Appunti di machine learning

Daniele Besozzi

Anno accademico 2025/2026

Contents

Premesse

Questi sono appunti realizzati per riassumere e schematizzare tutti i concetti presentati durante il corso di machine learning tenuto presso il corso di laurea magistrale in informatica presso l'università degli studi di Milano Bicocca. Lo scopo di questo documento non è quello di sostituire le lezioni del corso o di essere l'unica fonte di studio, bensì integrare le altri fonti con un documento riassuntivo.
Mi scuso in anticipo per eventuali errori e prego i lettori di segnalarli contattandomi via mail all'indirizzo d.besozzi@campus.unimib.it.

Chapter 1

Introduzione

In questo capitolo presenterò gli aspetti matematici fondamentali per andare ad affrontare gli argomenti del corso.

1.1 Vettori e matrici

Denotiamo un vettore riga e colonna rispettivamente con (a, b, c) e $\begin{bmatrix} a \\ b \\ c \end{bmatrix}$

dove $a, b, c \in \mathbb{R}$ sono scalari.
In generale denotiamo con le lettere maiuscole le matrici, e.g. X e i suoi elementi con X_{ij}.
$x \in \mathbb{R}^n$ è un vettore di n elementi e $X \in \mathbb{R}^{m \times n}$ è una matrice di dimensione $m \times n$.

1.2 Norme di vettori e matrici

Dato un vettore $x \in \mathbb{R}^n$ vi sono diversi tipi di norme comunemente utilizzate.

- $||x||_2 = (\sum_{i=1}^{n} x_i^2)^{\frac{1}{2}}$ è il tipo più comune e viene chiamato **norma 2** di un vettore o **norma Euclidea**. Normalmente è denotato semplicemente con $||x||$.
- $||x||_1 = |x_1| + |x_2| + \cdots + |x_n|$, detta la **norma 1** o **distanza di Manhattan**
- $||x||_\infty = \max_{1 \leq i \leq n} |x_i|$, detta la **norma** ∞.

Analogamente, per una matrice $X \in \mathbb{R}^{m \times n}$, si possono definire diverse norme:

- **Norma di Frobenius**: $||X||_F = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |X_{ij}|^2 \right)^{\frac{1}{2}}$
- **Norma spettrale (o norma 2)**: $||X||_2$
- **Norma 1**: $||X||_1 = \sum_{i,j} |X_{ij}|$

1.3 Notazioni generiche

Siano:

- u: variabile indipendente (input), non necessariamente un vettore o uno scalare
- v: variabile dipendente (output), come sopra

allora abbiamo che:

- $x = \phi(u)$, dove $x \in \mathbb{R}^d$ è il vettore di features e ϕ è la funzione di mapping o embedding.
- $y = \psi(v)$, dove $y \in \mathbb{R}^m$ è il vettore target (o di output) e ψ è la funzione di mapping di feature in output.

Siano $x^1, \dots x^n$ e $y^1, \dots y^n$ due dataset di n esempi, dove x^i e y^i formano la i-esima coppia di dati. Dunque n è il numero di campioni, allora posso associarvi le due matrici dei dati

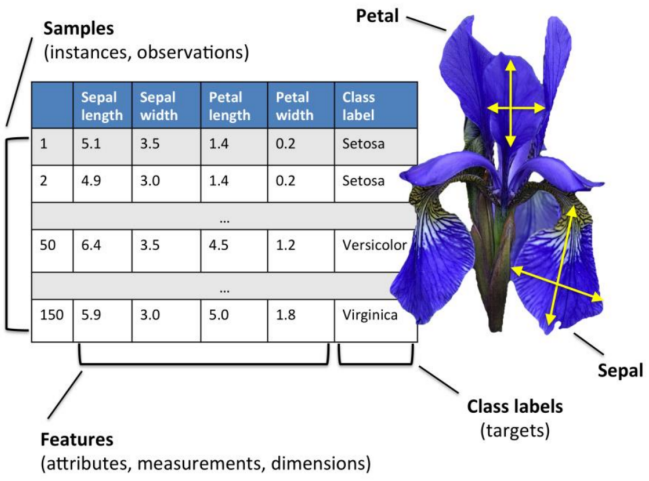
$$X = \begin{bmatrix} (x^1)^T \\ \vdots \\ (x^n)^T \end{bmatrix} \in \mathbb{R}^{n \times d}, \quad Y = \begin{bmatrix} (y^1)^T \\ \vdots \\ (y^n)^T \end{bmatrix} \in \mathbb{R}^{n \times m}$$

Le cui righe sono i vettori feature e i vettori target rispettivamente, trasposti.
Definiamo allora:

- $g_\theta : \mathbb{R}^d \to \mathbb{R}^m$ è un predittore.
- $\hat{y} = g_\theta(x)$ è la predizione di y, dato x.
- $\theta \in \mathbb{R}^p$ è il vettore di parametri del predittore.

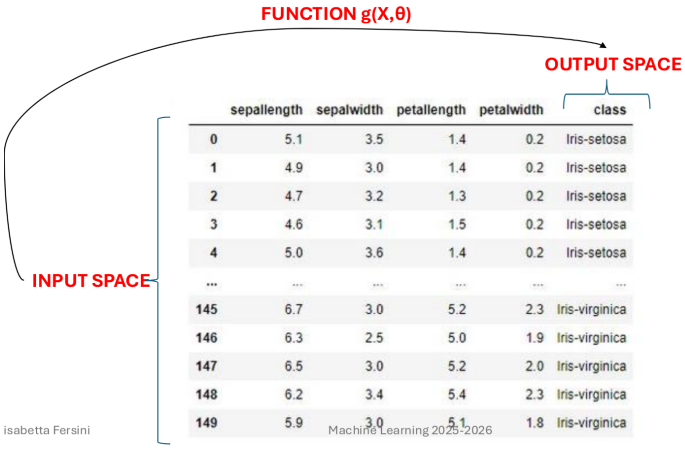
La scelta dei parametri θ a seconda dei dati viene chiamato *training* o *fitting* del predittore.
Separando le definizioni in base al dominio di riferimento, abbiamo le seguenti suddivisioni:

- Spazio di input:
  - Istanza (sample, oggetto, record): un esempio descritto da un certo numero di attributi.
  - Attributo (campo, caratteristica, variabile): misura di un aspetto di una istanza.
- Spazio di output:
  - Classe (label, target): categoria a cui appartiene una istanza.
- Predittore o modello:
  - Una funzione g con parametri θ che dato uno spazio di input X produce uno spazio di output Y.produce una predizione nello spazio di output Y.



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

Figure 1.1: Esempi pratici.



	sepallength	sepalwidth	petallength	petalwidth	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Figure 1.2: Esempi pratici 2.

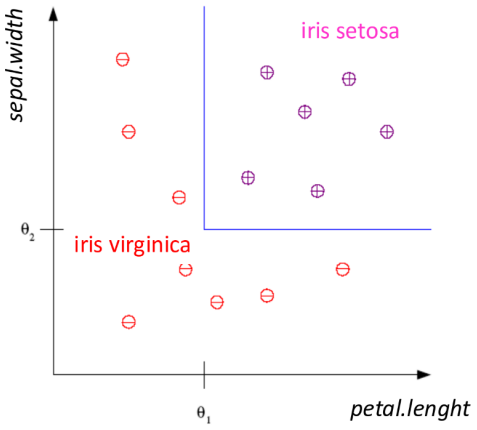


Figure 1.3: Esempi pratici 3.

1.4 Rischio atteso e rischio empirico

Quando un modello di machine learning g viene addestrato, vogliamo che abbia buone prestazioni non solo per i dati utilizzati per il training, ma anche per dati sconosciuti (*generalizzazione*). Dovremo stimare il **rischio atteso**, ovvero la loss media che dovrebbe presentarsi sulla reale distribuzione dei dati $P(x, y)$.
Supponiamo di avere un dataset di n coppie (x^i, y^i), dove x^i è il vettore di feature e y^i il vettore target associato alla i-esima istanza. Definiamo la **loss function** $L(g_\theta(x), y)$, che misura l'errore commesso dal modello g_θ.
Dunque il rischio atteso è definito come:

$$R(g_\theta) = E_{(x,y)\sim P}[L(g_\theta(x), y)]$$

Da notare però che la reale distribuzione P non è nota, dunque non possiamo stimare il rischio atteso. Le cause sono:

1. Distribuzione non nota
   Osserviamo solo un campione finito di campioni estratti dal mondo reale. La distribuzione completa che genera quei dati non è accessibile.

2. Impossibilità pratica
   Anche se conoscessimo il processo di generazione in teoria, calcolare esattamente la predizione del rischio è impossibile in generale perché richiederebbe di sommare/integrare tutte i possibili esempi.

3. Dati finiti e rumorosi
   Il dataset a disposizione è finito, spesso presenta rumore e errori di misura, o bias di raccolta. Dunque possiamo solo approssimare il rischio utilizzando una **stima empirica**.

Dunque, vogliamo stimare e minimizzare il rischio empirico. Supponiamo di avere un dataset di n coppie (x^i, y^i). Definiamo la loss function $L(g_\theta(x), y)$, che misura l'errore commesso dal modello g_θ. Allora il rischio empirico $\hat{R}(g_\theta)$ è definito come:

$$\hat{R}(g_\theta) = \frac{1}{n} \sum_{i=1}^{n} L(g_\theta(x^i), y^i)$$

La maggior parte degli algoritmi di machine learning minimizzano il rischio empirico.

$$g^* = arg \min_{g_\theta \in G} \hat{R}(g_\theta)$$

Dunque dato un dataset di training, x_i, y_i per $i = 1, \dots n$, per cui la loss sull' i-esima coppia di dati è $l(g_\theta(x_i), y_i)$, il rischio empirico è la loss media sul dataset di training. L'empirical risk minimization (ERM) si occupa di scegliere i parametri θ che minimizzano il rischio empirico.

1.5 Estrazione delle features

Utilizziamo u per denotare i dati di input grezzi, ad esempio un vettore, testo, immagine, audio, video, ecc. $x = \phi(u)$ è il vettore di features, ottenuto tramite la funzione ϕ, chiamata embedding, funzione feature o mappamento di feature. La funzione ϕ può variare molto nel livello di complessità in base al caso.

1.6 Modelli

Noi cerchiamo un predittore (o modello) $g_\theta : \mathbb{R}^d \to \mathbb{R}^m$ che dato un vettore di features $x \in \mathbb{R}^d$ produce una predizione $\hat{y} = g_\theta(x) \in \mathbb{R}^m$. La scelta del modello g_θ viene effettuata sia in base ai dati a disposizione che in base alle conoscenze pregresse. In termini di dati grezzi, il predittore è $\hat{v} = \psi^{-1}(g(\phi(u)))$, con qualche variante se la funzione ψ è invertibile. Se $\hat{y} \approx y$ allora il predittore ha fatto una buona predizione sulla i-esima coppia di dati. Però, l'obiettivo è avere $\hat{y} \approx y$ non solo sui dati di training, ma anche su dati mai visti prima. Due forme equivalenti per rappresentare un modello sono $\hat{y} = g_\theta(x)$ e $\hat{y} = g(x, \theta)$. Si dice che la funzione g da la struttura (o forma) del predittore, θ è un parametro per il modello predittivo La scelta di un particolare $\theta \in \mathbb{R}^p$ si chiama *tuning* o *fitting* o *addestramento* del modello. Un algoritmo di apprendimento è una ricetta per scegliere i parametri θ a partire dai dati di training.

1.7 Assunzione $i.i.d.$

Prima di presentare il concept learning è necessario presentare l'assunzione $i.i.d.$ (independent and identically distributed). Dunque si assume che i dati siano stati campionati in modo indipendente e che provengano dalla stessa distribuzione.

- In teoria : L'unione delle funzioni di distribuzione di probabilità possono essere scritte come il prodotto delle funzioni di distribuzione di probabilità marginali.

$$f(X, \theta) = \prod_{i=1}^{n} f(X_i, \theta)$$

- In pratica: Ogni modello considera un campione alla volta, ignorando le feature degli altri campioni.

Possiamo notare che nella figura 1.5 sono presenti dei campi nulli, questo è dovuto al fatto che non tutti i dati sono stati raccolti, ciò presenta un problema. Anche una rappresentazione relazionale dei dati può essere effettuata, come ad esempio in figura 1.6. Infine è anche possibile combinare le due rappresentazioni, come in figura 1.7.
Il processo di **flattening** su un file è detto **proposizionalizzazione**, ovvero l'unione di più relazioni

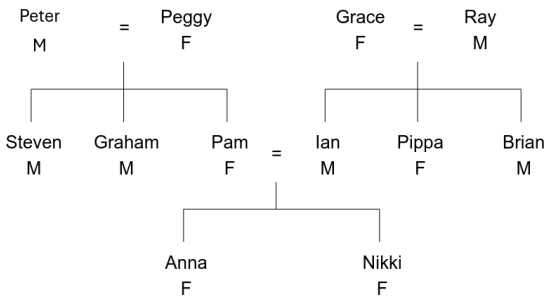


Figure 1.4: Esempio di struttura familiare

Name	Gender	Parent1	parent2
Peter	Male	?	?
Peggy	Female	?	?
Steven	Male	Peter	Peggy
Graham	Male	Peter	Peggy
Pam	Female	Peter	Peggy
Ian	Male	Grace	Ray
Pippa	Female	Grace	Ray
Brian	Male	Grace	Ray
Anna	Female	Pam	Ian
Nikki	Female	Pam	Ian

Figure 1.5: formato tabellare dei dati della figura 1.4

in una sola. Questa operazione è possibile per ogni insieme finito di relazioni finite, Un problema che emerge è la presenza di relazioni senza un numero pre specificato di oggetti. La proposizionalizzazione può introdurre delle regolarità fittizie che riflettono la struttura del database, inoltre possono introdurre dei **bias** causati da dati ripetuti.

First person	Second person	Sister of?
Peter	Peggy	No
Peter	Steven	No
...	...	...
Steven	Peter	No
Steven	Graham	No
Steven	Pam	Yes
...	...	...
Ian	Pippa	Yes
...	...	...
Anna	Nikki	Yes
...	...	...
Nikki	Anna	yes

First person	Second person	Sister of?
Steven	Pam	Yes
Graham	Pam	Yes
Ian	Pippa	Yes
Brian	Pippa	Yes
Anna	Nikki	Yes
Nikki	Anna	Yes
All the rest		No

Closed-world assumption

Figure 1.6: formato relazionale dei dati della figura 1.4

First person				Second person				Sister of?
Name	Gender	Parent1	Parent2	Name	Gender	Parent1	Parent2	
Steven	Male	Peter	Peggy	Pam	Female	Peter	Peggy	Yes
Graham	Male	Peter	Peggy	Pam	Female	Peter	Peggy	Yes
Ian	Male	Grace	Ray	Pippa	Female	Grace	Ray	Yes
Brian	Male	Grace	Ray	Pippa	Female	Grace	Ray	Yes
Anna	Female	Pam	Ian	Nikki	Female	Pam	Ian	Yes
Nikki	Female	Pam	Ian	Anna	Female	Pam	Ian	Yes
All the rest								No

Figure 1.7: formato misto dei dati della figura 1.4

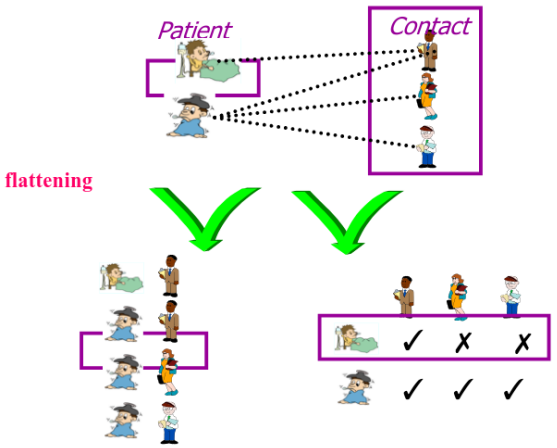


Figure 1.8: esempio di flattening

Nella colonna a sinistra della figura 1.8 possiamo notare che viene introdotto un bias dato dalla ripetizione di un dato, che in realtà è causato dalla natura relazionale dello stesso (dunque perdo la i.i.d.). Mentre colonna di destra viene introdotto un bias dato che viene considerato come non avvenuto un evento che in realtà potrebbe essere ancora non stato osservato, inoltre assumo a prescindere il numero di attributi e questo causa problemi nel momento in cui ne vengono aggiunti di nuovi.

1.8 tipi di attributi

Ogni istanza è descritta da un numero fisso predefinito di feature, i suoi attributi. Ci sono diversi tipi di attributi:

- **Quantità nominali**:Valori a simboli distinti, i valori in sé servono solo come etichette o nomi. Non vi è nessuna relazione implicita tra i valori nominali, come ordine o distanza. L'unico test effettuabile è quello di eguaglianza.
- **Quantità ordinali**: Valori che possono essere ordinati, ma non vi è alcuna informazione sulla distanza tra i valori (addizione e sottrazione non hanno significato).
- **Quantità di ratio**: Il metodo di misurazione definisce lo zero, ad esempio una distanza. Assumono valori in $\mathbb{R}$ e sono dotati di un ordine naturale. Sono possibili tutte le operazioni aritmetiche.

Perché è importante conoscere il tipo di attributo? Operazioni e confronti tra attributi di tipi diversi potrebbero non avere senso al livello semantico. Inoltre ci permettono di effettuare un check sulla validità dei valori, gestire la mancanza di essi ed effettuare confronti di eguaglianza.

Chapter 2

Inductive learning: Concept learning

Prima di iniziare con la spiegazione precisiamo la differenze tra apprendimento induttivo e deduttivo.

- **Apprendimento deduttivo**: Approccio di ragionamento dove un sistema applica regole generali per effettuare predizioni riguardanti casi specifici. È il contrario dell'apprendimento induttivo. L'idea principale è quella di partire da principi generali, teorie o regole note a priori. Applicare queste a situazioni specifiche per determinare risultati.
- **Apprendimento induttivo**:Un modello apprende regole generali o pattern a partire da esempi specifici o osservazioni. L'idea principale è quella di partire da data points specifici (esempi, dati di training) e cercare di generalizzare da questi esempi per formare un'ipotesi o regola che può prevedere casi sconosciuti.

	Inductive Learning	**Deductive Learning**
Definition	Learning patterns and models from data examples.	Applying known rules or logic to derive predictions.
Approach	Bottom-up: generalizes from specific training data.	Top-down: uses existing knowledge/rules to reason about new data.
Input	Labeled or unlabeled data (e.g., features and labels).	Formal rules, logic statements, or knowledge base.
Output	Predictive model or hypothesis (e.g., neural net).	Conclusions derived logically from existing rules.
Goal	Learn a function that maps inputs to outputs based on data.	Derive consequences or inferences from rules.
Example in ML	Training a classifier on labeled images (e.g., cat vs. dog).	Using a knowledge base and logical inference to classify an image.
Common Algorithms	Decision Trees, SVM, Neural Networks, k-NN, etc.	Logic programming, Expert Systems, Prolog-based systems.

Table 2.1: Confronto tra Inductive e Deductive Learning

Nell'apprendimento induttivo buona parte del processo di apprendimento involve la raccolta di concetti generali da esempi specifici di training. Ogni concetto può essere visto come descrivere un certo sotto insieme di oggetti o eventi definiti su un insieme più ampio. Una definizione alternativa considera ogni concetto come una funzione a valori booleani, definita su un insieme più ampio di oggetti (concept learning).

2.1 Concept learning

Consideriamo l'azione di apprendere il concetto target *"days on which my friend Aldo enjoys his favorite water sport"*. L'obiettivo è prevedere il valore *"enjoy sport"* per un giorno arbitrario, basandosi sui valori

degli altri attributi.

Sky	**Temp**	**Humid**	**Wind**	**Water**	**Forecast**	**EnjoySport**
Sunny	Warm	Normal	Strong	Warm	Same	Yes
Sunny	Mid	High	Strong	Warm	Same	Yes
Rainy	Cold	High	Strong	Warm	Change	No
Sunny	Warm	High	Strong	Cool	Change	Yes

Table 2.2: EnjoySport dataset

Cominciamo considerando una semplice rappresentazione nella quale ogni ipotesi consiste nella congiunzione dei requisiti sugli attributi delle istanze. Ogni ipotesi è un vettore di 6 requisiti, che specifica il valore dei 6 attributi. Per ogni attributo l'ipotesi può essere:

- ?: qualsiasi valore è accettabile
- specifico valore: l'attributo deve avere quel valore
- $\emptyset$: nessun valore è accettabile

Dunque il nostro obiettivo è trovare le ipotesi g in G tali che $g(x) = y$ per ogni istanza x.
L'ipotesi di apprendimento induttivo ci dice che ogni ipotesi trovata che approssima la funzione di target bene su un insieme di dati di training sufficientemente grande approssimerà anche la funzione target su dati non osservati. Il concept learning può essere visto come il processo di cercare in un grande spazio delle ipotesi implicitamente definite dalla rappresentazione delle ipotesi. L'obiettivo di questa ricerca è di trovare l'ipotesi che combacia meglio con gli esempi di training. Sia G lo spazio di ricerca, allora vogliamo trovare g che meglio "fitta" D. Riprendendo l'esempio di prima definiamo i requisiti $<?, \text{Cold}, \text{High}, ?, ?, ? >$. I possibili valori sono:

- **Sky:** Sunny, Cloudy, Rainy
- **AirTemp:** Warm, Cold, Mid
- **Humidity:** Normal, High
- **Wind:** Strong, Weak
- **Water:** Warm, Cold
- **Forecast:** Same, Change

Le possibili istanze sono $3 \times 3 \times 2 \times 2 \times 2 \times 2 = 288$
Le possibili ipotesi distinte sono $|G| = 5 \times 5 \times 4 \times 4 \times 4 \times 4 = 6400$ (bisogna considerare anche $\emptyset$ e ?), queste sono le ipotesi sintatticamente distinte.
Però dobbiamo rimuovere le possibilità che contengono $\emptyset$ in quanto non accettano nessun valore.
$|G| = 6400 - 5104 = 1296 = 4 \times 4 \times 3 \times 3 \times 3 \times 3$, che sono le ipotesi semanticamente distinte. Ciò viene descritto dalla formula $|G| = \prod_{i=1}^{n}(|D_i| + 1)$, dove D_i è l'insieme dei valori possibili per l'attributo i e n è il numero di attributi.
Data la visione del concept learning come problema di ricerca sorge naturalmente la questione dello studio di algoritmi efficienti per ridurre lo spazio delle ipotesi G.

Chapter 3

Feature Engineering

def. Trasformare i dati grezzi in feature significative, ovvero variabili che permettono al modello di capire meglio e modellare le relazioni nei dati. Idee di base:

- iniziare con delle feature
- processarle o trasformarle per creare delle nuove feature "ingegnerizate"
- usare queste nuove feature nel predittore

Il risultato ottenuto era soddisfacente? Ho migliorato le performance del modello?

- training del modello con le feature originali e validazione delle performance
- fare lo stesso con le feature nuove
- confrontare i risultati

3.1 Tipi di feature

3.1.1 Feature categoriche(nominali)

Prendono solo un numero finito di valori, ovvero $U = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$, dove ogni α_i è una etichetta di categoria. Per riferirci a queste possiamo usare anche semplicemente gli indici i. Per trattare queste feature $U = \{1, 2, \ldots, k\}$ possiamo effettuare un **one-hot embedding**, $\phi(i) = e_i \in \mathbb{R}^k$.

3.1.2 Feature ordinali

I dati sono categorici, con un ordine. Ad esempio: {strongly disagree, disagree, neutral, agree, strongly agree}
Ci sono due strategie:

- Embeddare in $\mathbb{R}$, con valori $\{-2, -1, 0, 1, 2\}$. **N.B.** sto assumendo distanza unitaria e uniforme.
- Usare un embedding one-hot, come per le feature categoriche.

Quando usare l'una o l'altra? Nella prima soluzione utilizzo una sola colonna, nella seconda ne uso k. Uso la prima strategia quando necessito della concezione di ordine, dato che è esplicitato con questo metodo. Non usiamo la prima strategia quando la distanza tra i valori non è uniforme. La seconda strategia viene utilizzata quando non è necessaria la nozione di ordine (che viene persa).
Cosa succede se codifico con one-hot e calcolo la distanza euclidea (radice della somma dei quadrati delle differenze)? **Attenzione!** La distanza euclidea è costante.

3.2 Tipi di trasformazioni di feature

1. **Modfica di feature individuali**: rimpiazzo la feature x_i con una feature trasformata o modificata x_i^{new} (utile quando voglio portare sulla stessa scala)
2. **Combinazione di feature**: creo nuove feature combinando più feature

3. **Creazione di nuove features da più vecchie features**. Che problemi ha? Viene modificata la relazione che sto apprendendo.

Con il metodo 2 e 3 ho il problema di **curse of dimensionality**, ovvero il numero di campioni (necessari per avere la stessa accuratezza) cresce esponenzialmente con il numero di feature. Ma nella pratica il numero di esempi di training è fisso, dunque le performance in generale calano per alti numeri di features. In molti casi l'informazione che viene persa dalla rimozione di variabili viene compensata da un miglior training in uno spazio di dimensione minore.

3.3 Feature reduction

In maniera ingenua possiamo pensare che più feature abbiamo, meglio è. In pratica, in molti casi, avere troppe feature può essere dannoso, vogliamo feature con un forte potere discriminante. Due strategie tipiche:

- **Feature selection**: selezionare un sottoinsieme delle feature esistenti, in modo da ridurre il numero di feature.

- **Feature extraction**: creare nuove feature a partire da quelle esistenti, in modo da ridurre il numero di feature. A partire da un insieme di feature $a = \{a_i | i = 1, \dots, n\}$, trovo un mapping $\hat{a} = f(a) : \mathbb{R}^n \to \mathbb{R}^m$, con $m < n$. Questo mapping deve essere fatto in modo da preservare la maggior parte delle informazioni e struttura presente in $\mathbb{R}^n$. Un **mapping ottimale** $\hat{a} = f(a)$ è quello che non risulta in un aumento della minima probabilità di errore. Un esempio di trasformazioni lineari è la principal component analysis.

3.4 Principal Component Analysis (PCA)

L'idea principale è quella di approssimare ogni dato x con una combinatone lineare di un numero ridotto di "vettori", chiamati **componenti principali**. Questi vettori formano il miglior sottospazio che fitta i dati. Il numero di componenti principali M è la nuova dimensione ridotta. Più formalmente:

- Siano $\phi_1, \dots, \phi_r \in \mathbb{R}^d$, inoltre $\Phi = [\phi_1, \dots, \phi_r]$ è una matrice $d \times r$.

- Le loro combinazioni lineari formano un sottospazio $S = \subseteq \mathbb{R}^d$, dove $S = \{\Phi a | a \in \mathbb{R}^r\}$

- La distanza da un punto x a S è definita come:

$$dist(x, S) = \min_a ||x - \Phi a||_2$$

- Il punto più vicino a x del sottospazio S è chiamato **proiezione** di x su S, ed è indicato con $\hat{x}$.

$$\hat{x} = \Phi(\Phi^T \Phi)^{-1} \Phi^T x$$

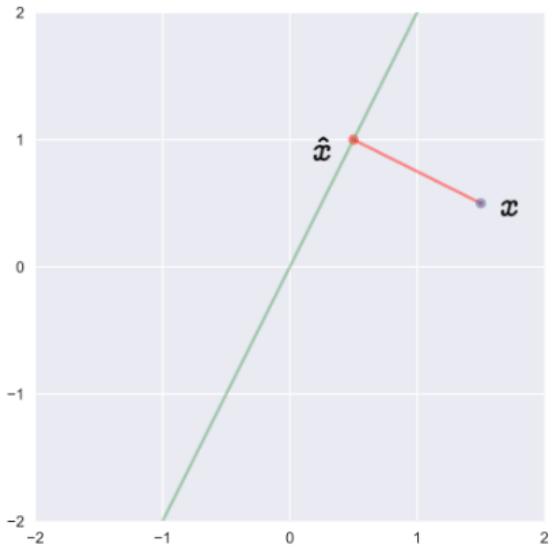


Figure 3.1: Proiezione di un punto su un sottospazio

- Ogni osservazione x viene assunta come "vicina" a una combinazione lineare di $\{\phi_1, \ldots, \phi_r\}$.
- Le colonne di Φ sono chiamate **componenti principali**.
- Il parametro $r < d$ è detto rango di riduzione, o dimensione del sottospazio.
- La funzione di loss associata è la distanza quadratica media tra i punti e le loro proiezioni:

$$l(\Phi) = \frac{1}{n}||x - \hat{x}||_2^2$$

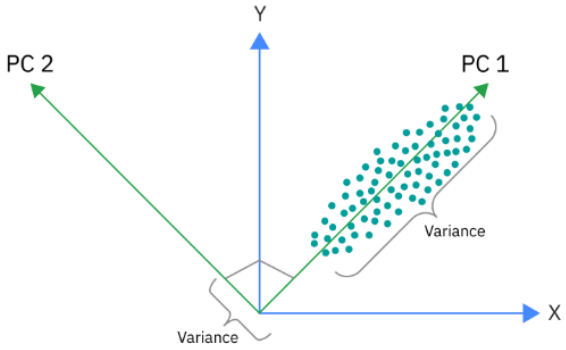


Figure 3.2: Esempio di PCA in 2D

Un nuovo possibile sistema di coordinate (quello verde in figura 3.2) è quell dove gli assi sono uguali agli autovettori della matrice di covarianza del dataset. Adesso immaginiamo di poter ruotare gli assi, in modo che nel nuovo sistema di coordinate la matrice di covarianza sia:

$$Cov(x, y) = \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix}$$

In questo modo le due variabili non sono correlate tra loro.

Termine	Espressione	Descrizione
Matrice di covarianza	$\Sigma = \frac{1}{n} X^T X$	Descrive la variabilità dei dati
Autovettore	ϕ_i	Direzione principale (componente principale)
Autovalore	λ_i	Varianza spiegata lungo ϕ_i
Matrice delle componenti	$\Phi = [\phi_1, \ldots, \phi_r]$	Base del sottospazio ridotto
Proiezione	$\hat{x} = \Phi\Phi^T x$	Punto proiettato sul sottospazio

Table 3.1: Riepilogo dei concetti chiave dell'Analisi delle Componenti Principali (PCA).

Quindi la PCA è una trasformazione lineare che proietta il dataset in un nuovo sistema di coordinate, dove la prima coordinata ha più varianza, ogni coordinata successiva ha varianza decrescente e tutte le coordinate sono ortogonali tra loro.
In pratica trasformiamo un insieme di x variabili correlate in un insieme di p componenti principali non correlate.

1. Creare la matrice X di dimensione $K \times N$ dei dati, dove ogni riga è un'osservazione e ogni colonna una feature.
2. Calcolare la matrice di covarianza S, basata su X. Questo cattura la ridondanza del dataset.
3. Risolvere $S\underline{e} = \lambda\underline{e}$ per trovare gli autovettori $\underline{e}$ e autovalori λ di S. L'autovettore è una direzione, l'autovalore ci dice quanta varianze c'è nei dati in quella direzione.
4. Risolvere $P = X\underline{e}$ per calcolare le componenti principali (N componenti)

Tecnicamente, la quantità di varianza catturata dalla i-esima componente principale è misurata dall'auto valore λ_i. La PCA è particolarmente utile quando le variabili nel dataset sono altamente correlate. La correlazione indica che c'è ridondanza nei dati. A causa della ridondanza, la PCA può essere usata per ridurre le variabili originali in un numero minore di variabili (componenti principali), spiegando la maggior parte della varianza delle variabili originali.
Gli autovalori possono essere usati per determinare il numero di PC da mantenere dopo la PCA.

- Un autovalore > 1 indica che le PC contano di più per la varianza che una delle variabili originali.
- In maniera alternativa, possiamo limitare il numero di componenti al numero che spiega una certa percentuale della varianza totale.

Se voglio ad esempio spiegare almeno il 70% dell'esempio in figura 3.3, scelgo le prime 3 componenti.

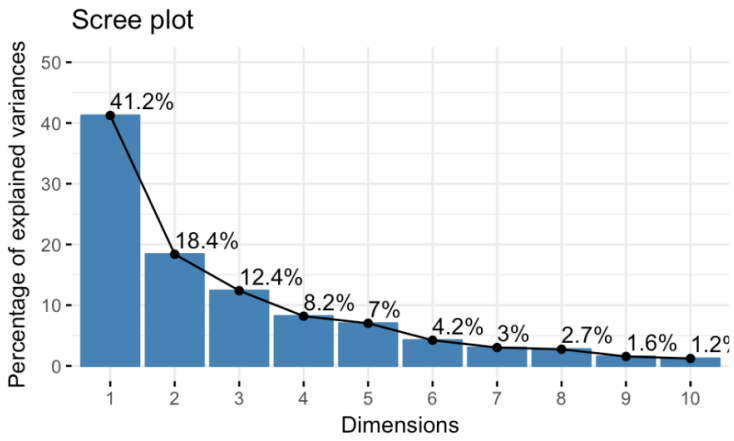


Figure 3.3: Varianza spiegata dalle componenti principali

3.4.1 Analisi esplorativa dei dati

Consideriamo uno dei dataset più famosi, l'iris dataset.

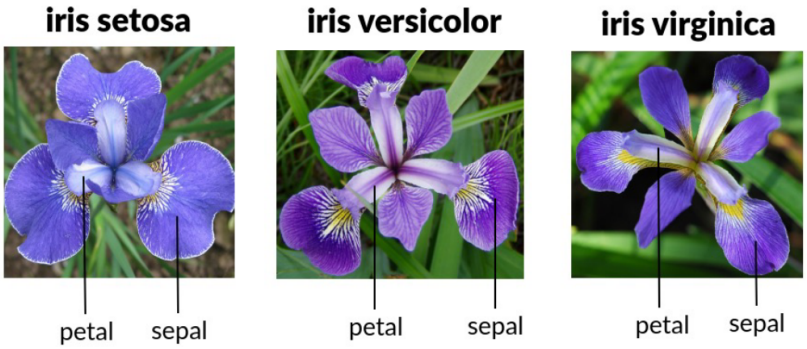


Figure 3.4: Oggetto del dataset iris

Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
1	5.1	3.5	1.4	0.2	Iris-setosa
2	4.9	3.0	1.4	0.2	Iris-setosa
3	4.7	3.2	1.3	0.2	Iris-setosa
4	4.6	3.1	1.5	0.2	Iris-setosa
5	5.0	3.6	1.4	0.2	Iris-setosa
⋮	⋮	⋮	⋮	⋮	⋮
146	6.7	3.0	5.2	2.3	Iris-virginica
147	6.3	2.5	5.0	1.9	Iris-virginica
148	6.5	3.0	5.2	2.0	Iris-virginica
149	6.2	3.4	5.4	2.3	Iris-virginica
150	5.9	3.0	5.1	1.8	Iris-virginica

Table 3.2: Iris dataset — sample rows

Adesso mostriamo la distribuzione della lunghezza del petalo rispetto le tre classi.

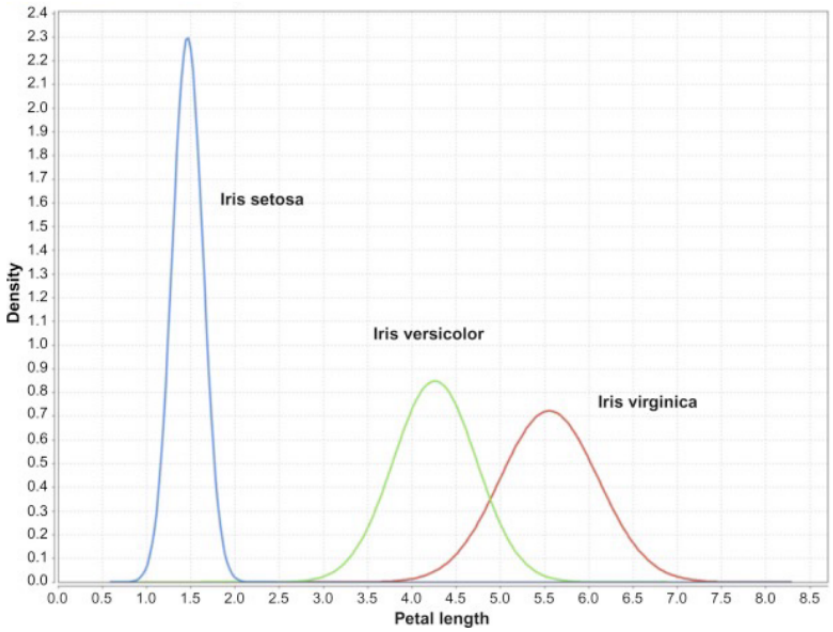


Figure 3.5: Distribuzione della lunghezza del petalo per le tre specie di iris

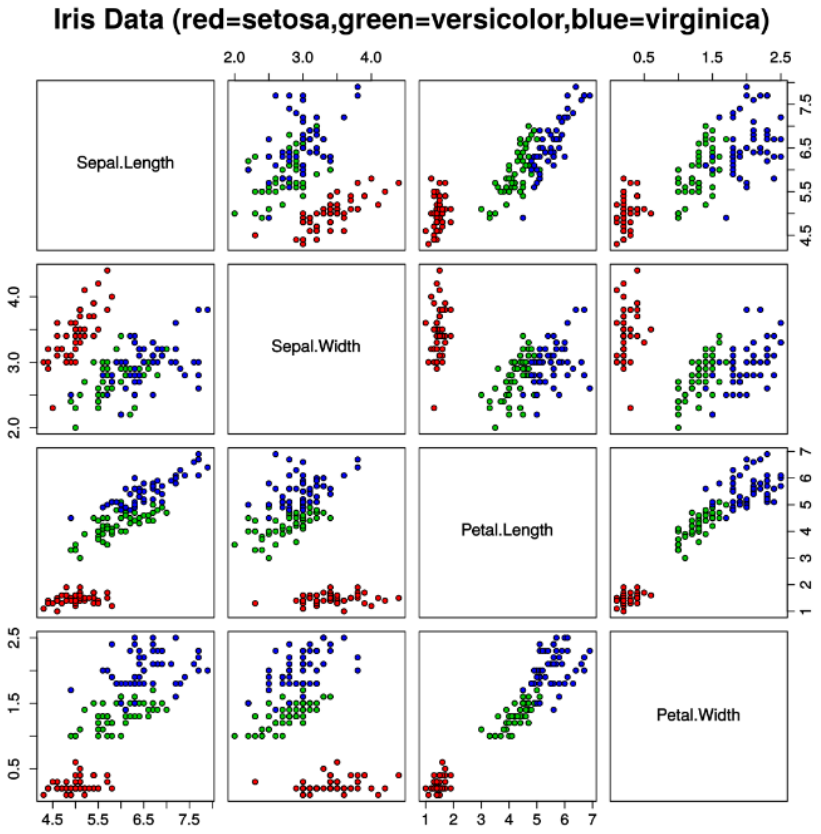


Figure 3.6: Distribuzione delle feature del dataset iris

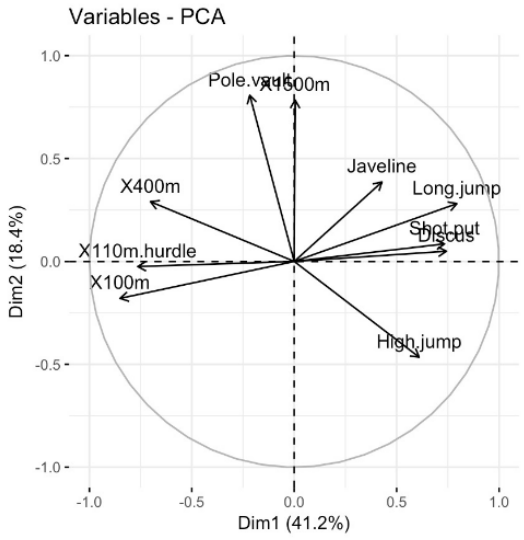


Figure 3.7: PCA del dataset iris

Possiamo osservare le componenti per analizzare la correlazione tra le feature. Le variabili correlate positivamente sono raggruppate insieme, quelle correlate negativamente sono posizionate ai poli opposti del plot rispetto l'origine. La distanza tra le variabili e l'origine misura la qualità della variabile, quelle lontane dell'origine sono ben rappresentate. Un vettore lungo indica una buona rappresentazione dell'individuo sulle componenti principali. Un vettore corto indica che l'individuo non è ben rappresentato dalle componenti principali.

Chapter 4

Alberi di decisione

Gli alberi di decisione sono uno dei metodi più utilizzati nella pratica per l'inferenza induttiva. L'obiettivo è quello di imparare ad approssimare funzioni su valori discreti, in modo da poter modellare espressioni disgiuntive e al tempo stesso essere robusti rispetto a dati rumorosi. La funzione appresa viene rappresentata come un albero, in maniera alternativa come usa serie di `if-then-else`. Riprendendo i concetti di approssimazione di una funzione:

- X è l'insieme delle istanze.
- Y è l'insieme delle etichette (classi).
- $f : X \to Y$ è la funzione target (sconosciuta).
- $G = \{g : X \to Y\}$ è l'insieme delle ipotesi.

L'input del problema è l'insieme di esempi della funzione target F di addestramento $\{< x_i, y_i >\}_{i=1}^{n}$ e l'output è l'ipotesi $h \in G$ che approssima meglio f. Come avviene l'approssimazione negli alberi di decisione? Ogni $x \in X$ è un'istanza, ovvero un vettore di feature. I valori di Y sono discreti. Ogni funzione g è un albero, dove x viene classificato scendendo nell'albero fino alle foglie, dove gli viene assegnata una y. Da notare che non tutti gli attributi necessariamente concorrono alla classificazione. Gli

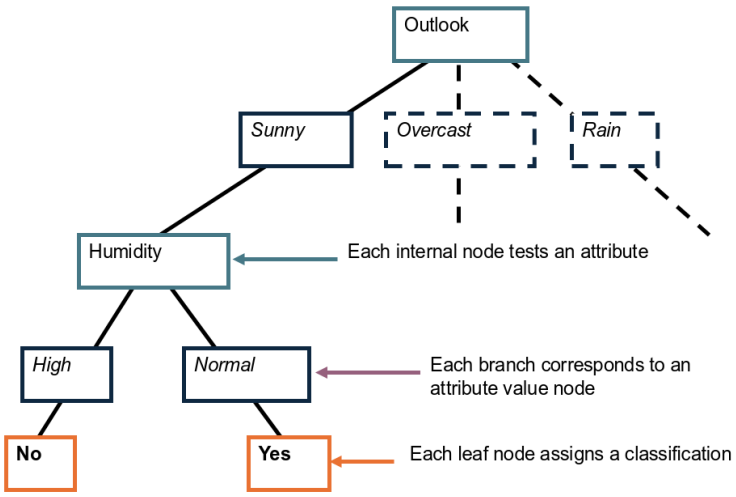


Figure 4.1: Esempio di albero di decisione per la classificazione

alberi di decisione rappresentano quindi disgiunzioni di congiunzioni di condizioni sugli attributi. Questo perché ogni cammino dalla radice a una foglia rappresenta una congiunzione di condizioni sugli attributi, l'insieme dei cammini che portano a una stessa etichetta rappresenta una disgiunzione di congiunzioni. Perché usare gli alberi di decisione? Esplicitano la relazione tra attributi e sono facilmente interpretabili.

Consideriamo allora l'insieme di disgiunzioni di congiunzioni, se non ci sono esempi contraddittori allora

la formula è vera per ogni esempio positivo e falsa per ogni esempio negativo, dunque è consistente con il dataset di addestramento. Vogliamo però inoltre che la formula sia il più piccola possibile (rimanendo valida), dato che è la nostra euristica per una buona generalizzazione. Vogliamo trovare la formula equivalente minima. Posso rimuovere controlli non necessari, ad esempio se ho la formula

$$(\text{Outlook} = \text{Rain}) \wedge (\text{Temperature} = \text{Hot}) \wedge (\text{Humidity} = \text{High}) \wedge (\text{Wind} = \text{Weak}) \vee$$

$$(\text{Outlook} = \text{Rain}) \wedge (\text{Temperature} = \text{Hot}) \wedge (\text{Humidity} = \text{High}) \wedge (\text{Wind} = \text{Strong})$$

dato che l'ultimo attributo non influenza il risultato (weak e strong sono tutti i possibili valori di wind) posso rimuoverlo, ottenendo

$$(\text{Outlook} = \text{Rain}) \wedge (\text{Temperature} = \text{Hot}) \wedge (\text{Humidity} = \text{High}).$$

Un altro caso è quando due o più congiunzioni contengono la stessa sottoformula, dunque non dovrebbero essere valutate nuovamente.
Gli alberi di decisione dividono lo spazio delle feature in iper rettangoli con assi paralleli agli assi delle feature. Ogni regione rettangolare viene etichettata con una classe (o una distribuzione di probabilità sulle classi). Questo partizionamento è chiamato *decision boundary*.

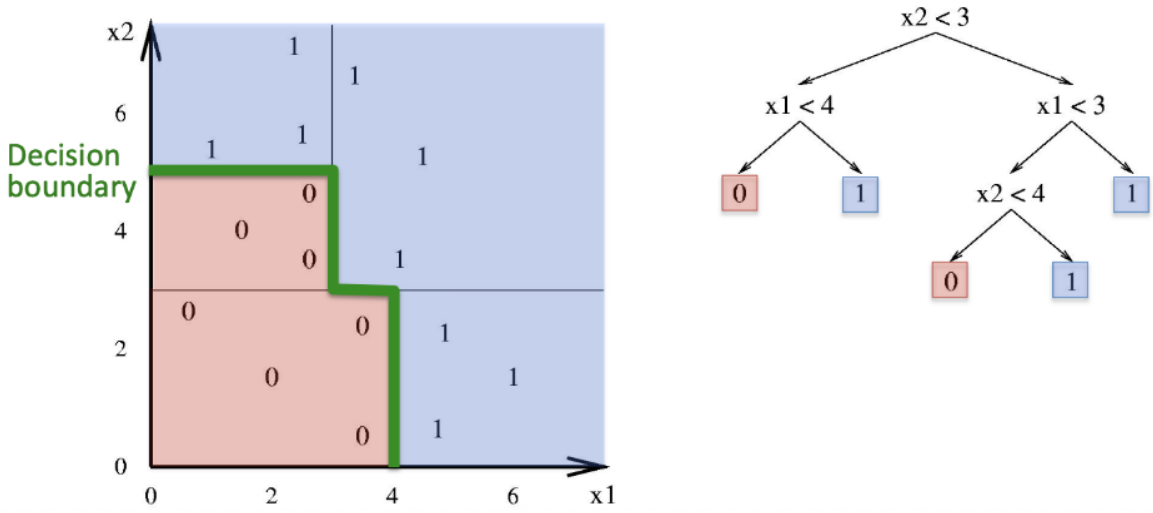


Figure 4.2: Esempio di partizionamento dello spazio delle feature tramite un albero di decisione

4.1 Espressività degli alberi di decisione

Gli alberi di decisione possono esprimere qualsiasi funzione sugli attributi di un input. Trivialmente, esiste un albero di decisione consistente per ogni insieme di training con un cammino verso una foglia per ogni esempio, ma probabilmente questo non generalizzerà per nuovi esempi. Un "buon" attributo separa gli esempi in sottoinsiemi che sono (idealmente) tutti negativi o tutti positivi.

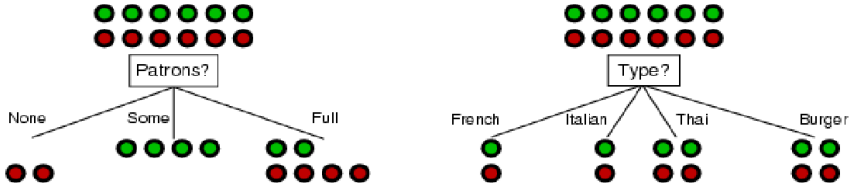


Figure 4.3: Esempio di albero di decisione che rappresenta la funzione XOR

Osservando la figura 4.3, quale attributo separa meglio gli esempi? Patrons, perché divide gli esempi in due insiemi, uno con tutti esempi positivi e uno con esempi negativi e positivi, tranne in un ramo in cui ho una distribuzione. Nel secondo esempio invece non siamo in grado di separare gli esempi basandoci su quell'attributo.

Al crescere del numero di nodi (o la profondità) dell'albero, cresce la dimensione dello spazio delle ipotesi. Quanti alberi di decisione distinti ci sono con n attributi booleani?

- Il numero di funzioni booleane su n argomenti.
- Il numero di tabelle di verità distinte con 2^n righe.
- 2^{2^n} tabelle di verità (dato che ogni riga può essere 0 o 1).

4.2 Apprendimento degli alberi di decisione

Gli alberi di decisione effettuano l'analisi del rischio empirico con diverse funzioni di loss in base al problema (regressione o classificazione).

4.2.1 Alberi di decisione di classificazione

Vediamo gli alberi di decisione per la classificazione. Vogliamo minimizzare il rischio empirico sulla funzione di loss 0-1:

$$R_{emp}(g) = \frac{1}{n}\sum_{i=1}^{n} 1(g(x_i) \neq y_i)$$

Dove la funzione di loss 0-1 è definita come:

$$L(y, g(x)) = \begin{cases} 0 & \text{se } y = g(x) \\ 1 & \text{se } y \neq g(x) \end{cases}$$

Qual è un problema con questa funzione di loss? La funzione di loss 0-1 non è differenziabile, dunque

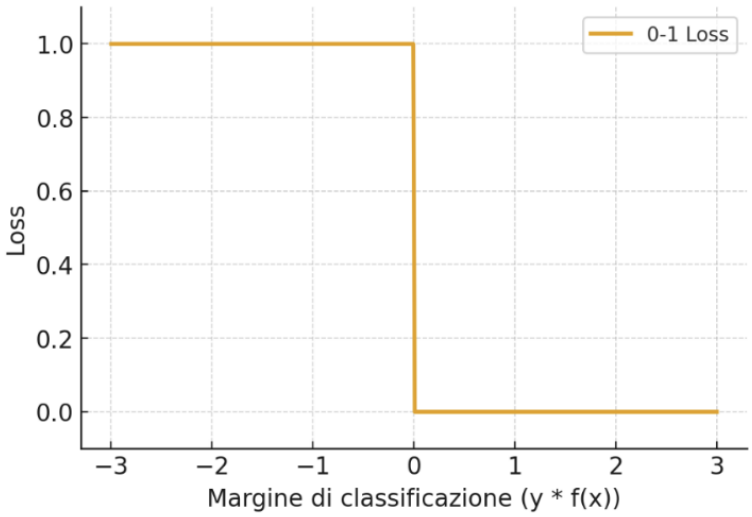


Figure 4.4: Funzione di loss 0-1

non posso usare metodi basati sul calcolo del gradiente per minimizzarla. Un modo per aggirare questo problema è usare funzioni di loss surrogate, più facili da computare, che siano differenziabili e che correlino con la funzione di loss originale.

4.2.2 Alberi di decisione di regressione

Come funzione di rischio empirico medio possiamo usare l'errore quadratico medio:

$$R_{emp}(g) = \frac{1}{n}\sum_{i=1}^{n}(y_i - g(x_i))^2$$

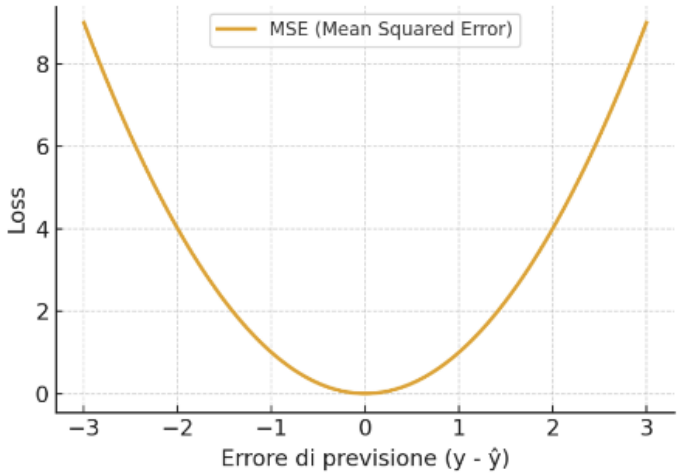


Figure 4.5: Funzione di loss MSE

4.2.3 Difficoltà nel training

In generale fare training di alberi di decisione è un problema NP-completo. Dunque si usano algoritmi che utilizzano euristiche greedy per costruire l'albero in modo incrementale. Si inizia da un albero vuoto, e a ogni passo si sceglie il miglior attributo su cui fare lo split, basandosi su una metrica, poi si ripete il processo ricorsivamente. Inoltre, è possibile fermarsi a un albero parziale, senza arrivare alle foglie, perché? Secondo il principio del **rasoio di Occam**, l'ipotesi più semplice che spiega i dati è da preferire.

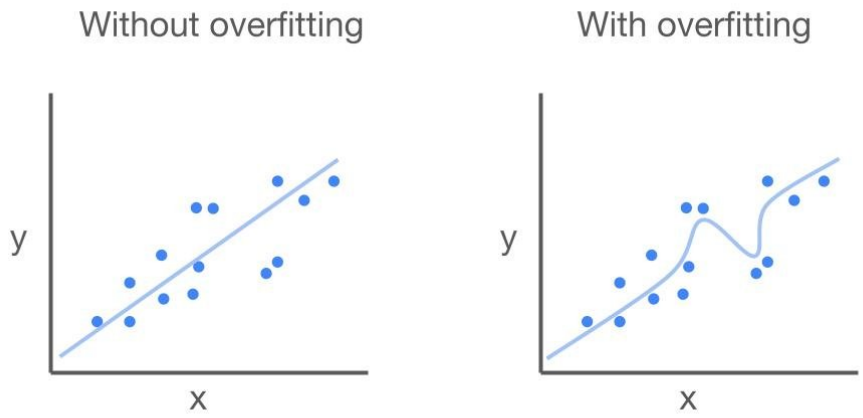


Figure 4.6: Esempio di overfitting

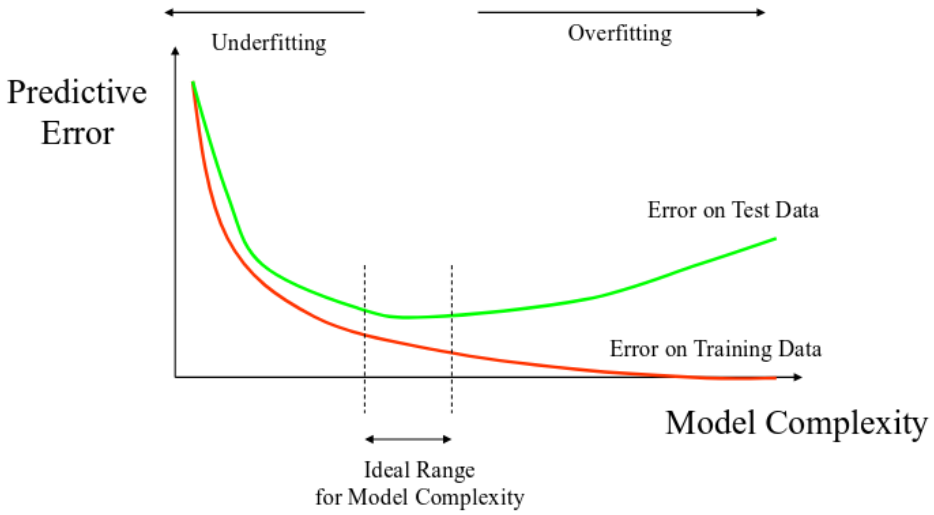


Figure 4.7: Errori in funzione della complessità del modello

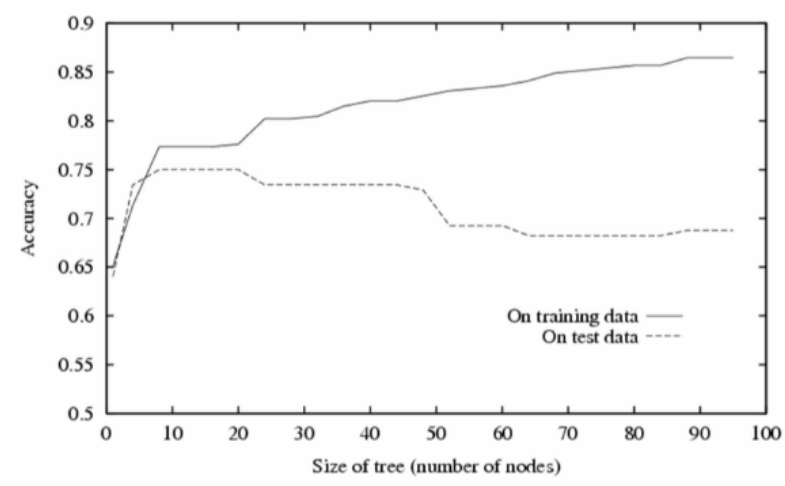


Figure 4.8: Accuratezza in funzione della profondità dell'albero

Come posso evitare l'overfitting?
Posso fermare la crescita dell'albero quando la separazione dei dati non è più significativa, oppure posso creare l'albero intero e poi potare i rami dopo.
Come seleziono il "miglior" albero?
Posso misurare le performance su un training set, oppure usare la usare un dataset di validazione.

Algorithm 1: Algoritmo base per l'apprendimento di alberi di decisione

Input: Esempi di addestramento S, insieme di attributi A
Output: Albero di decisione con radice `node`
node := radice dell'albero di decisione;
while *esistono nodi da espandere* **do**
  A := scegliere il "miglior" attributo di decisione per il nodo corrente;
  assegnare A come attributo di decisione per il nodo;
  foreach *valore v di A* **do**
    creare un nuovo discendente del nodo associato a v;
  smistare gli esempi di addestramento ai nodi foglia;
  if *gli esempi di addestramento nei nodi foglia sono perfettamente classificati* **then**
    fermarsi;
  else
    ricorrere sui nuovi nodi foglia;
return *node*;

f

4.3 Algoritmi di apprendimento su alberi di decisione

Come determinare il "miglior" attributo su cui fare lo split? L'idea di base è che un buon attributo

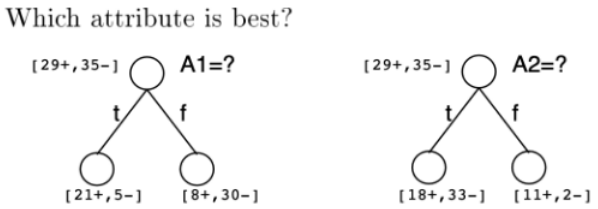


Figure 4.9: Determinazione dell'attributo

separa bene gli esempi in sottoinsiemi che sono (idealmente) tutti negativi o tutti positivi.

4.3.1 CART (1984)

Con questo algoritmo viene effettuata una divisione binaria ricorsivamente.

- Per ogni nodo, viene cercata la miglior separazione binaria tra tutte le possibili feature e tutti i possibili valori di soglia.
- La separazione viene scelta in base alla riduzione dell'impurità (misurata tramite Gini impurity index).

L'indice di Gini è una misura di quanto "puro" o "impuro" è un nodo in un albero di classificazione, ovvero misura la distribuzione delle classi al livello di nodo.

$$Gini(D) = \sum_{i=1}^{J}(p_i \sum_{k \neq i} p_k) = \sum_{i=1}^{J}(p_i - p_i^2) = \sum_{i=1}^{J} p_i - \sum_{i=1}^{J} p_i^2 = 1 - \sum_{i=1}^{J} p_i^2$$

Il livello massimo di purità è 0, quando tutti gli esempi appartengono a una sola classe. Il livello massimo di impurità è $\frac{m-1}{m}$, avendo m classi.

Problemi di CART

CART gestisce bene i dati numerici, ma non è ottimale per dati categorici con più di due valori distinti. Supponiamo di avere un attributo categorico con k valori distinti, allora il numero di possibili split binari è $2^{k-1} - 1$, che cresce esponenzialmente con k.

4.3.2 ID3 (1986)

L'algoritmo ID3 (Iterative Dichotomiser 3) è un altro algoritmo per la costruzione di alberi di decisione. Utilizza un approccio top-down per dividere le feature a ogni passo per costruire un albero di decisione. Il funzionamento è basato sul concetto di **entropia**.

Entropia

Sia S l'insieme degli esempi di addestramento, e $p_\oplus$ e $p_\ominus$ le proporzioni di esempi positivi e negativi in S. L'entropia misura l'impurità di S.

$$Entropy(S) = -p_\oplus \log_2 p_\oplus - p_\ominus \log_2 p_\ominus$$

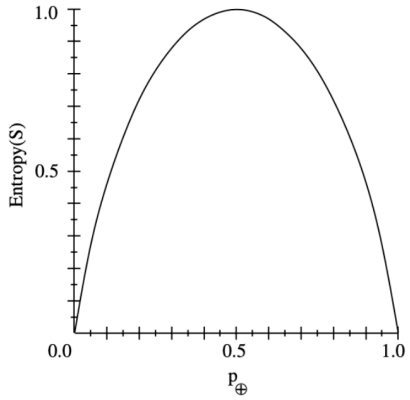


Figure 4.10: Valori di entropia in funzione della proporzione di esempi positivi

Una definizione alternativa dell'entropia ci dice che $Entropy(S)$ è la quantità media di informazione (in bit) necessaria per codificare la classe di un esempio scelto casualmente da S, in una codifica ottimale di lunghezza minima.
L'entropia misura quindi il livello di incertezza associato alla classificazione di un esempio scelto casualmente da S. Di conseguenza ci dice il numero di bit necessari in media per codificare la classe di un esempio scelto casualmente da S.

- Se tutti gli esempi appartengono alla stessa classe ("+"), non c'è incertezza → non servono bit per comunicare l'etichetta → entropia = 0.
- Se metà sono "+" e metà sono "−", l'incertezza è massima → servono più bit per codificare la classe → entropia massima (= 1 bit, nel caso binario).
- Se il 90% è "+" e il 10% è "−", in media servono meno di 1 bit: entropia ≈ 0.47.

Situazione	Probabilità di classe	Entropia (bit)	Significato
Tutti "+"	$(p_+ = 1,\ p_- = 0)$	0	Nessuna incertezza → non servono bit
50% "+", 50% "−"	$(p_+ = 0.5,\ p_- = 0.5)$	1	Incertezza massima → 1 bit
90% "+", 10% "−"	$(p_+ = 0.9,\ p_- = 0.1)$	≈ 0.47	In media, meno di 1 bit

Table 4.1: Esempi numerici di entropia per una distribuzione binaria

Secondo la teoria dell'informazione di Shannon, l'informazione associata a un evento con probabilità p è definita come:

$$I(p) = -\log_2(p)$$

Ciò rappresenta inoltre il numero minimo di bit (mediamente) necessari per codificare l'evento. Al crescere della probabilità di un evento, il numero di bit necessari per codificarlo diminuisce.

Dato che possiamo estrarre casualmente un esempio + con probabilità $p_\oplus$ e un esempio − con probabilità $p_\ominus$, il numero di bit necessari è:

$$Entropy(S) = -[p_\oplus \log_2(p_\oplus) + p_\ominus \log_2(p_\ominus)]$$

Qual è la relazione tra entropia e apprendimento degli alberi di decisione? L'algoritmo cerca di ridurre l'entropia scegliendo gli attributi che separano i dati in sottoinsiemi più puri. Ogni separazione fornisce informazioni, dunque riduce il numero medio di bit necessari per codificare la classe, ovvero riduce l'entropia.
Facciamo un esempio su un dataset con 10 esempi, 4 positivi e 6 negativi.

Person	Age	Income	Buy?
1	young	high	No
2	young	mid	No
3	young	low	Yes
4	adult	high	Yes
5	adult	mid	Yes
6	adult	low	Yes
7	elderly	high	No
8	elderly	mid	No
9	elderly	low	No
10	young	low	Yes

Table 4.2: Esempio: età, reddito e decisione di acquisto

L'entropia iniziale (al nodo radice) è:

$$p_{yes} = \frac{4}{10}, \quad p_{no} = \frac{6}{10}$$

$$H(S) = -\left(\frac{4}{10}\log_2\frac{4}{10} + \frac{6}{10}\log_2\frac{6}{10}\right) \approx 0.97 \text{ bit}$$

Questo vuol dire che se volessimo codificare la classe (si/no) di una persona scelta casualmente, in media servirebbero circa 0.97 bit, questo vuol dire che l'incertezza è alta. Vogliamo quindi trovare una divisione che riduce l'incertezza.
Proviamo a dividere in base all'attributo Age:

Age	Yes	No	Totale	p_{yes}	Entropia
young	2	2	4	0.5	1.0
adult	3	0	3	1.0	0.0
elderly	0	3	3	0.0	0.0

Table 4.3: Split per attributo Age: conteggi, probabilità di "Yes" ed entropia per ciascun sottoinsieme

Calcoliamo ora l'entropia media dopo lo split:

$$H_{after} = \frac{4}{10}\cdot 1.0 + \frac{3}{10}\cdot 0.0 + \frac{3}{10}\cdot 0.0 = 0.4 \text{ bit}$$

L'entropia è diminuita da 0.97 bit a 0.4 bit, dunque lo split ha ridotto l'incertezza. Misuriamo questa riduzione di entropia tramite l'**information gain**:

$$IG(S, \text{Age}) = H(S) - H_{after} = 0.97 - 0.4 = 0.57 \text{ bit}$$

Ora immaginiamo di creare un albero di decisione basandoci su un unico attributo. Vogliamo decidere se usare Humidity o Wind come attributo di split.

Day	Outlook	Temp.	Humidity	Wind	Play Tennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Table 4.4: Esempio: dataset "Play Tennis" (14 esempi)

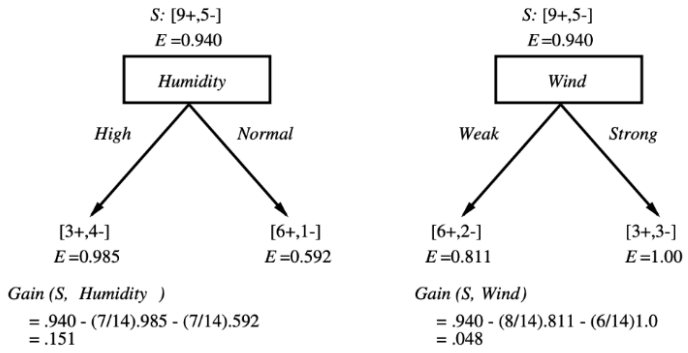


Figure 4.11: Alberi di decisione basati su Humidity e Wind

Dunque possiamo concludere che Humidity è un attributo migliore di Wind per fare lo split, dato che riduce maggiormente l'entropia.

Problemi di ID3

- L'entropia al livello teorico è una buona strategia, ma computazionalmente è costosa dato il calcolo del logaritmo (si può approssimare con le serie di Taylor).
- L'information gain è sensibile alle classi molto sbilanciate (entropia risultante molto alta dunque basso information gain).
- Bias rispetto agli attributi che possono assumere molti valori distinti (può causare overfitting).

4.3.3 C4.5 (1993)

Questo algoritmo utilizza il *gainratio* per regolare l'information gain dividendolo per una misura di **split information** che riflette quanto largamente un dato sia diviso.

$$Gain\ Ratio(S, A) = \frac{Information\ Gain(S, A)}{Split\ Information(S, A)}$$

Dove lo split information è definito come:

$$Split\ Information(S, A) = -\sum_{v \in Values(A)} p_v \log_2 p_v$$

Split information è alto quando un attributo divide i dati in molte piccole porzioni. Dividendo IG per lo split information, il gain ratio penalizza gli attributi che creano troppe partizioni.
Se l'attributo A divide i dati in modo equo (e.g. 50-50), lo split information è alto, perché i dati vengono "distribuiti" in maniera equa tra le partizioni. Se invece un attributo crea delle partizioni molto sbilanciate (e.g. 95-5), lo split information è basso, perché viene a malapena creata una partizione.

Attributo	Valori distinti	SplitInfo	Nota
A	60 / 40	≈ 0.97	Split bilanciato, informazione media
B	50 / 50	1.00	Split perfettamente bilanciato
C	100 valori unici	6.64	Split information molto alta $\Rightarrow$ l'Information Gain viene penalizzato
D	90 / 5 / 5	0.57	Split molto sbilanciato, scarsa informazione utile

Table 4.5: Confronto di split per diversi attributi

Attributo	Valori distinti	Information Gain	Split Info	Gain Ratio
Color	3	0.30	1.20	0.25
Student ID	100	1.00	6.60	0.15
Age Group	4	0.40	1.50	0.27

Table 4.6: Confronto di Information Gain, Split Info e Gain Ratio per alcuni attributi

Come vengono gestiti gli attributi numerici in C4.5?

1. Ordinare i valori numerici dell'attributo.
2. Considerare dei valori di soglia candidati, tipicamente i valori medi tra i valori ordinati dove la label cambia.
3. Separare in due gruppi per ogni soglia.
4. Calcolare il gain ratio per ogni soglia.
5. Scegliere la soglia con il gain ratio più alto.

Problemi di C4.5

Il problema principale è che il gain ratio favorisce gli split che dividono poco i dati, ad esempio con split info molto basso. Se split info è molto basso, il denominatore del gain ratio è piccolo, dunque il gain ratio diventa artificialmente alto anche se l'information gain è basso. Dunque il modello potrebbe scegliere divisioni con "rami sbilanciati" che non migliorano la classificazione, peggiorando le performance.
Inoltre, C4.5 non sceglie direttamente gli attributi con il gain ratio più alto, ma prima filtra gli attributi con information gain molto basso, poi seleziona gli attributi con il gain ratio più alto tra quelli rimanenti. Il gain ratio quindi corregge il bias dell'information gain rispetto agli attributi con molti valori distinti, ma introduce un bias verso attributi che dividono poco i dati (basso split info).

Gestione dei valori mancanti

Quando un valore di un attributo è mancante su delle istanze, l'istanza non viene scartata. Viene stimata la probabilità di ogni possibile valore dell'attributo mancante basandosi sulle altre istanze note. Quando viene computato l'information gain e il gain ratio per quell'attributo, l'istanza contribuisce frazionalmente a ogni possibile ramo, pesata dalla probabilità stimata di quel valore.

Chapter 5

Percettrone

Un percettrone è un classificatore lineare, ovvero classifica l'input disegnando una linea (o iperpiano in dimensioni più alte) che separa le classi. Prendendo in input un vettore di feature x, lo moltiplica per un vettore di pesi w e aggiunge un bias b, dal risultato classifica l'input in base a una soglia.

$$y = w \cdot x + b$$

Perché è necessario il bias?
Il bias permette di spostare la linea di decisione lontano dall'origine, altrimenti la linea passerebbe sempre per l'origine. Il bias si comporta come il termine noto in una equazione lineare, dando più flessibilità al modello nel posizionare la linea di decisione.

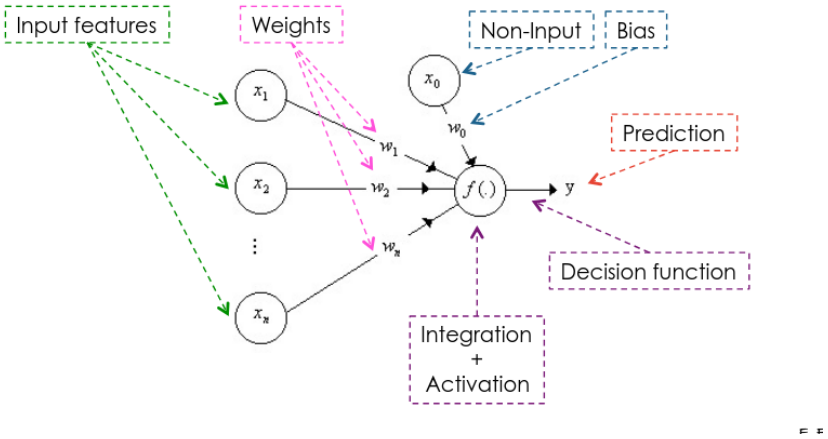


Figure 5.1: Schema di un percettrone

Il percettrone prende in input un vettore di feature $x = (x_1, x_2, \ldots, x_n)$, pesa ogni feature e fornisce in output una variabile booleana, $y \in \{-1, 1\}$, calcolata attraverso una funzine di attivazione.

$$f(x, w) = \underbrace{\text{sign}}_{\text{Attivazione}} \Big(\underbrace{w_1 x_1 + \cdots + w_n x_n + w_0 x_0}_{\text{Integrazione}} \Big)$$

dove $x_0 = 1$ è il termine di bias e w_0 è il peso associato al bias che modifica la soglia di attivazione.

5.1 Apprendimento sul percettrone

Etichettiamo le classi positive e negative come $+1$ e -1.
Determiniamo un insieme di parametri ottimali $\theta = \{w_1, w_2, \ldots, w_m\}$ che porti a una combinazione lineare delle feature x e pesi w tali che il modello classifichi tutte le istanze di training.
Usiamo una funzione di step come funzione di attivazione $f(w, x)$, dove se $f(w, x)$ è maggiore di un valore di soglia α allora l'output è 1, altrimenti è -1.

Una **unit step function** può essere vista come un classificatore basato su threshold, ovvero decide se il neurone si attiva o meno in base al valore di input. $f(x) = \begin{cases} +1 & \text{se } wx + b \geq 0 \\ -1 & \text{se } wx + b < 0 \end{cases}$

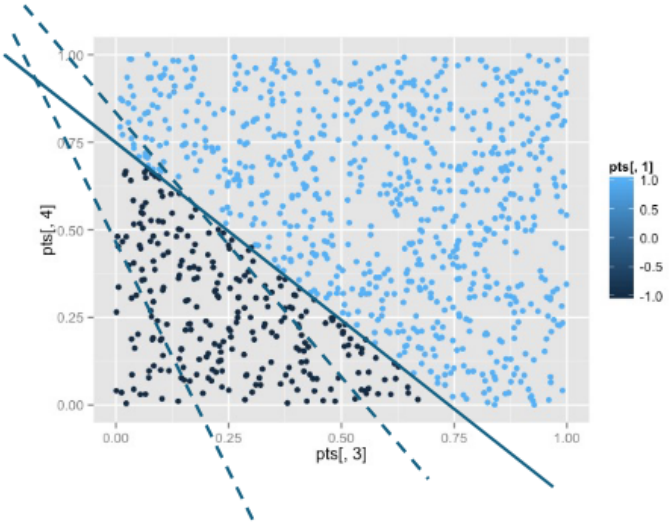


Figure 5.2: Classificazione tramite percettrone

Allora dando un algoritmo di apprendimento per il percettrone:

1. Vettore di feature $x = (x_1, x_2, \ldots, x_n)$.
2. Inizializza i pesi w a valori casuali piccoli.
3. Inizializzo il bias b.
4. Per ogni esempio di training x_i ho una label target y_i.
5. Calcolo l'output del percettrone $\hat{y_i} = sign(wx + b)$.

obiettivo: aggiornare il vettore w solo quando $\hat{y_i} \neq y_i$. La regola di aggiornamento è:

$$w_{\text{new}} = w_{\text{old}} + yx$$
$$b_{\text{new}} = b_{\text{old}} + y$$

se $y(wx + b) > 0$ la classificazione è corretta, altrimenti si aggiorna w spostandolo nella direzione di x se $y = 1$ o nella direzione opposta se $y = -1$.
In altre parole:

$$\text{Se } y(wx + b) \leq 0 \quad \text{allora aggiorniamo: } \begin{cases} w \leftarrow w + yx \\ b \leftarrow b + y \end{cases}$$

Quando $y = 1$ e viene predetto -1, il peso viene aumentato di x, spostando la linea di decisione verso gli esempi positivi. Quando $y = -1$ e viene predetto 1, il peso viene diminuito di x, spostando la linea di decisione verso gli esempi negativi.

Quali sono le forti limitazioni del percettrone semplice?
L'aggiornamento dei pesi viene effettuato di una magnitudine fissa, determinata unicamente dall'input vector x e dalla sua label y.
Possiamo elencare i seguenti problemi:

1. Grandezza del passo non controllabile:
   - La grandezza dell'aggiornamento dipende solo da $||x||$.

- Se gli input variano molto nella loro scala, gli aggiornamenti potrebbero essere troppo grandi (overshooting) o troppo piccoli (convergenza lenta).
- Non c'è un meccanismo di controllo o normalizzazione di questo effetto.

2. Oscillazioni vicino al decision boundary:
   - Senza un fattore di smorzamento (learning rate), il percettrone può oscillare sul confine di decisione, quindi diverge.
3. Sensibilità alla scala di input:
   - Se una feature ha una scala molto più grande rispetto alle altre, può dominare l'aggiornamento dei pesi, rendendo non stabile numericamente l'aggiornamento.

5.1.1 Learning rate

Una soluzione a questi problemi è l'introduzione di un **learning rate** ($\eta > 0$ piccolo), un fattore di scala che modula la grandezza dell'aggiornamento dei pesi. La regole di aggiornamento diventano:

$$w_{\text{new}} = w_{\text{old}} + \eta y x$$
$$b_{\text{new}} = b_{\text{old}} + \eta y$$

Un η grande porta a un apprendimento più veloce ma rischia di fare overshooting. Un η piccolo porta a un apprendimento più stabile ma più lento.
η può essere adattato durante l'addestramento, iniziando con un valore più grande e diminuendolo nel tempo per convergere, questa è la fondazione delle tecniche di ottimizzazioni moderne.

L'aggiornamento dei pesi può avvenire in due modi:

- **Online**: i pesi vengono aggiornati dopo ogni esempio di training → **Stochastic Gradient Descent**.
- **Batch**: gli aggiornamenti vengono accumulati su tutti gli esempi che vengono classificati male.

Theorem 1 ***Teorema di Rosenblatt***
Se il dataset di training è linearmente separabile, l'addestramento del percettrone converge sempre a una ipotesi consistente dopo un numero finito di epoche, per ogni $\eta > 0$. Se il dataset non è linearmente separabile, dopo un certo numero di epoche i pesi cominceranno a oscillare.

5.1.2 Delta rule

Invece di aggiornare i pesi solo quando una classificazione viene fatta in modo errato, possiamo usare la **delta rule** che aggiorna i pesi in proporzione alla magnitudo dell'errore (ovvero la differenza tra l'output desiderato e l'output reale).
Dato un vettore di input x, la label target y, l'output del percettrone $\hat{y} = wx + b$ e il learning rate η, la regola di aggiornamento diventa:

$$w_{\text{new}} = w_{\text{old}} + \eta(y - \hat{y})x$$
$$b_{\text{new}} = b_{\text{old}} + \eta(y - \hat{y})$$

Vogliamo che il neurone produca un output $\hat{y}$ il più vicino possibile alla label target y. Dunque, definiamo una funzione di loss che misura quanto si sia sbagliato il modello:

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

Quindi l'obbiettivo è minimizzare questo errore rispetto ai pesi w e al bias b. Calcoliamo come l'errore L varia rispetto a ogni peso w_j:

$$\frac{\partial L}{\partial w_j} = -(y - \hat{y})x_j$$

e rispetto al bias:

$$\frac{\partial L}{\partial b} = -(y - \hat{y})$$

Queste derivate ci dicono in quale direzione spostare ogni parametro per ridurre l'errore.

La discesa del gradiente ci dice che dobbiamo aggiornare i pesi e il bias nella direzione opposta al gradiente (visto che il gradiente punta nella direzione di massima crescita dell'errore). Dunque aggiorniamo i pesi e il bias come segue:

$$w_j \leftarrow w_j - \eta \frac{\partial L}{\partial w_j}$$

Sostituendo le derivate nella regola di aggiornamento otteniamo:

$$w_j \leftarrow w_j + \eta(y - \hat{y})x_j$$
$$b \leftarrow b + \eta(y - \hat{y})$$

La spiegazione intuitiva del perché funziona è che:

- Se l'output $\hat{y}$ è troppo basso rispetto a y, allora $(y - \hat{y})$ è positivo, quindi aumentiamo i pesi per aumentare l'output.
- Se l'output $\hat{y}$ è troppo alto rispetto a y, allora $(y - \hat{y})$ è negativo, quindi diminuiamo i pesi per ridurre l'output.

5.2 Limiti dei modelli lineari

La classe di ipotesi di modelli lineari (o log-lineari) è seriamente limitata. Ad esempio, non possono risolvere problemi non linearmente separabili come l'operazione XOR.

$$\text{xor}(0,0) = 0$$
$$\text{xor}(0,1) = 1$$
$$\text{xor}(1,0) = 1$$
$$\text{xor}(1,1) = 0$$

Non esiste un $w \in \mathbb{R}^2$ e un $b \in \mathbb{R}$ tali che:

$$(0,0) \cdot w + b < 0$$
$$(1,0) \cdot w + b \geq 0$$
$$(0,1) \cdot w + b \geq 0$$
$$(1,1) \cdot w + b < 0$$

Questo perché non esiste una linea che possa separare i punti (0,1) e (1,0) dai punti (0,0) e (1,1) in un piano 2D. Però è possibile trasformare l'input utilizzando una funzione non lineare per rendere i punti separabili linearmente.
In generale, addestriamo un classificatore lineare su un dataset che non è linearmente separabile definendo una funzione che "linearizza" i dati. Tuttavia, nella maggior parte dei casi la dimensione dello spazio delle feature è molto più alta che lo spazio originale degli input, inoltre dobbiamo definire una funzione di mapping ϕ. Le support vector machines (SVM) approcciano il problema definendo un insieme di mapping, che mappano i dati in uno spazio di grande dimensione.
Un esempio è il mapping polinomiale $\phi(x) = x^d$, per $d = 2$:

$$\phi(x_1, x_2) = (x_1, x_2, x_1x_1, x_2x_2, x_1x_2)$$

Ovvero tutte le combinazioni delle due variabili.
Nonostante ora siamo in grado di addestrare un classificatore lineare per il problema dello XOR, abbiamo un incremento polinomiale nel numero dei parametri (non efficiente).

Un approccio diverso è quello di definire una funzione di mapping non lineare addestrabile, può prendere la forma di un modello parametrizzato lineare, accoppiato a una funzione non lineare di attivazione g.

$$\hat{y} = \phi(x)W + b$$
$$\phi(x) = g(xW' + b)$$

Chapter 6

Reti neurali

Una rete neurale artificiale (Artificial Neural Network, ANN) incorpora i due componenti fondamentali delle reti neurali biologice: i neuroni (nodi) e le sinapsi (pesi).

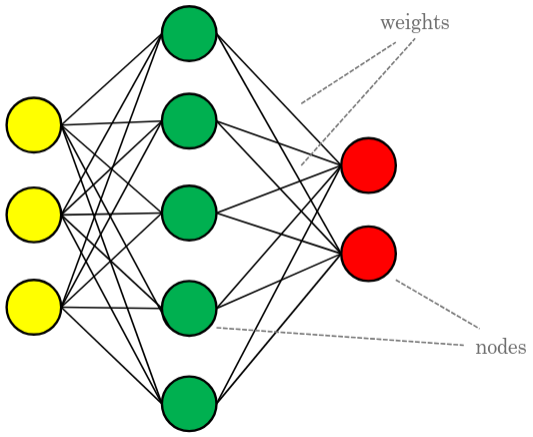


Figure 6.1: Rappresentazione schematica di una rete neurale artificiale.

Dove a ogni nodo corrisponde una funzione di attivazione che elabora gli input ricevuti dai nodi collegati tramite i pesi sinaptici. Lo scopo delle funzioni di attivazione è quello di introdurre non linearità nel modello, permettendo alla rete di apprendere e rappresentare relazioni complesse tra input e output.

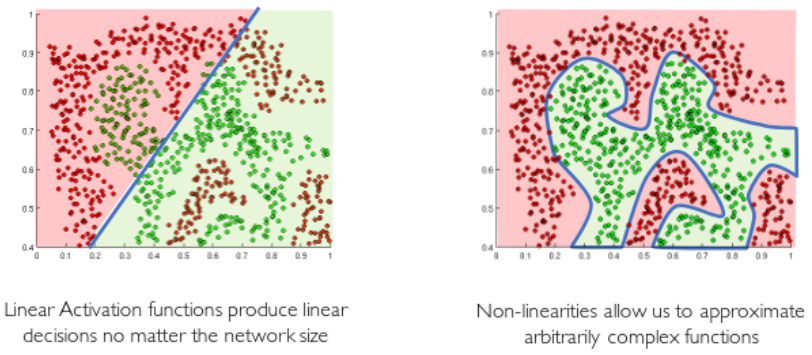


Figure 6.2: Esempi di funzioni di attivazione lineare e non lineari.

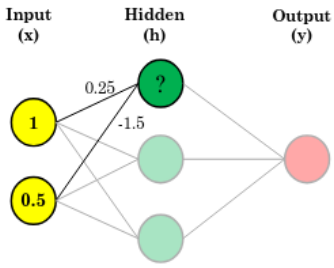


Figure 6.3: Esempio di rete neurale artificiale semplice con due input, un layer nascosto e un output.

Nella figura 6.3 possiamo vedere un esempio di rete neurale dove in un nodo viene calcolato a partire dagli input x_1 e x_2 il valore:

$$(1 \cdot x_1 + 0.5 \cdot x_2) = -0.5$$

$$\text{Con funzione di attivazione: } \frac{1}{1+e^{0.5}} = -0.3775$$

Nella figura 6.4 possiamo vedere un esempio di errore nella classificazione, come possiamo addestrare il

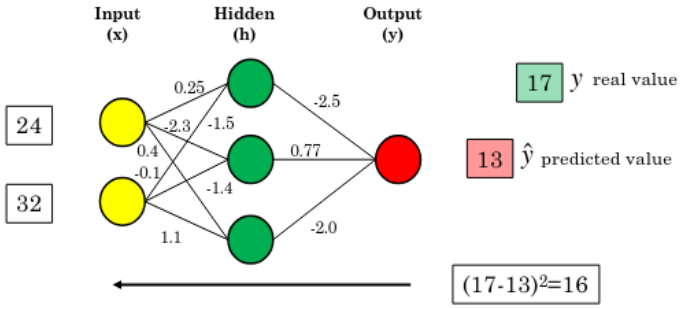


Figure 6.4: Esempio di errore nella classificazione

modello per correggere questo errore?

6.1 Backpropagation

La backpropagation è un algoritmo principale per addestrare le reti neurali artificiali. L'obiettivo è aggiustare i pesi per minimizzare la funzione di loss (differenza tra prediction e target).
Quando una rete neurale fa una previsione:

1. L'input viene passato avanti attraverso la rete per produrre un output (forward pass).
2. L'output viene confrontato con il target utilizzando una funzione di loss per calcolare l'errore.
3. L'algoritmo calcola quanto ogni peso ha contribuito all'errore totale.
4. L'algoritmo propaga l'errore all'indietro attraverso la rete (backward pass).
5. I pesi vengono aggiornati utilizzando la discesa del gradiente per ridurre l'errore.

La regola della catena (analisi matematica) è la base della backpropagation. Se abbiamo delle funzioni composte $y = f(u)$ e $u = g(x)$, quindi $y = f(g(x))$, per vedere quanto y cambia al variare di x usiamo la regola della catena:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

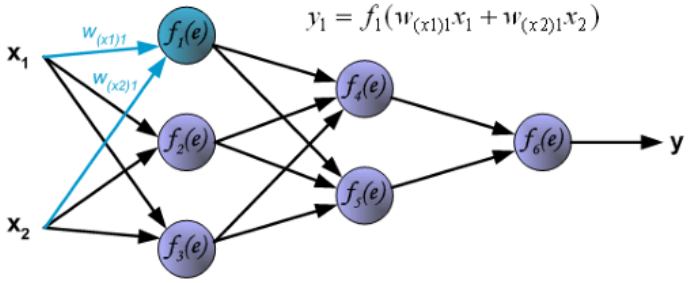
Dove $\frac{dy}{du}$ è la derivata di f rispetto a u (mostra quanto y cambia al variare di u) e $\frac{du}{dx}$ è la derivata di g rispetto a x (mostra quanto u cambia al variare di x). Se abbiamo un numero maggiore di funzioni possiamo sempre applicare la regola della catena.
Supponiamo che la nostra rete neurale abbia L layer:

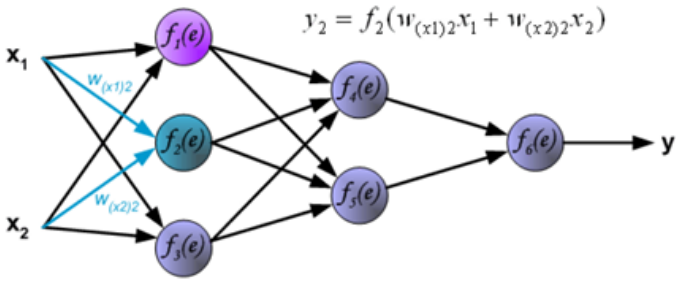
- **Forward pass:** Per ogni layer l da 1 a L:
  - Calcoliamo la combinazione lineare degli input: $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$
  - Applichiamo la funzione di attivazione: $a^{[l]} = \sigma^{[l]}(z^{[l]})$
  - N.B. $a^{[0]}$ è l'input della rete.
- **Backward pass:**
  1. Calcoliamo il gradiente della funzione di loss rispetto all'output: $\delta^{[L]} = \nabla_a \mathcal{L} \odot \sigma'^{[L]}(z^{[L]})$
  2. Per ogni layer l da $L-1$ a 1 calcolo
  $$\delta^{[l]} = (W^{[l+1]})^T \delta^{[l+1]} \odot \sigma'^{[l]}(z^{[l]})$$
  3. Calcoliamo i gradienti dei pesi e dei bias:
  $$\begin{aligned} \frac{\partial \mathcal{L}}{\partial W^{[l]}} &= \delta^{[l]}(a^{[l-1]})^T \\ \frac{\partial \mathcal{L}}{\partial b^{[l]}} &= \delta^{[l]} \end{aligned}$$
  4. Aggiorniamo i pesi (usando gradient descent):
  $$\begin{aligned} W^{[l]} &= W^{[l]} - \eta \frac{\partial \mathcal{L}}{\partial W^{[l]}} \\ b^{[l]} &= b^{[l]} - \eta \frac{\partial \mathcal{L}}{\partial b^{[l]}} \end{aligned}$$

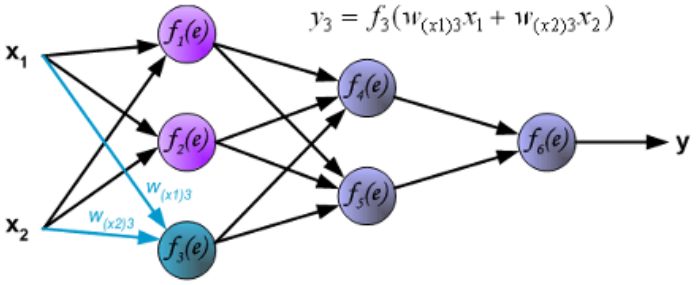
6.2 Esempio di simulazione

Forward pass

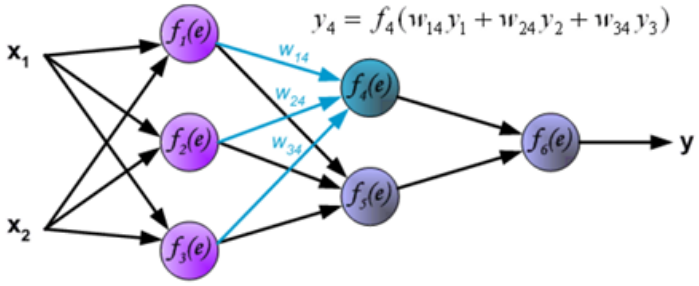
Ora mostriamo come un segnale si propaga nella rete, i simboli $w_{(xm)n}$ rappresentano i pesi delle connessioni tra l'input x_m e il neurone n.

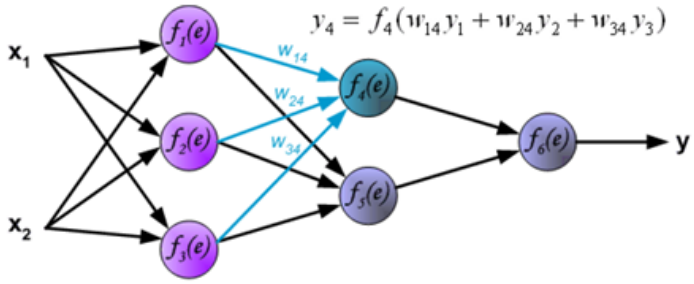


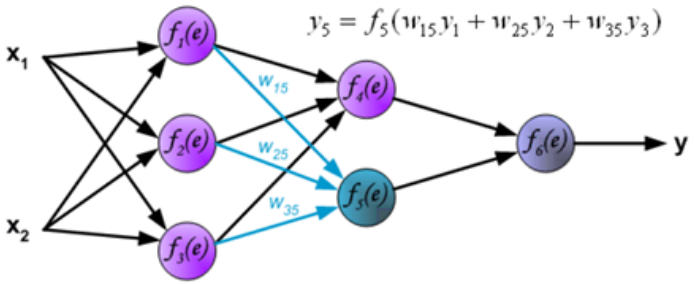




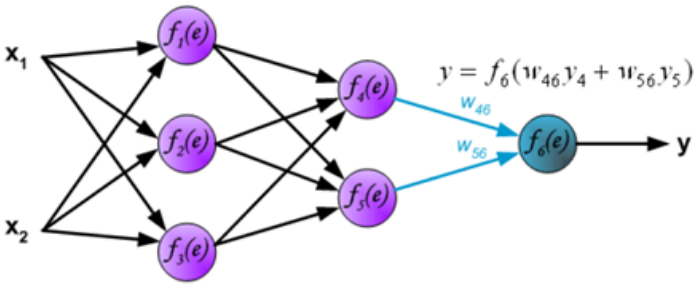
Propagazione dei segnali attraverso il layer nascosto. I simboli w_{mn} rappresentano i pesi delle connessioni tra l'output del neurone m e input del neurone n del layer dopo.





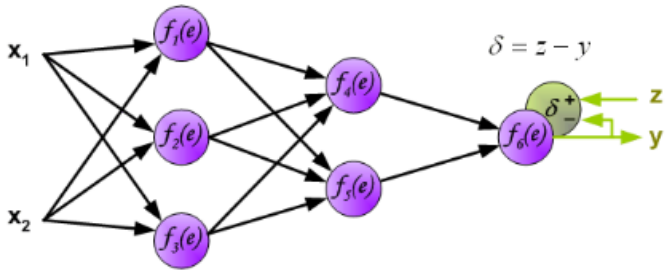


Il segnale viene propagato fino all'output.

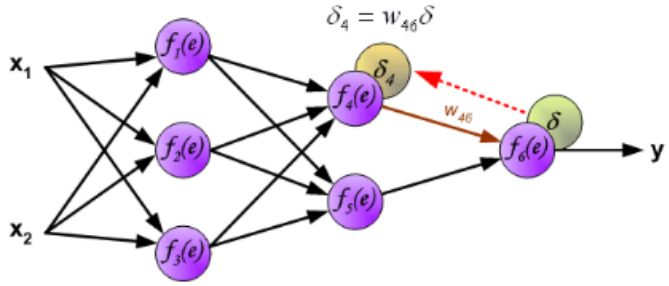


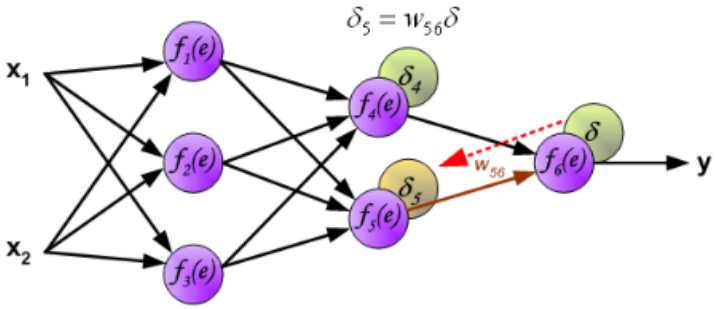
Backward pass

In questo step il segnale di output della rete y viene confrontato con il il valore di output desiderato (target), trovato nel dataset di addestramento. La differenza si chiama segnale di errore δ del neurone di output, nota che δ è la derivata della funzione di loss.

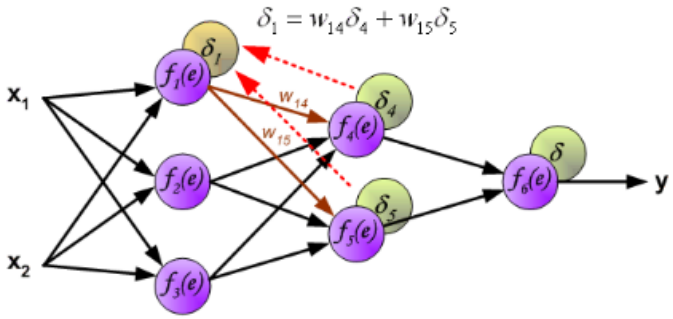


Viene propagato il segnale δ di errore all'indietro attraverso la rete.

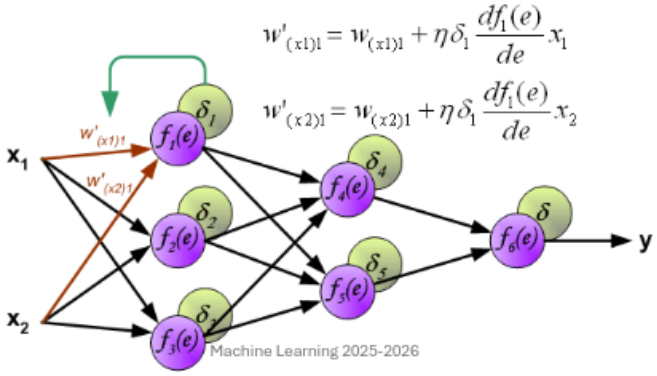


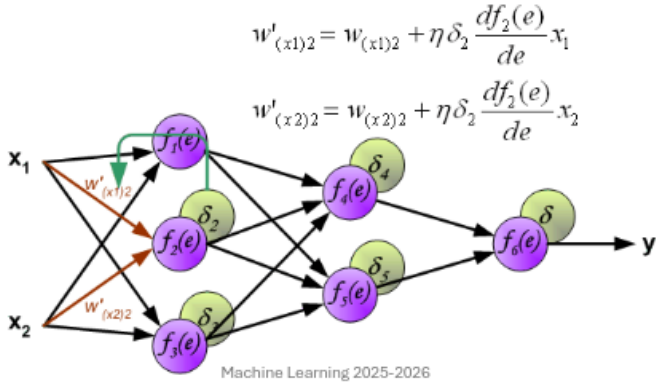


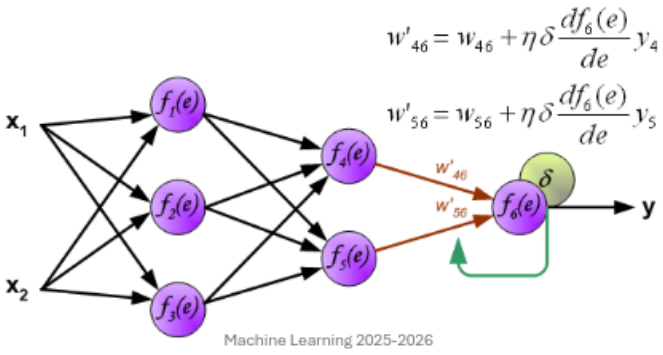
I pesi usati per propagare l'errore all'indietro sono uguali a quelli usati durante la computazione in avanti. Solo la direzione del flusso di dati cambia.



Quando il segnale di errore di ogni neurone viene calcolato, i pesi di ogni nodo di input vengono aggiornati in base alla derivata della funzione di attivazione (di cui modifico i pesi).







6.3 Stochastic gradient descent (SGD)

Algoritmo di ottimizzazione generico, riceve una funzione f parametrizzata secondo θ, una funzione di loss $\mathcal{L}$ e delle coppie input-target (x_i, y_i). L'obiettivo è di impostare i parametri θ in modo che la loss cumulativa di f sugli esempi di training sia minimizzata.

$$\mathcal{L}(\theta) = \sum_{i=1}^{n} L(f(x_i; \theta), y_i)$$

Algorithm 2: Addestramento online con Stochastic Gradient Descent (SGD)

Input:
- Funzione parametrizzata $f(x; \Theta)$
- Funzione di loss $L(\hat{y}, y)$
- Esempi di addestramento (x_i, y_i)
- Tasso di apprendimento η_t

Output: Parametri ottimizzati Θ
while *criteri di arresto non soddisfatti* **do**
  campiona un esempio di addestramento (x_i, y_i);
  calcola la loss locale $L\big(f(x_i; \Theta), y_i\big)$;
  calcola il gradiente stimato $\hat{g} \leftarrow \nabla_\Theta L\big(f(x_i; \Theta), y_i\big)$;
  aggiorna i parametri: $\Theta \leftarrow \Theta - \eta_t \hat{g}$;
return Θ;

L'algoritmo comincia estraendo un esempio di addestramento alla volta e calcolando la loss locale. Successivamente calcola il gradiente stimato della loss rispetto ai parametri Θ usando l'esempio corrente. Si assume che l'input e la predizione siano parametri fissi, quindi la loss viene trattata come una funzione solo di Θ. Infine, i parametri θ vengono aggiornati nella direzione opposta al gradiente, scalato per il learning rate η_t.
Si noti che l'errore è calcolato su un singolo esempio, dunque una stima grezza dell'intero dataset di addestramento che vogliamo minimizzare, questo portebbe portare a calcolare dei gradienti poco accurati.

Per migliorare la stima del gradiente, si può usare il mini-batch gradient descent, che calcola la loss su un piccolo sottoinsieme di esempi di addestramento (mini-batch) invece che su un singolo esempio.

Algorithm 3: Addestramento con Minibatch Stochastic Gradient Descent

Input:
- Funzione parametrizzata $f(x; \Theta)$
- Funzione di loss $L(\hat{y}, y)$
- Esempi di addestramento $\{(x_i, y_i)\}_{i=1}^{n}$
- Dimensione del minibatch m
- Tasso di apprendimento η_t

Output: Parametri ottimizzati Θ
while *criteri di arresto non soddisfatti* **do**
  campiona un minibatch di m esempi $\{(x_1, y_1), \ldots, (x_m, y_m)\}$;
  $\hat{g} \leftarrow 0$;
  for $i \leftarrow 1$ **to** m **do**
    calcola il gradiente locale $g_i \leftarrow \nabla_\Theta L\big(f(x_i; \Theta), y_i\big)$;
    $\hat{g} \leftarrow \hat{g} + \frac{1}{m} g_i$;
  aggiorna i parametri: $\Theta \leftarrow \Theta - \eta_t \hat{g}$;
return Θ;