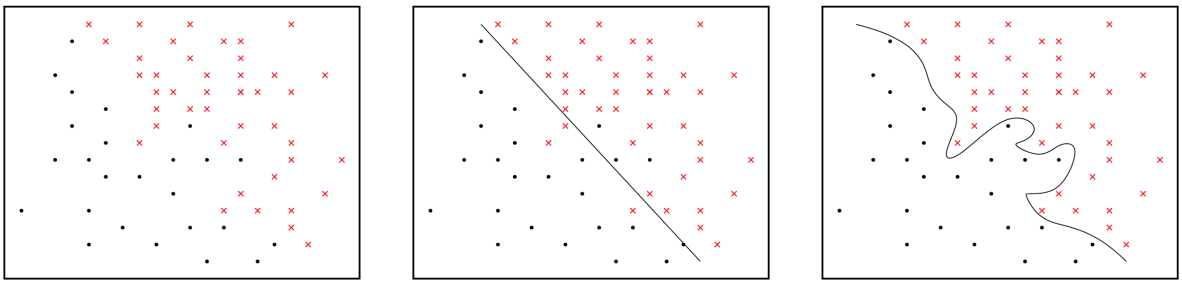


Figura 1.4: Esempio di classificazione binaria. Nell'immagine di sinistra è raffigurato l'insieme di addestramento, al centro è mostrato un esempio di classificatore lineare che separa le due classi mediante una linea retta. Si noti che sei elementi del training set (tre rossi e tre neri) sono classificati male. A destra, le classi sono separate da una curva più complessa e non ci sono errori di classificazione. (Immagine presa da [14])

Esiste un'importante classe di metodi in cui questi bordi, detti *superfici di decisione*, sono lineari e tali metodi sono definiti metodi lineari per la classificazione.
Vi sono differenti metodi attraverso cui queste superfici possono essere individuate.
Si supponga vi siano K classi, etichettate con $1, 2, ..., K$, e che il modello lineare per il k-esimo indicatore sia $\hat{f}_k(x) = \hat{\beta}_{k_0} + \hat{\beta}_k^T x$.
La superficie di decisione tra la classe k e l è quell'insieme di punti per i quali $\hat{f}_k(x) = \hat{f}_l(x)$, ossia l'insieme affine o iperpiano $\{x : (\hat{\beta}_{k_0} - \hat{\beta}_{l_0}) + (\hat{\beta}_k - \hat{\beta}_l)^T(x) = 0\}$.
Dal momento che quanto appena detto rimane valido per qualunque coppia di classi, lo spazio di input viene suddiviso in regioni di classificazione costante.
Questo approccio fa parte di una classe di metodi che modella le *funzioni discriminanti* $\delta_k(x)$ per ciascuna classe, e che quindi classifica x come appartenente alla classe con il più grande valore della relativa funzione discriminante.
I metodi che modellano le probabilità a posteriori $Pr(G = k|X = x)$ sono contenuti in questa classe.
Se $\delta_k(x)$ o $Pr(G = k|X = x)$ sono lineari rispetto alla variabile x, allora le superfici di decisone saranno lineari.
In realtà, ciò che viene richiesto è che qualche trasformazione monotona di δ_k o $Pr(G = k|X = x)$ sia lineare affinchè anche le superfici di decisione lo siano.

Per esempio, se vi sono due classi, un comune metodo per calcolare le probabilità a posteriori è dato da

$$Pr(G = 1|X = x) = \frac{exp(\beta_0 + \beta^T x)}{1 + exp(\beta_0 + \beta^T x)} \tag{1.14}$$

$$Pr(G = 2|X = x) = \frac{1}{1 + exp(\beta_0 + \beta^T x)}. \tag{1.15}$$

In questo caso la trasformazione monotona è data dalla funzione *logit*, definita come

$$logit(p) = log\left(\frac{p}{1-p}\right), \tag{1.16}$$

e infatti si vede che

$$log\frac{Pr(G = 1|X = x)}{Pr(G = 2|X = x)} = \beta_0 + \beta^T x. \tag{1.17}$$

La superficie di decisione è l'insieme dei punti per i quali i *log-odds* sono zero e tale insieme è un iperpiano definito come $\{x : \beta_0 + \beta^T x = 0\}$.

1.3.1 Regressione logistica

Uno dei metodi di classificazione maggiormente utilizzati è la regressione logistica. Il modello di regressione logistica nasce con l'obiettivo di modellare le probabilità a posteriori delle K classi mediante funzioni lineari in x che garantiscano che la somma delle probabilità sia uno e che ciascuna di esse rimanga all'interno dell'intervallo [0,1]. Tale modello ha la seguente forma

$$log\frac{Pr(G=1|X=x)}{Pr(G=K|X=x)} = \beta_{10} + \beta_1^T x \tag{1.18}$$

$$log\frac{Pr(G=2|X=x)}{Pr(G=K|X=x)} = \beta_{20} + \beta_2^T x \tag{1.19}$$

$$\vdots$$

$$log\frac{Pr(G=K-1|X=x)}{Pr(G=K|X=x)} = \beta_{(K-1)0} + \beta_{K-1}^T x. \tag{1.20}$$

Questo modello è specificato in termini di $K-1$ log-odds o trasformazioni di logit. Sebbene il modello usi l'ultima classe come denominatore nei rapporti di probabilità, la scelta del denominatore è arbitraria in quanto le stime sono equivalenti sotto questa scelta.
Un semplice calcolo mostra che

$$Pr(G=k|X=x) = \frac{exp(\beta_{k_0} + \beta_k^T x)}{1 + \sum_{l=1}^{K-1} exp(\beta_{l_0} + \beta_l^T x)}, \quad k = 1, ..., K-1 \tag{1.21}$$

$$Pr(G=K|X=x) = \frac{1}{1 + \sum_{l=1}^{K-1} exp(\beta_{l_0} + \beta_l^T x)} \tag{1.22}$$

in cui la somma è chiaramente uno.
Per enfatizzare la dipendenza dall'insieme dei parametri $\theta = \{\beta_{10}, \beta_1^T, ..., \beta_{(K-1)0}, \beta_{K-1}^T\}$, le probabilità possono essere denotate come $Pr(G=k|X=x) = p_k(x;\theta)$.
Quando $K=2$, questo modello è particolarmente semplice siccome vi è solo una funzione lineare.

I metodi di regressione logistica solitamente vengono utilizzati per la massima verosimiglianza avvalendosi della probabilità condizionata di G su X.

La funzione di *log-verosimiglianza* per N osservazioni è

$$l(\theta) = \sum_{\mathrm{i=1}}^{\mathrm{N}} logp_{g_i}(x^{\{i\}};\theta), \tag{1.23}$$

dove $p_k(x^{\{i\}};\theta) = Pr(G = k|X = x^{\{i\}};\theta)$.
In questa sezione viene discusso in dettaglio il caso con due classi siccome l'algoritmo si semplifica notevolmente. È conveniente codificare le due classi g_i attraverso una risposta binaria (0/1) y_i in cui $y_i = 1$ quando $g_i = 1$ e $y_i = 0$ quando $g_i = 2$.
Siano $p_1(x;\theta) = p(x;\theta)$ e $p_2(x;\theta) = 1 - p(x;\theta)$.
La funzione log-verosimiglianza può essere riscritta come

$$l(\beta) = \sum_{\mathrm{i=1}}^{\mathrm{N}} \Big\{ y_i logp(x^{\{i\}};\beta) + (1-y_i)log(1-p(x^{\{i\}};\beta)) \Big\} \tag{1.24}$$

$$= \sum_{\mathrm{i=1}}^{\mathrm{N}} \Big\{ y_i \beta^T x^{\{i\}} - log(1+e^{\beta^T x^{\{i\}}})) \Big\}. \tag{1.25}$$

Qui $\beta = \{\beta_{10}, \beta_1\}$ e si assume che il vettore di inputs $x^{\{i\}}$ includa il termine costante 1 per adattarsi all'intercetta.
Per massimizzare la funzione log-verosimiglianza si pongono le sue derivate uguali a zero, ossia

$$\frac{\partial l(\beta)}{\partial \beta} = \sum_{\mathrm{i=1}}^{\mathrm{N}} x^{\{i\}}(y_i - p(x^{\{i\}};\beta)) = 0, \tag{1.26}$$

che corrisponde a $p+1$ equazioni non lineari in β.

Per risolvere le equazioni 1.26 si utilizza il metodo Newton-Raphson, o metodo delle tangenti, che richiede il calcolo delle derivate seconde o della matrice Hessiana,

$$\frac{\partial^2 l(\beta)}{\partial \beta \partial \beta^T} = -\sum_{\mathrm{i=1}}^{\mathrm{N}} x^{\{i\}} {x^{\{i\}}}^T p(x^{\{i\}};\beta)(1-p(x^{\{i\}};\beta)). \tag{1.27}$$

Partendo da β^{old}, la soluzione dell'algoritmo di Newton è

$$\beta^{new} = \beta^{old} - \left(\frac{\partial^2 l(\beta)}{\partial \beta \partial \beta^T} \right)^{-1} \frac{\partial l(\beta)}{\partial \beta}, \tag{1.28}$$

dove le derivate sono valutate in β^{old}.

È conveniente scrivere i risultati ottenuti e l'Hessiana con le notazioni matriciali; sia $\mathbf{y}$ il vettore costituito dai valori y_i, sia $\mathbf{X}$ la matrice Nx$(p+1)$ degli $x^{\{i\}}$, $\mathbf{p}$ il vettore delle distribuzioni di probabilità adattate con i-esimo elemento $p(x^{\{i\}}; \beta^{old})$ e $\mathbf{N}$ una matrice NxN diagonale di pesi con i-esimo elemento diagonale $p(x^{\{i\}}; \beta^{old})(1 - p(x^{\{i\}}; \beta^{old}))$.
In questo modo si ottiene

$$\frac{\partial l(\beta)}{\partial \beta} = \mathbf{X}^T(\mathbf{y} - \mathbf{p}) \tag{1.29}$$

$$\frac{\partial^2 l(\beta)}{\partial \beta \partial \beta^T} = -\mathbf{X}^T\mathbf{W}\mathbf{X} \tag{1.30}$$

La soluzione dell'algoritmo di Newton è

$$\begin{aligned}
\beta^{new} &= \beta^{old} - (\mathbf{X}^T\mathbf{W}\mathbf{X})^{-1}\mathbf{X}^T(\mathbf{y} - \mathbf{p}) && (1.31)\\
&= (\mathbf{X}^T\mathbf{W}\mathbf{X})^{-1}\mathbf{X}^T\mathbf{W}(\mathbf{X}\beta^{old} + \mathbf{W}^{-1}(\mathbf{y} - \mathbf{p})) && (1.32)\\
&= (\mathbf{X}^T\mathbf{W}\mathbf{X})^{-1}\mathbf{X}^T\mathbf{W}\mathbf{z} && (1.33)
\end{aligned}$$

Nella seconda e terza riga la soluzione ottenuta è stata riscritta utilizzando i minimi quadrati ponderati con

$$\mathbf{z} = \mathbf{X}\beta^{old} + \mathbf{W}^{-1}(\mathbf{y} - \mathbf{p}). \tag{1.34}$$

Queste equazioni vengono risolte in modo ripetitivo, dal momento che ad ogni iterazione $\mathbf{p}$ cambia, così come $\mathbf{W}$ e $\mathbf{z}$.
Questo metodo è chiamato metodo dei *minimi quadrati iterativamente ripesati* (IRLS) in quanto in ciascuna iterazione si risolve il problema dei minimi quadrati riponderati:

$$\beta^{new} \leftarrow arg\min_{\beta}(\mathbf{z} - \mathbf{X}\beta)^T\mathbf{W}(\mathbf{z} - \mathbf{X}\beta). \tag{1.35}$$

Sembra che $\beta = 0$ sia un buon valore iniziale per il metodo iterativo, anche se la convergenza non è mai garantita.

Solitamente l'algoritmo converge, in quanto la log-verosimiglianza è una funzione concava, ma po' accadere l'overshooting che si verifica quando il valore in uscita supera il valore finale che si vuole predire. L'overshoot, o sovraelongazione, è definito come la differenza tra il massimo valore di scostamento in uscita ed il valore di regime, il tutto espresso solitamente in percentuale.
Nei rari casi in cui la log-verosimiglianza decresce, la convergenza è garantita dimezzando il numero di passi.
I modelli di regressione logistica sono utilizzati principalmente come analisi dati in cui l'obiettivo centrale è comprendere il ruolo delle variabili di input nello spiegare il risultato ottenuto.
La produzione di enormi quantità di dati e l'elevato tempo impiegato dall'unità di elaborazione per generare i risultati crea numerose problematiche durante il processo di analisi legato alla fase di classificazione.
Per questo motivo, la classificazione di insiemi di dati costituiti da un'elevata cardinalità porta a problematiche di notevole importanza. Considerando le attuali tecniche di classificazione, la maggior parte di esse ha la necessità di utilizzare la memoria principale dell'unità di elaborazione per consentire l'apprendimento del classificatore, ma, dato il numero elevato di dati, occorre utilizzare delle memorie di capacità sempre maggiori e questo non è sempre possibile, dato anche l'elevato costo.
Quindi, maggiore è la quantità di dati, maggiore sarà la quantità di memoria richiesta per poterne effettuare la classificazione.
Per questo motivo, se fosse possibile effettuare la classificazione con la memoria disponibile sull'unità elaborativa, bisognerebbe stimare l'effettivo tempo impiegato nella classificazione.
Inoltre, per procedere alla classificazione, occorre dapprima la costruzione di un modello in grado di classificare i dati e questo introduce ulteriori problematiche che si aggiungono a quelle presentate in precedenza.
Oltre a tenere conto del tempo impiegato dal processo di classificazione, bisogna quindi considerare anche il tempo necessario per la costruzione di questo modello.

1.4 Metodi Machine Learning per l'apprendimento supervisionato

È possibile individuare altri metodi Machine Learning per l'apprendimento supervisionato, oltre a quelle precedentemente descritti, tra cui

- *metodi k-nearest neighbor*: tali metodi, dato un input x, per determinare $\hat{Y}$ individuano le osservazioni più vicine a x provenienti dall'insieme di addestramento. In particolare, il modello per determinare $\hat{Y}$ è definito nel modo seguente

$$\hat{Y}(x) = \frac{1}{k} \sum_{x^{\{i\}} \in N_k(x)} y_i, \tag{1.36}$$

  dove $N_k(x)$ è l'intorno di x definito dai k punti più vicini ad $x^{\{i\}}$ nell'insieme di addestramento.

- *regressione locale*: tale regressione rientra tra i *metodi kernel* e ha come obiettivo quello di fornire in modo esplicito stime della funzione di regressione o dell'attesa condizionata specificando la natura dell'intorno locale e della classe di funzioni regolari utilizzate localmente. L'intorno locale è specificato dalla *funzione kernel* $K_\lambda(x^{\{0\}}, x)$ che assegna pesi ai punti x in una regione attorno a $x^{\{0\}}$.

- *regressione ridge*: tale regressione fa parte dei *metodi di regolarizzazione*, metodi continui e non soggetti ad alta irregolarità. La regressione ridge regolarizza i coefficienti di regressione imponendo una penalità sulla loro dimensione.

- *lasso* (operatore di selezione e ritiro assoluto minimo): anch'esso fa parte dei *metodi di regolarizzazione* e riduce i coefficienti di regressione verso lo zero penalizzando il modello di regressione con un termine di penalità che è la somma dei coefficienti assoluti di regressione.

Capitolo 2

Deep Learning

Il Deep Learning è una branca del Machine Learning che studia l'apprendimento automatico delle macchine tramite l'utilizzo delle *reti neurali*.
L'apprendimento profondo consiste in un insieme di tecniche basate su reti neurali artificiali organizzate su diversi strati, ciascuno dei quali calcola i valori per quello successivo affinchè l'informazione venga elaborata in maniera sempre più completa.

In questo capitolo si analizza il ruolo svolto dalle reti neurali nell'apprendimenti profondo. In particolare, viene descritta la loro struttura architettonica, alla cui base vi sono due diverse tipologie di neuroni.
Infine, vengono introdotti il concetto di funzione costo, relativo al processo di apprendimento della rete neurale, e l'algoritmo di retropropagazione, utilizzato per l'ottimizzazione dei pesi della rete.

2.1 Reti neurali

I contenuti di questo paragrafo sono tratti da [5].

Le reti neurali permettono alle macchine di estrarre informazioni molto complesse da un insieme di dati, rendendo possibile lo svolgimento di compiti molto complicati. Questo avviene mediante l'esecuzione di operazioni sequenziali sui dati di input che consistono

in trasformazioni lineari seguite da funzioni non lineari [8].
Una rete neurale standard è costituita da semplici processori collegati tra loro, chiamati neuroni artificiali, ciascuno dei quali produce una sequenza di valori reali di attivazione [6].
Il termine "reti neurale" deriva dal fatto che esse si ispirano al funzionamento biologico del cervello umano che elabora informazioni attraverso una rete di neuroni.
In figura 2.1 è possibile osservare un parallelismo tra una rete neurale biologica e una rete neurale artificiale.
Un neurone biologico, infatti, riceve in ingresso segnali da diversi altri neuroni tramite connessioni sinaptiche e li integra. Se l'attivazione che ne risulta supera una certa soglia, viene generato un potenziale d'azione che si propaga attraverso il suo assone ad uno o più neuroni.

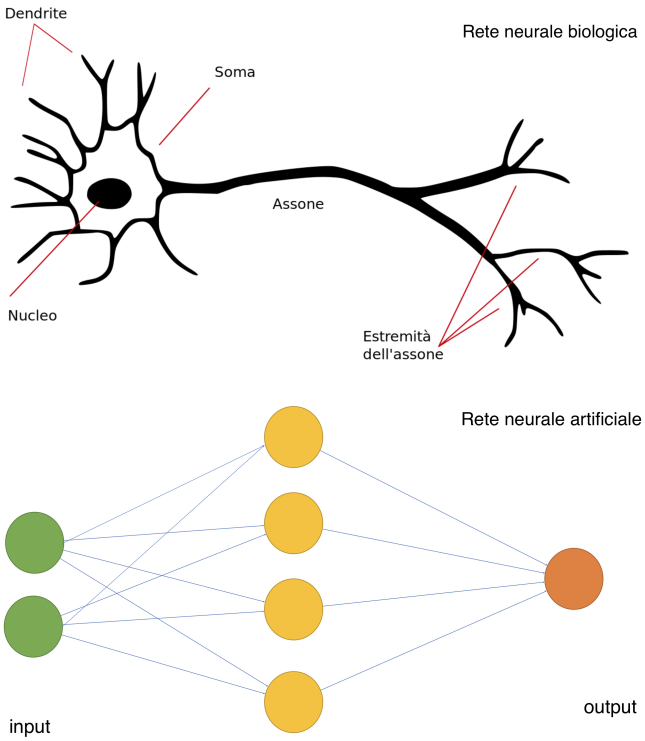


Figura 2.1: Confronto tra una rete neurale biologica e una rete neurale artificiale. (Immagine presa da [10])

Il punto di partenza è capire che la logica non è sempre necessaria per comprendere. Infatti, si immagini un bambino di pochi mesi: se la mamma gli sorride, lui risponde con un altro sorriso proprio perchè ha capito che la madre sta sorridendo. Nessuna glielo ha insegnato, ha semplicemente imparato, nel corso del tempo, a riconoscerlo e a riprodurlo. Il fattore che interviene in questo processo di apprendimento è rappresentato dalla pura e semplice esperienza che offre al cervello i dati necessari per comprendere.
Il bambino, tramite l'esperienza, impara, ad esempio, a riconoscere e distinguere i diversi toni di voce, a compiere una determinata azione piuttosto che un'altra.

Nei software "neuronali" il programmatore svolge la medesima funzione attraverso l'immissione nella macchina di dati conosciuti. L'esperienza all'interno della macchina e' data dalla modifica dei parametri di riferimento dello specifico neurone al fine di ottenere un preciso risultato già noto.
Questo meccanismo consente alla macchina di rispondere in modo corretto anche davanti a dati totalmente nuovi e mai visti.
La rete neurale, apprendendo attraverso l'esperienza e la lettura dei dati, costruisce architetture gerarchiche e fornisce livelli avanzati di input-output.
La macchina, pertanto, non viene programmata quanto piuttosto addestrata tramite l'apprendimento supervisionato, non supervisionato e per rinforzo.

2.1.1 Struttura delle reti neurali

Le reti neurali su cui si basano tutti gli sviluppi e gli studi degli ultimi anni e che stanno spopolando nel campo del Deep Learning sono le *feed-forward neural networks* (FFNN), dette anche "backpropagation" neural networks, così formalmente definite

Definizione 1. *La feed-forward neural network è una rete neurale artificiale i cui i vertici sono numerati, permettendo così a tutte le connessioni di andare da un vertice ad un altro di numero maggiore senza formare cicli.*
In particolare, i vertici sono strutturati in strati, con connessioni solamente verso strati più alti, ed il segnale viaggia sempre in avanti dallo strato di ingresso a quello di uscita.

Tal definizione evidenzia il fatto che i nodi sono raggruppati in *strati* e formano una composizione gerarchica. Infatti, la struttura delle reti neurali così delineata si articola su più livelli, come mostrato in figura 2.2:

- il livello più a sinistra della rete è chiamato *strato di input* o *input layer* ed i neuroni al suo interno sono detti *neuroni di input*
- il livello più a destra della rete è chiamato *strato di output* o *output layer* e contiene *neuroni di output*
- i livelli centrali sono chiamati *strati latenti* o *hidden layers* ed i neuroni in questi strati non sono nè di input nè di output.

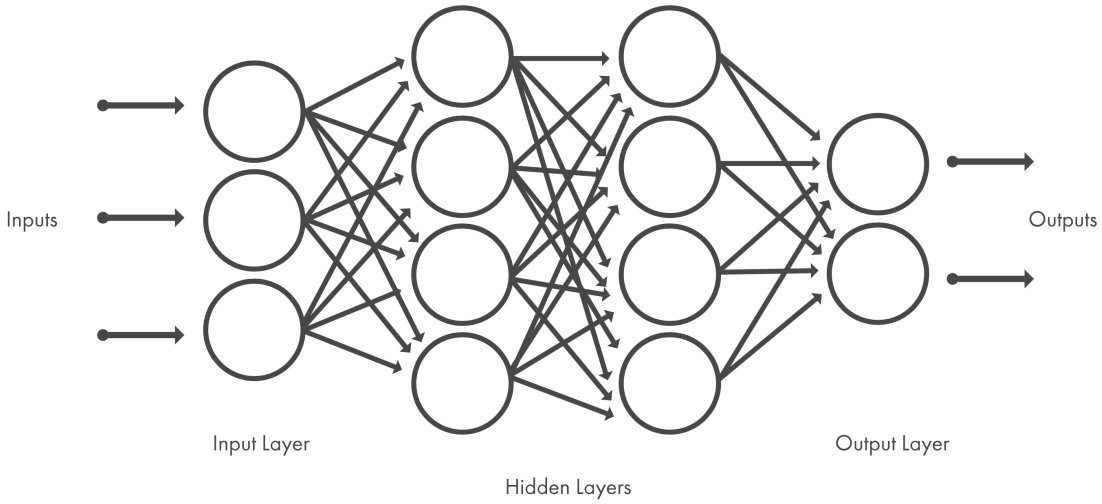


Figura 2.2: Tipica architettura di una rete neurale in cui è possibile osservare uno strato di input, due strati latenti ed uno strato di output. (Immagine presa da [11])

L'output di uno strato è utilizzato come input per lo strato successivo.
Le connessioni sono unidirezionali e sono rappresentate da archi, ai quali è permesso solamente il collegamento tra nodi di uno strato e quelli dello strato successivo. Inoltre, ad ogni arco è associato un numero reale chiamato *peso* o *weight* usato per esprimere l'importanza dell'input relativo all'output risultante [18].

È possibile distinguere due diverse tipologie di neuroni artificiali: il *percettrone* ed il *neurone sigmoidale* o *sigmoid neuron*.

I seguenti contenuti sono tratti da [7] e [3].

Il *percettrone* è considerato il neurone artificiale più semplice.
Esso è un classificatore binario che mappa i suoi inputs $x^{\{1\}}, x^{\{2\}}, ..., x^{\{n\}}$ in un singolo valore binario di output y, come è possibile osservare in figura 2.3.

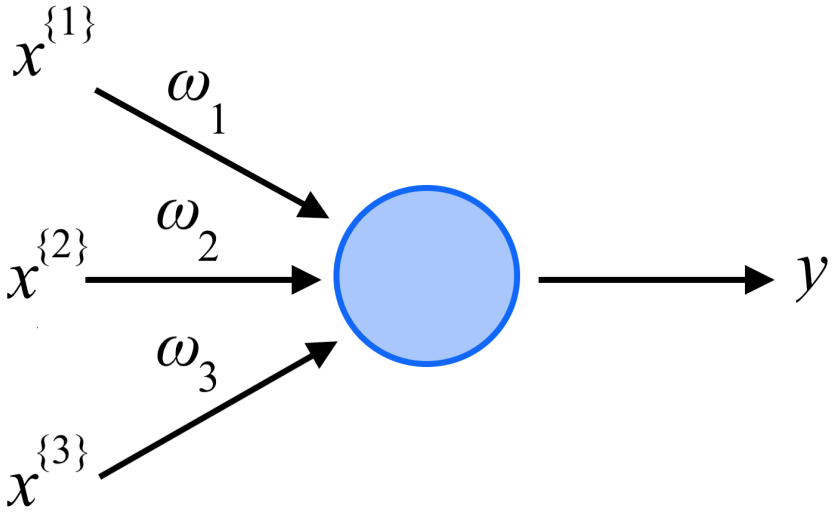


Figura 2.3: Percettrone che riceve in ingresso tre valori di input $x^{\{1\}}, x^{\{2\}}, x^{\{3\}}$ insieme ai relativi pesi w_1, w_2, w_3 e restituisce un unico valore di output y.

Esso può essere considerato come il più semplice modello di rete neurale *feed-forward* in quanto gli input alimentano direttamente l'unità di output attraverso connessioni pesate, determinando in tal modo la propagazione degli impulsi in avanti (unica direzione resa possibile).
Il campo di applicazione del percettrone è limitato al riconoscimento di forme, alla loro classificazione in gruppi separati e al calcolo di semplici funzioni.

Ad ogni input $x^{\{i\}}$ è associato un peso w_i con $i = 1, ..., n$; i pesi sono dinamici e ciò permette alla macchina di apprendere con modalità sommariamente simili alle reti neurali biologiche, anche se in maniera più elementare.
Il valore di output può essere 0 o 1 ed è determinato, oltre che dagli inputs e dai corrispettivi pesi, da un numero reale detto *valore di soglia* o *threshold value*. Tale valore deve essere superato dalla somma pesata degli inputs affinchè il dispositivo sia attivo.

In particolare,

$$output = \begin{cases} 0 & \text{se } \sum_{j=1}^{n} w_j x^{\{j\}} \leq -b \\ 1 & \text{se } \sum_{j=1}^{n} w_j x^{\{j\}} > -b \end{cases} \tag{2.1}$$

Apportando modifiche al vettore dei pesi $w = (w_1, \ldots, w_n)$ e al valore di soglia è possibile modulare l'output di un percettrone, ottenendo così diversi tipi di modelli.

La formula (2.1) può essere semplificata considerando $w = (w_1, \ldots, w_n)$ come vettore dei pesi e $x = (x^{\{1\}}, \ldots, x^{\{n\}})$ come vettore di input.
In questo modo si ottiene

$$output = \begin{cases} 0 & \text{se } w \cdot x + b \leq 0 \\ 1 & \text{se } w \cdot x + b > 0 \end{cases} \tag{2.2}$$

dove $w \cdot x = \sum_j w_j x^{\{j\}}$ utilizzando il prodotto scalare.

Il termine b è detto *bias* e può essere considerato come una misurazione della probabilità di ottenere 1 dall'output del percettrone. Se b è positivo, è estremamente semplice ottenere 1 come valore di output mentre, se b risulta molto negativo, è più difficile.

Si supponga che si abbia un problema da risolvere e che si voglia utilizzare una rete costituita da percettroni per determinarne la soluzione.
Per fare ciò, si applicano piccoli cambiamenti nei pesi e nei biases con l'obiettivo di ottenere piccoli cambiamenti anche nell'output. Questo permetterebbe alla macchina di apprendere e quindi sarebbe possibile farla comportare nel modo migliore per riuscire ad ottenere outputs sempre più vicini alla soluzione del problema.

Ciò, però, non avviene quando si utilizzano i percettroni in quanto un piccolo cambiamento nei pesi o nei biases di un singolo percettrone all'interno della rete può provocare un grande cambiamento nell'output, facendolo spostare da 0 a 1.
Questo salto potrebbe comportare una totale variazione in tutta la rete, non permettendo così un avvicinamento al risultato desiderato.

È possibile risolvere questo problema introducendo un nuovo tipo di neurone artificiale chiamato *neurone sigmoidale*.
Tale neurone è simile al percettrone ma riveste un'importante ruolo all'interno della rete neurale in quanto piccoli cambiamenti nei pesi o nei biases provocano sempre piccoli cambiamenti nel relativo output.
Il neurone sigmoidale, così come il percettrone, ha come inputs i valori $x^{\{1\}}, x^{\{2\}}, \ldots, x^{\{n\}}$ e a ciascun $x^{\{j\}}$ viene associato un peso w_j e un bias complessivo b.
Come output, invece, può assumere valori compresi tra 0 e 1, anzichè solo 0 o 1.
In particolare, il valore dell'output è dato da $\sigma(wx + b)$ dove σ è chiamata *funzione sigmoidea* o *sigmoid function* ed è definita come

$$\sigma(z) \equiv \Big(\frac{1}{1 + e^{-\mathrm{z}}}\Big). \tag{2.3}$$

Quindi, l'output di un sigmoid neuron è dato da

$$\frac{1}{1 + \exp(-w \cdot x - b)}. \tag{2.4}$$

Di particolare interesse è il grafico di σ in quanto permette di cogliere il legame tra neuroni sigmoidali e percettroni, sebbene, a prima vista, appaiano molto diversi tra loro.
La funzione sigmoidea, come mostrato in figura 2.4, può essere considerata come una versione liscia di una funzione a scala.
Se σ fosse stata una funzione a scala, allora il neurone sigmoidale sarebbe stato un percettrone in quanto il valore di output sarebbe stato 0 o 1 in base al valore assunto da $w \cdot x + b$. In particolare,

- con $w \cdot x + b$ negativo si sarebbe ottenuto 0
- con $w \cdot x + b$ positivo si sarebbe ottenuto 1.

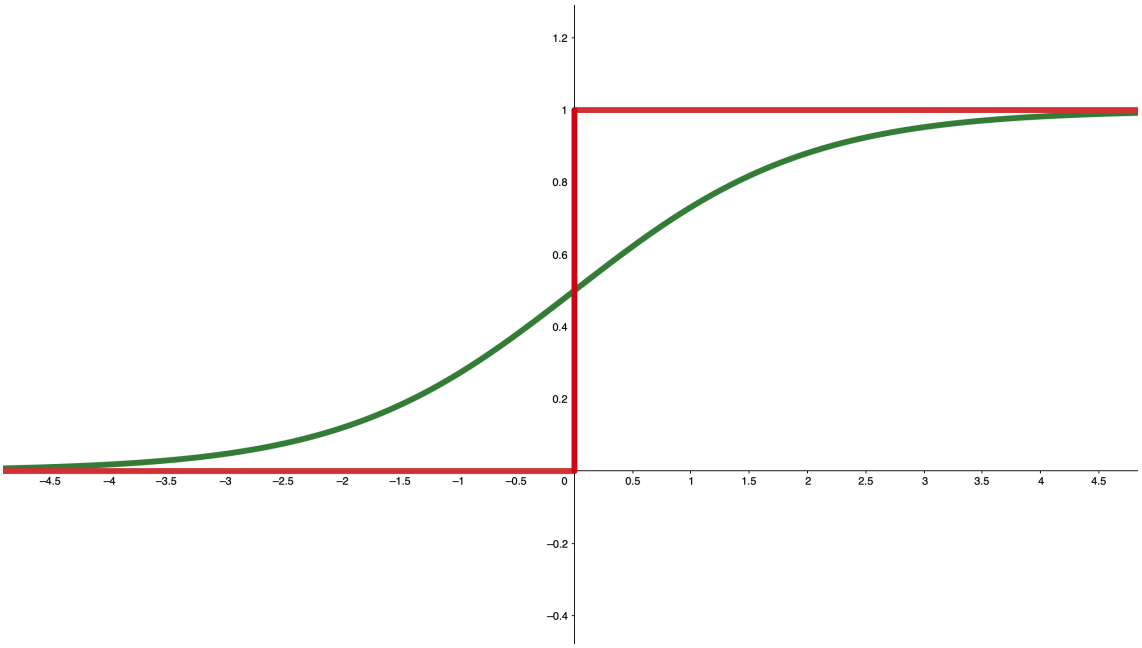


Figura 2.4: La curva verde rappresenta il plot della funzione sigmoidea σ mentre la curva rossa raffigura una funzione a scala.

La derivata della funzione sigmoidea assume una forma molto semplice data da

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)). \tag{2.5}$$

La proprietà più importante di cui gode σ è l'uniformità, espressa nel modo seguente: a piccoli cambiamenti nei pesi Δw_j e nel bias Δb corrisponde una piccola variazione nell'output $\Delta output$, ossia

$$\Delta output \approx \sum_j \frac{\partial output}{\partial w_j} \Delta w_j + \frac{\partial output}{\partial b} \Delta b. \tag{2.6}$$

$\Delta output$ è quindi una funzione lineare nelle variabili Δw_j e Δb e tale linearità permette di scegliere piccoli cambiamenti nei pesi e nei biases per ottenere qualunque piccolo cambiamento desiderato nell'output.

Per comodità è possibile interpretare la funzione sigmoidea in forma vettoriale. Se $z \in \mathbb{R}^m$, σ: $\mathbb{R}^m \to \mathbb{R}^m$ è definita applicando la funzione sigmoidea componente per componente nel seguente modo: $(\sigma(z))_{\mathrm{i}} = \sigma(z_{\mathrm{i}})$.

Con questa notazione è possibile organizzare diversi livelli di neuroni.
In ciascun livello ogni neurone produce un singolo numero reale che viene mandato ad ogni neurone del livello successivo. Qui ciascun neurone forma la propria combinazione pesata di questi valori, aggiungendo il proprio bias e applicando la funzione sigmoidea, e il valore di output ottenuto viene a sua volta mandato a ciascun neurone del livello successivo.
In particolare, se i numeri reali prodotti dai neuroni in un determinato livello vengono raccolti in un unico vettore a, allora il vettore degli outputs prodotto nel livello successivo ha la forma $\sigma(Wa + b)$, dove W è una matrice contenente i pesi mentre b è un vettore di biases. Il numero di colonne in W corrisponde al numero di neuroni che hanno prodotto il vettore a nel livello precedente, mentre il numero di righe di W, così come il numero delle componenti del vettore b, corrisponde al numero di neuroni presenti nel livello attuale.

Si supponga ora che la rete sia composta da L strati di cui lo strato 1 e lo strato L siano rispettivamente lo strato di input e lo strato di output mentre i restanti *L-2* strati siano latenti. Si assuma inoltre che lo strato l, con $l = 1, 2, \ldots, L$, contenga n_l neuroni, ossia che n_1 sia la dimensione dell'input iniziale.
Complessivamente, quindi, la rete mappa da $\mathbb{R}^{n_1}$ a $\mathbb{R}^{n_L}$.

La matrice dei pesi relativi allo strato l si denota con $W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$.
Più precisamente, $w^{[l]}_{ij}$ è il peso che il neurone i dello strato l applica all'output del neurone j dello strato *l-1*.
Similmente, $b^{[l]} \in \mathbb{R}^{n_l}$ è il vettore dei biases per lo strato l, quindi il neurone j dello strato l utilizza il bias dato dalla componente vettoriale $b^{[l]}_j$.
Dato un input $x \in \mathbb{R}^{n_1}$, si potrebbe ordinatamente riassumere l'azione di una rete denotando con $a^{[l]}_j$ l'output del neurone j dello strato l.
In questo modo si ottiene

$$a^{[1]} = x \in \mathbb{R}^{n_1} \tag{2.7}$$

$$a^{[l]} = \sigma(W^{[l]} a^{[l-1]} + b^{[l]}) \in \mathbb{R}^{n_l} \qquad per\ l = 2, 3, \ldots, L. \tag{2.8}$$

2.2 Funzione costo

I contenuti di questo paragrafo sono tratti da [3].

Durante la fase di apprendimento della rete è di fondamentale importanza trovare i valori ottimali dei pesi per determinare il mapping desiderato tra i valori di input e quelli di output. In particolare, se si è interessati ad addestrare una rete neurale che operi come classificatore, l'output desiderato è l'etichetta corretta della classe del pattern di input. Se invece la rete neurale deve risolvere un problema di regressione, l'output desiderato è il valore corretto della variabile dipendente, in corrispondenza del valore della variabile indipendente fornita in input.
L'obiettivo dell'algoritmo di apprendimento è quello di modificare i pesi della rete in modo da minimizzare l'errore medio sui pattern del training set, dove l'errore è dato dalla differenza tra l'output prodotto dalla rete e l'output desiderato [15].

Calcolare perfettamente i pesi per una rete, però, è altamente improbabile poichè sono molteplici le incognite da valutare e considerare.
Per ovviare a questo problema si utilizza un algoritmo in grado di navigare nello spazio dei pesi che il modello può utilizzare per compiere previsioni precise.
L'addestramento di una rete neurale artificiale e la configurazione dei suoi algoritmi di apprendimento rappresentano le sfide principali nel campo del Machine Learning.
In questa fase viene fornito in ingresso alla rete un *training set* di dati con l'obiettivo di allenarla e un *validation set* per verificare la riuscita dell'addestramento stesso. Successivamente, la rete produce in uscita un output sotto forma di vettori di risultati.
La distanza tra i risultati in uscita dalla rete e i pattern desiderati viene calcolata da una funzione, detta *funzione costo*. Per ridurre la discrepanza tra il risultato ottenuto e quello atteso, e quindi per minimizzare la funzione costo in fase di addestramento, la rete modifica i pesi, i quali fungono da regolatori della funzione entrata/uscita del sistema. Nello specifico, l'algoritmo di apprendimento crea un vettore gradiente con lo scopo di modificare il vettore dei pesi.
Tale vettore calcola l'aumentare o il diminuire dell'errore con l'incremento infinitesimale dei pesi, indicando dove l'imprecisione in uscita è minore in media.

Nel calcolare l'errore del modello durante il processo di ottimizzazione è necessario scegliere un'adeguata funzione costo. Questo può essere complicato poichè la funzione deve acquisire le proprietà del problema ed è quindi importante che la funzione rappresenti fedelmente gli obiettivi di progettazione. Non esiste una funzione costo unica per tutti gli algoritmi ma esistono vari fattori coinvolti nella scelta, quali il tipo di algoritmo di apprendimento automatico scelto, la facilità di calcolo e la percentuale di valori anomali nel set di dati.

Si considerino ora N osservazioni, o *training points*, in $\mathbb{R}^{n_1}$ identificate con $\{x^{\{i\}}\}_{i=1}^{N}$ a cui sono associati gli outputs $\{y(x^{\{i\}})\}_{i=1}^{N}$ in $\mathbb{R}^{n_L}$.

La *funzione costo quadratica Cost* che si vuole minimizzare è

$$Cost = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left\|y(x^{\{i\}}) - a^{[L]}(x^{\{i\}})\right\|_2^2. \tag{2.9}$$

Tale funzione ha come variabili i pesi e i biases mentre i training points sono fissati.

2.3 Teorema di approssimazione universale

I contenuti di questo paragrafo sono tratti da [12] e [13].

Nella teoria delle reti neurali artificiali il *teorema di approssimazione universale* afferma che una rete feed-forward con un singolo strato latente contenente un numero finito di neuroni può approssimare arbitrariamente bene funzioni continue su sottoinsiemi compatti di $\mathbb{R}^n$.
Tale teorema afferma che semplici reti neurali possono rappresentare una grande varietà di funzioni interessanti, qualora vengano assegnati parametri o pesi appropriati.
Tuttavia, esso garantisce solo l'esistenza di una rete che soddisfi determinate condizioni ma purtroppo non fornisce alcun metodo pratico per trovarne la struttura o i parametri corrispondenti.
Per questo motivo non è un teorema costruttivo ma è solo un teorema di esistenza.

L'enunciato canonico del teorema di approssimazione universale è il seguente:

Teorema 2.3.1 (Teorema di approssimazione universale)**.** *Sia $\varphi : \mathbb{R} \to \mathbb{R}$ una funzione continua, non costante, limitata e monotona crescente.*
Sia I_m un ipercubo unitario m-dimensionale $[0,1]^m$.
Lo spazio delle funzioni continue a valori reali su I_m si denota con $C(I_m)$.
Allora, date una qualunque funzione $f \in C(I_m)$ e una $\varepsilon > 0$, esistono un intero N, delle costanti reali $v_i, b_i \in \mathbb{R}$ e dei vettori reali $w_i \in \mathbb{R}^m$, con $i = 1, ..., N$, tali che è possibile definire

$$F(x) = \sum_{i=1}^{N} v_i \varphi(w_i^T x + b_i) \tag{2.10}$$

come una realizzazione approssimata della funzione f.
In particolare,

$$|F(x) - f(x)| < \varepsilon \qquad \forall x \in I_m.$$

In altre parole, le funzioni della forma $F(x)$ sono dense in $C(I_m)$.

Tale teorema rimane valido se si sostituisce I_m con un qualunque sottoinsieme compatto di $\mathbb{R}^m$.
Il teorema di approssimazione universale non spiega come calcolare l'insieme delle costanti necessarie per creare F. Esso si limita ad affermare che uno strato latente è sufficiente ad un percettrone multistrato per una ε-approssimazione di un generico training set rappresentato dall'insieme di inputs $x = \{x^{\{1\}}, ..., x^{\{m\}}\}$ e outputs desiderati $f(x^{\{1\}}, ..., x^{\{m\}})$.
L'equazione (2.10) può rappresentare l'uscita di una rete multistrato dove si denota con

- m la dimensione dello spazio del vettore di ingresso
- N la dimensione dello strato latente
- w_i i pesi che connettono lo strato di input con lo strato latente
- v_i i pesi che connettono lo strato latente con lo strato di output.

2.4 Retropropagazione

I contenuti di questo paragrafo sono tratti da [3] e [9].

La *retropropagazione*, o *backpropagation*, è uno degli algoritmi maggiormente utilizzati per addestrare le reti neurali, sia quelle a strato singolo che multi-strato. Tale algoritmo ha come obiettivo quello di modificare i pesi dei neuroni della rete per minimizzare l'errore dato dalla differenza tra il vettore di output ottenuto e quello desiderato. La maggior parte degli algoritmi di ottimizzazione numerica è iterativa e necessita del calcolo del gradiente e dell'operatore hessiano della funzione costo rispetto ai pesi e ai biases. Molto spesso, però, una rete neurale multistrato ha un numero di parametri molto elevato. In particolare, con L strati la rete possiede L matrici di parametri $\mathrm{W}^{[\mathrm{l}]}$, ognuna delle quali contiene $n_{\mathrm{l}} \times n_{\mathrm{l}+1}$ coefficienti, dove il numero di nodi $\mathrm{n_l}$ può raggiungere qualche migliaio.
Per ogni iterazione dell'algoritmo, il calcolo del gradiente richiede un numero di operazioni pari al numero di coefficienti, mentre le operazioni richieste per il calcolo dell'hessiano crescono in modo quadratico all'aumentare del numero dei parametri.

L'algoritmo di backpropagation, invece, oltre a non necessitare del calcolo dell'operatore hessiano, richiede il calcolo del gradiente solamente nell'ultimo strato, il quale viene poi propagato all'indietro per ottenere il gradiente negli strati precedenti.
L'obiettivo dell'algoritmo è determinare le derivate parziali della funzione costo sfruttando la sua struttura: dal momento che (2.9) è una combinazione lineare di termini che vengono calcolati sull'insieme di addestramento, lo stesso vale per le sue derivate parziali.
Quindi, per un fissato training point, si consideri $C_{x^{\{i\}}}$ in (3.4) come una funzione di pesi e biases e la si riscriva togliendo la dipendenza da $x^{\{i\}}$.
In questo modo si ottiene

$$C = \frac{1}{2}\left\|y - a^{[\mathrm{L}]}\right\|_2^2. \tag{2.11}$$

Si prenda ora in esame (2.8) in cui $a^{[\mathrm{L}]}$ è l'output della rete neurale artificiale.
La dipendenza di C dai pesi e dai biases proviene esclusivamente da $a^{[\mathrm{L}]}$.

Per derivare al meglio le espressioni al fine di ottenere derivate parziali, è utile introdurre due insiemi di variabili:

1. $$z^{[\mathrm{l}]} = W^{[\mathrm{l}]} a^{[\mathrm{l}-1]} + b^{[\mathrm{l}]} \in \mathbb{R}^{\mathrm{n_l}} \quad per\ l = 2, 3, \ldots, L, \tag{2.12}$$

   dove $\mathrm{z_j}^{[\mathrm{l}]}$ è l'*input ponderato* del neurone j allo strato l.
   Dunque, la relazione fondamentale (2.8) che propaga informazioni attraverso la rete può essere riscritta come

   $$a^{[\mathrm{l}]} = \sigma\big(z^{[\mathrm{l}]}\big) \qquad per\ l = 2, 3, \ldots, L. \tag{2.13}$$

2. $\delta^{[\mathrm{l}]} \in \mathbb{R}^{\mathrm{n_l}}$, definita in ogni sua componente da

   $$\delta_{\mathrm{j}}^{[\mathrm{l}]} = \frac{\partial C}{\partial z_{\mathrm{j}}^{[\mathrm{l}]}} \qquad per\ \ 1 \le j \le n_l \ \ e \ \ 2 \le l \le L. \tag{2.14}$$

   Questa espressione, chiamata spesso *errore* nel j-esimo neurone allo strato l, è una quantità intermedia. Tuttavia, l'idea di riferirsi a $\delta_{\mathrm{j}}^{[\mathrm{l}]}$ in (2.14) come un errore è dovuta al fatto che la funzione costo può essere in un punto di minimo solo se tutte le sue derivate parziali sono zero, quindi $\delta_{\mathrm{j}}^{[\mathrm{l}]} = 0$ è un buon obiettivo.
   Precisamente, $\delta_{\mathrm{j}}^{[\mathrm{l}]}$ misura la sensibilità della funzione costo sull'input ponderato del neurone j allo strato l.

È importante ora definire il *prodotto di Hadamard* tra due vettori: se $x, y \in \mathbb{R}^{\mathrm{n}}$, allora $x \circ y \in \mathbb{R}^{\mathrm{n}}$ è dato da $(x \circ y)_i = x_i y_i$.
Introdotte queste notazioni, è possibile enunciare ora un importante risultato:

Teorema 2.4.1.

$$\delta^{[\mathrm{L}]} = \sigma'(z^{[\mathrm{L}]}) \circ (a^L - y) \tag{2.15}$$

$$\delta^{[\mathrm{l}]} = \sigma'(z^{[\mathrm{l}]}) \circ (W^{[\mathrm{l}+1]})^T \delta^{[\mathrm{l}+1]} \qquad per\ \ 2 \le l \le L-1 \tag{2.16}$$

$$\frac{\partial C}{\partial b_{\mathrm{j}}^{[\mathrm{l}]}} = \delta_{\mathrm{j}}^{[\mathrm{l}]} \qquad per\ \ 2 \le l \le L \tag{2.17}$$

$$\frac{\partial C}{\partial w_{\mathrm{jk}}^{[\mathrm{l}]}} = \delta_{\mathrm{j}}^{[\mathrm{l}]} a_{\mathrm{k}}^{[\mathrm{l}-1]} \qquad per\ \ 2 \le l \le L \tag{2.18}$$

Dimostrazione. Si inizia dimostrando (2.15).
La relazione (2.13) con $l = L$ mostra che $z_{\mathrm{j}}^{[\mathrm{L}]}$ e $a_{\mathrm{j}}^{[\mathrm{L}]}$ sono connesse dalla relazione $a^{[\mathrm{L}]} = \sigma'(z^{[\mathrm{L}]})$ e perciò

$$\frac{\partial a_{\mathrm{j}}^{[\mathrm{L}]}}{\partial z_{\mathrm{j}}^{[\mathrm{L}]}} = \sigma'(z_{\mathrm{j}}^{[\mathrm{L}]}). \tag{2.19}$$

Inoltre, da (2.11)

$$\frac{\partial C}{\partial a_{\mathrm{j}}^{[\mathrm{L}]}} = \frac{\partial}{\partial a_{\mathrm{j}}^{[\mathrm{L}]}} \frac{1}{2} \sum_{\mathrm{k}=1}^{\mathrm{n_L}} (y_k - a_{\mathrm{k}}^{[\mathrm{L}]})^2 = -(y_j - a_{\mathrm{j}}^{[\mathrm{L}]}) \tag{2.20}$$

Così, usando la regola della catena, si ottiene

$$\delta_{\mathrm{j}}^{[\mathrm{L}]} = \frac{\partial C}{\partial z_{\mathrm{j}}^{[\mathrm{L}]}} = \frac{\partial C}{\partial a_{\mathrm{j}}^{[\mathrm{L}]}} \frac{\partial a_{\mathrm{j}}^{[\mathrm{L}]}}{\partial z_{\mathrm{j}}^{[\mathrm{L}]}} = (a_{\mathrm{j}}^{[\mathrm{L}]} - y_j)\sigma'(z_{\mathrm{j}}^{[\mathrm{L}]}) \tag{2.21}$$

che è la componente j-esima di (2.15).

Per mostrare (2.16) si usa la regola della catena per trasformare $z_{\mathrm{j}}^{[\mathrm{l}]}$ in $\{z_{\mathrm{k}}^{[\mathrm{l}+1]}\}_{\mathrm{k}=1}^{\mathrm{n_{l+1}}}$. Applicando tale regola e usando la definizione (2.14)

$$\delta_{\mathrm{j}}^{[\mathrm{l}]} = \frac{\partial C}{\partial z_{\mathrm{j}}^{[\mathrm{l}]}} = \sum_{\mathrm{k}=1}^{\mathrm{n_{l+1}}} \frac{\partial C}{\partial z_{\mathrm{k}}^{[\mathrm{l}+1]}} \frac{\partial z_{\mathrm{k}}^{[\mathrm{l}+1]}}{\partial z_{\mathrm{j}}^{[\mathrm{l}]}} = \sum_{\mathrm{k}=1}^{\mathrm{n_{l+1}}} \delta_{\mathrm{k}}^{[\mathrm{l}+1]} \frac{\partial z_{\mathrm{k}}^{[\mathrm{l}+1]}}{\partial z_{\mathrm{j}}^{[\mathrm{l}]}} \tag{2.22}$$

Ora, da (2.12) si sa che $z_{\mathrm{k}}^{[\mathrm{l}+1]}$ e $z_{\mathrm{j}}^{[\mathrm{l}]}$ sono in relazione tramite

$$z_{\mathrm{k}}^{[\mathrm{l}+1]} = \sum_{\mathrm{s}=1}^{\mathrm{n_l}} w_{\mathrm{ks}}^{[\mathrm{l}+1]} \sigma\big(z_{\mathrm{s}}^{[\mathrm{l}]}\big) + b_{\mathrm{k}}^{[\mathrm{l}+1]}. \tag{2.23}$$

Quindi,

$$\frac{\partial z_{\mathrm{k}}^{[\mathrm{l}+1]}}{\partial z_{\mathrm{j}}^{[\mathrm{l}]}} = w_{\mathrm{kj}}^{[\mathrm{l}+1]} \sigma'\big(z_{\mathrm{j}}^{[\mathrm{l}]}\big) \tag{2.24}$$

In (2.22), sostituendo la relazione appena trovata, si ottiene

$$\delta_{\mathrm{j}}^{[\mathrm{l}]} = \sum_{\mathrm{k}=1}^{\mathrm{n_{l+1}}} \delta_{\mathrm{k}}^{[\mathrm{l}+1]} w_{\mathrm{kj}}^{[\mathrm{l}+1]} \sigma'\Big(z_{\mathrm{j}}^{[\mathrm{l}]}\Big) \tag{2.25}$$

che potrebbe essere riscritta come

$$\delta_{\mathrm{j}}^{[\mathrm{l}]} = \sigma'\left(z_{\mathrm{j}}^{[\mathrm{l}]}\right)\left((W^{[\mathrm{l}+1]})^T \delta^{[\mathrm{l}+1]}\right)_j. \tag{2.26}$$

Questa è proprio la componente del vettore dell'equazione (2.16).

Per mostrare (2.17) si può notare da (2.12) e (2.13) che $z_{\mathrm{j}}^{[\mathrm{l}]}$ è relazionato con $b_{\mathrm{j}}^{[\mathrm{l}]}$ tramite

$$z_{\mathrm{j}}^{[\mathrm{l}]} = \left(W^{[\mathrm{l}]}\sigma\left(z^{[\mathrm{l}-1]}\right)\right)_j + b_{\mathrm{j}}^{[\mathrm{l}]}. \tag{2.27}$$

Poichè $z^{[\mathrm{l}-1]}$ non dipende da $b_{\mathrm{j}}^{[\mathrm{l}]}$, si trova che

$$\frac{\partial z_{\mathrm{j}}^{[\mathrm{l}]}}{\partial b_{\mathrm{j}}^{[\mathrm{l}]}} = 1. \tag{2.28}$$

Dunque, con la regola della catena, si ottiene che

$$\frac{\partial C}{\partial b_{\mathrm{j}}^{[\mathrm{l}]}} = \frac{\partial C}{\partial z_{\mathrm{j}}^{[\mathrm{l}]}}\frac{\partial z_{\mathrm{j}}^{[\mathrm{l}]}}{\partial b_{\mathrm{j}}^{[\mathrm{l}]}} = \frac{\partial C}{\partial z_{\mathrm{j}}^{[\mathrm{l}]}} = \delta_{\mathrm{j}}^{[\mathrm{l}]} \tag{2.29}$$

usando la definizione (2.14). Questo dà (2.17).

Infine, per ottenere (2.18), si considera la componente j-esima di $z^{[\mathrm{l}]}$ in (2.13) data da

$$z_{\mathrm{j}}^{[\mathrm{l}]} = \sum_{\mathrm{k}=1}^{\mathrm{n}_{\mathrm{l}-1}} w_{\mathrm{jk}}^{[\mathrm{l}+1]} a_{\mathrm{k}}^{[\mathrm{l}-1]} + b_{\mathrm{j}}^{[\mathrm{l}]} \tag{2.30}$$

che consente di avere

$$\frac{\partial z_{\mathrm{j}}^{[\mathrm{l}]}}{\partial w_{\mathrm{jk}}^{[\mathrm{l}]}} = a_{\mathrm{k}}^{[\mathrm{l}-1]}, \tag{2.31}$$

indipendentemente da j, e anche

$$\frac{\partial z_{\mathrm{s}}^{[\mathrm{l}]}}{\partial w_{\mathrm{jk}}^{[\mathrm{l}]}} = 0 \qquad \text{per } s \neq j. \tag{2.32}$$

In altre parole, (2.31) e (2.32) seguono dal fatto che il j-esimo neurone dello strato l usa solo i pesi della riga j di $W^{[\mathrm{l}]}$ e applica questi pesi linearmente.
Allora, tramite la regola della catena, (2.31) e (2.32) danno

$$\frac{\partial C}{\partial w_{\mathrm{jk}}^{[\mathrm{l}]}} = \sum_{\mathrm{s}=1}^{\mathrm{n_l}} \frac{\partial C}{\partial z_{\mathrm{s}}^{[\mathrm{l}]}} \frac{\partial z_{\mathrm{s}}^{[\mathrm{l}]}}{\partial w_{\mathrm{jk}}^{[\mathrm{l}]}} = \frac{\partial C}{\partial z_{\mathrm{j}}^{[\mathrm{l}]}} \frac{\partial z_{\mathrm{j}}^{[\mathrm{l}]}}{\partial w_{\mathrm{jk}}^{[\mathrm{l}]}} = \frac{\partial C}{\partial z_{\mathrm{j}}^{[\mathrm{l}]}} a_{\mathrm{k}}^{[\mathrm{l}-1]} = \delta_{\mathrm{j}}^{[\mathrm{l}]} a_{\mathrm{k}}^{[\mathrm{l}-1]}, \tag{2.33}$$

dove nell'ultimo passaggio si è usato la definizione di $\delta_{\mathrm{j}}^{[\mathrm{l}]}$ in(2.14). □

Dal teorema appena enunciato segue che l'output $a^{[L]}$ può essere valutato tramite un *passo in avanti*, attraverso la rete, che permette di calcolare $a^{[1]}, z^{[2]}, a^{[2]}, z^{[3]}, ..., a^{[L]}$ in questo ordine. Avendo fatto ciò, $\delta^{[L]}$ è immediatamente disponibile.
Inoltre, $\delta^{[L-1]}, \delta^{[L-2]}, ..., \delta^{[2]}$ possono essere calcolati tramite un *passo all'indietro*.
In questo modo si ha accesso alle derivate parziali.

L'algoritmo di backpropagation alterna quindi due passi in modo iterativo:

1. *il passo in avanti*, che corrisponde a punto 1 dell'algoritmo sottostante
2. *il passo all'indietro* che, a sua volta, comprende una *fase di propagazione*, che corrisponde ai punti 2-4, ed una *fase di aggiornamento* che corrisponde al punto 5.

Algoritmo di Backpropagation

1. Calcolare il valore dei nodi $a^{[l]}$ per ogni strato $l = 2, \ldots, L$ utilizzando i valori correnti di $W^{[l]}$ e $b^{[l]}$;

2. Per lo strato $l = L$, calcolare

$$\delta^{[L]} = \sigma'(z^{[L]}) \circ (a^L - y); \tag{2.34}$$

3. Per gli strati latenti $l = L - 1, \ldots, 2$ ricavare

$$\delta^{[l]} = \sigma'(z^{[l]}) \circ (W^{[l+1]})^T \delta^{[l+1]}; \tag{2.35}$$

4. Avendo $\delta^{[2]}, \ldots, \delta^{[L]}$ è possibile ottenere le derivate parziali come

$$\frac{\partial C}{\partial w^{[l]}_{jk}} = \delta^{[l]}_j a^{[l-1]}_k \qquad e \qquad \frac{\partial C}{\partial b^{[l]}_j} = \delta^{[l]}_j; \tag{2.36}$$

5. Aggiornare i parametri $W^{[l]}$ e $b^{[l]}$ usando la discesa del gradiente;

6. Ricominciare con una nuova iterazione dal passo l utilizzando i nuovi valori per i parametri $W^{[l]}$ e $b^{[l]}$.

Capitolo 3

Discesa del gradiente e algoritmi per l'ottimizzazione di SGD

Quest'ultimo capitolo confronta i tre differenti metodi di discesa del gradiente che vengono utilizzati per trovare i punti di ottimo della funzione costo. Per ciascuno di essi viene descritto il funzionamento, insieme ai punti di forza e di debolezza.
Infine, vengono introdotti alcuni algoritmi di ottimizzazione del metodo SGD che consentono una più veloce convergenza al punto di minimo della funzione costo.

3.1 Discesa del gradiente

I contenuti di questo capitolo sono tratti da [3] e [4].

In ottimizzazione ed analisi numerica il *metodo di discesa del gradiente*, detto anche *metodo di discesa più ripida*, è una tecnica che consente di determinare i punti di massimo e minimo di una funzione di più variabili.
Questo metodo è il più utilizzato per minimizzare la funzione costo, e quindi ottimizzare le reti neurali, e ciò avviene aggiornando i parametri relativi alla funzione in direzione opposta rispetto al gradiente della funzione stessa.

Come detto precedentemente, addestrare una rete significa scegliere i parametri, ossia pesi e biases, che minimizzano la funzione costo. Essi assumono la forma di matrici e vettori, ma è più conveniente immaginarli memorizzati come un singolo vettore che viene indicato con p. Generalmente, si suppone $p \in \mathbb{R}^s$ e si scrive la funzione costo (2.9) come $Cost(p)$ per enfatizzare la dipendenza dai parametri, con $Cost : \mathbb{R}^s \longrightarrow \mathbb{R}$.

È possibile individuare tre varianti del metodo di discesa del gradiente che differiscono per il numero di dati utilizzati nel calcolo del gradiente della funzione. La quantità di dati di cui l'algoritmo si avvale crea un trade-off tra l'accuratezza del parametro aggiornato ed il tempo impiegato per ottenere tale aggiornamento.

3.1.1 Discesa del gradiente batch

Il metodo di discesa del gradiente convenzionale, noto anche come *metodo di discesa del gradiente batch*, funziona in modo iterativo e calcola una sequenza di vettori in $\mathbb{R}^s$ con l'obiettivo di farli convergere ad un vettore che minimizzi la funzione costo.
Partendo dal vettore p si vuole determinare una perturbazione Δp affinchè il vettore $p + \Delta p$ rappresenti un miglioramento del costo.
Se Δp è piccolo, ignorando i termini di ordine $\|\Delta p\|^2$, si ottiene

$$Cost(p + \Delta p) \approx Cost(p) + \sum_{r=1}^{s} \frac{\partial Cost(p)}{\partial p_r} \Delta p_r, \tag{3.1}$$

dove $\frac{\partial Cost(p)}{\partial p_r}$ denota la derivata parziale della funzione costo rispetto all'r-esimo parametro.
Per convenzione, il vettore delle derivate parziali si indica con $\nabla Cost(p) \in \mathbb{R}^s$ ed è chiamato *gradiente*. Con questa notazione si ha che

$$(\nabla Cost(p))_r = \frac{\partial Cost(p)}{\partial p_r}$$

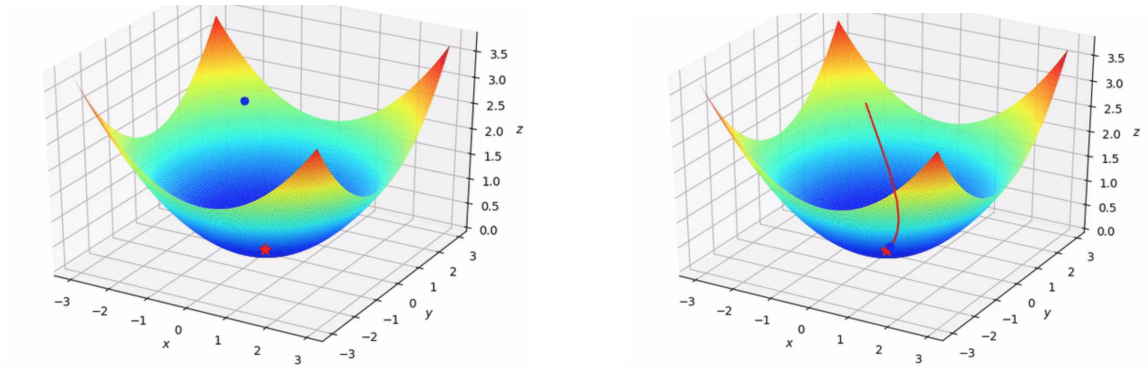
e quindi la formula (3.1) diventa

$$Cost(p + \Delta p) \approx Cost(p) + \nabla Cost(p)^T \Delta p \tag{3.2}$$

L'equazione (3.2) mostra che per ridurre il valore della funzione costo si può scegliere Δp affinchè $\nabla Cost(p)^T \Delta p$ sia più negativo possibile.
Per determinare tale valore di Δp si utilizza la disuguaglianza di Cauchy-Schwarz che afferma che per ogni $f, g \in \mathbb{R}^s$ si ha $|f^T g| \leq \|f\|_2 \|g\|_2$. Da ciò segue che il valore più negativo che $f^T g$ può assumere è $-\|f\|_2 \|g\|_2$, ottenuto per $f = -g$.
Dunque, si considera Δp nella direzione $-\nabla Cost(p)$.
Questo conduce all'aggiornamento

$$p \longrightarrow p - \eta \nabla Cost(p), \tag{3.3}$$

in cui il parametro η è chiamato *tasso di apprendimento* e determina la dimensione dello spostamento compiuto per raggiungere il punto di minimo (locale).
Se il gradiente è negativo, la funzione di perdita in quel punto è decrescente, il che significa che il parametro deve spostarsi verso valori più grandi per raggiungere un punto di minimo. Al contrario, se il gradiente è positivo, i parametri si spostano verso valori più piccoli per raggiungere valori inferiori della funzione di perdita.
Questa equazione individua il metodo di discesa più ripida, come si può osservare in figura 3.1.



(a) (b)

Figura 3.1: Partendo dal punto blu raffigurato in 3.1a è possibile osservare in 3.1b il percorso compiuto dall'algoritmo di discesa del gradiente per raggiungere il punto di minimo della funzione. (Immagine presa da [17])

Scelto un vettore iniziale, lo si itera con (3.3) finchè non si raggiungono specifici criteri di arresto oppure non si supera un numero massimo di iterazioni prestabilito.
La funzione costo (2.9) è data da una somma di termini da cui segue che anche la derivata parziale di tale funzione è somma di derivate parziale.
Esplicitamente, sia

$$C_{\mathrm{x}^{\{\mathrm{i}\}}} = \frac{1}{2}\left\|y(x^{\{\mathrm{i}\}}) - a^{[\mathrm{L}]}(x^{\{\mathrm{i}\}})\right\|_2^2. \tag{3.4}$$

Da (2.9) si ottiene

$$\nabla Cost(p) = \frac{1}{N}\sum_{i=1}^{\mathrm{N}} \nabla C_{\mathrm{x}^{\{\mathrm{i}\}}}(p). \tag{3.5}$$

Per poter ottenere un unico aggiornamento di p è necessario calcolare i gradienti per l'intero insieme di dati. Questo comporta che il metodo di discesa del gradiente batch possa essere molto lento e intrattabile per insiemi di dati che non rientrano nella memoria. Inoltre, come si può osservare in figura 3.2, tale metodo garantisce la convergenza al minimo globale per superfici convesse e ad un minimo locale per superfici non convesse.

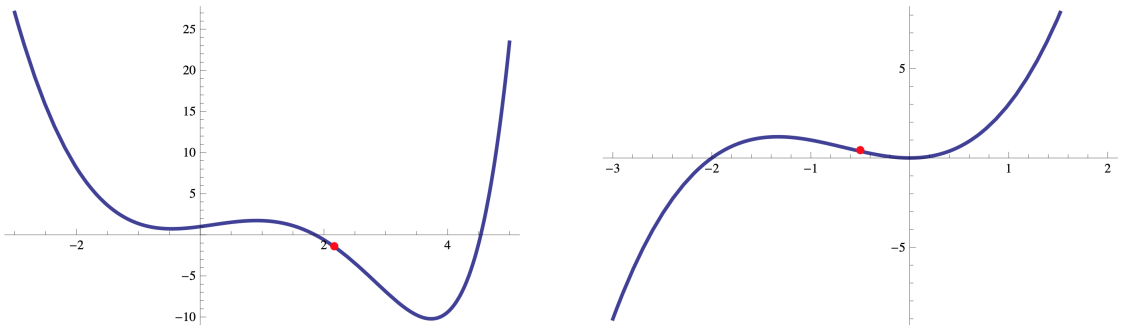


Figura 3.2: Per entrambe le funzioni, il metodo di discesa del gradiente inizia in corrispondenza del punto rosso. Per la funzione di sinistra il punto di minimo trovato è globale, mentre per la funzione di destra è locale. Siccome in entrambi i punti la derivata è zero, l'algoritmo non riesce a stabilire se si tratta di minimi locali o globali.

Il metodo di discesa del gradiente batch converge spesso ad un punto di minimo locale della funzione costo. L'algoritmo, infatti, non riesce evitare tale minimo locale, che però rappresenta una soluzione non soddisfacente del problema.
In situazioni concrete, il metodo del gradiente batch potrebbe rimanere bloccato in un'area in cui il costo è estremamente mal condizionato, come un punto in cui la funzione forma un profondo pendio. Tale punto è un minimo locale in un sottospazio definito dai più grandi autovalori della matrice hessiana della funzione costo [16].

Il metodo di discesa stocastica del gradiente è in grado di risolvere questo problema.

3.1.2 Discesa stocastica del gradiente

Quando si ha un grande numero di parametri e di training points, calcolare il vettore gradiente (3.5) ad ogni iterazione del metodo di discesa del gradiente batch può essere estremamente costoso.
Una alternativa più economica può essere sostituire la media dei gradienti individuali calcolata su tutti i training points con il gradiente di un singolo training point scelto casualmente. Questa modifica apportata introduce nuovo metodo conosciuto come *metodo di discesa stocastica del gradiente* o *stochastic gradient descent* (SGD).
Esso è un metodo iterativo per l'ottimizzazione di funzioni differenziabili e per l'approssimazione stocastica del metodo di discesa del gradiente (GD) quando la funzione costo ha la forma di una somma.
Una singola iterazione di questo metodo può essere riassunta nel modo seguente:

1. si sceglie casualmente ed in maniera uniforme un intero i da $\{1, 2, 3, \ldots, N\}$

2. si aggiorna
$$p \longrightarrow p - \eta \nabla C_{\mathrm{x}\{\mathrm{i}\}}(p). \tag{3.6}$$

Ad ogni step, quindi, SGD utilizza un training point scelto casualmente per rappresentare l'intero insieme di addestramento.
Si nota che, anche per η molto piccolo, l'aggiornamento (3.6) non garantisce una riduzione della funzione costo nel suo complesso.

La versione del metodo del gradiente stocastico che è stata introdotta in (3.6) è la più semplice. In particolare, l'indice i in (3.6) è scelto tramite un *campionamento con reinserimento* ossia, dopo aver scelto e utilizzato un training point, esso viene reintrodotto nel training set e quindi nel passaggio successivo può essere scelto con la stessa probabilità degli altri punti.

Un'alternativa a ciò è il *campionamento senza reinserimento* che consiste nello scorrere ciclicamente ciascuno degli N training points in ordine casuale.
Eseguire N passaggi in questo modo, ciascuno dei quali viene indicato come il termine *epoca*, può essere riassunto come segue:

1. si ridistribuiscono gli interi $\{1, 2, 3, \ldots, N\}$ in un nuovo ordine $\{k_1, k_2, k_3, \ldots, k_N\}$
2. per $i = 1, \ldots, N$ si aggiorna

$$p \longrightarrow p - \eta \nabla C_{\mathrm{x}\{\mathrm{k_i}\}}(p). \tag{3.7}$$

Il metodo SGD elimina la ridondanza nell'ottenere un aggiornamento per volta, rendendolo così molto veloce. Inoltre, SGD compie frequenti aggiornamenti caratterizzati da un'alta varianza che portano la funzione costo a fluttuare fortemente, come si può osservare in figura 3.3.

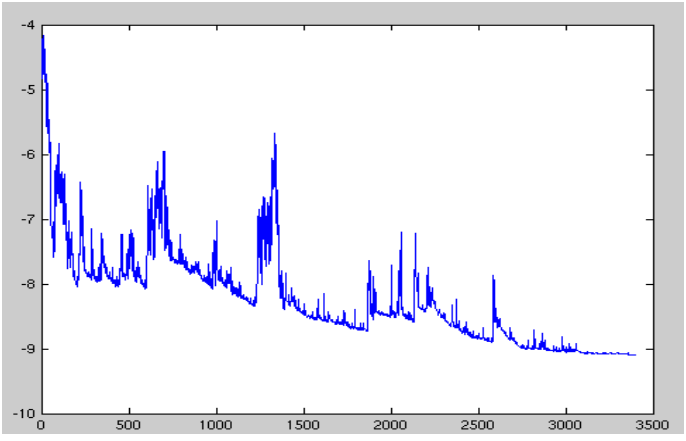


Figura 3.3: Andamento della funzione costo nel corso delle iterazioni in un problema di ottimizzazione risolto tramite SGD. (Immagine presa da [4])

Questo comporta alcune conseguenze:

- da una parte, mentre la discesa del gradiente batch converge al minimo del bacino in cui i parametri sono situati, la fluttuazione della funzione dovuta a SGD permette di individuare nuovi e potenzialmente migliori punti di minimo locale

- dall'altra, la fluttuazione complica la convergenza al punto di minimo esatto, in quanto l'overshooting è frequente. Nonostante ciò, è stato mostrato che diminuire lentamente il tasso di apprendimento permette a SGD di avere lo stesso comportamento di convergenza della discesa del gradiente batch.

Il metodo SGD è ampliamente usato per l'addestramento di una varietà di modelli probabilistici e di apprendimento automatico come macchine a vettori di supporto, regressione logistica e modelli grafici.

3.1.3 Discesa del gradiente mini-batch

Se si considera il metodo del gradiente stocastico come un'approssimazione della media su tutti i training points in (3.5) fatta da un singolo campione, allora è naturale considerare un compromesso in cui si utilizza una piccola media campionaria.
Per qualche $m \ll N$ è possibile eseguire i seguenti steps:

1. si scelgono m interi $k_1, k_2, \ldots, k_m$ presi casualmente ed in maniera uniforme dall'insieme $\{1, 2, 3, \ldots, N\}$

2. si aggiorna
$$p \longrightarrow p - \eta \frac{1}{m} \sum_{\mathrm{i}=1}^{\mathrm{m}} \nabla C_{\mathrm{x}^{\{\mathrm{i}\}}}(p). \tag{3.8}$$

In questa iterazione l'insieme $\{x^{\{\mathrm{k_i}\}}\}_{i=1}^{m}$ è noto come *minibatch*.
Vi è anche in questo caso un'alternativa senza reinserimento in cui, assumendo $N = Km$ per qualche K, si suddivide casualmente il training set in K minibatch distinti e si scorre ciclicamente su di essi.
Poichè il metodo del gradiente stocastico è di solito utilizzato su un calcolo in larga scala, scelte algoritmiche come la dimensione di un minibatch e la forma di randomizzazione

sono spesso legate ai requisiti di prestazioni architettoniche di calcolo. Tuttavia, oltre a variare in maniera dinamica queste scelte, è possibile cambiare il tasso di apprendimento η così da accelerare la convergenza.
La discesa del gradiente mini-batch sintetizza i punti di forza dei due metodi precedenti e fornisce un aggiornamento per ogni mini-batch di m punti di addestramento.
In questo modo tale metodo

- riduce la varianza dei parametri aggiornati, permettendo così una più stabile convergenza
- basandosi solo su una parte delle osservazioni, permette un'esplorazione più ampia dello spazio con maggiore possibilità di trovare nuovi e potenzialmente migliori punti di minimo
- è computazionalmente molto più rapido, garantendo così una convergenza più veloce verso il punto di minimo
- è in grado di calcolare le stime dei parametri caricando in memoria solamente una parte del dataset alla volta, permettendo così l'applicazione di questo metodo anche a grandi insiemi di dati.

La discesa del gradiente mini-batch è solitamente il metodo prediletto per l'addestramento di una rete neurale.

3.2 Algoritmi per l'ottimizzazione di SGD

Il metodo SGD non garantisce sempre una buona convergenza e fa emergere alcune sfide che necessitano di essere affrontate, tra cui

- la scelta del tasso di apprendimento η. Un tasso di apprendimento troppo piccolo comporta una lenta e difficile convergenza, mentre un tasso di apprendimento troppo grande può ostacolare la convergenza e causare una fluttuazione della funzione costo attorno al minimo o, perfino, la divergenza

- come modificare il tasso η durante l'apprendimento della macchina. Durante l'addestramento, si cerca di modificare η in base a piani predefiniti o a valori di soglia che devono essere stabiliti in anticipo e che quindi non sono in grado di adattarsi alle caratteristiche dell'insieme di dati che si sta analizzando
- come adattare il tasso di apprendimento per gli aggiornamenti dei parametri nel caso in cui l'insieme dei dati non sia uniforme
- come evitare i punti di minimo locale.

Il Deep Learning fa largamente uso di algoritmi che sono in grado di trattare le sfide sopracitate.

3.2.1 Momento

Il metodo SGD riscontra alcune difficoltà in corrispondenza di aree in cui la superficie curva molto più ripidamente in una dimensione piuttosto che nell'altra. Tali aree sono frequenti attorno a punti di ottimo locale. SGD oscilla intorno alle pendenze che si creano, compiendo così solo esitanti ed incerti progressi verso il punto di ottimo locale, come si può osservare in 3.4.

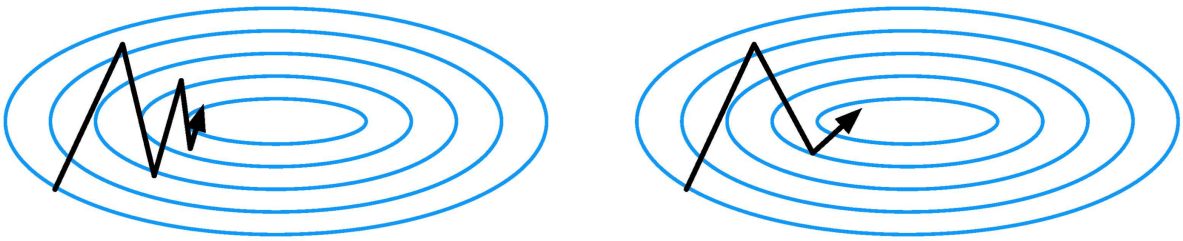


Figura 3.4: Nell'immagine di sinistra si osserva il metodo SGD senza il momento, mentre nell'immagine di destra si osserva il metodo SGD con il momento. (Immagine presa da [4])

Il *Momento* è un metodo che permette a SGD di accelerare nella direzione rilevante e di mitigare così le oscillazione, come si può osservare in figura 3.4.

Questo avviene aggiungendo al vettore aggiornato corrente una frazione γ del vettore aggiornato nell'iterazione precedente:

$$v_t = \gamma v_{t-1} + \eta \Delta Cost(p) \tag{3.9}$$

$$p = p - v_t. \tag{3.10}$$

Il termine momento γ solitamente è fissato a 0.9 o ad un valore simile.
Sostanzialmente, quando si utilizza il momento è come se si spingesse una palla giù da una collina: la palla, durante la sua discesa, accumula il momento e diventa sempre più veloce.
La stessa cosa succede agli aggiornamenti dei parametri: il termine momento aumenta le dimensioni degli aggiornamenti quando i gradienti puntano nella stessa direzione mentre riduce le dimensioni degli aggiornamenti quando i gradienti cambiano direzione.
In questo modo si ottiene una più veloce convergenza e si riduce l'oscillazione.

3.2.2 Nesterov accelerated gradient

Una palla che rotola giù da una collina, seguendo in modo inconsapevole la discesa, è altamente insoddisfacente.
Si preferirebbe infatti avere una palla più intelligente che avesse un'idea della sua direzione. In questo modo essa saprebbe di dover rallentare prima che la collina si rivolga nuovamente vero l'alto.

Il metodo *Nesterov Accelerated Gradient* (NAG) è in grado di conferire al termine momento questa consapevolezza.
Per aggiornare i parametri p viene utilizzato il termine momento γv_{t-1}.
Calcolando $p - \gamma v_{t-1}$ si ottiene una approssimazione della posizione successiva occupata dai parametri, ossia una idea approssimativa di dove i parametri stanno andando.
Ora è effettivamente possibile guardare avanti attraverso il calcolo del gradiente, non rispetto ai parametri correnti p, ma rispetto all'approssimazione della loro posizione futura:

$$v_t = \gamma v_{t-1} + \eta \Delta_p Cost(p - \gamma v_{t-1}) \tag{3.11}$$

$$p = p - v_t \tag{3.12}$$

Anche in questo caso il termine momento viene fissato intorno al valore 0.9.

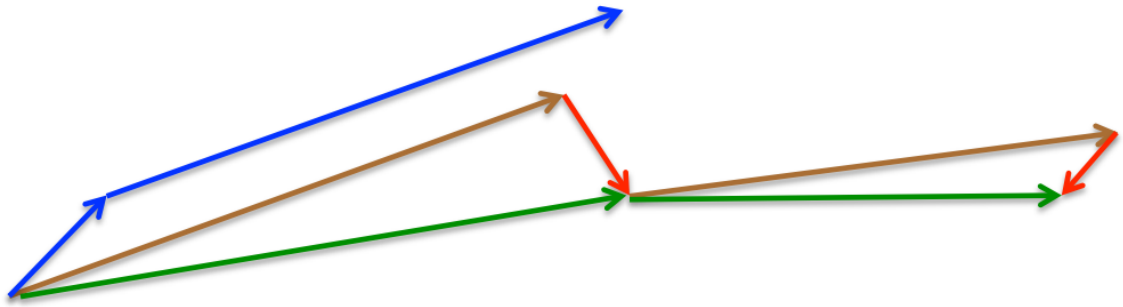


Figura 3.5: In questa immagine è possibile confrontare l'aggiornamento ottenuto con il metodo momento (vettore blu) e l'aggiornamento ottenuto con il metodo NAG (vettore verde). (Immagine presa da [4])

Il metodo momento calcola inizialmente il gradiente corrente, che corrisponde al vettore blu più piccolo nell'immagine 3.5, e successivamente compie un grande salto nella direzione del gradiente aggiornato, che corrisponde al vettore blu più grande.
Al contrario, il metodo NAG compie inizialmente un grande salto nella direzione del gradiente precedente, che corrisponde al vettore marrone in 3.5, misura il gradiente ed infine compie una correzione, che corrisponde al vettore rosso, ottenendo coì l'aggiornamento cercato, che corrisponde al vettore verde.
Questo aggiornamento anticipatorio impedisce di andare troppo veloce e dà origine ad una maggiore reattività. In questo modo si è in grado di adattare gli aggiornamenti alla pendenza della funzione costo ed accelerare, a sua volta, il metodo SGD.

L'obiettivo successivo che si vuole raggiungere è quello di adattare gli aggiornamenti ai singoli parametri per eseguire aggiornamenti più grandi o più piccoli in base alla loro importanza.

3.2.3 Adagard

Adagard è un algoritmo in grado di adattare il tasso di apprendimento η ai parametri, ottenendo così grandi aggiornamenti per parametri infrequenti e piccoli aggiornamenti per parametri frequenti. Per questo motivo tale metodo è particolarmente indicato nel trattare insiemi di dati sparsi.
Adagard, inoltre, migliora molto la robustezza di SGD, permettendo così di addestrare reti neurali su larga scala.

Nei metodi visti precedentemente viene calcolato un unico aggiornamento per tutti i parametri p contemporaneamente, in quanto viene utilizzato lo stesso tasso di apprendimento η.
Dal momento che Adagard fa uso di un differente tasso di apprendimento per ogni parametro p_i in ciascuno step t, il metodo inizialmente calcola un aggiornamento per parametro che poi viene vettorizzato.
Si pone $g_{t,i}$ il gradiente della funzione costo rispetto al parametro p_i allo step t:

$$g_{t,i} = \Delta_{p_t} Cost(p_{t,i}). \tag{3.13}$$

Il metodo SGD compie un aggiornamento per ciascun parametro p_i ad ogni step t, ottenendo così

$$p_{t+1,i} = p_{t,i} - \eta \cdot g_{t,i}. \tag{3.14}$$

Adagard modifica il tasso di apprendimento η per ciascun parametro p_i ad ogni step t basandosi sui gradienti precedenti calcolati per p_i:

$$p_{t+1,i} = p_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \varepsilon}} \cdot g_{t,i}, \tag{3.15}$$

dove

- $G_t \in \mathbb{R}^{\mathrm{dxd}}$ è una matrice diagonale in cui ciascun elemento diagonale i, i è della forma $G_{t,ii} = \sum_{k=1}^{t} \sqrt{\Delta_{p_k} Cost(p_{k,i})}$, in cui k rappresenta lo step
- ε è un termine che permette di non fare le divisioni per zero.

Dal momento che G_t contiene la somma delle radici dei gradienti precedenti rispetto a tutti i parametri p, è possibile vettorizzare l'implementazione compiendo una moltiplicazione $\odot$ matrice-vettore elemento per elemento tra G_t e g_i:

$$p_{t+1} = p_t - \frac{\eta}{\sqrt{G_t + I\varepsilon}} \odot g_t. \tag{3.16}$$

Uno dei principali benefici che apporta Adagard è che elimina la necessità di regolare manualmente il tasso di apprendimento.
D'altra parte, la principale debolezza di tale metodo è che accumula gradienti sotto radice al denominatore: dal momento che ogni temine che viene aggiunto è positivo, la somma accumulata continua a crescere durante l'apprendimento. Questo porta il tasso di apprendimento a restringersi, fino a diventare infinitamente piccolo, impedendo così all'algoritmo di acquisire conoscenze ulteriori.

Il seguente algoritmo cerca di risolvere questa imperfezione.

3.2.4 Adadelta

Adadelta è una estensione di Adagard che cerca di ridurre il suo aggressivo e monotonicamente decrescente tasso di apprendimento.
Al posto di accumulare tutti i precedenti gradienti sotto radice, Adadelta restringe il numero di gradienti che vengono accumulati ad una dimensione fissata w.
In particolare, la somma dei gradienti è ricorsivamente definita come una media pesata di tutti i precedenti gradienti sotto radice.
La media $E[g^2]_t$ relativa allo step t dipende esclusivamente dalla media precedente e dall'attuale gradiente:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2, \tag{3.17}$$

dove il termine γ solitamente viene posto uguale a 0.9, similmente al termine momento.
Riscrivendo l'aggiornamento di SGD in termini del vettore aggiornato Δp_t si ottiene

$$\Delta p_t = -\eta \cdot g_{t,i} \tag{3.18}$$

$$p_{t+1} = p_t + \Delta p_t. \tag{3.19}$$

Il vettore di Adagard aggiornato che si deriva precedentemente ha quindi la seguente forma:

$$\Delta p_t = -\frac{\eta}{\sqrt{G_t + \varepsilon}} \odot g_t. \tag{3.20}$$

La matrice diagonale G_t viene sostituita con la media pesata $E[g^2]_t$ calcolata sui precedenti gradienti sotto radice:

$$\Delta p_t = -\frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}} g_t. \tag{3.21}$$

Dal momento che il denominatore è la radice dell'errore quadratico medio (RMS) del gradiente, è possibile riscriverla come

$$\Delta p_t = -\frac{\eta}{RMS[g]_t} g_t. \tag{3.22}$$

Si osserva che le unità in questo aggiornamento, così come in SGD, Momento o Adagard, non combaciano in quanto l'aggiornamento dovrebbe avere le stesse ipotetiche unità dei parametri.
Per poter realizzare questo si definisce un'altra media pesata, non più di radici di gradienti, bensì di radici di aggiornamenti del parametro:

$$E[\Delta p^2]_t = \gamma E[\Delta p^2]_{t-1} + (1-\gamma)\Delta p_t^2. \tag{3.23}$$

La radice dell'errore quadratico medio degli aggiornamenti del parametro è

$$RMS[\Delta p]_t = \sqrt{E[\Delta p^2]_t + \varepsilon}. \tag{3.24}$$

Dal momento che $RMS[\Delta p]_t$ non è noto, esso viene approssimato con l'RMS degli aggiornamenti del parametro fino allo step precedente.
Sostituendo il tasso di apprendimento η nell'aggiornamento precedente con $RMS[\Delta p]_{t-1}$ si ottiene l'aggiornamento Adadelta:

$$\Delta p_t = -\frac{RMS[\Delta p]_{t-1}}{RMS[g]_t} g_t \tag{3.25}$$

$$p_{t+1} = p_t + \Delta p_t. \tag{3.26}$$

Il metodo Adadelta non necessita di stabilire un tasso di apprendimento predefinito in quanto η è stato eliminato dall'aggiornamento.

3.2.5 RMSprop

RMSprop è un metodo in cui il tasso di apprendimento è adattabile ed il suo algoritmo è uguale al primo aggiornamento di Adadelta, ossia

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2 \tag{3.27}$$

$$p_{t+1} = p_t - \frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}} g_t. \tag{3.28}$$

RMSprop divide il tasso di apprendimento dalla media pesata della radice dei gradienti. In questo metodo il termine γ è posto uguale a 0.9 mentre un buon valore per il tasso di apprendimento è 0.001.

3.2.6 Adam

Il metodo *Adam* (Adaptive Moment Estimation) è un altro metodo che calcola i tassi di apprendimento adattabili per ciascun parametro.
In aggiunta, oltre che memorizzare la media pesata delle precedenti radici dei gradienti v_t, come Adadelta e RMSprop, Adam conserva anche una media pesata dei precedenti gradienti m_t, simile al momento:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{3.29}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2, \tag{3.30}$$

dove m_t e v_t sono stime rispettivamente del primo momento e del secondo momento dei gradienti.
Quando m_t e v_t vengono inizializzati come vettori di zeri, essi tendono a propendere verso zero specialmente durante gli steps iniziali e quando i tassi di decadimento sono piccoli (ad esempio quando β_0 e β_1 sono vicini a 1).
Per contrastare questi biases si calcolano stime corrette del primo momento e del secondo momento:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{3.31}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}. \tag{3.32}$$

Tali stime vengono utilizzate per aggiornare i parametri nel seguente modo:

$$p_{t+1} = p_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon}\hat{m}_t. \tag{3.33}$$

In questo caso i valori predefiniti che vengono proposti sono 0.9 per β_1, 0.999 per β_2 e 10^{-8} per ε.

Il metodo Adam funziona bene e, anche paragonato agli altri metodi per l'apprendimento adattabile, risulta essere efficiente e vantaggioso.

3.2.7 AdaMax

Il fattore v_t, presente nell'aggiornamento del metodo Adam, bilancia il gradiente in modo inversamente proporzionale rispetto alla norma l_2 dei gradienti precedenti e del gradiente corrente $|g_t|^2$:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)|g_t|^2. \tag{3.34}$$

È possibile generalizzare questo aggiornamento utilizzando la norma l_p.
In questo modo si ottiene

$$v_t = \beta_2^p v_{t-1} + (1 - \beta_2^p)|g_t|^p. \tag{3.35}$$

Generalmente, norme l_p con p molto grande sono numericamente instabili e questo comporta che le norme più comunemente usate siano l_1 e l_2.
Nonostante ciò, la norma l_∞ mostra un comportamento stabile e per questo motivo viene utilizzata dal metodo *AdaMax*, in cui è possibile osservare che v_t con l_∞ converge ad un valore stabile.

Per evitare ambiguità con il metodo Adam, viene utilizzata la notazione u_t per denotare il vettore v_t dotato della norma infinito:

$$u_t = \beta_2^\infty v_{t-1} + (1 - \beta_2^\infty)|g_t|^\infty \tag{3.36}$$

$$= max(\beta_2 \cdot v_{t-1}, |g_t|). \tag{3.37}$$

È possibile inserire il risultato appena ottenuto nell'aggiornamento del metodo Adam, sostituendo $\sqrt{\hat{v}_t} + \varepsilon$ con u_t per ottenere l'aggiornamento AdaMax:

$$p_{t+1} = p_t - \frac{\eta}{u_t}\hat{m}_t. \tag{3.38}$$

Buoni valori di default sono $\eta = 0.002$, $\beta_1 = 0.9$ e $\beta_2 = 0.999$.

3.2.8 Nadam

Nadam è un metodo che combina Adam e NAG, modificando il termine momento m_t.
La regola per aggiornare il momento con le notazioni correnti è la seguente:

$$g_t = \Delta_{p_t} Cost(p_t) \tag{3.39}$$

$$m_t = \gamma m_{t-1} + \eta g_t \tag{3.40}$$

$$p_{t+1} = p_t - m_t, \tag{3.41}$$

dove *Cost* è la funzione costo, γ è il termine di decadimento del momento e η è la dimensione dello step.
Sviluppando la terza equazione si ottiene

$$p_{t+1} = p_t - (\gamma m_{t-1} + \eta g_t). \tag{3.42}$$

Questo dimostra nuovamente che il momento permette di compiere uno step nella direzione del precedente momento e uno step nella direzione del gradiente corrente.

Inoltre, NAG permette di compiere uno step più accurato nella direzione del gradiente aggiornando i parametri con il momento prima di calcolare il gradiente.
Per fare ciò è necessario modificare soltanto il gradiente g_t, ottenendo così l'algoritmo NAG:

$$g_t = \Delta_{p_t} Cost(p_t - \gamma m_{t-1}) \tag{3.43}$$

$$m_t = \gamma m_{t-1} + \eta g_t \tag{3.44}$$

$$p_{t+1} = p_t - m_t. \tag{3.45}$$

Anzichè applicare il momento due volte, una per aggiornare il gradiente g_t e l'altra per aggiornare i parametri p_{t+1}, è possibile modificare NAG applicando direttamente il vettore momento per aggiornare i parametri correnti, ossia

$$g_t = \Delta_{p_t} Cost(p_t) \tag{3.46}$$

$$m_t = \gamma m_{t-1} + \eta g_t \tag{3.47}$$

$$p_{t+1} = p_t - (\gamma m_t + \eta g_t). \tag{3.48}$$

In questo caso si nota che, anzichè utilizzare il vettore momento precedente m_{t-1} come in (3.42), si usa il vettore momento corrente m_t per guardare avanti.
Per aggiungere il momento Nesterov al metodo Adam è possibile sostituire il vettore momento precedente con quello corrente.
L'algoritmo di aggiornamento del metodo Adam è il seguente:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{3.49}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{3.50}$$

$$p_{t+1} = p_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t. \tag{3.51}$$

Sviluppando la seconda equazione utilizzando le definizioni di m_t e $\hat{m}_t$ si ottiene

$$p_{t+1} = p_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} (\frac{\beta_1 m_{t-1}}{1 - \beta_1^t} + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t}), \tag{3.52}$$

dove $\frac{\beta_1 m_{t-1}}{1-\beta_1^t}$ è la stima corretta del vettore momento dello step precedente.
Tale termine può essere sostituito con $\hat{m}_{t-1}$, ottenendo

$$p_{t+1} = p_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} (\beta_1 \hat{m}_{t-1} + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t}). \tag{3.53}$$

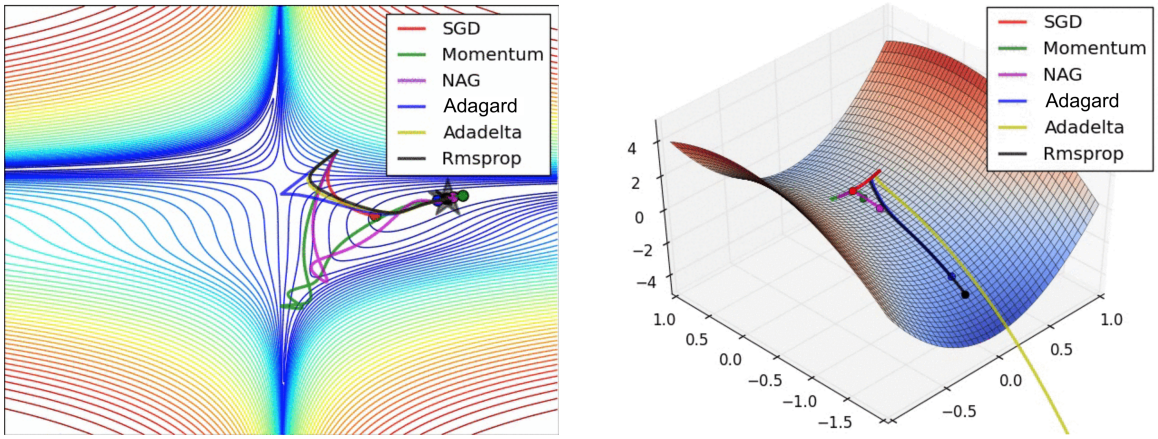
Questa equazione è molto simile al termine momento in (3.42).

È possibile ora aggiungere il momento Nesterov semplicemente sostituendo la stima corretta del vettore momento $\hat{m}_{t-1}$ relativo allo step precedente con la stima corretta del vettore momento corrente $\hat{m}_t$, ottenendo così l'aggiornamento Nadam:

$$p_{t+1} = p_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} (\beta_1 \hat{m}_t + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t}). \tag{3.54}$$

3.3 Confronto tra gli algoritmi di ottimizzazione

Per poter paragonare il comportamento di ottimizzazione degli algoritmi precedentemente introdotti è utile ricorrere ad una visualizzazione grafica di tali algoritmi.



(a) Ottimizzazione SGD su una superficie di perdita.

(b) Ottimizzazione SGD in corrispondenza di un punto di sella.

Figura 3.6: (Immagine presa da [4])

In figura 3.6a si osserva il percorso compiuto da ciascun algoritmo su una superficie di perdita. Tutti partono dallo stesso punto ma ognuno segue un cammino differente per raggiungere il punto di minimo.
Si nota che

- *Adagard, Adadelta* e *RMSprop* si dirigono immediatamente nella giusta direzione e convergono velocemente in modo simile
- *SGD* è in grado di determinare la corretta direzione per raggiungere il punto di minimo ma impiega un tempo sensibilmente maggiore rispetto a quello che impiegherebbe con l'ausilio di un algoritmo di ottimizzazione
- *momento* e *NAG* vanno fuori strada, evocando l'immagine di una palla che rotola giù da una collina. Nonostante ciò, NAG riesce a correggere velocemente il suo

percorso, grazie alla sua crescente reattività nel guardare avanti, e a raggiunge così il minimo.

La figura 3.6b mostra il comportamento degli algoritmi in corrispondenza di un punto di sella, ossia un punto in cui una direzione ha pendenza positiva mentre l'altra negativa. Si nota che

- *SGD, Momento* e *NAG* riscontrano difficolà nel rompere la simmetria, sebbene gli ultimi due metodi alla fine riescano ad evitare il punto di sella
- *Adagard, RMSprop* e *Adadelta* scendono velocemente verso la pendenza negativa, con Adadelta che guida tale discesa.

Come si può osservare, i metodi con tasso di apprendimento adattabile, ossia Adagard, Adadelta e RMSprop, sono più idonei e forniscono la migliore convergenza in entrambi i casi analizzati.

In conclusione, se l'insieme dei dati di input è sparso allora i metodi con tasso di apprendimento adattabile sono i più indicati per raggiungere migliori risultati.
RMSprop è un'estensione di Adagard che tratta la sua radicale diminuzione del tasso di apprendimento. Inoltre, è uguale ad Adadelta con la differenza che Adadelta usa l'RMS degli aggiornamenti dei parametri nell'aggiornamento del numeratore.
Infine, Adam aggiunge una correzione di bias e momento a RMSprop.
Da ciò si evince che RMSprop, Adadelta e Adam sono algoritmi molto simili che funzionano bene in circostanze analoghe ma, complessivamente, Adam è considerato il migliore.

Conclusioni

In questo elaborato sono stati esaminati differenti metodi di analisi dati per l'addestramento di una macchina.
Inizialmente è stato introdotto il concetto di Machine Learning, differenziando le sue categorie di apprendimento in regressione e classificazione in base al tipo di training set a disposizione.

È stato osservato che il modello statistico più utilizzato nell'ambito del Deep Learning supervisionato è la rete neurale multi-strato, la quale presenta strutture differenti in base al tipo di apprendimento che si vuole attuare.
Durante il processo di addestramento di una rete neurale è stato constatato che è di fondamentale importanza trovare i valori ottimali dei pesi della rete per poter minimizzare l'errore tra i risultati di output ottenuti e quelli desiderati. Tra gli algoritmi maggiormente utilizzati per la modifica di tali pesi vi è l'algoritmo di backpropagation.

La parte centrale della tesi è stata incentrata nell'introdurre e confrontare differenti metodi per la determinazione dei punti di ottimo della funzione costo con l'obiettivo di individuare quelli con caratteristiche di convergenza migliori.
Da tale analisi è emerso che, tra i metodi di discesa del gradiente, il metodo di discesa stocastica del gradiente è in grado di determinare con maggiore facilità un punto di minimo. Esso, però, impiega un tempo sensibilmente maggiore rispetto a quello che impiegherebbe con l'ausilio di un algoritmo di ottimizzazione. Inoltre, il metodo di discesa stocastica del gradiente fa molto affidamento ad una solida inizializzazione e potrebbe facilmente rimanere bloccato in punti di sella piuttosto che raggiungere i punti di minimo.

In conclusione, se si è interessati ad avere una veloce convergenza e ad addestrare nel modo migliore una rete neurale profonda o complessa si consiglia di utilizzare uno tra i metodi di ottimizzazione del metodo di discesa stocastica del gradiente precedentemente introdotti con tasso di apprendimento adattabile.

Bibliografia

[1] Friedman, J., Hastie, T., & Tibshirani, R. (2001). The elements of statistical learning (Vol. 1, No. 10). New York: Springer series in statistics.

[2] Gareth, J. (2010). An introduction to statistical learning: with applications in R. Springer Verlag.

[3] Higham, C. F., & Higham, D. J. (2019). Deep learning: An introduction for applied mathematicians. SIAM Review, 61(4), 860-891.

[4] Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747.

[5] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444.

[6] Schmidhuber, J. (2015). Deep learning in neural networks: An overview. Neural networks, 61, 85-117.

[7] Nielsen, M. A. (2015). Neural networks and deep learning (Vol. 2018). San Francisco, CA, USA: Determination press.

[8] Vidal, R., Bruna, J., Giryes, R., & Soatto, S. (2017). Mathematics of deep learning. arXiv preprint arXiv:1712.04741.

[9] Rumelhart, D. E., Hinton, G. E., Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

[10] Rete neurale. (N.d.). In *Wikipedia*. Retrived June 13, 2020, from *https* : *//it.wikipedia.org/wiki/Rete_neurale*.

[11] *https* : *//it.mathworks.com/discovery/neuralnetwork.html*.

[12] Universal approximation theorem. (N.d.). In *Wikipedia*. Retrived June 15, 2020, from *https* : *//en.wikipedia.org/wiki/Universal_approximation_theorem*.

[13] Angelo Bacino. Reti neurali artificiali. Retrived June 15, 2020, from *http* : *//xoomer.virgilio.it/angelobacino/Neural/capitolo3b.pdf*.

[14] Giorgio Gambosi. Apprendimento supervisionato. Retrived June 11, 2020, from *https* : *//tvml.github.io/ml*1819/*store/book.pdf*.

[15] Davide Maltoni. Reti Neurali. Retrived June 28, 2020, from *http* : *//bias.csr.unibo.it/maltoni/ml/DispensePDF/*8*_ML_RetiNeurali.pdf*.

[16] Bottou, L. (1991). Stochastic gradient learning in neural networks. Proceedings of Neuro-Nimes, 91(8), 12.

[17] The outline of Gradient Descent. Retrived June 30, 202, from *https* : *//suniljangirblog.wordpress.com/*2018/12/03/*the − outline − of − gradient − descent/*.

[18] Alessandro Aere, Proprietà statistiche di modelli per il deep learning. Tesi di laurea magistrale in scienze statistiche, Università degli studi di Padova, 2016/2017, Bruno Scarpa.