

Fundamentals Of Machine Learning

2022-2023

Indice

1 INTRODUZIONE	2
2 PRELIMINARI MATEMATICI	9
2.1 Algebra lineare	10
2.2 Probabilità e statistica	15
3 MODELLI LINEARI PER LA REGRESSIONE	22
3.1 Geometric Curve Fitting	22
3.2 Maximum Likelihood Estimation e Least Squares	23
3.3 Regressione lineare Bayesiana	30
3.3.1 Decomposizione bias-varianza	31
3.3.2 Regressione lineare Bayesiana	34
4 MODELLI LINEARI PER LA CLASSIFICAZIONE	39
4.1 Funzioni discriminanti lineari	39
4.1.1 Due classi	39
4.1.2 Multiclasse	41
4.1.3 Least Squares per la classificazione	42
4.1.4 Discriminante lineare di Fisher	44
4.2 Modelli Generativi Probabilistici	47
4.2.1 Modelli generativi con input continui	48
4.3 Modelli Discriminativi Probabilistici	50
4.3.1 Regressione logistica	52
4.3.2 Regressione logistica Bayesiana	53
5 KERNEL MACHINES	55
5.1 Maximum Margin Classifiers	55
5.2 Soft Margin Classifier	60
5.3 SVM per problemi non lineari	62
5.3.1 Modo alternativo di ottenere la SVM primale	63
5.3.2 Sguardo più approfondito alla forma duale della SVM	64
5.3.3 Kernel trick	65
5.4 SVM nella pratica	68
6 MODELLI NON PARAMETRICI	70
6.1 Tecniche di valutazione del classificatore	70
6.2 Iistogrammi	72
6.3 Kernel Density Estimator	75
6.3.1 Nadaraya-Watson (o Kernel Regression)	77

6.4	K-Nearest Neighbors	78
6.4.1	K-Nearest Neighbors Classification	78
7	UNSUPERVISED LEARNING	80
7.1	K-Means Clustering	80
7.2	Gaussian Mixture Models	82
7.3	Principal Component Analysis	89
7.3.1	Formulazione della massima varianza	91
7.3.2	Formulazione del minimo errore	92
7.3.3	Applicazioni della PCA	94
8	ENSEMBLE MODELS	98
8.1	Committees e Bagging	98
8.2	Boosting	100
8.3	Tree Models	102
8.4	Conditional Mixture Models	105
9	DEEP LEARNING	108
9.1	Connectionism	108
9.2	Fondamenti del Deep Learning	117
9.2.1	Convolutional Neural Networks	143

Capitolo 1

INTRODUZIONE

Il **Machine Learning** (o *apprendimento automatico*) è una branca dell’Intelligenza Artificiale (IA) che mira a dotare le macchine della capacità di apprendere informazioni dai dati in maniera autonoma.

Il Machine Learning può essere approssimativamente suddiviso in due macro categorie di approcci di apprendimento:

- **Supervised Learning** (*apprendimento supervisionato*), si riferisce ad una classe di approcci di apprendimento che mirano ad imparare a prevedere gli output a partire da informazioni in input fornite da un dataset costituito da coppie di input e di output. In altre parole, al modello vengono forniti degli esempi in input ed i rispettivi output desiderati con l’obiettivo di definire una regola generale che associa l’input all’output corretto.
- **Unsupervised Learning** (*apprendimento non supervisionato*), si riferisce ad una classe di approcci di apprendimento che cercano di imparare a prevedere gli output a partire da dati in input non etichettati, ovvero dati le cui classi non sono note a priori ma devono essere apprese automaticamente. Tale tecnica di apprendimento consiste quindi nel riclassificare ed organizzare i dati in input sulla base di caratteristiche comuni per cercare di effettuare previsioni sui successivi input. In altre parole, il modello ha lo scopo di trovare una struttura negli input forniti, senza che questi siano etichettati in alcun modo.

N.B: in realtà esiste uno spettro più ampio di tecniche di apprendimento, come weakly-supervised, semi-supervised, self-supervised, ecc...

Idealmente, con il Machine Learning vogliamo apprendere a partire da dati di addestramento, detti **training data**, per fare inferenza (**inferences**) su nuovi dati. In questo modo potremo modellare problemi di apprendimento, valutare le prestazioni dei sistemi di apprendimento e quantificare la fiducia (**belief**) nelle soluzioni ottenute.

Le componenti matematiche principali di un problema di Machine Learning sono:

- Uno **spazio di input** (*input space*) $\mathcal{X}(\in \mathbb{R}^m)$, ovvero lo spazio dei dati in ingresso, ed uno **spazio di output** (*output space*) $\mathcal{Y}(\in \mathbb{R}^n)$, ovvero lo spazio dei dati in uscita.
- Un’**ipotesi generativa** (*generative assumption*) di una funzione $h : \mathcal{X} \rightarrow \mathcal{Y}$ (spesso considereremo una funzione lineare $y = h(x) + \varepsilon$ che rappresenta la funzione utilizzata dal modello per prevedere gli output). Tale ipotesi rappresenta la funzione che descrive al meglio l’obiettivo dell’apprendimento.

- Una **densità di probabilità congiunta** (*joint probability density*) sconosciuta $p(x, y)$ su \mathcal{X} ed \mathcal{Y} .
- Uno **spazio di ipotesi** (*hypothesis space*) \mathcal{H} di funzioni da \mathcal{X} in \mathcal{Y} , rappresenta l'insieme di tutte le possibili ipotesi legali del problema. Da tale insieme l'algoritmo di apprendimento determinerà la funzione (solo una) che descrive al meglio il modello.
- Una **funzione loss** (*loss function*) $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}$, rappresenta una *funzione di costo* che mappa un evento (ovvero un insieme di risultati al quale viene assegnata una certa probabilità che accada) su un numero reale che rappresenta il costo associato a tale evento. La funzione loss quindi permette di misurare il grado di accuratezza del modello associato.

N.B: un problema di ottimizzazione cerca di minimizzare tale funzione loss.

Assumendo l'ipotesi $h \in \mathcal{H}$, l'**obiettivo dell'apprendimento** (*learning objective*) sarà:

$$h^* = \arg \min_{h \in \mathcal{H}} \mathbb{E}_p[\mathcal{L}(h(x), y)] = \arg \min_{h \in \mathcal{H}} \int \mathcal{L}(h(x), y) p(x, y) dx dy$$

Dovremo quindi studiare p che risulta essere sconosciuto. Non avendo informazioni su p dobbiamo ricorrere al campionamento, cioè campioneremo le coppie $(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}$, per $i \in \{1, \dots, N\}$ (dove N rappresenta la dimensione del dataset). In questo modo avremo che $(x_i, y_i) \sim p(x, y)$ (ovvero consideriamo il campione simile alla densità di probabilità congiunta) e quindi possiamo approssimare l'obiettivo con la **loss empirica attesa** (*empirical expected loss*), ovvero:

$$h^* = \arg \min_{h \in \mathcal{H}} \int \mathcal{L}(h(x), y) p(x, y) dx dy \approx \arg \min_{h \in \mathcal{H}} \frac{1}{N} \sum_{i=1}^N \mathcal{L}(h(x_i), y_i)$$

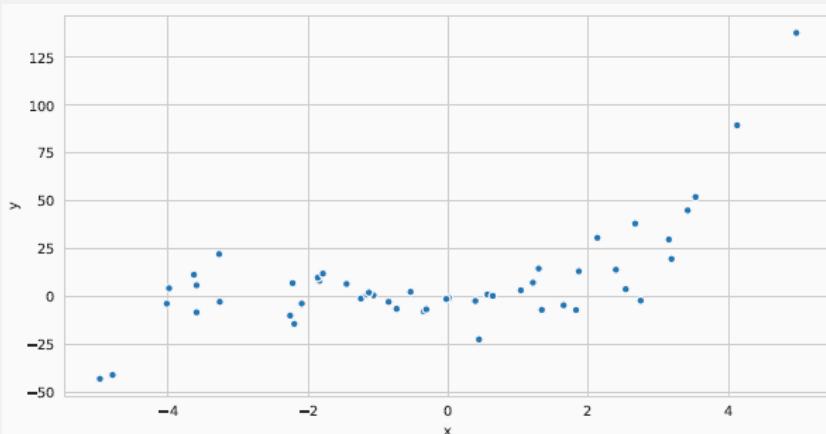
Dovremo inoltre fare assunzioni su \mathcal{L} , \mathcal{H} , sulla minimizzazione e sugli spazi \mathcal{X} ed \mathcal{Y} .



ES (problema di regressione lineare):

Consideriamo un esempio in cui i dati sono distribuiti come di seguito.

Idealmente $y = f(x) + \varepsilon$ (dove ε rappresenta il **Gaussian noise**).



Il nostro obiettivo è quello di utilizzare questo dataset di addestramento (**training set**) per effettuare delle previsioni, cioè per prevedere l'obiettivo (detto **target**) $\hat{y} = f(\hat{x})$ per un nuovo dato (\hat{x}).

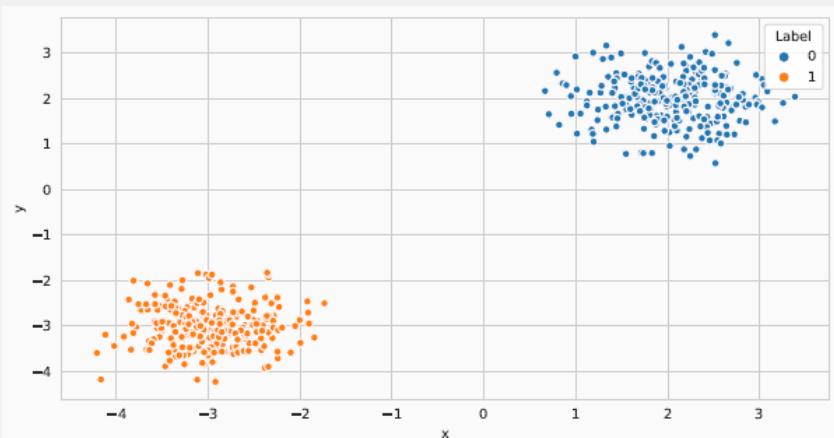
In questo modo stiamo implicitamente cercando di capire quale sia la funzione f sottostante.

N.B: l'apprendimento dovrebbe risultare indipendente da ε .

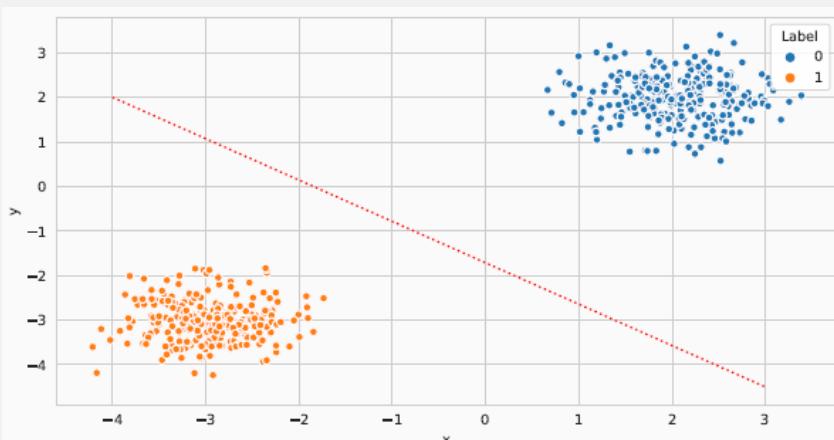


ES:

A volte vorremo capire come i dati vengono generati da N sorgenti, con l'obiettivo di discriminare le sorgenti l'una dall'altra (attribuendo delle classi ad ogni sorgente simile).

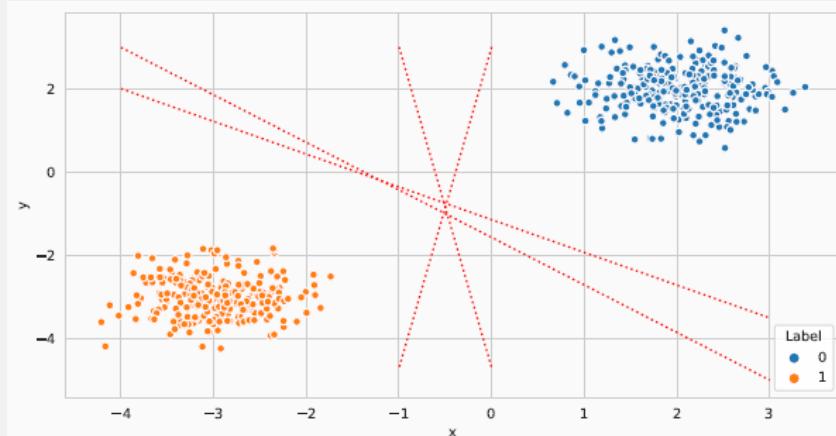


L'idea generale è quella di trovare un **iperpiano di separazione** (*separating hyperplane*) che separa una classe dall'altra.



Uno degli obiettivi di questo **problema di discriminazione** (*discrimination problem*) risulta essere quella di individuare quale sia il migliore piano

di separazione tra le classi, poiché potrebbero esistere più iperpiani contemporaneamente.



N.B: quando stimiamo i parametri di un modello (processo di **inferenza** (*inference*)) è necessario applicare tutte le informazioni a nostra disposizione. In particolare dobbiamo fare attenzione a quantificare ogni volta che è possibile la nostra credenza (*belief*).



ES:

Tornando al problema di regressione, osserviamo una variabile di input x a valori reali e vogliamo prevedere una variabile target t a valori reali. A scopo dimostrativo consideriamo un esempio in cui i dati sono generati sinteticamente secondo $y = f(x | \vec{w}) + \varepsilon$ (dove y rappresenta l'output del modello, x rappresenta il dato in input e \vec{w} rappresenta un vettore di parametri del modello).

Quindi ci viene fornito un training set di coppie (x, y) campionate da $p(x, y)$ con l'obiettivo di imparare una funzione f sottostante che ha generato tali dati.

In questo modo, per un nuovo dato \hat{x} non ancora visto in precedenza possiamo utilizzare la funzione imparata $f(\hat{x} | \vec{w}^*)$ per predire il target \hat{y} .

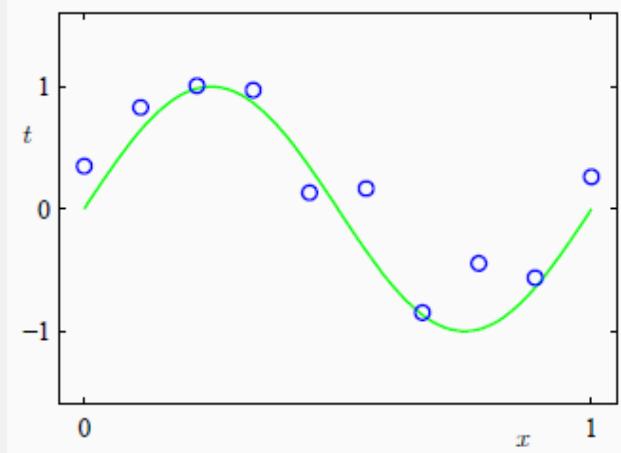


Figura 1.1: Grafico di un dataset di addestramento di $N = 10$ punti, rappresentati da cerchi blu, ognuno dei quali è costituito da un’osservazione della variabile d’ingresso x e della corrispondente variabile target t .

La curva verde mostra la funzione $\sin(2x)$ utilizzata per generare i dati.

Il nostro obiettivo è quello di prevedere il valore di \hat{t} per un nuovo valore di input \hat{x} , senza conoscere la curva verde.

Modelliamo questo problema come un adattamento di una curva (*curve fitting*), per esempio utilizzando un modello modello polinomiale:

$$y(x \mid \vec{w}) = w_0 + w_1 x + w_2 x^2 + \dots + w_M x^M = \sum_{j=0}^M w_j x^j$$

Notiamo che, anche se $y(x \mid \vec{w})$ è una funzione non lineare di x , risulta comunque essere una funzione lineare nei coefficienti \vec{w} (ovvero i parametri del modello).

Quindi, per apprendimento si intende la stima dei migliori parametri \vec{w} a partire dal dataset $\mathcal{D} = \{(x_i, y_i) \mid i = 1, \dots, N\}$.

Per fare ciò possiamo iniziare pensando di misurare l’errore della funzione stimata in termini dei data osservati, ovvero:

$$\mathcal{L}(\vec{w} \mid \mathcal{D}) = \frac{1}{2} \sum_{(x,t) \in \mathcal{D}} \{y(x, \vec{w}) - t\}^2$$

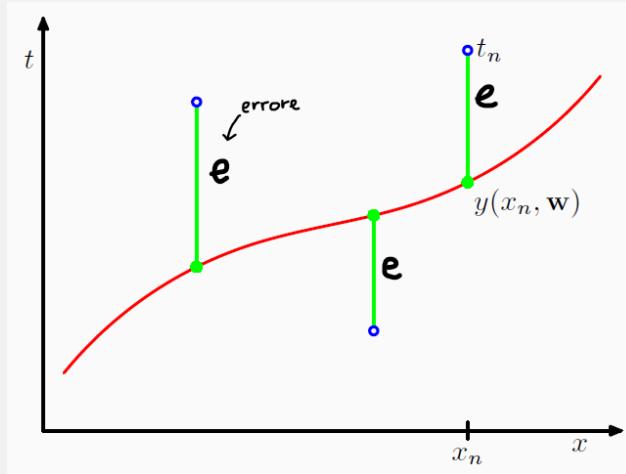
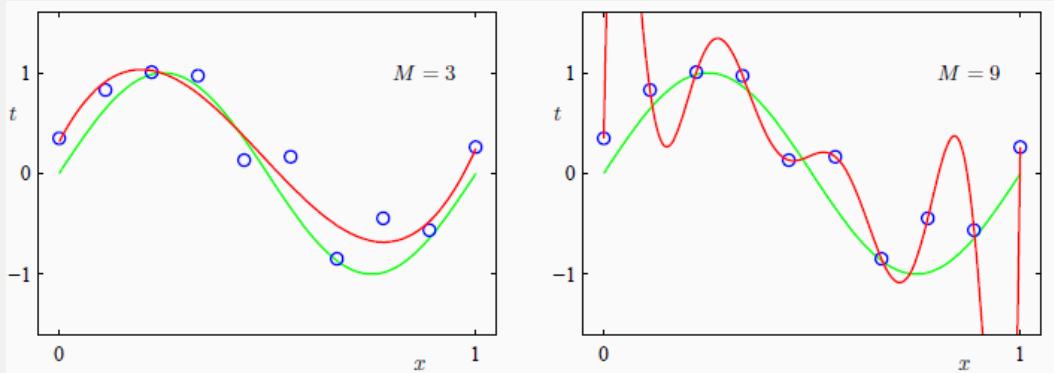
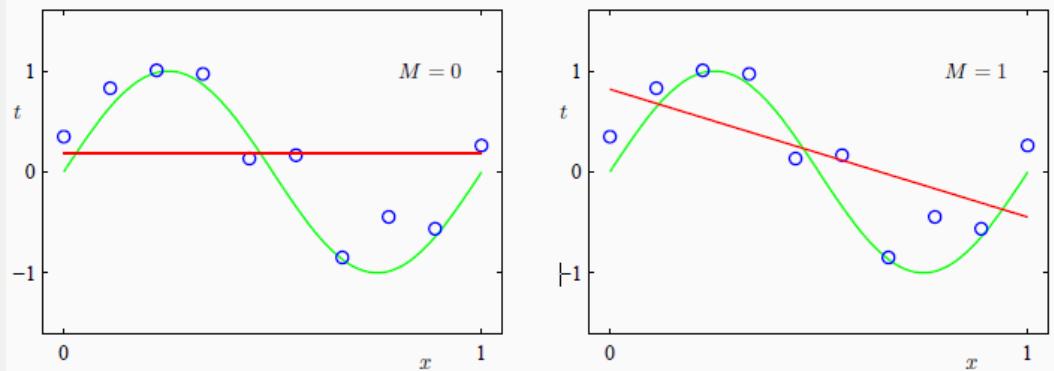


Figura 1.2: Funzione di errore corrispondente a metà della somma dei quadrati degli spostamenti (indicati dalle barre verdi verticali) di ciascun input dalla funzione $y(x, \mathbf{w})$.

Tale funzione risulta essere quadratica in \vec{w} e di conseguenza le sue derivate $\left(\nabla_{\vec{w}} \mathcal{L}(\vec{w} \mid \mathcal{D}) = \left[\frac{\partial \mathcal{L}}{\partial w_0}, \frac{\partial \mathcal{L}}{\partial w_1}, \dots, \frac{\partial \mathcal{L}}{\partial w_M} \right] \right)$ sono lineari. Inoltre $\mathcal{L}(\vec{w} \mid \mathcal{D})$ ha un unico minimizzatore \vec{w}^* .

N.B.: c'è un iperparametro del modello che abbiamo dimenticato, ovvero l'ordine del polinomio M.



Più tale iperparametro cresce, più la nostra funzione si adatterà ai dati.
Notiamo che:

- Se tale iperparametro risulta essere troppo piccolo (ad esempio $M = 0$ oppure $M = 1$), allora la funzione di apprendimento non riesce a modellare i dati di addestramento e di conseguenza non può generare nuovi dati in maniera corretta (tale problema è detto **underfitting**).
- Se tale iperparametro risulta essere troppo elevato ($M = 9$), allora la funzione di apprendimento sarà troppo vincolata ai dati di addestramento (avremo appreso anche i rumori del modello) e quindi non risulta essere una buona preditrice (tale problema è detto **overfitting**).

Per capire l'underfitting e l'overfitting possiamo utilizzare la radice quadratica media (*Root Mean Square*) $E_{RMS} = \sqrt{\frac{2\mathcal{L}(\vec{w}^*|\mathcal{D})}{N}}$, in cui la divisione per N ci permette di confrontare in modo paritario dataset di dimensioni diverse e la radice quadrata garantisce che l' E_{RMS} sia misurato sulla stessa scala della variabile target t .

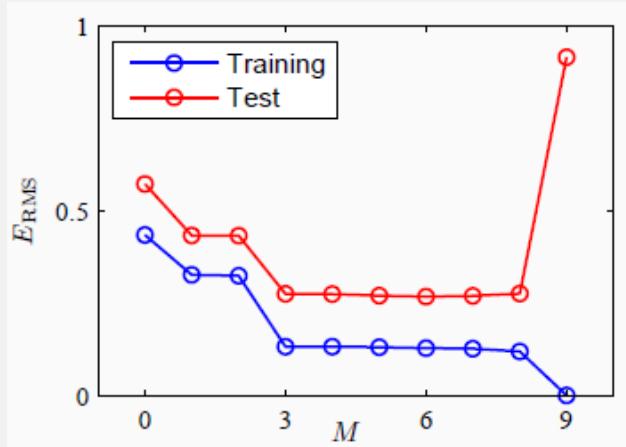


Figura 1.3: Grafici dell'errore quadratico medio valutato su un training dataset e su un testing dataset indipendente per vari valori di M .

Dalla figura si nota che piccoli valori di M danno valori relativamente grandi di errore nel testing dataset (questo perché i polinomi corrispondenti non sono in grado di catturare le oscillazioni della funzione $\sin(2\pi x)$), mentre valori di M compresi nell'intervallo $3 \leq M \leq 8$ danno valori piccoli di errore per il testing dataset e forniscono rappresentazioni ragionevoli della funzione generatrice $\sin(2x)$. Per $M = 9$ l'errore del training dataset si azzerà, perché questo polinomio contiene gradi di libertà corrispondenti ai 10 coefficienti w_0, \dots, w_9 e quindi può essere regolata esattamente sui dati dell'insieme di addestramento. Tuttavia, l'errore del testing dataset diventa molto grande.

Il problema rimanente è quello della selezione del modello più adatto (elemento fondamentale del Machine Learning).

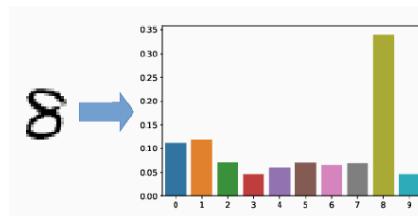
Capitolo 2

PRELIMINARI MATEMATICI

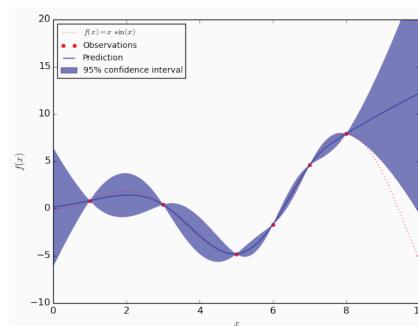
L’algebra lineare risulta essere un argomento centrale del Machine Learning poiché ci permette di trattare i dati con un’**alta dimensionalità**. Questo ci consentirà di modellare gli input agli algoritmi di Machine Learning come punti in spazi ad alta dimensionalità e successivamente modellare le trasformazioni funzionali di questi input in spazi di **features**. Infine ci consentirà di modellare le trasformazioni successive che portano agli output.

La probabilità e la statistica risultano essere argomenti meno centrali per il Machine Learning rispetto all’algebra lineare. A volte si vuole poter dare un’interpretazione probabilistica ad un modello. Tuttavia la maggior parte dei modelli di Deep Learning (particolari modelli di Machine Learning) sono definiti come pure trasformazioni degli input in output. Perciò, la statistica e la probabilità risulteranno essere strumenti molto utili per analizzare i risultati.

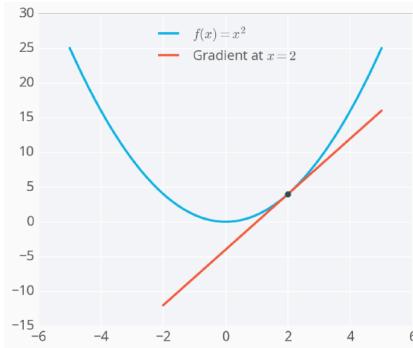
Per molti problemi vorremo che i nostri modelli producano una distribuzione di probabilità sui possibili risultati. Ad esempio, un semplice problema di classificazione risulta essere quello di classificare la cifra raffigurata in un’immagine data in input al modello.



Per altri problemi, invece, vorremo poter qualificare gli output del modello. Ad esempio, in molti problemi di regressione in cui gli output di alcuni punti potrebbero risultare più certi di altri output.



Molti problemi di apprendimento sono formulati come problemi di ottimizzazione in variabili multiple e quindi in questi casi apprendere significa stimare questi problemi minimizzando qualche funzione obiettivo.



2.1 Algebra lineare

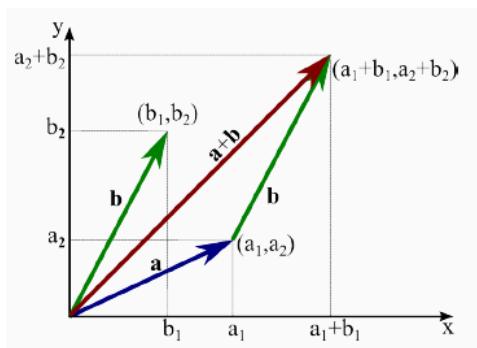
Un **vettore** è un oggetto matematico che ha sia una *grandezza* che una *direzione*. I vettori descrivono rette, piani ed iperpiani nello spazio permettendo di eseguire calcoli in spazi multidimensionali.

Adottiamo la seguente notazione per specificare un vettore di dimensione n :

$$\vec{v} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_n \end{bmatrix} \quad \vec{v}^T = [v_0 \ v_1 \ \dots \ v_n] \quad (\vec{v} \in \mathbb{R}^n)$$

Definizioni (operazioni fondamentali sui vettori):

- **Somma tra vettori:** dati due vettori \vec{u} e \vec{v} (con $\vec{u}, \vec{v} \in \mathbb{R}^n$), allora $\vec{w} = \vec{u} + \vec{v}$ (dove definiamo $w_i = u_i + v_i$).



- **Moltiplicazione scalare:** dato $\vec{v} \in \mathbb{R}^n$ un vettore, allora $\vec{w} = c\vec{v}$ per ogni $c \in \mathbb{R}$ (dove definiamo $w_i = cv_i$).
- **Prodotto scalare (o dot):** dati due vettori \vec{u} e \vec{v} (con $\vec{u}, \vec{v} \in \mathbb{R}^n$), allora definiamo il prodotto scalare tra \vec{u} e \vec{v} come:

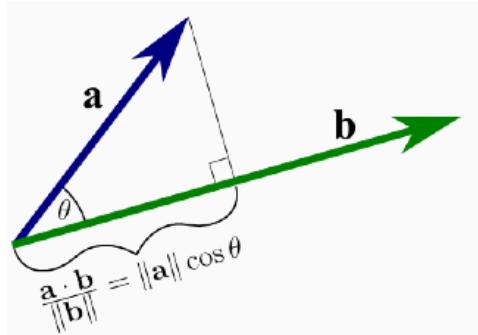
$$\vec{u} \cdot \vec{v} = \sum_{i=1}^n u_i v_i$$

N.B: il prodotto scalare tra due vettori è relativo alle direzioni e alle grandezze dei due vettori.

N.B: possiamo facilmente trovare il coseno tra due qualsiasi vettori \vec{u} e \vec{v} considerando l'angolo θ compreso tra questi. Di conseguenza possiamo riscrivere il prodotto scalare come:

$$\vec{u} \cdot \vec{v} = \|\vec{u}\| \cdot \|\vec{v}\| \cdot \cos\theta \quad \text{ovvero} \quad \frac{\vec{u} \cdot \vec{v}}{\|\vec{v}\|} = \|\vec{u}\| \cos\theta$$

Notiamo che per valutare se due vettori sono ortogonali possiamo vedere se il loro prodotto scalare è nullo (poiché questo dipende dal coseno e quindi abbiamo che se \vec{u} e \vec{v} sono ortogonali allora $\theta = 90^\circ$ e quindi $\cos\theta = 0$).



- **Norma del vettore** (oppure *grandezza* oppure *lunghezza*): dato $\vec{v} \in \mathbb{R}^n$ un vettore, allora definiamo la norma di \vec{v} come $\|\vec{v}\|_2 = \sqrt{\vec{v} \cdot \vec{v}}$.

N.B: queste proprietà sono generalizzabili per qualsiasi dimensione.

Mappa bilineare (bilinear map): una funzione $\Omega : V \times V \rightarrow \mathbb{R}$ è una *mappa bilineare* dallo spazio vettoriale V ad \mathbb{R} se e solo se:

- $\Omega(\lambda\vec{x} + \psi\vec{y}, \vec{z}) = \lambda\Omega(\vec{x}, \vec{z}) + \psi\Omega(\vec{y}, \vec{z})$
- $\Omega(\vec{x}, \lambda\vec{y} + \psi\vec{z}) = \lambda\Omega(\vec{x}, \vec{y}) + \psi\Omega(\vec{x}, \vec{z})$

per ogni $\vec{x}, \vec{y}, \vec{z} \in V$.

N.B: Ω è detta **simmetrica** se $\Omega(\vec{x}, \vec{y}) = \Omega(\vec{y}, \vec{x})$ per ogni $\vec{x}, \vec{y} \in V$.

N.B: Ω è detta **definita positiva** se $\Omega(\vec{x}, \vec{x}) \geq 0$ per ogni \vec{x} ed $\Omega(\vec{x}, \vec{x}) = 0$ se e solo se $\vec{x} = 0$.

Prodotto interno e spazio del prodotto interno (inner product): sia V uno spazio vettoriale qualsiasi e $\Omega : V \times V \rightarrow \mathbb{R}$ una mappa bilineare da V ad \mathbb{R} . Allora:

- Se Ω è *simmetrica* e *definita positiva*, allora Ω è detta un *prodotto interno* su V . Perciò il prodotto interno rappresenta un'applicazione $\Omega : V \times V \rightarrow \mathbb{R}$ tale che $(\vec{x}, \vec{y}) \mapsto \vec{x} \cdot \vec{y}$. Solitamente lo indicheremo con $\langle \vec{x}, \vec{y} \rangle$ invece di $\Omega(\vec{x}, \vec{y})$.
- La coppia (V, Ω) (oppure $(V, \langle \cdot, \cdot \rangle)$) per il prodotto interno Ω è detta *spazio del prodotto interno* (oppure spazio vettoriale con prodotto interno). In particolare, se $\Omega(\vec{x}, \vec{y}) = \vec{x}^T \vec{y}$, allora la coppia (V, Ω) è detta *spazio vettoriale Euclideo*.

N.B: il prodotto interno ci permette di formalizzare le intuizioni geometriche sulla lunghezza, sull'ortogonalità e sulla distanza.

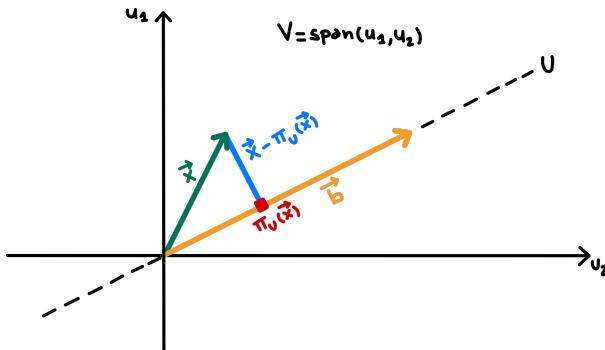


ES (prodotto in \mathbb{R}^n):

Sia $X = (x_1, x_2, \dots, x_n)$ ed $Y = (y_1, y_2, \dots, y_n)$ allora il prodotto standard in \mathbb{R}^n , ovvero $X \cdot Y = x_1y_1 + \dots + x_ny_n$ è un prodotto interno.

Proiezione ortogonale su un sottospazio: siano \vec{x} e \vec{b} due vettori, sia V uno spazio vettoriale e sia U un sottospazio di V tale che $\vec{b} \in U$. Allora la *proiezione ortogonale* di \vec{x} su un sottospazio U rappresenta il punto più vicino ad \vec{x} che si trova su tale sottospazio e si indica con $\pi_U(\vec{x})$.

In particolare, osserviamo che:



- $\langle \vec{x} - \pi_U(\vec{x}), \vec{b} \rangle = 0$, poiché $\vec{x} - \pi_U(\vec{x})$ (che rappresenta la distanza tra \vec{x} e la sua proiezione ortogonale su \vec{b}) e \vec{b} sono ortogonali per ipotesi.
- $\pi_U(\vec{x}) = \lambda \vec{b}$, poiché la proiezione ortogonale di \vec{x} su \vec{b} è multiplo di \vec{b} .

Allora possiamo calcolare λ :

$$\begin{aligned} \langle \vec{x} - \lambda \vec{b}, \vec{b} \rangle = 0 &\Leftrightarrow \langle \vec{x}, \vec{b} \rangle + \langle -\lambda \vec{b}, \vec{b} \rangle = 0 \Leftrightarrow \langle \vec{x}, \vec{b} \rangle = \langle \lambda \vec{b}, \vec{b} \rangle \Leftrightarrow \langle \vec{x}, \vec{b} \rangle = \lambda \langle \vec{b}, \vec{b} \rangle \\ &\Rightarrow \lambda = \frac{\langle \vec{x}, \vec{b} \rangle}{\langle \vec{b}, \vec{b} \rangle} = \frac{\langle \vec{x}, \vec{b} \rangle}{||\vec{b}||} \end{aligned}$$

Perciò otteniamo che la proiezione ortogonale di \vec{x} su un sottospazio vettoriale U è definita come:

$$\pi_U(\vec{x}) = \lambda \vec{b} = \frac{\langle \vec{x}, \vec{b} \rangle}{||\vec{b}||} \vec{b}$$

Una **matrice** è una struttura di elementi disposti su **righe** e **colonne**.

Per indicare una matrice si utilizzano lettere maiuscole, mentre per indicare gli elementi di una matrice ci riferiamo alla stessa lettera in minuscolo e specifichiamo a pedice della lettera il valore della riga e della colonna.

Adottiamo quindi la seguente notazione per specificare una matrice di dimensione $m \times n$ valori reali (ovvero $A \in \mathbb{R}^{n \times m}$) dove n indica il numero di righe della matrice ed m indica il numero di colonne:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{bmatrix}$$

N.B.: la **trasposta** di una matrice è la matrice stessa alla quale viene cambiato l'orientamento delle righe e delle colonne e si indica con il simbolo T posto in apice:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{bmatrix} \quad A^T = \begin{bmatrix} a_{1,1} & a_{2,1} & \dots & a_{n,1} \\ a_{1,2} & a_{2,2} & \dots & a_{n,2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,m} & a_{2,m} & \dots & a_{n,m} \end{bmatrix}$$

N.B.: la matrice **identità** I è una matrice quadrata (cioè di dimensione $n \times n$) che contiene tutti valori 1 sulla diagonale e valori 0 altrove:

$$I = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

Di conseguenza valgono le relazioni $I \cdot A = A \cdot I$ ed $I \cdot v = v$.

N.B.: l'**inversa** di una matrice quadrata A si denota con A^{-1} ed è definita come l'unica matrice (se esiste) tale che $A^{-1} \cdot A = A \cdot A^{-1} = I$.

Le matrici supportano le comuni operazioni aritmetiche ma con alcune osservazioni:

- **Somma tra matrici**: date due matrici $A \in \mathbb{R}^{n \times m}$ ed $B \in \mathbb{R}^{n \times m}$ allora $C = A + B$ dove $C \in \mathbb{R}^{n \times m}$ (in particolare $c_{i,j} = a_{i,j} + b_{i,j}$, ovvero si sommano gli elementi corrispondenti di ciascuna matrice).
- **N.B.**: la dimensione delle matrici A e B deve essere uguale per poter eseguire la somma.
- **N.B.**: la **negazione** di una matrice non è altro che la matrice a cui viene cambiato il segno di ogni elemento:

$$C = \begin{bmatrix} c_{1,1} & c_{1,2} & \dots & c_{1,m} \\ c_{2,1} & c_{2,2} & \dots & c_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \dots & c_{n,m} \end{bmatrix} \quad -C = \begin{bmatrix} -c_{1,1} & -c_{1,2} & \dots & -c_{1,m} \\ -c_{2,1} & -c_{2,2} & \dots & -c_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ -c_{n,1} & -c_{n,2} & \dots & -c_{n,m} \end{bmatrix}$$

In questo modo possiamo interpretare le sottrazioni come somma tra matrici in cui vengono negate le matrici che devono essere sottratte.

- **Moltiplicazione scalare**: dato $\lambda \in \mathbb{R}$ un valore reale ed $A \in \mathbb{R}^{n \times m}$ una matrice a valori reali, allora:

$$\lambda \cdot A = \lambda \cdot \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{bmatrix} = \begin{bmatrix} \lambda \cdot a_{1,1} & \lambda \cdot a_{1,2} & \dots & \lambda \cdot a_{1,m} \\ \lambda \cdot a_{2,1} & \lambda \cdot a_{2,2} & \dots & \lambda \cdot a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ \lambda \cdot a_{n,1} & \lambda \cdot a_{n,2} & \dots & \lambda \cdot a_{n,m} \end{bmatrix}$$

- **Prodotto scalare**: date due matrici $A \in \mathbb{R}^{n \times m}$ ed $B \in \mathbb{R}^{m \times l}$ allora $C = A \cdot B$ dove $C \in \mathbb{R}^{n \times l}$ (in particolare $c_{i,j} = \sum_{k=1}^n a_{i,k} \cdot b_{k,j}$, ovvero si calcola il prodotto scalare tra righe e colonne).

N.B.: per poter eseguire il prodotto scalare tra due matrici è importante che il numero di colonne di A sia uguale al numero di righe di B .

Con tali nozioni è possibile risolvere equazioni lineari a più dimensioni (come per il calcolo dei parametri w di un problema di apprendimento).



ES:

Consideriamo di voler individuare i parametri dell'equazione lineare così descritta ($A \cdot \vec{x} = b$):

$$\begin{bmatrix} 67.9 & 1.0 \\ 61.9 & 1.0 \end{bmatrix} \cdot \begin{bmatrix} m \\ b \end{bmatrix} = \begin{bmatrix} 170.85 \\ 122.50 \end{bmatrix}$$

Allora, moltiplicando entrambi i lati per la matrice inversa di A :

$$\begin{aligned} \begin{bmatrix} 67.9 & 1.0 \\ 61.9 & 1.0 \end{bmatrix}^{-1} \begin{bmatrix} 67.9 & 1.0 \\ 61.9 & 1.0 \end{bmatrix} \cdot \begin{bmatrix} m \\ b \end{bmatrix} &= \begin{bmatrix} 67.9 & 1.0 \\ 61.9 & 1.0 \end{bmatrix}^{-1} \begin{bmatrix} 170.85 \\ 122.50 \end{bmatrix} \\ \Rightarrow I \cdot \begin{bmatrix} m \\ b \end{bmatrix} &= \begin{bmatrix} m \\ b \end{bmatrix} = \begin{bmatrix} 67.9 & 1.0 \\ 61.9 & 1.0 \end{bmatrix}^{-1} \begin{bmatrix} 170.85 \\ 122.50 \end{bmatrix} \end{aligned}$$

N.B.: come abbiamo appena osservato nell'esempio, la moltiplicazione tra matrici permette di calcolare le *trasformazioni lineari* di spazi vettoriali.

N.B.: possiamo essere interessati anche alle *trasformazioni affini* che non necessariamente preservano l'origine. Una **trasformazione affine** è una semplice trasformazione lineare seguita da una *traslazione* (detta **bias**), ovvero $f(x) = Ax + b$ (dove b rappresenta il bias).

Notiamo inoltre che una trasformazione affine in n dimensioni può essere modellata da una trasformazione lineare in $n+1$ dimensioni.



ES:

Si modella la trasformazione affine $Ax + b$ (di grado n) incorporando b in A ed aggiungendo 1 in \vec{x} (in questo modo si ottiene una trasformazione lineare di grado $n+1$)

$$Ax + b = \begin{bmatrix} a_1 & a_2 & b_1 \\ a_3 & a_4 & b_2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$$

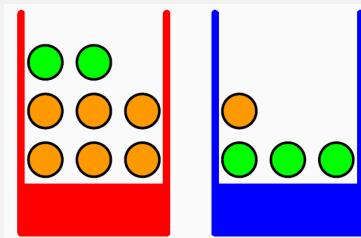
Gli strumenti che abbiamo riportato sono completamente generici e permettono di definire matrici dense ed omogenee di qualsiasi dimensionalità. Il termine generico per queste matrici è **tensore** ed in esso tutta la matematica si generalizza a dimensioni arbitrarie.

2.2 Probabilità e statistica



ES:

Consideriamo due urne, una rossa ed una blu, contenenti pezzi di frutta (mele e arance). Il contenuto esatto di ogni urna è sconosciuto. Si vuole pescare un pezzo di frutta da una delle due urne scelta in modo casuale, con probabilità di pescare dall'urna rossa 0.4 e probabilità di pescare dall'urna blu 0.6.



In questo caso abbiamo una variabile aleatoria U (urna) definita dalla sua distribuzione, cioè $\Pr(U = \text{rossa}) = 0.4$ ed $\Pr(U = \text{blu}) = 0.6$.

Inoltre abbiamo un ulteriore variabile aleatoria F (frutta) che è dipende da U (poiché la sua distribuzione dipende dalla scatola scelta).

Vogliamo calcolare la probabilità complessiva di scegliere una mela, ovvero $\Pr(F = \text{mela})$ (probabilità che dipende chiaramente anche da $\Pr(U)$).

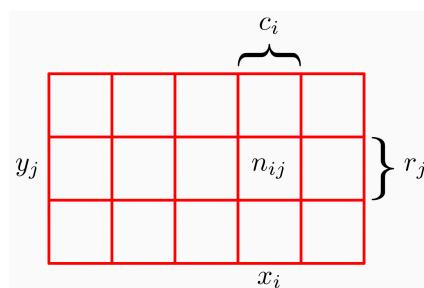
Inoltre, se la frutta scelta è un'arancia, vogliamo calcolare la probabilità che sia stata estratta dall'urna blu, ovvero $\Pr(U = \text{blu} | F = \text{arancia})$.

Per poter rispondere a queste domande utilizzeremo la definizione di **distribuzione di probabilità congiunta** (*joint distribution*) di tutte le variabili coinvolte.

Consideriamo un caso generale. Vogliamo stimare la distribuzione congiunta delle variabili aleatorie X ed Y senza avere informazioni preliminari.

Si estrae un campione $\{(x_i, y_i) \mid i = 1, 2, \dots, N\}$ in modo indipendente dalla distribuzione congiunta $p(X, Y)$ e si crea un **istogramma**. Tale istogramma è generico per ogni distribuzione congiunta e si utilizza per raccogliere i dati (ovvero è la rappresentazione normalizzata della distribuzione congiunta $p(X, Y)$).

N.B: $p(X, Y)$ indica la $\Pr(X, Y)$.



Definiamo:

- $n_{i,j}$ il numero di campioni che si osservano nella cella dell'istogramma (i, j) relativa ad (x_i, y_j) (rappresenta quindi il numero di volte che otteniamo (x_i, y_j) nel campione).
- c_i il numero totale di volte che X assume il valore x_i (rappresenta quindi la somma degli elementi presenti nella colonna i dell'istogramma).
- r_j il numero totale di volte che Y assume il valore y_j (rappresenta quindi la somma degli elementi presenti nella riga j dell'istogramma).

N.B: c_i ed r_j si dicono probabilità marginali.

Da questo istogramma possiamo quindi valutare:

- La **probabilità congiunta** $p(X, Y)$ calcolata come $p(X = x_i, Y = y_i) = \frac{n_{i,j}}{N}$. Indica la probabilità che si verifichi (x_i, y_j) e si calcola come il rapporto tra il numero di volte in cui si osserva (x_i, y_j) ed il numero totale di osservazioni N .
- Le **probabilità marginali**

$$- p(X = x_i) = \frac{c_i}{N} = \sum_j p(X = x_i, Y = y_j).$$

Indica la probabilità che X assuma il valore x_i e si calcola come il rapporto tra il numero totale di volte in cui si osserva x_i ed il numero totale di osservazioni N .

$$- p(Y = y_i) = \frac{r_j}{N} = \sum_i p(X = x_i, Y = y_j).$$

Indica la probabilità che Y assuma il valore y_j e si calcola come il rapporto tra il numero totale di volte in cui si osserva y_j ed il numero totale di osservazioni N .

Perciò, per calcolare le probabilità condizionate, consideriamo solamente gli *eventi congiunti* (**joint events**) per i quali $X = x_i$ e scriviamo la frazione di tali eventi per i quali $Y = y_j$ come $p(Y = y_j | X = x_i) = \frac{n_{i,j}}{c_i}$ (ovvero la probabilità di osservare y_j quando si verifica x_i non è altro che il rapporto tra il numero di volte in cui si ha (x_i, y_j) ed il numero totale di volte in cui si osserva x_i).

N.B: possiamo derivare la probabilità condizionata direttamente dalla probabilità congiunta. Infatti $p(X = x_i, Y = y_j) = \frac{n_{i,j}}{N} = \frac{n_{i,j}}{c_i} \cdot \frac{c_i}{N} = p(Y = y_j | X = x_i)p(X = x_i)$ da cui otteniamo $p(Y = y_j | X = x_i) = \frac{p(Y = y_j, X = x_i)}{p(X = x_i)}$.

In Machine Learning ricorrono le seguenti leggi di probabilità:

- **Regola della somma:** $p(X) = \sum_Y p(X, Y)$
- **Regola del prodotto:** $p(X, Y) = p(Y | X)p(X)$

- **Regola di Bayes:** $p(Y | X) = \frac{p(X|Y)p(Y)}{p(X)}$

N.B: tale regola assume un significato particolare se viene applicata all'inferenza dei parametri, ovvero $p(\vec{w} | D) = \frac{p(D|\vec{w})p(\vec{w})}{p(D)}$ che rappresenta la relazione posterior \propto data likelihood \times prior dove posterior indica la distribuzione di probabilità condizionata che rappresenta come sono i parametri dopo che sono

stati osservati i dati, data likelihood indica la probabilità di assegnare una specifica classe e prior indica la distribuzione di probabilità che rappresenta la conoscenza di un dato prima di osservarlo.

- **Valore medio o valore atteso o aspettativa** (*expectation*) della funzione $f(x)$ sulla distribuzione di probabilità $p(x) : \mathbb{E}[f] = \sum_x p(x)f(x) = \int p(x)f(x)dx$.
N.B: se disponiamo di un campione finito di N punti dalla distribuzione $p(x)$ possiamo approssimare l'aspettativa come $\mathbb{E}[f] \approx \sum_i p(x_i)f(x_i)$.

Poiché in Machine Learning è importante la distribuzione Gaussiana allora risulta rilevante definire la **covarianza** come:

$$\text{cov}(\vec{x}, \vec{x}) = \mathbb{E}_x[\{\vec{x} - \mathbb{E}[\vec{x}]\}\{\vec{x}^T - \mathbb{E}[\vec{x}^T]\}] = \mathbb{E}_x[\vec{x}\vec{x}^T] - \mathbb{E}[\vec{x}]\mathbb{E}[\vec{x}^T]$$

N.B: la **distribuzione Gaussiana univariata** è molto importante e si definisce come:

$$\mathcal{N}(x | \mu, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left\{-\frac{1}{2\sigma^2}(x - \mu)^2\right\}$$

dove μ rappresenta il *valore medio* e σ rappresenta la *varianza*.

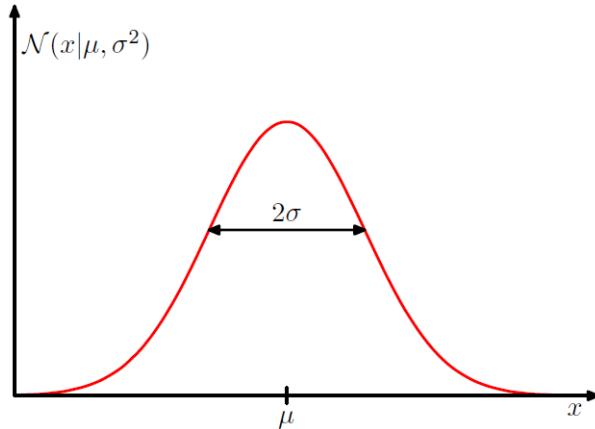


Figura 2.1: Grafico della distribuzione Gaussiana univariata di media μ e deviazione standard σ .

N.B: allo stesso modo risulta importante la **distribuzione Gaussiana multivariata** e si definisce come:

$$\mathcal{N}(x | \vec{\mu}, \Sigma) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma|^{1/2}} \exp\left\{-\frac{1}{2}(x - \vec{\mu})^T \Sigma^{-1} (x - \vec{\mu})\right\}$$

dove $\vec{\mu}$ è un vettore di dimensione D (rappresenta il *valore medio*) ed Σ è una matrice di dimensione $D \times D$ (rappresenta la *matrice di covarianza*).

La teoria delle probabilità ci offre un modo per rappresentare e quantificare l'incertezza. Supponiamo di avere un input \vec{x} ed un vettore \vec{y} di variabili target. Per i *problemi di regressione* \vec{y} sarà una variabile continua, mentre per i *problemi di classificazione* \vec{y} sarà una variabile discreta (poiché rappresenterà le etichette delle classi). Perciò la distribuzione congiunta $p(\vec{x}, \vec{y})$ ci fornisce un quadro completo dell'incertezza associata a queste variabili.



ES:

Consideriamo una radiografia di 512×512 pixel (il vettore di ingresso x è l'insieme delle intensità dei pixel dell'immagine) e supponiamo che si voglia decidere se un paziente ha il cancro (classe \mathcal{C}_1) oppure non ha il cancro (classe \mathcal{C}_2) valutando tale radiografia.

Perciò avremo una funzione che assumerà un valore 1 se viene diagnosticato il cancro al paziente e 0 altrimenti, ovvero:

$$f(x) = \begin{cases} 1 & \text{se il paziente ha il cancro} \\ 0 & \text{altrimenti} \end{cases}$$

Allora il dataset \mathcal{D} sarà definito da un'insieme di coppie $\mathcal{D} = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), \dots, (\vec{x}_N, y_N)\}$ dove \vec{x} rappresenta un vettore di dimensione 512×512 ed y rappresenta la classe (vale 0 o 1).

Innanzitutto dobbiamo affrontare il problema dell'inferenza, ovvero dobbiamo determinare la distribuzione congiunta $p(\vec{x}, y)$ (di solito estremamente difficile). Dopodiché dobbiamo decidere come agire in modo ottimale per una specifica distribuzione congiunta $p(\vec{x}', y)$ (spesso molto facile).

Quando otteniamo una radiografia \vec{x} di un nuovo paziente, vogliamo decidere quale delle due classi assegnare al paziente (ovvero $p(\mathcal{C}_k | \vec{x})$).

Quindi dato un input \vec{x} , il nostro obiettivo è quello di decidere a quale classe appartiene. Possiamo ottenere alcune informazioni su questa decisione applicando la **posterior distribution**, ovvero

$$p(\mathcal{C}_k | \vec{x}) = \frac{p(\vec{x} | \mathcal{C}_k)p(\mathcal{C}_k)}{p(\vec{x})} = \frac{\text{data likelihood} \times \text{prior}}{\text{evidence}}$$

Se l'obiettivo è quello di ridurre al minimo la possibilità di assegnare ad \vec{x} la classe sbagliata, allora scegliamo la classe che ha la posterior distribution più alta. Quindi, per determinare la classe di appartenenza di \vec{x} dobbiamo minimizzare il *tasso di errore* (**misclassification rate**) di classificazione previsto, ovvero:

$$\begin{aligned} p(\text{missclass}) &= p(\vec{x} \in \mathcal{R}_1, \mathcal{C}_2) + p(\vec{x} \in \mathcal{R}_2, \mathcal{C}_1) = \\ &\int_{\mathcal{R}_1} p(\vec{x}, \mathcal{C}_2) d\vec{x} + \int_{\mathcal{R}_2} p(\vec{x}, \mathcal{C}_1) d\vec{x} \end{aligned}$$

Ovvero, abbiamo bisogno di una regola che assegna \vec{x} a una delle classi disponibili. Tale regola dividerà lo spazio di input in regioni \mathcal{R}_k chiamate **regioni di decisione**, una per ogni classe, in modo che tutti i punti in \mathcal{R}_k siano assegnati alla classe \mathcal{C}_k . Per trovare la regola decisionale ottimale nel caso di due classi osserviamo che un errore si verifica quando un vettore di input appartenente alla classe \mathcal{C}_1 viene assegnato alla classe \mathcal{C}_2 o viceversa (la probabilità che ciò avvenga è data dalla precedente equazione).

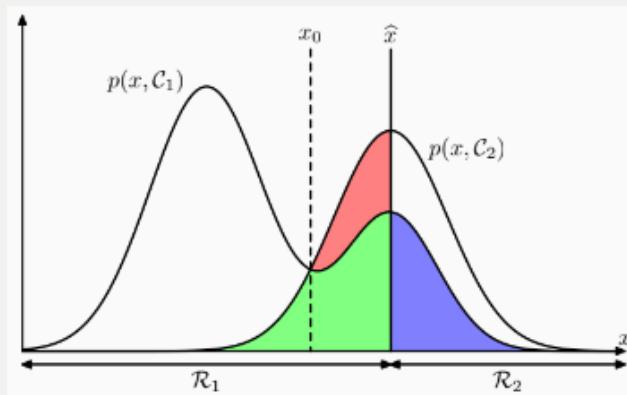


Figura 2.2: Illustrazione schematica delle probabilità congiunte $p(\vec{x}, \mathcal{C}_k)$ per ciascuna delle due classi tracciate rispetto a \vec{x} .

I valori di $x \geq \hat{x}$ sono classificati come classe \mathcal{C}_2 e quindi appartengono alla regione decisionale \mathcal{R}_2 , mentre i punti $x < \hat{x}$ sono classificati come \mathcal{C}_1 e appartengono a \mathcal{R}_1 .

Gli errori sono localizzati nelle regioni blu, verde e rossa. In particolare per $x < \hat{x}$ gli errori sono dovuti input appartenenti alla classe \mathcal{C}_2 classificati erroneamente come \mathcal{C}_1 (rappresentati dalla somma delle regioni rossa e verde) e, viceversa, per $x \geq \hat{x}$ gli errori sono dovuti a input appartenenti alla classe \mathcal{C}_1 classificati erroneamente come \mathcal{C}_2 (rappresentati dalla regione blu).

Variando la posizione del confine decisionale \hat{x} , le dimensioni delle regioni blu e verde rimangono costanti, mentre le dimensioni della regione rossa varia. L'obiettivo è quello di minimizzare le dimensioni della regione rossa (il caso ottimo si verifica per $\hat{x} = x_0$).

Possiamo quindi seguire tre possibili opzioni:

- **Classificatore Bayesiano:** si stimano le densità condizionali di ogni singola classe $p(\vec{x} | \mathcal{C}_k)$ insieme alle probabilità *prior* $p(\mathcal{C}_k)$ e quindi si utilizza il teorema di Bayes per determinare il *posterior*. Ovvero:

$$p(\mathcal{C}_k | \vec{x}) = \frac{p(\vec{x} | \mathcal{C}_k)p(\mathcal{C}_k)}{p(\vec{x})}$$

- **Neural Networks:** si stimano direttamente le probabilità *posterior*.
- Si stima direttamente la funzione discriminante saltando tutte le formalità Bayesiane.

N.B: la scelta sull'approccio da utilizzare viene effettuata attraverso varie ragioni pratiche.

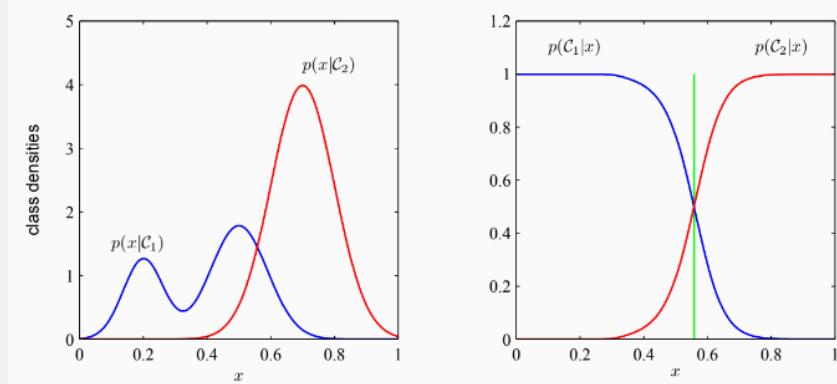


Figura 2.3: Esempio delle densità condizionali di due classi con una sola variabile di input x (grafico di sinistra) e delle corrispondenti probabilità posteriori (grafico di destra). La linea verde verticale nel grafico di destra mostra il confine decisionale in x che dà il minimo tasso di errore di classificazione.

Effettuiamo inoltre alcune osservazioni riguardo al problema della dimensionalità. Consideriamo un problema di classificazione su tre classi che prende in input un vettore con solo due dimensioni.

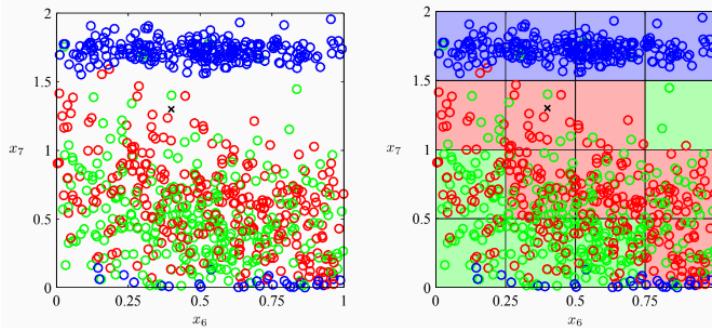


Figura 2.4: Illustrazione di un problema di classificazione in cui lo spazio di input è suddiviso in celle. Ogni nuovo input viene assegnato alla classe che ha la maggioranza dei rappresentanti nella stessa cella dell'input (in questo caso associamo alla croce nera la classe rossa poiché è quella che risulta più vicina).

Se aggiungiamo dimensioni in input, allora il numero di caselle in qualsiasi discretizzazione dello spazio cresce esponenzialmente.

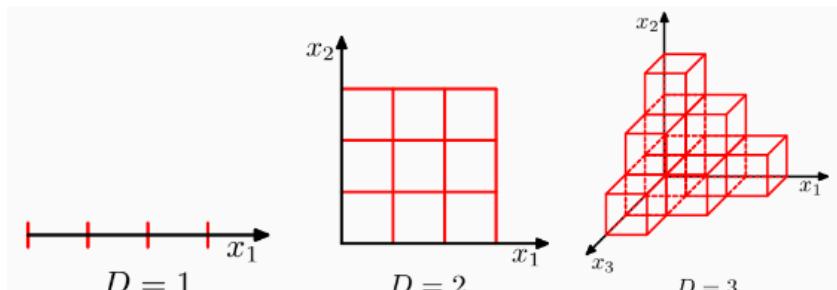


Figura 2.5: Il numero di regioni di una griglia regolare cresce esponenzialmente con la dimensionalità D dello spazio.

Perciò, incrementando il numero di variabili in input aumentano le features e quindi diventa un problema più complesso (per avere buoni risultati dovremmo aumentare esponenzialmente anche i dati).

Notazioni.

- Il **valore atteso** (*expectation*) di $f(x, y)$ rispetto ad una variabile aleatoria x si scrive $\mathbb{E}_X[f(x, y)]$.
- Se x è condizionata da z allora il **valore atteso condizionale** si scrive $\mathbb{E}_X[f(x) \mid z]$.
- La **varianza** di $f(x)$ è denotata con $\text{var}[f(x)]$ e la **covarianza** con $\text{cov}[x, y]$.

Capitolo 3

MODELLI LINEARI PER LA REGRESSIONE

3.1 Geometric Curve Fitting

Iniziamo lo studio dei modelli di regressione lineare da una prospettiva geometrica. A questo scopo definiremo il problema dell'apprendimento come un problema di ricerca dei parametri ottimali \vec{w} (dove il vettore \vec{w} è detto **weights**) di uno spazio di ipotesi parametrizzato \mathcal{H} .

Supponiamo di osservare una variabile di input x a valore reale. L'obiettivo è quello di voler utilizzare questa osservazione per prevedere il valore di una variabile target t a valore reale che rappresenta la predizione di x .

Consideriamo innanzitutto la classe dei problemi di regressione univariata in cui analizziamo le coppie di osservazioni (x_i, t_i) , dove:

- $x_i \in \mathbb{R}$ sono le osservazioni della variabile indipendente (rappresentano i dati in input i quali possiedono alcune informazioni caratteristiche, dette **features**).
- $t_i \in \mathbb{R}$ sono le osservazioni della variabile dipendente (rappresentano i target dei dati in input x_i , ovvero la classe assegnata ad x_i).

Assumiamo che le variabili dipendenti siano correlate alle variabili indipendenti attraverso una funzione f incognita che permette di adattare il modello ai dati. Tale funzione, sebbene non sia lineare in \vec{x} , risulta essere lineare nei parametri \vec{w} .

N.B: Le funzioni che risultano essere lineari nei parametri incogniti \vec{w} hanno importanti proprietà e sono chiamate **modelli lineari**.

Una semplice ma efficace scelta di f è rappresentata dalla funzione polinomiale della seguente forma (dove M rappresenta il grado del polinomio):

$$y(x | \vec{w}) = w_0 + w_1 x + w_2 x^2 + \dots + w_M x^M = \sum_{j=0}^M w_j x_j$$

Ipotizzando di avere un dataset di addestramento (ovvero un set di coppie di osservazioni (x_i, t_i)) e di avere uno spazio di ipotesi \mathcal{H} parametrizzato da \vec{w} , allora possiamo determinare i valori dei parametri \vec{w} adattando il modello (ovvero la funzione polinomiale) ai dati di addestramento. Questo può essere fatto utilizzando una **funzione loss \mathcal{L}** che misura l'inadeguatezza tra la funzione $y(x | \vec{w})$, per qualsiasi valore di \vec{w} , e i dati di addestramento.

N.B.: una loss elevata indica un cattivo adattamento del modello ai dati mentre una loss bassa indica un buon adattamento del modello ai dati.

In particolare per i problemi di regressione lineare la funzione loss si chiama *funzione di errore (error function)* e si indica con $E(\vec{w} | \mathcal{D})$. Tale funzione di errore misura l'errore tra la funzione $y(x | \vec{w})$ per un certo valore di \vec{w} ed il training set

$\mathcal{D} = \{(x_i, y_i) | i = 1, \dots, N\}$. Una semplice scelta della funzione di errore è data dalla metà della somma dei quadrati degli errori (**square error**) tra le predizioni $y(x_i | \vec{w})$ per ogni input x_i ed il corrispondente target t_i . Perciò vogliamo minimizzare la seguente funzione:

$$\mathcal{L}(\vec{w} | \mathcal{D}) = E(\vec{w} | \mathcal{D}) = \frac{1}{2} \sum_{i=1}^N [y(x_i | \vec{w}) - t_i]^2$$

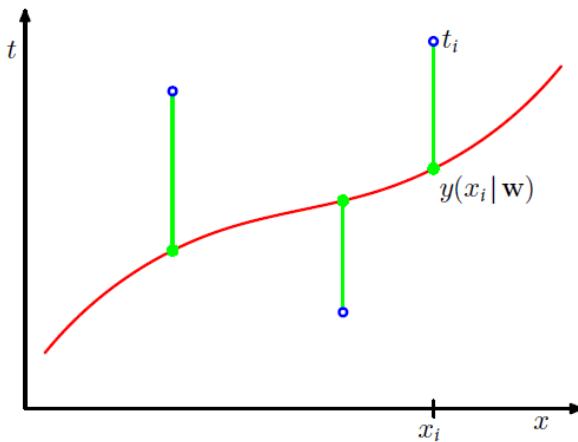


Figura 3.1: Funzione di errore corrispondente alla somma degli errori dei quadrati degli spostamenti (indicati dalle linee verdi verticali) di ciascun punto della funzione $y(x_i | \vec{w})$.

Possiamo risolvere il problema dell'adattamento della curva (**curve fitting**) ai dati di addestramento scegliendo il valore di \vec{w} per il quale $E(\vec{w} | \mathcal{D})$ è il più piccolo possibile.

N.B.: se consideriamo le assunzioni effettuate sulla linearità della funzione rispetto ai parametri \vec{w} otteniamo $E(\vec{w} | \mathcal{D}) = \frac{1}{2} \sum_{i=1}^N [(w_0 + w_1 x_i + w_2 x_i^2 + \dots + w_M x_i^M) - t_i]^2$.

La risoluzione con questa tecnica risulta più semplice rispetto ad altre tecniche (come ad esempio il gradiente), ma è molto più sensibile ai dati.

N.B.: poiché la funzione di errore è una funzione quadratica dei parametri \vec{w} , allora le sue derivate rispetto a tali parametri saranno lineari negli elementi di \vec{w} e quindi la minimizzazione della funzione di errore ha un'unica soluzione indicata con \vec{w}^* .

Quindi questa interpretazione geometrica è un'esempio di stima puntuale dei parametri di un modello (poiché si ottiene una unica soluzione plausibile \vec{w}^*).

3.2 Maximum Likelihood Estimation e Least Squares

Vogliamo sviluppare un modello probabilistico di regressione lineare. Tale modello dovrebbe:

- Permetterci di ragionare in modo probabilistico sulla qualità della nostra soluzione.
- Permettere di quantificare la fiducia nella qualità delle singole predizioni.
- Consentirci di integrare conoscenze pregresse sulle soluzioni probabili.

Il primo passo per lo sviluppo di tale modello è la *stima della massima verosimiglianza* (**Maximum Likelihood Estimate** (MLE)) dei parametri ottimali \vec{w} a partire dai dati osservati \mathcal{D} .

Il modello lineare più semplice per la regressione utilizza semplicemente combinazioni lineari delle variabili di input $\vec{x} = (x_1, \dots, x_D)^T$, ovvero:

$$y(\vec{x}, \vec{w}) = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_D x_D$$

Tale funzione risulta essere lineare sia in \vec{w} che in \vec{x} . La linearità in \vec{x} però rappresenta una grossa limitazione, quindi ammettiamo combinazioni lineari di funzioni non lineari dell'ingresso, ovvero:

$$y(\vec{x}, \vec{w}) = w_0 + \sum_{j=1}^{M-1} w_j \phi_j(\vec{x})$$

dove w_0 è detto **bias** e consente lo spostamento dell'output.

Spesso risulta conveniente definire una funzione base fittizia $\phi_0(x) = 1$ in modo da poter riscrivere in modo più compatto la funzione (si incorpora w_0 alla somma):

$$y(\vec{x}, \vec{w}) = \sum_{j=0}^{M-1} w_j \phi_j(\vec{x}) = \vec{w}^T \phi(\vec{x})$$

dove $\vec{w} = (w_0, \dots, w_{M-1})^T$ ed $\phi = (\phi_0, \dots, \phi_{M-1})^T$.

N.B.: le funzioni base più comuni sono:

- Funzione polinomiale $\phi_i(x) = x^i$
- Funzione Gaussiana $\phi_i = \exp\left\{-\frac{(x-\mu_j)^2}{2\sigma^2}\right\}$
- Funzione Sigmoide $\phi_i(x) = \tanh(x)$

Ipotizzando che la nostra variabile target t sia la realizzazione di una funzione deterministica $y(\vec{x}, \vec{w})$ con un rumore Gaussiano additivo ε , cioè $t = y(\vec{x}, \vec{w}) + \varepsilon$ dove ε è una variabile aleatoria Gaussiana a media nulla con precisione β (solitamente uguale alla varianza inversa, cioè $\frac{1}{\sigma^2}$), allora possiamo scrivere la distribuzione di probabilità come:

$$p(t \mid \vec{x}, \vec{w}, \beta) = \mathcal{N}(t \mid y(\vec{x}, \vec{w}), \beta^{-1})$$

Quello che stiamo osservando è abbastanza semplice ed intuitivo. Ad ogni punto x_0 il predittore valuta la sua previsione ponendo una Gaussiana intorno ad esso.

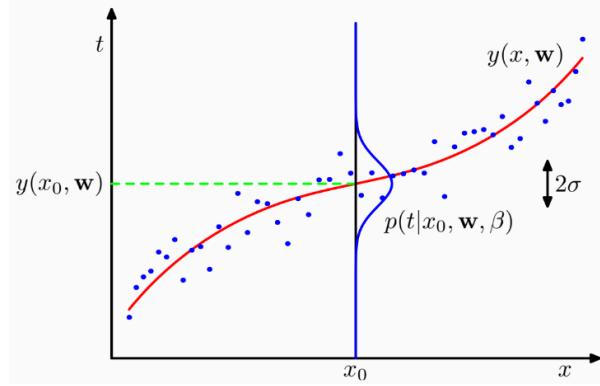


Figura 3.2: Illustrazione di una distribuzione condizionata Gaussiana per t dato x , in cui la media è data dalla funzione polinomiale $y(x, \vec{w})$ e la precisione è data dal parametro β , che è correlato alla varianza da $\beta^{-1} = \sigma^2$.

Consideriamo quindi un dataset di input $X = \{\vec{x}_1, \dots, \vec{x}_N\}$ insieme ai corrispondenti target $\vec{t} = (t_1, \dots, t_N)^T$. Assumendo che i campioni vengano estratti in modo identico ed indipendente da $p(t | \vec{x}, \vec{w}, \beta)$, allora otteniamo la seguente espressione per la *funzione di verosimiglianza*:

$$p(\vec{t} | \vec{X}, \vec{w}, \beta) = \prod_{n=1}^N \mathcal{N}(t_n | y(\vec{x}_n, \vec{w}), \beta^{-1}) = \prod_{n=1}^N \mathcal{N}(t_n | \vec{w}^T \phi(\vec{x}_n), \beta^{-1})$$

N.B.: osserviamo che nei problemi di supervised learning, come la regressione e la classificazione, non siamo interessati a modellare la distribuzione delle variabili di ingresso. Perciò X apparirà sempre nell'insieme delle variabili condizionanti e quindi verrà omesso per non appesantire la notazione.

Una volta definita la funzione di verosimiglianza, allora possiamo utilizzarla per determinare \vec{w} e β .

Ipotizziamo di voler determinare \vec{w} . L'obiettivo è quello di massimizzare la funzione di verosimiglianza in \vec{w} (ottenendo la funzione di **massima verosimiglianza**). Per fare ciò dobbiamo innanzitutto calcolare il gradiente della funzione di verosimiglianza rispetto a \vec{w} (ovvero $\nabla_{\vec{w}} p(\vec{t} | \vec{w}, \beta)$), dopodiché dobbiamo determinare il massimo ponendo il gradiente uguale a 0.

Però, tale funzione di verosimiglianza risulta essere molto complessa nel calcolo del gradiente (poiché dovremmo calcolare derivate di prodotti). Quindi, per semplificarla si considera il logaritmo della funzione e si riscrive utilizzando la forma esponenziale della Gaussiana (ovvero $\mathcal{N}(x | \mu, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\{-\frac{1}{2\sigma^2}(x - \mu)^2\}$).

Otteniamo quindi:

$$\ln p(\vec{t} | \vec{w}, \beta) = \sum_{n=1}^N \ln \mathcal{N}(t_n | \vec{w}^T \phi(\vec{x}_n), \beta^{-1}) = \frac{N}{2} (\ln \beta - \ln(2\pi)) - \frac{\beta}{2} \sum_{n=1}^N \{t_n - \vec{w}^T \phi(\vec{x}_n)\}^2$$

A questo punto massimizziamo la funzione di verosimiglianza in \vec{w} seguendo il procedimento illustrato in precedenza per ottenere le *equazioni normali (normal equations)* per il problema dei *minimi quadrati (least squares)*.

Per prima cosa ne calcoliamo il gradiente:

$$\nabla_{\vec{w}} \ln p(\vec{t} | \vec{w}, \beta) = \beta \sum_{n=1}^N \{t_n - \vec{w}^T \phi(\vec{x}_n)\} \phi(\vec{x}_n)^T$$

Quindi poniamo il gradiente uguale a 0 (ovvero $\nabla_{\vec{w}} \ln p(\vec{t} | \vec{w}, \beta) = 0$):

$$0 = \sum_{n=1}^N t_n \phi(\vec{x}_n)^T - \vec{w}^T \left(\sum_{n=1}^N \phi(\vec{x}_n) \phi(\vec{x}_n)^T \right)$$

Risolvendo per \vec{w} otteniamo il valore di massima verosimiglianza (ML) per \vec{w} , ovvero:

$$\vec{w}_{\text{ML}} = (\Phi^T \Phi)^{-1} \Phi^T \cdot \vec{t}$$

dove \vec{t} è il vettore dei target mentre Φ è una matrice di dimensione $N \times M$, detta **design matrix**, i cui elementi sono dati da $\Phi_{nj} = \phi_j(\vec{x}_n)$. Ovvero:

$$\Phi = \begin{bmatrix} \phi_0(\vec{x}_1) & \phi_1(\vec{x}_1) & \dots & \phi_{M-1}(\vec{x}_1) \\ \phi_0(\vec{x}_2) & \phi_1(\vec{x}_2) & \dots & \phi_{M-1}(\vec{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(\vec{x}_N) & \phi_1(\vec{x}_N) & \dots & \phi_{M-1}(\vec{x}_N) \end{bmatrix}$$

N.B.: la quantità $\Phi^\dagger = (\Phi^T \Phi)^{-1} \Phi^T$ è detta **pseudo-inversa di Moore-Penrose** e può essere considerata una generalizzazione della nozione di matrice inversa alle matrici non quadrate. Infatti se Φ è una matrice quadrata ed invertibile, allora utilizzando la proprietà delle matrici $(AB)^{-1} = B^{-1}A^{-1}$ osserviamo che $\Phi^\dagger = \Phi^{-1}$.

Un modo facile di pensare Φ può essere il seguente:

- Una riga è la valutazione di tutte le funzioni base sul campione di addestramento corrispondente (di dimensionalità M).
- Una colonna è la funzione base corrispondente valutata su tutti i campioni di addestramento (di dimensionalità N).

Allo stesso modo, la stima di massima verosimiglianza del parametro di precisione β può essere ricavata massimizzando $p(\vec{t} | \vec{w}, \beta)$ rispetto a β ottenendo:

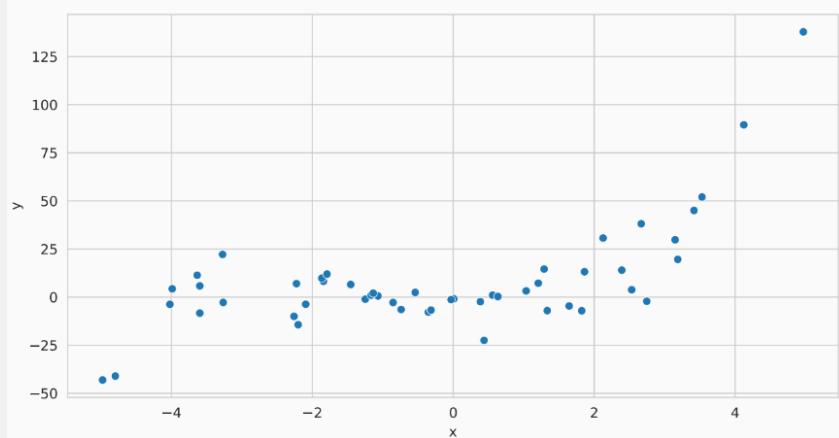
$$\frac{1}{\beta_{\text{ML}}} = \frac{1}{N} \sum_{n=1}^N \{t_n - \vec{w}_{\text{ML}}^T \phi(\vec{x}_n)\}^2$$

Quindi vediamo che l'inverso della precisione del rumore è dato dalla varianza residua dei valori target rispetto alla funzione di regressione.



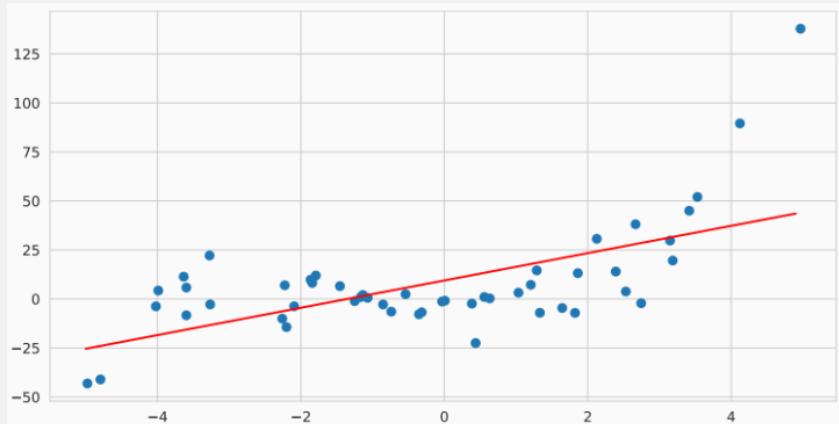
ES:

Consideriamo la seguente distribuzione dei dati:

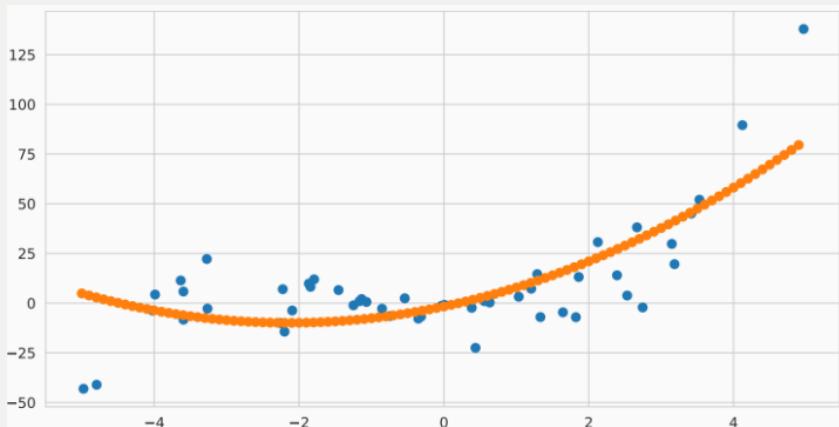


Possiamo quindi adattare il modello ai dati applicando l'approccio di curve fitting visto in precedenza utilizzando una funzione base polinomiale. A seconda del grado del polinomio scelto avremo un adattamento più o meno preciso.

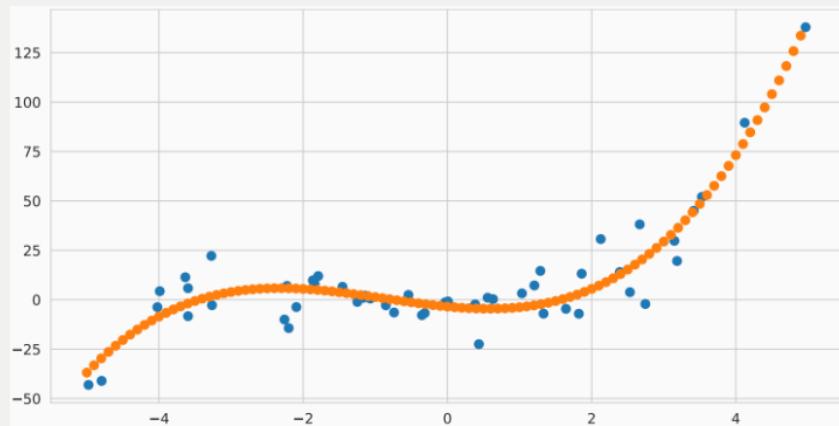
Se ad esempio consideriamo $M = 1$, avremo:



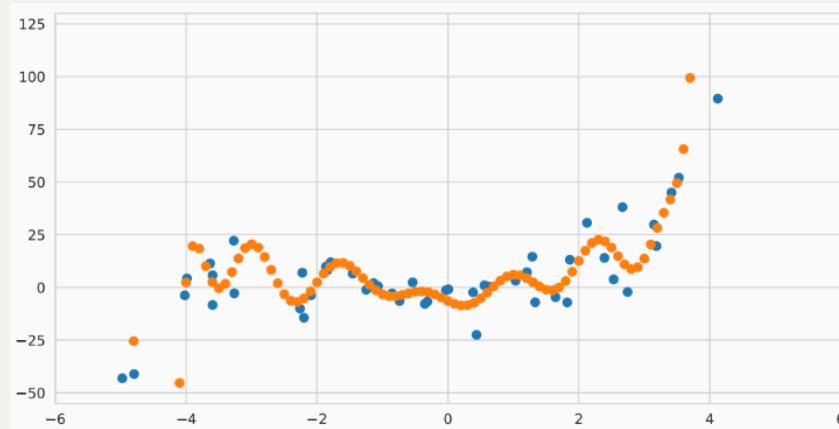
Se invece $M = 2$ (funzione quadratica) avremo:



Se $M = 3$ (funzione cubica) otteniamo:



Mentre se $M = 20$ (funzione polinomiale di grado 20) avremo:



Dobbiamo quindi individuare quale di questi modelli risulta essere il migliore.

N.B.: in questi grafici vediamo esempi di **overfitting** (quando M risulta essere troppo grande) e di **underfitting** (quando M risulta essere troppo piccolo).

Per individuare il modello migliore dobbiamo valutare la capacità del modello di catturare l'errore ε presente nelle osservazioni.

Per controllare l'overfitting di un modello vogliamo regolarizzare i minimi quadrati (**regularized least squares**). A questo scopo aggiungiamo un termine di regolarizzazione ad una funzione di errore, in modo tale che la funzione di errore totale da minimizzare assuma la seguente forma:

$$E_D(\vec{w}) + \lambda E_W(\vec{w})$$

dove il termine E_D rappresenta la funzione di errore ai minimi quadrati con le funzioni base ϕ , il coefficiente λ rappresenta il coefficiente di regolarizzazione ed il termine E_W è detto *regolarizzatore dei pesi* (**weight regularizer**) ed è comunemente definito come la norma quadratica dei pesi del modello, ovvero $E_W(\vec{w}) = \frac{1}{2} \vec{w}^T \vec{w} = \frac{1}{2} \|\vec{w}\|_2^2$.

Questa particolare scelta di regolarizzatore dei pesi (anche abbreviato come regolarizzatore) è nota come **weight decay** (*decadimento dei pesi*), perché negli algoritmi di apprendimento sequenziale spinge i valori dei pesi a decadere verso lo zero, a meno che non siano supportati dai dati.

Quindi la funzione di errore totale da minimizzare diventa:

$$E(\vec{w}) = \frac{1}{2} \sum_{n=1}^N \{t_n - \vec{w}^T \phi(\vec{x}_n)\}^2 + \frac{\lambda}{2} \vec{w}^T \vec{w}$$

Tale funzione ha il vantaggio di essere ancora una funzione quadratica di \vec{w} . In particolare, massimizzando questa funzione, otteniamo il seguente vettore dei parametri \vec{w} ottimale:

$$\vec{w}_{ML} = (\lambda I + \Phi^T \Phi)^{-1} \Phi^T \cdot \vec{t}$$

A volte viene utilizzato un regolarizzatore più generale, per il quale l'errore regolarizzato assume la seguente forma:

$$E(\vec{w}) = \frac{1}{2} \sum_{n=1}^N \{t_n - \vec{w}^T \phi(\vec{x}_n)\}^2 + \frac{\lambda}{2} \sum_{j=1}^n |w_j|^q$$

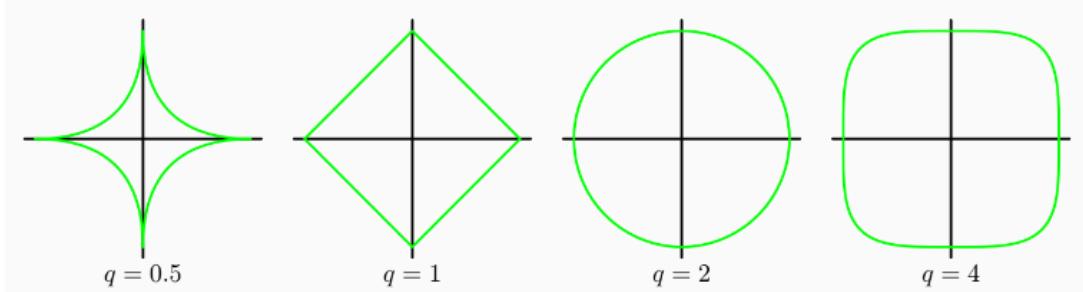


Figura 3.3: Curve del termine di regolarizzazione per diversi valori del parametro q .

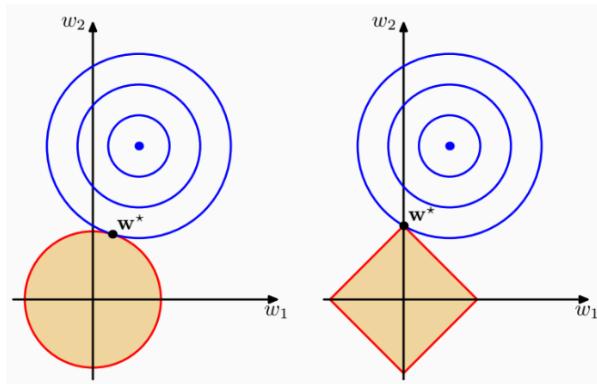


Figura 3.4: Grafico delle curve della funzione di errore non regolarizzata (in blu) insieme alla regione definita dal termine di regolarizzazione (caso $q = 2$ e $q = 1$). Il valore ottimo di \vec{w} è denotato con \vec{w}^* .

La regolarizzazione permette di addestrare modelli complessi su insiemi di dati di dimensioni limitate senza un eccessivo overfitting. Tuttavia, il problema di determinare la complessità ottimale del modello si sposta da quello di trovare il numero appropriato di funzioni base a quello di determinare il valore adeguato del coefficiente di regolarizzazione λ .

3.3 Regressione lineare Bayesiana



ES (lancio di una moneta Bayesiano):

Vogliamo determinare se una moneta è equa, ovvero se $p(H) = p(T)$.

Consideriamo il dataset di osservazioni $\mathcal{D} = \{x_1, x_2, \dots, x_N\}$ dove:

$$x_i = \begin{cases} 1 & \text{se nell'}i\text{-esimo lancio esce testa} \\ 0 & \text{altrimenti} \end{cases}$$

Dopo aver raccolto i dati in \mathcal{D} su N lanci, conosciamo il numero di “teste”

$$n_h = \sum_{i=1}^N x_i \quad (\text{e quindi il numero di “croci” sarà } n_t = N - n_h).$$

Sia θ la probabilità sconosciuta di “testa”. L’obiettivo è quello di stimare θ .

La probabilità di osservare n_h “teste” in N lanci sarà $p(\mathcal{D} | \theta) = \binom{N}{n_h} \theta^{n_h} (1 - \theta)^{n_t}$. Dobbiamo quindi trovare il valore ottimo di θ che definiamo come:

$$\begin{aligned} \theta^* &= \arg \max_{\theta} \binom{N}{n_h} \theta^{n_h} (1 - \theta)^{n_t} \\ &= \binom{N}{n_h} \arg \max_{\theta} \ln[\theta^{n_h} (1 - \theta)^{n_t}] \end{aligned}$$

Dobbiamo quindi valutare la stima di massima verosimiglianza del parametro θ .

Quindi poniamo la derivata rispetto a θ uguale a 0 e risolviamo, ovvero:

$$\begin{aligned} \binom{N}{n_h} \frac{d}{d\theta} \ln[\theta^{n_h} (1 - \theta)^{n_t}] &\propto \frac{d}{d\theta} [\ln \theta^{n_h} + \ln(1 - \theta)^{n_t}] \\ \Rightarrow \frac{d}{d\theta} [\ln \theta^{n_h} + \ln(1 - \theta)^{n_t}] &= \frac{n_h}{\theta} - \frac{n_t}{1 - \theta} \end{aligned}$$

Otteniamo che $\theta_{ML} = \frac{n_h}{n_h + n_t}$.

N.B.: osserviamo ad esempio il caso particolare $N = 5$ ed $n_h = 5$, allora avremo $\theta_{ML} = 1$.

Abbiamo visto quindi che nell’approccio di massima verosimiglianza assumiamo che i dati vengano generati da un parametro θ che dobbiamo stimare.

Diversamente, nell’approccio Bayesiano assumiamo che θ sia tratto da una distribuzione sconosciuta su tutti i parametri (detta **parameter distribution**) che dobbiamo stimare.

Conoscendo la probabilità $p(\mathcal{D} | \theta) = \binom{N}{n_h} \theta^{n_h} (1 - \theta)^{n_t}$ detta **data likelihood**, vogliamo stimare la distribuzione dei parametri $p(\theta | \mathcal{D}) =$

$\frac{p(\mathcal{D} | \theta)p(\theta)}{p(\mathcal{D})}$ dove $p(\mathcal{D} | \theta)$ è detta **data likelihood**, $p(\theta)$ è detta **prior distribution** ed $p(\mathcal{D})$ è detta **evidence** (rappresenta la probabilità marginale di vedere il dataset \mathcal{D}).

Tale distribuzione dei parametri definisce il prior per un nuovo esperimento aggiornando il belief.

A tale scopo utilizziamo la funzione Beta definita come:

$$\text{Beta}(\theta; \alpha, \beta) := \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha, \beta)}$$

dove α e β sono detti **shape parameters** ed $B(\alpha, \beta)$ è detta **beta function** (non dipende da θ).

Di conseguenza possiamo calcolare la distribuzione dei parametri come:

$$\begin{aligned} p(\theta | \mathcal{D}) &= \frac{p(\mathcal{D} | \theta)p(\theta)}{p(\mathcal{D})} \\ &\propto \theta^{n_h}(1-\theta)^{n_t}\theta^{\alpha-1}(1-\theta)^{\beta-1} \\ &\propto \text{Beta}(\theta; n_h + \alpha, n_t + \beta) \end{aligned}$$

Risolvendo per il θ ottimo abbiamo $\frac{d}{d\theta}p(\theta | \mathcal{D}) = \frac{n_h + \alpha - 1}{\theta} - \frac{n_t + \beta - 1}{1-\theta}$.

N.B.: se $\alpha = \beta = 1$ otteniamo nuovamente il caso precedente.

3.3.1 Decomposizione bias-varianza

Questa discussione sulla regolarizzazione delle soluzioni ai problemi dei minimi quadrati ci porta ad un importante strumento concettuale del Machine Learning.

L'utilizzo della massima verosimiglianza, o equivalentemente dei minimi quadrati, può portare ad un grave overfitting per modelli complessi che può essere risolto assumendo fissate la forma ed il numero delle funzioni base. Tuttavia, limitare il numero di funzioni base per evitare l'overfitting ha l'effetto collaterale di limitare la flessibilità del modello nel catturare tendenze importanti nei dati.

Sebbene l'introduzione di termini di regolarizzazione possa controllare l'overfitting dei modelli con molti parametri, si pone il problema di come determinare un valore adeguato per il coefficiente di regolarizzazione λ .

Dobbiamo quindi sviluppare una teoria che ci permette di analizzare e risolvere questo problema.

Come abbiamo osservato in precedenza, possiamo avere diverse funzioni loss \mathcal{L} . Una scelta comune della funzione loss risulta essere la funzione **square loss** definita come:

$$h(\vec{x}) := \mathbb{E}[t | \vec{x}] = \int t \cdot p(t | \vec{x}) dt$$

che rappresenta il valore atteso condizionale del target t rispetto all'input \vec{x} .

L'obiettivo è quindi quello di stimare la distribuzione $p(t | \vec{x})$. A questo scopo vogliamo trovare una y che minimizzi l'errore $\{y(\vec{x}; \mathcal{D})h(\vec{x})\}^2$ (esprime la perdita subita per un singolo input x quando si utilizza la stima $y(\vec{x}; \mathcal{D})$).

Considerando la previsione rispetto a \mathcal{D} e considerando tutti i possibili input, si ottiene:

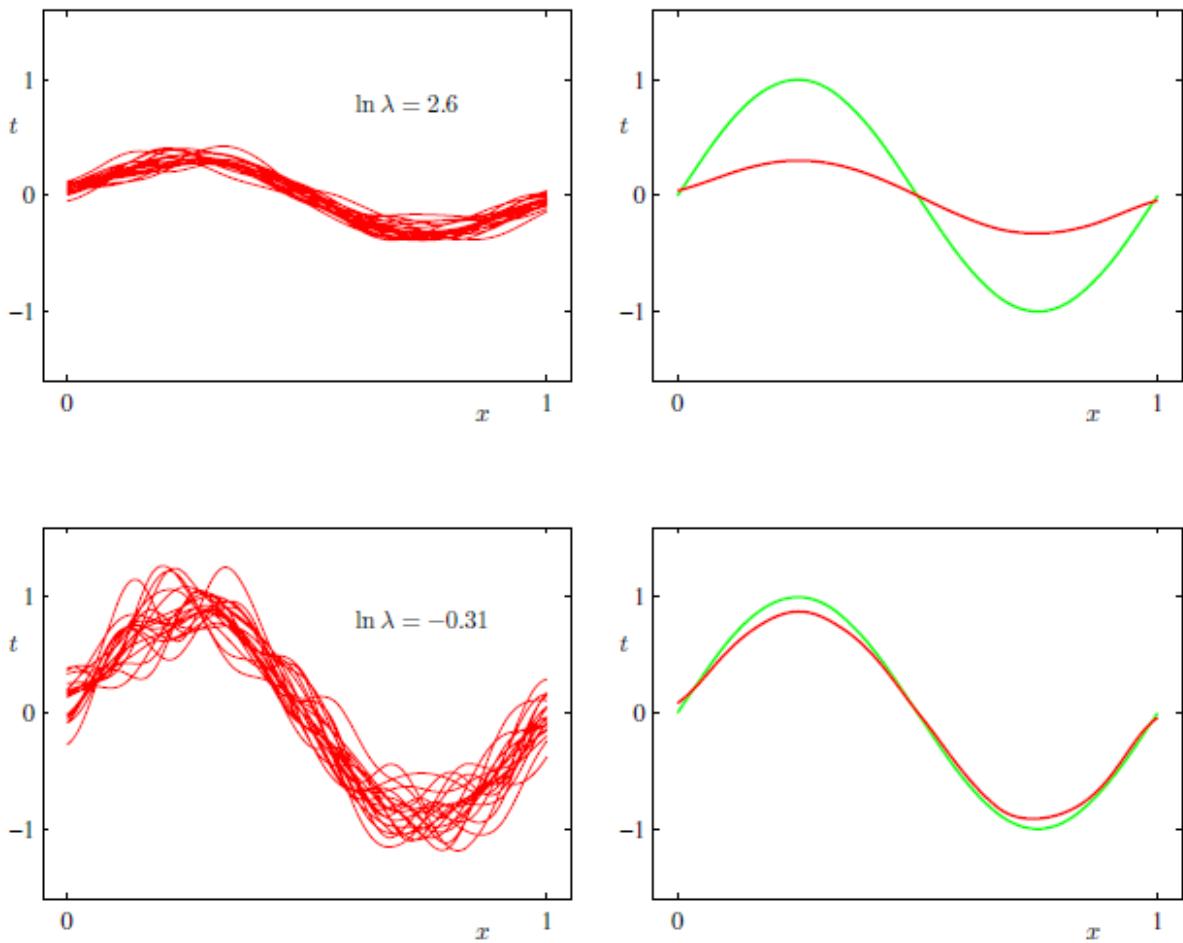
$$\text{expected loss} = (\text{bias})^2 + \text{variance} + \text{noise}$$

dove:

- $(\text{bias})^2 = \int \{\mathbb{E}_{\mathcal{D}}[y(\vec{x}; \mathcal{D})] - h(\vec{x})\}^2 p(\vec{x}) d\vec{x}$
- variance = $\int \mathbb{E}_{\mathcal{D}}[\{y(\vec{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\vec{x}; \mathcal{D})]\}^2] p(\vec{x}) d\vec{x}$
- noise = $\int \int \{h(\vec{x}) - t\}^2 d\vec{x} dt$

Il nostro obiettivo è quello minimizzare la loss attesa, che abbiamo decomposto nella somma di un *bias* (al quadrato), una *varianza* ed un termine di *rumore* costante.

N.B.: il bias e la varianza dipendono dalla complessità del modello. Perciò esiste un compromesso tra questi due parametri. I modelli più flessibili hanno un basso bias e un'alta varianza, mentre i modelli più rigidi hanno un alto bias e una bassa varianza (rilevando il coefficiente di regolazione si riduce il bias e si aumenta la varianza).



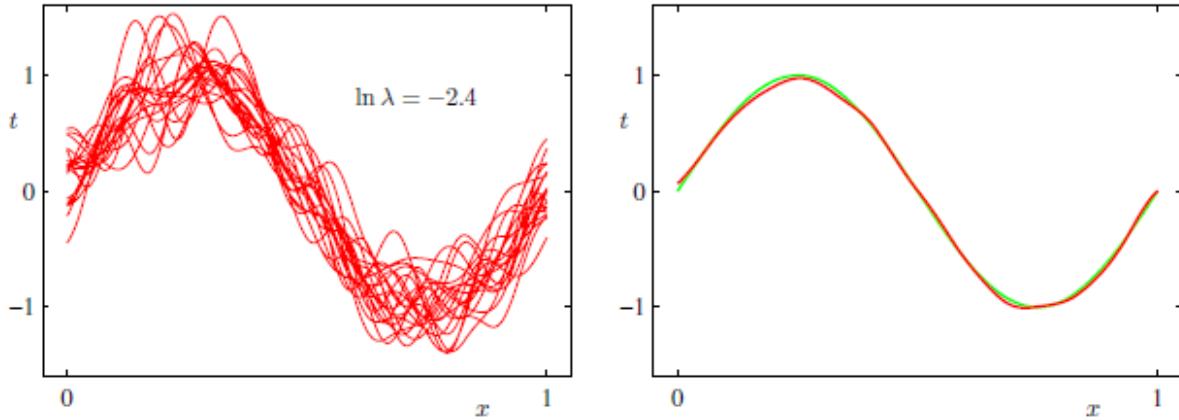


Figura 3.5: Dipendenza del bias e della varianza con la complessità del modello, regolata da un parametro di regolarizzazione λ . Le immagini a sinistra mostrano il risultato dell'adattamento del modello ai dati per diversi valori di λ , mentre le immagini di destra mostrano la corrispondente media (in rosso) insieme alla funzione sinusoidale di partenza (in verde) da cui è stato generato il dataset \mathcal{D} .

Come osserviamo dall'immagine, nella prima riga abbiamo un elevato valore del coefficiente di regolarizzazione λ che comporta una bassa varianza (poiché le curve rosse nel grafico di sinistra risultano essere molto simili) ma un elevato bias (poiché le due curve nel grafico di destra sono molto diverse).

Al contrario, nell'ultima riga abbiamo un basso valore di λ che comporta un'elevata varianza (mostrata dall'alta variabilità tra le curve rosse nel grafico di sinistra) ma un basso bias (mostrato dal buon adattamento tra il modello medio e la sinusoide originale).

Sebbene la decomposizione bias-varianza possa fornire alcuni spunti interessanti sul problema della complessità del modello, ha un valore pratico limitato poiché si basa sulle medie rispetto a insiemi di set di dati (mentre noi possediamo solo un singolo set di dati osservati). Perciò questi integrali risultano essere intrattabili.

N.B.: se si dispone di elevato numero di training set indipendenti di una certa dimensione, sarebbe meglio combinarli in un unico training set di grandi dimensioni, che ridurrebbe il livello di overfitting per una data complessità del modello.

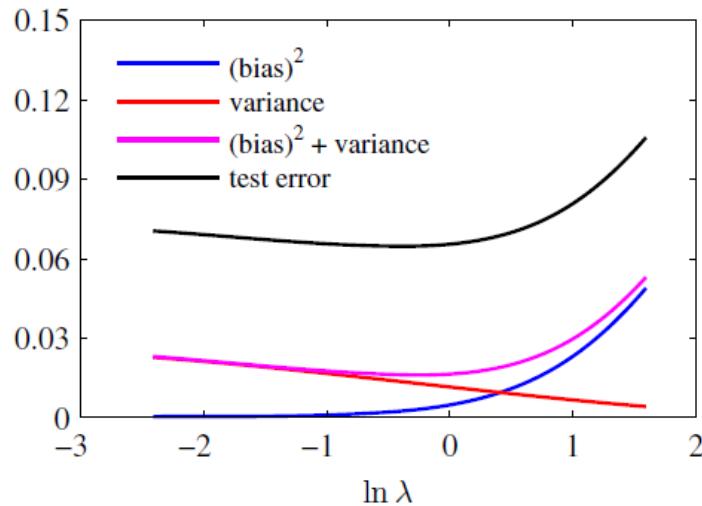


Figura 3.6: Grafico del bias (al quadrato), della varianza e della loro somma corrispondente ai risultati mostrati nella precedente figura.

Dobbiamo quindi definire metodi più pratici per stimare i compromessi empirici ottimali.

3.3.2 Regressione lineare Bayesiana

Nella discussione sulla massima verosimiglianza per la definizione dei parametri di un modello di regressione lineare, abbiamo visto che la complessità del modello deve essere controllata in base alla dimensione dell'insieme di dati. Aggiungendo un termine di regolarizzazione alla funzione di verosimiglianza, significa che la complessità effettiva del modello può essere controllata dal valore del coefficiente di regolarizzazione, anche se la scelta del numero e della forma delle funzioni base è importante per determinare il comportamento complessivo del modello.

Questo lascia il problema di decidere la complessità del modello appropriato per il problema specifico, che non può essere deciso semplicemente massimizzando la funzione di verosimiglianza perché questo porta sempre a modelli eccessivamente complessi e a un overfitting. I dati indipendenti possono essere utilizzati per determinare la complessità del modello, ma questo può essere sia computazionalmente costoso sia uno spreco di dati preziosi.

Il meccanismo matematico di massima verosimiglianza è ottimo, ma non permette di stabilire quanto dovremmo credere in una particolare soluzione.

N.B: il significato di credenza assume diversi aspetti:

- se la regressione produce un w_{ML} dai dati \mathcal{D} , vogliamo capire quanto risulta affidabile w_{ML} (ovvero quanto crediamo che sia vicino al vero \vec{w}^* che ha generato il dataset \mathcal{D}).
- se prediciamo un target t' su un nuovo input \vec{x}' utilizzando il classificatore $t' = y(\vec{x}, \vec{w}_{ML})$, vogliamo capire quanto possiamo credere in t' e se tale credenza risulta valida su tutto lo spazio.
- se possediamo una conoscenza preliminare, detta prior (cioè una convinzione sulla distribuzione dei parametri $p(\vec{w})$), vogliamo capire se è possibile incorporarla nella stima di w_{ML} .

Per questo motivo, ci rivolgiamo a un trattamento Bayesiano della regressione lineare, che eviterà il problema dell'overfitting della massima verosimiglianza, e che porterà anche a metodi automatici per determinare la complessità del modello utilizzando solo i dati di addestramento sfruttando la *verosimiglianza* (**likelihood**), l'*antecedenza* (**prior**) e l'*evidenza* (**evidence**).

Distribuzione dei parametri

Vogliamo quantificare il **belief** su uno specifico modello \vec{w}^* stimato da \mathcal{D} .

Ricordando la regola di Bayes $p(\vec{w} | \vec{t}) = \frac{\text{data likelihood} \times \text{prior}}{\text{evidence}}$, possiamo ricavare la data likelihood di un determinato modello nel seguente modo:

$$p(\vec{t} | \vec{w}, \beta) = \prod_{n=1}^N \mathcal{N}(t_n | \vec{w}^T \phi(\vec{x}_n), \beta^{-1})$$

Abbiamo bisogno di una distribuzione prior $p(\vec{w})$ che esprima la nostra convinzione sui valori che \vec{w} potrebbe assumere.

Scegliamo innanzitutto il prior in modo che abbia un valore atteso ragionevole. Per esempio potremmo aspettarci che i pesi siano vicini allo zero, in media, con una varianza attesa intorno allo zero. Scegliamo inoltre la forma del prior in modo che si adatti bene alla likelihood (poiché dobbiamo moltiplicare il prior alla likelihood). Perciò scegliamo un prior del tipo $p(\vec{w} | \alpha) = \mathcal{N}(\vec{w} | \vec{0}, \alpha^{-1}I)$ che si tratta di un **Prior Gaussian Conjugate** (significa che quando lo moltiplichiamo con una likelihood Gaussiana otteniamo un posterior Gaussiano). Quindi la distribuzione posterior su \vec{w} assume la seguente forma:

$$p(\vec{w} | \vec{t}) = p(\vec{t} | \vec{w}, \beta^{-1})p(\vec{w} | \alpha) = \mathcal{N}(\vec{w} | \vec{m}_N, S_N)$$

dove:

- $\vec{m}_N = \beta S_N \Phi^T \vec{t}$
- $S_N^{-1} = \alpha I + \beta \Phi^T \Phi$

Poiché stiamo trattando termini Gaussiani, otteniamo che il logaritmo la distribuzione posterior in funzione di \vec{w} è data dalla somma del logaritmo della likelihood e del logaritmo del prior, ovvero:

$$\ln p(\vec{w} | \vec{t}) = \frac{\beta}{2} \sum_{n=1}^N \{t_n - \vec{w}^T \phi(\vec{x}_n)\}^2 - \frac{\alpha}{2} \vec{w}^2 \vec{w} + \text{const}$$

La massimizzazione di questa distribuzione posterior rispetto a \vec{w} è quindi equivalente alla minimizzazione della funzione di errore della somma dei quadrati (che è la stessa soluzione ottenuta in precedenza con least squares) con l'aggiunta di un termine di regolarizzazione quadratico $\lambda = \frac{\alpha}{\beta}$. Però adesso abbiamo ottenuto delle conseguenze ancora più potenti. Infatti possiamo quantificare il belief di una certa soluzione \vec{w}^* e possiamo imparare in modo incrementale all'arrivo di nuovi dati.



ES:

Consideriamo di non avere dati di partenza (prima riga dell'immagine). Di conseguenza il posterior risulta essere inizialmente uguale al prior.

Quando però iniziamo ad osservare un dato (seconda riga dell'immagine), possiamo utilizzare la regola di Bayes per aggiornare il belief (ovvero moltiplichiamo la funzione likelihood con il prior della riga superiore e lo normalizziamo, ottenendo la distribuzione posterior mostrata nel grafico centrale della seconda riga).

N.B: il dato osservato è indicato con un cerchio blu nei grafici della colonna di destra dell'immagine.

N.B: nella colonna di sinistra sono riportati i grafici della funzione likelihood $p(t | x, \vec{w})$.

Via via che si osservano i dati, continuiamo ad utilizzare la regola di Bayes per aggiornare il belief.

Alla fine, la varianza si riduce e si stabilizza la stima del posterior intorno alla soluzione di massima verosimiglianza.

N.B: se consideriamo il limite di un numero infinito di dati osservati, la distribuzione posterior risulterebbe centrata nel parametro vero \vec{w}^* rappresentato con una croce bianca.

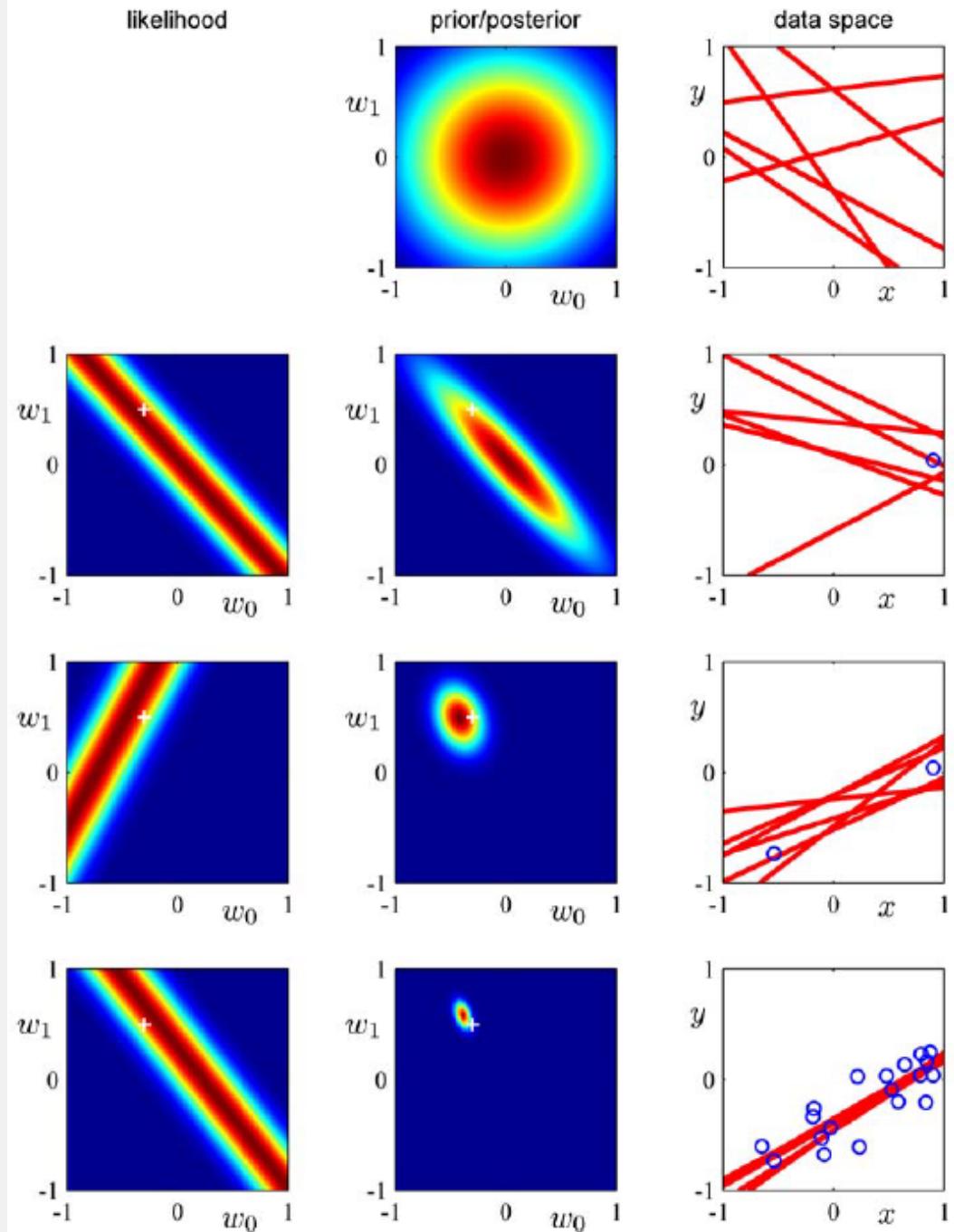


Figura 3.7: Illustrazione dell'apprendimento Bayesiano sequenziale per un semplice modello lineare della forma $y(x, \vec{w}) = w_0 + w_1 x$ all'aumentare della dimensione del dataset (la distribuzione posterior viene valutata a partire dal prior quando viene generata una nuova osservazione).

Distribuzione predittiva

Di solito non siamo interessati al valore effettivo di \vec{w} , ma piuttosto a fare previsioni su t per nuovi valori di x . Vogliamo quindi valutare quanto siamo soddisfatti di un output $y(\vec{x}, \vec{w}^*)$.

A tale scopo definiamo la *distribuzione predittiva* (**predictive distribution**) come:

$$p(t | \vec{t}, \alpha, \beta) = \int p(t | \vec{w}, \beta) p(\vec{w} | \vec{t}, \alpha, \beta) d\vec{w}$$

dove \vec{t} rappresenta il vettore dei target del training set. Inoltre ricordiamo che per semplificare la notazione abbiamo omesso i vettori di input \vec{x} .

Possiamo interpretare tale distribuzione come una media (cioè valore atteso) delle probabilità condizionali, dove il valore atteso è rispetto al posterior (distribuzione dei parametri).

Se approfondiamo la natura gaussiana della distribuzione predittiva notiamo che essa rappresenta una convoluzione di due gaussiane e quindi possiamo derivare la seguente forma analitica:

$$p(t | \vec{t}, \alpha, \beta) = \mathcal{N}(t | \vec{m}_N^T \phi(\vec{x}), \sigma_N^2(\vec{x}))$$

dove $\sigma_N^2(\vec{x}) = \frac{1}{\beta} + \phi(\vec{x})^T S_N \phi(\vec{x})$ in cui $\frac{1}{\beta}$ rappresenta il rumore dei dati e σ_N^2 rappresenta l'incertezza nella stima dei parametri \vec{w} .

N.B.: poiché il rumore e la distribuzione di \vec{w} sono gaussiane indipendenti, allora le loro varianze sono additive.

N.B.: via via che si osservano altri dati, la distribuzione posterior si restringe. Di conseguenza si può dimostrare che $\sigma_{N+1}^2(\vec{x}) < \sigma_N^2(\vec{x})$. Questo significa che un maggior numero di dati comporta sempre una soluzione migliore.

Come illustrazione della distribuzione predittiva per i modelli di regressione lineare Bayesiana, consideriamo l'insieme di dati generati (rappresentati dai cerchi blu) da una funzione sinusoidale (rappresentata dalla curva verde).

Per ogni grafico, la curva rossa mostra la media della corrispondente distribuzione predittiva gaussiana e la regione rossa ombreggiata si estende per una deviazione standard su entrambi i lati della media. Tali grafici rappresentano la varianza predittiva puntuale in funzione di x .

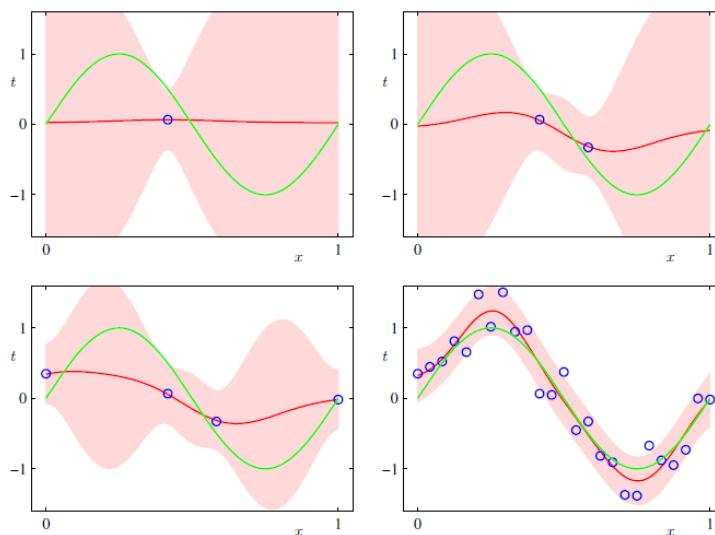


Figura 3.8: Esempio di distribuzione predittiva.

Al fine di comprendere la covarianza tra le previsioni per diversi valori di x possiamo estrarre campioni dalla distribuzione posteriore su \vec{w} e tracciare le funzioni corrispondenti $y(x, \vec{w})$. Come abbiamo osservato, via via che si osservano nuovi dati il secondo termine di $\sigma_N^2(\vec{x})$ si azzera, lasciando solo il contributo del rumore $\frac{1}{\beta}$.

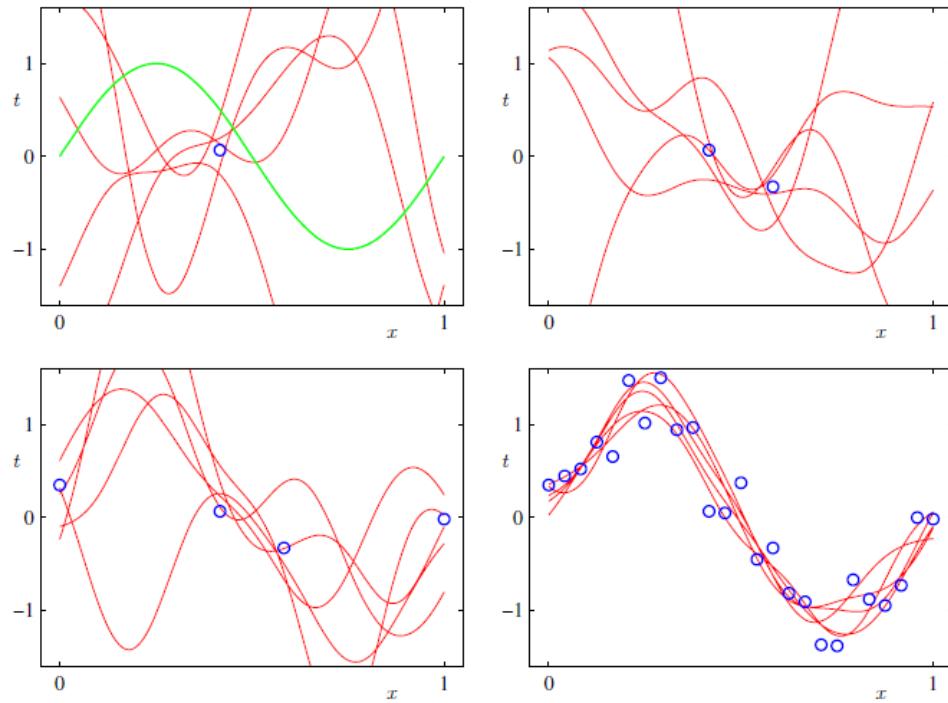


Figura 3.9: Grafici della funzione $y(x, \vec{w})$ campionando dalle distribuzioni posteriori su \vec{w} .

Capitolo 4

MODELLI LINEARI PER LA CLASSIFICAZIONE

Vogliamo analizzare una classe di modelli lineari per la risoluzione di problemi di classificazione. L'obiettivo della classificazione è prendere un vettore di input \vec{x} e assegnarlo a una classe \mathcal{C}_k (insieme discreto di K classi), dove $k = 1, \dots, K$.

In questo modo lo spazio di input viene suddiviso in *regioni decisionali* (**decision regions**) i cui confini sono chiamati *confini decisionali* (**decision boundaries**) o *superfici decisionali* (**decision surfaces**).

N.B.: considereremo il caso più semplice e comune, detto *classificazione a singola etichetta* (**single label classification**), in cui le classi vengono considerate disgiunte. In questo modo ad ogni input \vec{x} viene assegnata una e una sola classe.

Consideriamo innanzitutto dei modelli lineari per il problema della classificazione in cui le superfici decisionali sono funzioni lineari del vettore di input \vec{x} .

I dataset le cui classi possono essere separate esattamente da superfici decisionali lineari si dicono *linearmente separabili* (**linearly separable**).

Per i problemi di regressione lineare, la variabile target \vec{t} è semplicemente il vettore di numeri reali di cui se ne vuole prevedere il valore. Nel caso dei problemi di classificazione, invece, esistono vari modi di definire i target \vec{t} per rappresentare le etichette di classe.

Ad esempio, per i modelli probabilistici, nel caso di problemi a due classi, si utilizza la rappresentazione binaria per la quale esiste una singola variabile target $t \in \{0, 1\}$ tale che $t = 1$ rappresenta la classe \mathcal{C}_1 e $t = 0$ rappresenta la classe \mathcal{C}_2 .

Per problemi con $K > 2$ classi, invece, è conveniente utilizzare uno schema di codifica 1-di- K in cui \vec{t} è un vettore di lunghezza K tale che se la classe assegnata è \mathcal{C}_j , allora tutti gli elementi t_k di \vec{t} sono nulli ad eccezione dell'elemento t_j , che assume il valore 1 (ad esempio se consideriamo $K = 5$, allora per un modello della classe \mathcal{C}_2 verrà assegnato il vettore target $\vec{t} = [0, 1, 0, 0, 0]^T$).

4.1 Funzioni discriminanti lineari

Un **discriminante** è una funzione che prende un vettore di input \vec{x} e lo assegna a una delle K classi (indicate con \mathcal{C}_k). Osserviamo in modo particolare i discriminanti lineari, cioè quelli per i quali le superfici decisionali risultano essere iperpiani.

4.1.1 Due classi

Consideriamo il caso semplice con $K = 2$ classi.

La rappresentazione più semplice di una funzione discriminante lineare si ottiene considerando la seguente funzione lineare del vettore di input \vec{x} :

$$y(\vec{x}) = \vec{w}^T \vec{x} + w_0$$

dove \vec{w} è detto vettore **weight** e w_0 è detto **bias**.

N.B.: l'opposto del bias (ovvero il bias negato) è detto **threshold**.

In particolare un vettore di input \vec{x} viene assegnato alla classe \mathcal{C}_1 se $y(\vec{x}) \geq 0$ (ovvero se $\vec{w}^T \vec{x} + w_0 \geq 0 \Rightarrow \vec{w}^T \vec{x} \geq -w_0$) e alla classe \mathcal{C}_2 altrimenti (ovvero se $\vec{w}^T \vec{x} + w_0 < 0 \Rightarrow \vec{w}^T \vec{x} < -w_0$). Quindi:

$$\text{classe}(\vec{x}) = \begin{cases} \mathcal{C}_1 & \text{se } \vec{w}^T \vec{x} + w_0 \geq 0 \\ \mathcal{C}_2 & \text{se } \vec{w}^T \vec{x} + w_0 < 0 \end{cases}$$

Il confine decisionale corrispondente è quindi definito dalla relazione $y(\vec{x}) = 0$.

Consideriamo due punti x_A e x_B che giacciono entrambi sulla superficie di decisione. Poiché $y(x_A) = y(x_B) = 0$, si ha $\vec{w}^T(x_A - x_B) = 0$ e quindi il vettore \vec{w} è ortogonale a ogni vettore che giace sulla superficie di decisione e determina l'orientamento della superficie di decisione.

Analogamente, se x è un punto sulla superficie di decisione, allora $y(x) = 0$ e quindi la distanza ortogonale dall'origine alla superficie di decisione è data da $\frac{y(\vec{x})}{\|\vec{w}\|_2}$, ovvero da:

$$\frac{\vec{w}^T \vec{x}}{\|\vec{w}\|_2} = -\frac{w_0}{\|\vec{w}\|_2}$$

Vediamo quindi che il parametro di bias w_0 determina la posizione della superficie decisionale.

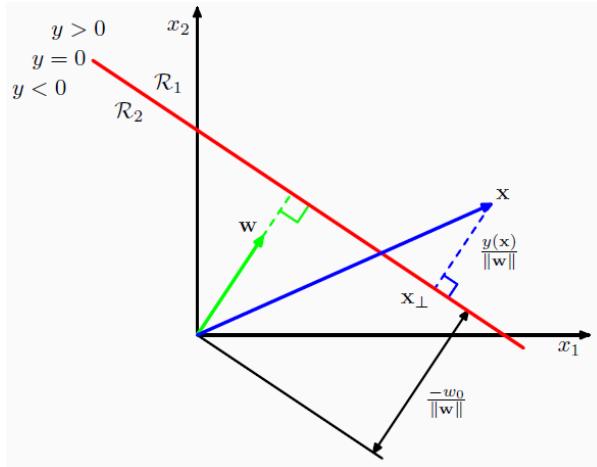


Figura 4.1: Illustrazione della geometria di una funzione discriminante lineare in due dimensioni. La superficie di decisione (in rosso) è perpendicolare a \vec{w} (in verde) e il suo spostamento dall'origine è determinato dal parametro bias w_0 . Inoltre, la distanza ortogonale di un generico punto x dalla superficie decisionale è data da $\frac{y(\vec{x})}{\|\vec{w}\|_2}$.

Come nel caso dei modelli di regressione lineare, a volte è conveniente utilizzare una notazione più compatta in cui si introduce un ulteriore valore di input fittizio $x_0 = 1$ e si definiscono $\tilde{w} = (w_0, \vec{w})^T$ e $\tilde{x} = (x_0 = 1, \vec{x})^T$ in modo che $y(\vec{x}) = \tilde{w}^T \tilde{x}$.

N.B.: le superfici decisionali in questo spazio sono iperpiani di dimensione D che passano per l'origine dello spazio aumentato a $(D + 1)$ dimensioni.

4.1.2 Multiclasse

Consideriamo adesso l'estensione dei discriminanti lineari a $K > 2$ classi.

Potremmo essere tentati di costruire un discriminante di classe K combinando una serie di funzioni discriminanti di due classi. Tuttavia, ciò comporta alcune serie difficoltà che adesso vediamo.

Consideriamo l'uso di $K - 1$ classificatori, ciascuno dei quali risolve un problema di classificazione a due classi di separare i punti di una particolare classe \mathcal{C}_k dai punti non appartenenti a quella classe, noto come classificatore *uno-contro-tutti* (**one-versus-rest**).

In alternativa, consideriamo l'uso di $K(K - 1)/2$ funzioni discriminanti binarie, una per ogni possibile coppia di classi, in cui ogni punto viene quindi classificato in base a un voto di maggioranza tra le funzioni discriminanti. Tale classificatore è detto *uno-contro-uno* (**one-versus-one**).

Queste soluzioni però comportano delle regioni ambigue di indecisione.

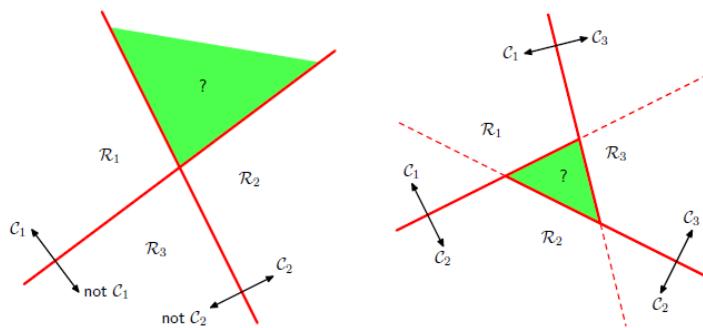


Figura 4.2: Il tentativo di costruire un discriminante di classe K da un insieme di due discriminanti comporta delle regioni ambigue (evidenziate in verde).

Nella figura di sinistra è riportato un esempio che prevede l'uso di due discriminanti progettati per distinguere i punti della classe \mathcal{C}_k da quelli non appartenenti alla classe \mathcal{C}_k .

A destra viene riportato un esempio che prevede l'uso di tre funzioni discriminanti, ognuna delle quali viene utilizzata per separare una coppia di classi \mathcal{C}_k e \mathcal{C}_j .

Possiamo evitare queste difficoltà considerando un singolo discriminante di K classi che utilizza k funzioni lineari della seguente forma:

$$y_k(\vec{x}) = \tilde{w}_k^T \vec{x} + w_{k0}$$

Che possiamo anche impacchettare in un'unica moltiplicazione matriciale:

$$y(\vec{x}) = \widetilde{W}^T \tilde{x}$$

dove \widetilde{W} è la matrice dei parametri la cui k -esima colonna contiene il vettore $\tilde{w}_k = (w_{k0}, \tilde{w}_k^T)^T$ ed \tilde{x} corrisponde al vettore di input aumentato $(x_0 = 1, \vec{x}^T)^T$.

Quindi assegniamo un punto \vec{x} alla classe \mathcal{C}_k se $y_k(x) > y_j(x) \forall j \neq k$.

N.B.: il confine decisionale tra la classe \mathcal{C}_k e la classe \mathcal{C}_j è dato da $y_k(x) = y_j(x)$.

N.B.: le regioni di decisione di un tale discriminante sono sempre convesse e singolarmente connesse.

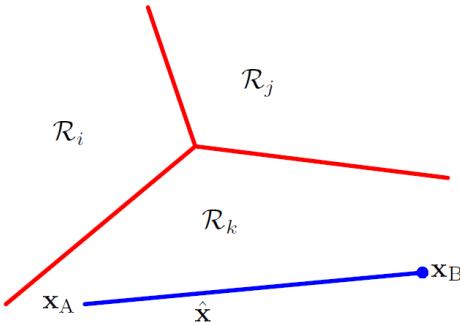


Figura 4.3: Illustrazione delle regioni di decisione per un discriminante lineare multiesemplificato, i cui confini di decisione sono indicati in rosso.

Se due punti x_A e x_B si trovano entrambi all'interno della stessa regione decisionale \mathcal{R}_k , allora ogni punto \hat{x} che si trova sulla retta che collega questi due punti deve trovarsi anch'esso in \mathcal{R}_k , e quindi la regione di decisione deve essere connessa e convessa.

4.1.3 Least Squares per la classificazione

Per la regressione lineare abbiamo considerato modelli che fossero funzioni lineari dei parametri e abbiamo visto che la minimizzazione di una funzione di errore della somma dei quadrati porta a una semplice soluzione in forma chiusa per i valori dei parametri. Vogliamo quindi vedere se possiamo applicare lo stesso formalismo ai problemi di classificazione.

Consideriamo un problema generale di classificazione con K classi, con uno schema di codifica binaria 1-di- K per il vettore target \vec{t} .

Ogni classe \mathcal{C}_k è descritta dal suo modello lineare $y_k(\vec{x}) = \vec{w}_k^T \vec{x} + w_{k0}$ che, come abbiamo già osservato, possiamo riscrivere nella forma $y(\vec{x}) = \widetilde{W}^T \tilde{x}$.

Determiniamo ora la matrice dei parametri \widetilde{W} minimizzando una funzione di errore della somma dei quadrati.

Si consideri un dataset di addestramento $\{\vec{x}_n, \vec{t}_n\}$ per $n \in \{1, \dots, N\}$ e definiamo una matrice T la cui n -esima riga è costituita dal vettore \vec{t}_n^T ed una matrice \widetilde{X} la cui n -esima riga è costituita dal vettore aumentato \tilde{x}_n^T .

Allora la funzione di errore della somma dei quadrati può quindi essere scritta nel seguente modo (dove Tr è la funzione *traccia* definita come la somma degli elementi sulla diagonale principale della matrice corrispondente):

$$E(\widetilde{W}) = \frac{1}{2} \text{Tr} \left\{ (\widetilde{X} \widetilde{W} - T)^T (\widetilde{X} \widetilde{W} - T) \right\}$$

Ponendo a zero la derivata rispetto a \widetilde{W} e riordinando, si ottiene la soluzione per \widetilde{W} nella forma seguente:

$$\widetilde{W} = (\widetilde{X}^T \widetilde{X})^{-1} \widetilde{X}^T T = \widetilde{X}^\dagger T$$

dove \widetilde{X}^\dagger è la pseudo-inversa di \widetilde{X} (come abbiamo già osservato).

Quindi otteniamo una funzione discriminante della seguente forma (forma analitica per un classificatore di classe K a partire dai dati in input):

$$y(\vec{x}) = \widetilde{W}^T \tilde{x} = T^T (\widetilde{X}^\dagger)^T \tilde{x}$$

L'approccio dei minimi quadrati fornisce una soluzione esatta in forma chiusa per i parametri della funzione discriminante. Tuttavia, anche come funzione discriminante

(usata per prendere decisioni a prescindere da ogni interpretazione probabilistica) soffre di alcuni gravi problemi. Abbiamo già visto per i problemi di regressione che le soluzioni ai minimi quadrati non risultano essere robuste nei confronti degli outlier, e questo vale anche per problemi di classificazione.

Osserviamo infatti che i punti di dati aggiuntivi nella figura di destra destra producono un cambiamento significativo nella posizione del confine decisionale, anche se questi punti sarebbero stati comunque classificati correttamente dal confine decisionale originale nella figura di sinistra. Questo perché la funzione di errore della somma dei quadrati penalizza le predizioni che risultano essere “troppo corrette”, in quanto si trovano molto lontano dal lato corretto del confine decisionale.

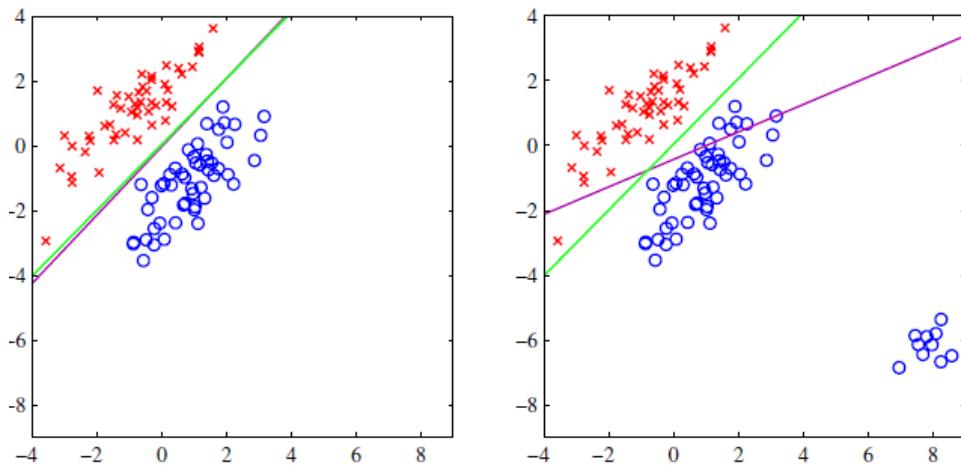


Figura 4.4: L’illustrazione mostra la classificazione per un dataset di due classi (rappresentate da croci rosse e cerchi blu), insieme ai confini decisionali trovati dai minimi quadrati (curva magenta) e dal modello di regressione logistica (curva verde).

Il grafico di destra mostra i risultati ottenuti quando vengono aggiunti altri dati in basso a sinistra del diagramma, detti *valori anomali* (*outliers*). Si osserva quindi che i minimi quadrati risultano essere molto sensibili agli outliers, a differenza della regressione logistica.

Tuttavia, i problemi con i minimi quadrati possono essere più gravi della semplice mancanza di robustezza. Infatti, osserviamo una serie di dati appartenenti a tre classi diverse in uno spazio di input bidimensionale (x_1, x_2) , con la proprietà che i confini di decisione lineari forniscano un’eccellente separazione tra le classi. In effetti, la tecnica della regressione logistica (descritta più avanti), fornisce una soluzione soddisfacente, come si vede nel grafico di destra. Tuttavia la soluzione dei minimi quadrati produce dei risultati scarsi, con solo una piccola regione dello spazio di input assegnata alla classe verde (come si osserva nel grafico di sinistra)

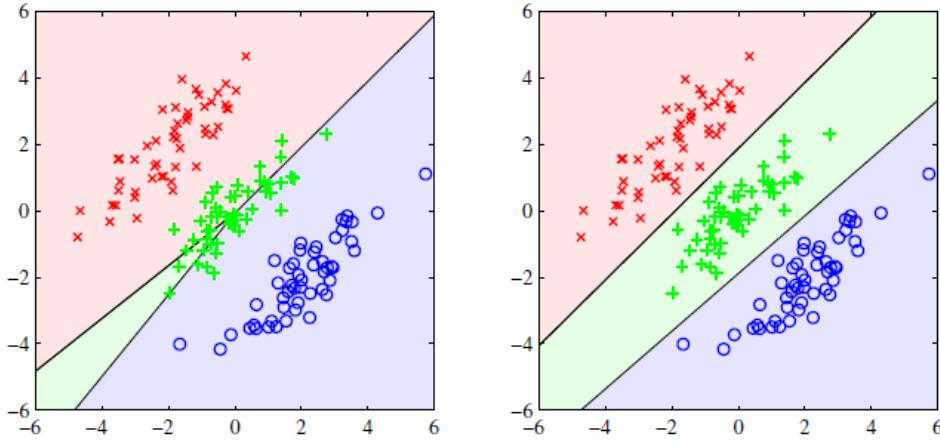


Figura 4.5: Esempio di un dataset di training di tre classi (distinte in rosso, verde e blu). Le linee indicano i confini della decisione e i colori di sfondo indicano le rispettive classi delle regioni di decisione. A sinistra è riportato il risultato dell'utilizzo di un discriminante ai minimi quadrati. Si nota che la regione dello spazio di input assegnata alla classe verde è troppo piccola e quindi la maggior parte dei punti di questa classe vengono classificati in modo errato. A destra, il risultato dell'utilizzo delle regressioni logistiche che mostra la corretta classificazione dei dati di addestramento.

4.1.4 Discriminante lineare di Fisher

Un modo di vedere un modello di classificazione lineare è in termini di riduzione della dimensionalità a una sola dimensione.

In generale, la proiezione su una dimensione comporta una notevole perdita di informazioni e le classi che risultano essere ben separate nello spazio D -dimensionale originale possono diventare fortemente sovrapposte in uno spazio ad una sola dimensione. Tuttavia, regolando le componenti del vettore peso \vec{w} , possiamo selezionare una proiezione che massimizzi la separazione delle classi.

Consideriamo innanzitutto il caso di due classi ($K = 2$) e supponiamo di prendere il vettore di input \vec{x} a D dimensioni e di proiettarlo in una sola dimensione utilizzando $y = \vec{w}^T \vec{x}$. Se poniamo una soglia su y e classifichiamo $y \geq -w_0$ come classe \mathcal{C}_∞ e $y < -w_0$ come classe \mathcal{C}_2 , otteniamo il classificatore lineare standard discusso precedentemente.

Ipotizziamo di avere N_1 punti appartenenti alla classe \mathcal{C}_1 ed N_2 punti appartenenti alla classe \mathcal{C}_2 , in modo tale che i vettori medi delle due classi siano definiti da:

$$\vec{m}_1 = \frac{1}{N_1} \sum_{n \in \mathcal{C}_1} \vec{x}_n = \frac{1}{N_1} \sum_{n=1}^{N_1} \vec{x}_{1,n} \quad \vec{m}_2 = \frac{1}{N_2} \sum_{n \in \mathcal{C}_2} \vec{x}_n = \frac{1}{N_2} \sum_{n=1}^{N_2} \vec{x}_{2,n}$$

La misura più semplice della separazione delle classi proiettata su \vec{w} è la separazione delle medie delle classi proiettate. Questo suggerisce che potremmo scegliere \vec{w} in modo che la proiezione di questi punti su \vec{w} massimizzi la distanza tra le proiezioni, ovvero:

$$m_2 - m_1 = \vec{w}^T (\vec{m}_2 - \vec{m}_1)$$

dove $m_k = \vec{w}^T \vec{m}_k$ rappresenta la media dei dati proiettati della classe \mathcal{C}_k .

Tuttavia, questa espressione può essere resa arbitrariamente grande semplicemente aumentando la grandezza di \vec{w} .

Per calcolare la massimizzazione, utilizzando un moltiplicatore di Lagrange, potremmo imporre il vincolo a \vec{w} di avere una lunghezza unitaria, ovvero $\|\vec{w}\|_2 = \sum_i w_i^2 = 1$. Da tale massimizzazione otteniamo sorprendentemente che $\vec{w} \propto (\vec{m}_2 - \vec{m}_1)$.

Tuttavia questo approccio presenta un problema. Consideriamo due classi ben separate nello spazio bidimensionale originale (x_1, x_2) ma che si sovrappongono in modo considerevole quando vengono proiettate sulla linea che unisce le loro medie. Questo difficoltà deriva dalle distribuzioni di classe che hanno matrici di covarianza fortemente non diagonali.

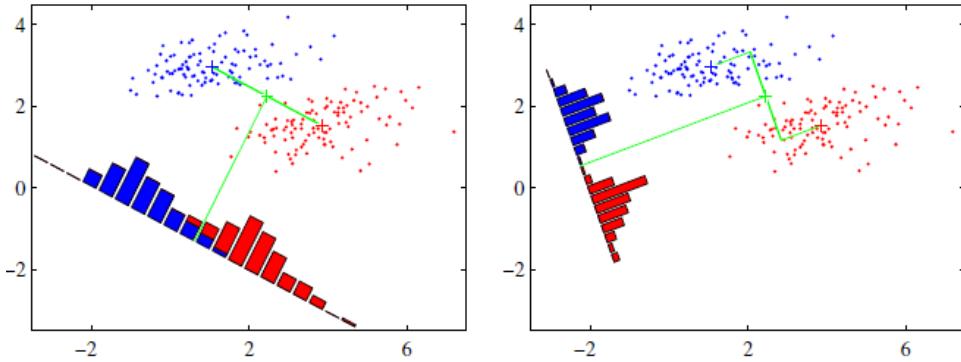


Figura 4.6: Il grafico di sinistra mostra i campioni di due classi (distinte in rosso e blu) e gli istogrammi risultanti dalla proiezione sulla linea che unisce le medie delle classi. Si noti che c'è una notevole sovrapposizione di classi nello spazio proiettato.

Il grafico di destra mostra la proiezione corrispondente basata sul discriminante lineare di Fisher, in cui la separazione delle classi risulta notevolmente migliorata.

L'idea proposta da Fisher è quella di massimizzare una funzione che dia un'ampia separazione tra le medie delle classi proiettate e allo stesso tempo una piccola varianza all'interno di ciascuna classe, minimizzando così la sovrapposizione tra le classi.

La varianza interna alla classe \mathcal{C}_k , detta *compattezza* (**compactness**), è quindi data da:

$$s_K^2 = \sum_{n \in \mathcal{C}_k} (y_n - m_k)^2 = \sum_{n=1}^{N_k} (\vec{w}^T \vec{x}_{k,n} - \vec{w}^T \vec{m}_k)^2$$

Possiamo definire la varianza totale all'interno della classe per l'intero dataset semplicemente come $s_1^2 + s_2^2$.

Il **criterio di Fisher** viene quindi definito come il rapporto della varianza tra le classi e della varianza totale all'interno della classe, ovvero:

$$J(\vec{w}) = \frac{(\vec{m}_2 - \vec{m}_1)^2}{s_1^2 + s_2^2} = \frac{(\vec{w}^T \vec{m}_2 - \vec{w}^T \vec{m}_1)^2}{s_1^2 + s_2^2}$$

Possiamo inoltre rendere più esplicita la dipendenza da \vec{w} riscrivendo il criterio di Fisher nella seguente forma:

$$J(\vec{w}) = \frac{\vec{w}^T S_B \vec{w}}{\vec{w}^T S_W \vec{w}}$$

dove S_B è la matrice di covarianza tra le classi ed S_W è la matrice di covarianza totale all'interno della classe.

Differenziando rispetto a \vec{w} , troviamo che $J(\vec{w})$ è massimizzato quando $(\vec{w}^T S_B \vec{w}) S_W \vec{w} = (\vec{w}^T S_W \vec{w}) S_B \vec{w}$.

Osserviamo che $S_B \vec{w}$ sta sempre sulla direzione di $(\vec{m}_2 - \vec{m}_1)$. Inoltre per adesso non ci interessa la grandezza di \vec{w} , ma solo la sua direzione. Quindi possiamo eliminare i fattori scalari $(\vec{w}^T S_B \vec{w})$ e $(\vec{w}^T S_W \vec{w})$. Quindi moltiplicando entrambi i membri per S_W^{-1} otteniamo che $\vec{w} \propto S_W^{-1}(\vec{m}_2 - \vec{m}_1)$.

N.B: se la covarianza totale all'interno della classe è isotropa in modo tale che S_W sia proporzionale alla matrice unitaria, allora si scopre che \vec{w} è proporzionale alla differenza delle medie della classe (come abbiamo appena dimostrato).

Tale risultato è noto come **discriminante lineare di Fisher** (anche se non si tratta di un discriminante vero e proprio, ma piuttosto di una scelta specifica della direzione per la proiezione dei dati in una dimensione). Tuttavia, i dati proiettati possono essere successivamente utilizzati per costruire un discriminante, scegliendo una soglia y_0 in modo tale da classificare un nuovo punto come appartenente a \mathcal{C}_∞ se $y(\vec{x})$ appartiene a \mathcal{C}_∞ oppure appartenente a \mathcal{C}_ϵ altrimenti.

Relazione con Least Squares

L'approccio ai minimi quadrati per la determinazione di un discriminante lineare si basava sull'apprendimento di funzioni lineari che si avvicinano il più possibile a un insieme di valori target. Il criterio di Fisher, invece, cerca di massimizzare la separazione delle classi nello spazio discriminante.

Potrebbe risultare interessante vedere la relazione tra questi due approcci. In particolare mostriamo che, per il problema delle due classi, il criterio di Fisher può essere ottenuto come caso particolare dei minimi quadrati.

Fino ad ora abbiamo considerato la codifica 1-di- K per i valori target. Tuttavia, se adottiamo uno schema di codifica del target leggermente diverso, allora la soluzione ai minimi quadrati per i pesi \vec{w} diventa equivalente alla soluzione di Fisher.

In particolare consideriamo i target per la classe \mathcal{C}_1 come $\frac{N}{N_1}$, dove N_1 rappresenta il numero di campioni della classe \mathcal{C}_1 ed N rappresenta il numero totale di campioni, mentre consideriamo i target per la classe \mathcal{C}_2 come $-\frac{N}{N_2}$, dove N_2 rappresenta il numero di campioni della classe \mathcal{C}_2 .

Formalmente, possiamo riscrivere queste assunzioni come:

$$t_n = \begin{cases} \frac{N}{N_1} & \text{se } \vec{x}_n \in \mathcal{C}_1 \\ -\frac{N}{N_2} & \text{se } \vec{x}_n \in \mathcal{C}_2 \end{cases}$$

N.B: questa assunzione del valore del target può essere interpretata come una stima del prior reciproco per una certa classe.

La funzione di errore della somma dei quadrati può essere scritta come:

$$E = \frac{1}{2} \sum_{n=1}^N (\vec{w}^T \vec{x}_n + w_0 - t_n)^2$$

Ponendo a zero le derivate di E rispetto a w_0 e \vec{w} , si ottengono rispettivamente:

$$\begin{aligned} \sum_{n=1}^N (\vec{w}^T \vec{x}_n + w_0 - t_n) &= 0 \\ \sum_{n=1}^N (\vec{w}^T \vec{x}_n + w_0 - t_n) \vec{x}_n &= 0 \end{aligned}$$

$$\text{N.B: } \sum_{n=1}^N t_n = N_1 \frac{N}{N_1} - N_2 \frac{N}{N_2} = 0.$$

Dalla prima equazione e considerando la precedente scelta dello schema di codifica del target t_n , otteniamo un'espressione per il bias della seguente forma:

$$w_0 = -\vec{w}^T \vec{m}$$

dove $\vec{m} = \frac{1}{N} \sum_{n=1}^N \vec{x}_n = \frac{1}{N} (N_1 \vec{m}_1 + N_2 \vec{m}_2)$ è la media del dataset.

Otteniamo quindi che:

$$w_0 = -\frac{\vec{w}^T (N_1 \vec{m}_1 + N_2 \vec{m}_2)}{N}$$

Invece dalla seconda equazione, facendo ancora riferimento alla stessa scelta di t_n , otteniamo:

$$\left(S_W + \frac{N_1 N_2}{N} S_B \right) \vec{w} = N(\vec{m}_1 - \vec{m}_2)$$

dove S_B ed S_W sono definiti come in precedenza.

Osserviamo che $S_B \vec{w}$ è sempre sulla stessa direzione di $(\vec{m}_2 - \vec{m}_1)$. Quindi possiamo dire che $\vec{w} \propto S_W^{-1} (\vec{m}_2 - \vec{m}_1)$

Pertanto il vettore dei pesi \vec{w} coincide con quello trovato dal criterio di Fisher. Inoltre, abbiamo trovato un'espressione per il valore di bias w_0 che mostra come un nuovo vettore di input \vec{x} dovrebbe essere classificato (alla classe \mathcal{C}_1 se $y(\vec{x}) = \vec{w}^T (\vec{x} - \vec{m}) > 0$ e alla classe \mathcal{C}_2 altrimenti).

4.2 Modelli Generativi Probabilistici

Passiamo adesso ad una visione probabilistica della classificazione lineare e mostriamo come i modelli con decision boundaries lineari derivano da semplici ipotesi sulla distribuzione dei dati. Per adesso adottiamo un approccio generativo in cui modelliamo le densità condizionali delle classi $p(\vec{x} | \mathcal{C}_k)$ (cioé le likelihood) ed i prior delle classi $p(\mathcal{C}_k)$ per calcolare le probabilità posteriori $p(\mathcal{C}_k | \vec{x})$ attraverso il teorema di Bayes.

Consideriamo innanzitutto il caso con $K = 2$ classi.

La probabilità posteriori per la classe \mathcal{C}_1 può essere scritta come:

$$p(\mathcal{C}_1 | \vec{x}) = \frac{p(\vec{x} | \mathcal{C}_1)p(\mathcal{C}_1)}{p(\vec{x} | \mathcal{C}_1)p(\mathcal{C}_1) + p(\vec{x} | \mathcal{C}_2)p(\mathcal{C}_2)} = \frac{1}{1 + \exp(-a)} \equiv \sigma(a(\vec{x}))$$

dove definiamo $a = \ln \frac{p(\vec{x} | \mathcal{C}_1)p(\mathcal{C}_1)}{p(\vec{x} | \mathcal{C}_2)p(\mathcal{C}_2)}$ e $\sigma(a) = \frac{1}{1 + \exp(-a)}$ detta funzione *sigmoide logistica (logistic sigmoid)* che possiamo rappresentare con il seguente grafico:

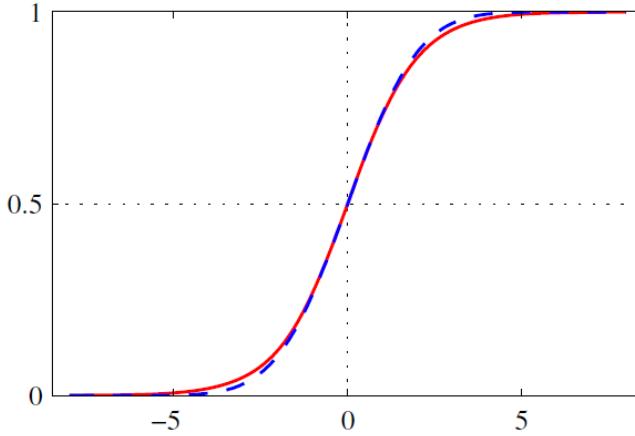


Figura 4.7: Grafico della funzione sigmoide logistica $\sigma(a)$ (in rosso) e della funzione probit scalare $\phi(\lambda a)$, per $\lambda^2 = \frac{\pi}{8}$ (tratteggiata in blu). Il fattore di scala $\frac{\pi}{8}$ è scelto in modo tale che le derivate delle due curve siano uguali per $a = 0$.

Tale funzione gioca un ruolo importante in molti modelli di classificazione. Il termine sigmoide significa "a forma di S". Questo tipo di funzione è anche chiamata anche funzione di schiacciamento (*squashing function*), perché mappa l'intero asse reale $(-\infty, +\infty)$ in un intervallo finito $[0, 1]$. L'inverso della sigmoide logistica è dato da $a = \ln \frac{\sigma}{1-\sigma}$, detta funzione *logit*, e come abbiamo visto rappresenta il logaritmo del rapporto delle probabilità per le due classi (ovvero $a = \ln \frac{p(\vec{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\vec{x}|\mathcal{C}_2)p(\mathcal{C}_2)}$).

N.B.: notiamo che abbiamo semplicemente riscritto le probabilità posteriori in una forma equivalente, e quindi la comparsa della funzione sigmoide logistica può sembrare piuttosto vacua (cioè riscrivere i posteriori in questo modo potrebbe sembrare una perdita di tempo). Tuttavia vedremo che questo aiuta a generalizzare i risultati quando $a(\vec{x})$ assume una forma semplice.

Per il caso multiclass con $K > 2$ abbiamo:

$$p(\mathcal{C}_k | \vec{x}) = \frac{p(\vec{x} | \mathcal{C}_k)p(\mathcal{C}_k)}{\sum_j p(\vec{x} | \mathcal{C}_j)p(\mathcal{C}_j)} = \frac{\exp(a_k)}{\sum_j \exp(a_j)}$$

Tale relazione può essere considerata una generalizzazione multiclass della sigmoide logistica ed è nota come *esponenziale normalizzata* (**normalized exponential**) o **softmax** (poiché rappresenta una versione attenuata della funzione max, infatti se $a_k \gg a_j$ per ogni $j \neq k$ allora $p(\mathcal{C}_k | \vec{x}) \simeq 1$ ed $p(\mathcal{C}_j | \vec{x}) \simeq 0$).

In questo caso le quantità a_k sono definite da $a_k = \ln p(\vec{x} | \mathcal{C}_k)p(\mathcal{C}_k)$.

4.2.1 Modelli generativi con input continui

Ora analizziamo le conseguenze della scelta di forme specifiche per le densità delle classi (ovvero le likelihood), esaminando le variabili di input continue \vec{x} .

Assumiamo che le densità condizionali delle classi siano Gaussiane e che condividano la stessa matrice di covarianza (ovvero abbiano tutti uguali matrice di covarianza Σ). Quindi esploriamo la forma risultante per le probabilità posteriori. In particolare la densità per la classe \mathcal{C}_k è data da:

$$p(\vec{x} | \mathcal{C}_k) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma|^{1/2}} \exp\left\{-\frac{1}{2}(\vec{x} - \vec{\mu}_k)^T \Sigma^{-1} (\vec{x} - \vec{\mu}_k)\right\}$$

Consideriamo innanzitutto il caso di due classi. Dalle precedenti analisi che abbiamo effettuato sulla forma del posterior, si ha:

$$p(\mathcal{C}_1 | \vec{x}) = \sigma(\vec{w}^T \vec{x} + w_0)$$

dove abbiamo definito:

$$\begin{aligned}\vec{w} &= \Sigma^{-1}(\vec{\mu}_1 - \vec{\mu}_2) \\ w_0 &= -\frac{1}{2}\vec{\mu}_1^T \Sigma^{-1} \vec{\mu}_1 + \frac{1}{2}\vec{\mu}_2^T \Sigma^{-1} \vec{\mu}_2 + \ln \frac{p(\mathcal{C}_1)}{p(\mathcal{C}_2)}\end{aligned}$$

N.B.: si nota che i termini quadratici in \vec{x} derivanti dagli esponenti delle densità Gaussiane si sono annullati (causa di avere assunto matrici di covarianza Σ comuni) portando a una funzione lineare di \vec{x} nell'argomento della sigmoide logistica.

N.B.: logicamente avremo $p(\mathcal{C}_2 | \vec{x}) = 1 - p(\mathcal{C}_1 | \vec{x})$.

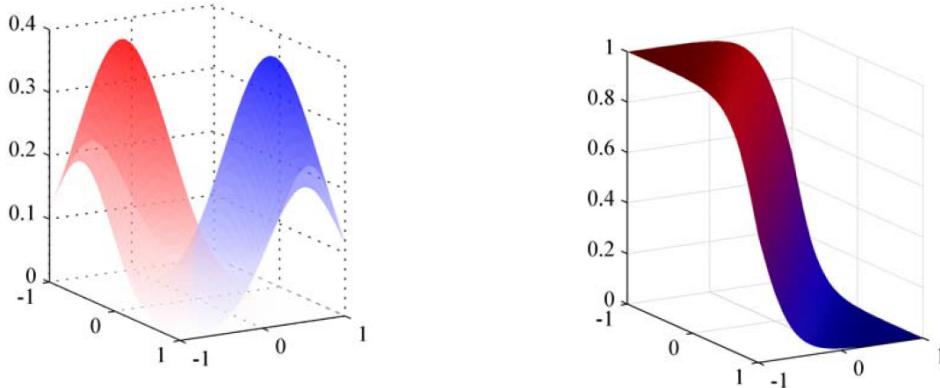


Figura 4.8: Il grafico di sinistra mostra le densità condizionali per due classi (rappresentate in rosso per la classe \mathcal{C}_1 e in blu per la classe \mathcal{C}_2).

Nel grafico di destra viene riportata la corrispondente probabilità posterior $p(\mathcal{C}_1 | \vec{x})$ data da una sigmoide logistica di una funzione lineare di \vec{x} . In questo caso la superficie è stata colorata utilizzando una proporzione di inchiostro rosso data da $p(\mathcal{C}_1 | \vec{x})$ e una proporzione di inchiostro blu data da $p(\mathcal{C}_2 | \vec{x}) = 1 - p(\mathcal{C}_1 | \vec{x})$.

Osserviamo che i confini decisionali risultanti corrispondono a superfici lungo le quali le probabilità posterior $p(\mathcal{C}_k | \vec{x})$ risultano costanti e quindi sono date da funzioni lineari di \vec{x} . Di conseguenza i confini decisionali sono lineari nello spazio degli input. Le probabilità prior $p(\mathcal{C}_k)$ entrano solo attraverso il parametro bias w_0 in modo tale che i cambiamenti nei prior comportino l'effetto di spostamento parallelo del confine decisionale o, più in generale, dei contorni paralleli della probabilità posterior costante.

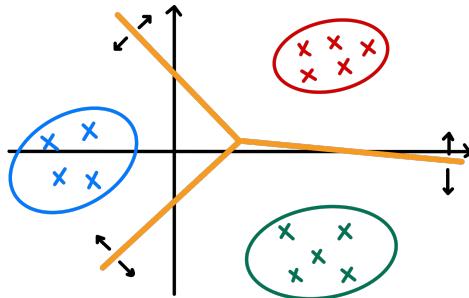
Per il caso generale multiclass con $K > 2$ classi abbiamo:

$$a_k(\vec{x}) = \vec{w}_k^T \vec{x} + w_{k0}$$

dove abbiamo definito:

$$\begin{aligned}\vec{w}_k &= \Sigma^{-1} \vec{\mu}_k \\ w_{k0} &= -\frac{1}{2} \vec{\mu}_k^T \Sigma^{-1} \vec{\mu}_k + \ln p(\mathcal{C}_k)\end{aligned}$$

Vediamo che gli $a_k(\vec{x})$ sono di nuovo funzioni lineari di \vec{x} come conseguenza dell'annullamento dei termini quadratici dovuti alle covarianze condivise. I confini decisionali risultanti, corrispondenti al minimo tasso di misclassificazione, si verificheranno quando due delle probabilità posteriori (le due più grandi) risultano essere uguali (poiché saranno definite da funzioni lineari di \vec{x}). Quindi ancora una volta abbiamo un modello lineare generalizzato.



Se rilassiamo l'ipotesi che le densità condizionali delle classi abbiano una matrice di covarianza condivisa e, invece, permettiamo che ogni densità condizionale $p(\vec{x} | \mathcal{C}_k)$ abbia una propria matrice di covarianza Σ_k distinta, allora le precedenti cancellazioni del termine quadratico non si verificheranno più e quindi otterremo funzioni quadratiche di \vec{x} (e non più lineari) dando luogo a un discriminante quadratico. Il risultato è un classificatore di Bayes quadratico.

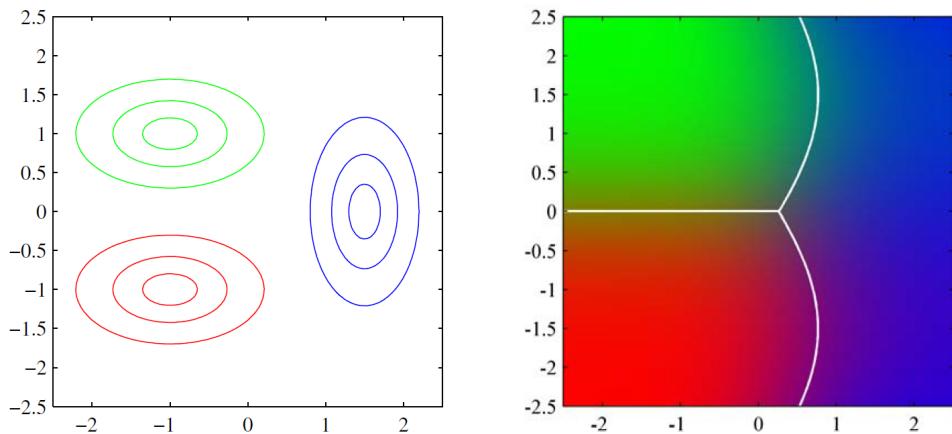


Figura 4.9: Il grafico di sinistra mostra le densità condizionali delle classi per tre classi, ciascuna con distribuzione gaussiana (colorate in rosso, verde e blu), in cui le classi rosse e verdi possiedono la stessa matrice di covarianza.

Il grafico di destra mostra le corrispondenti probabilità posteriori (rappresentate da vettori di colori RGB) per le rispettive tre classi. Sono mostrati anche i confini decisionali. Si noti che il confine tra le classi rossa e verde che hanno la stessa matrice di covarianza, è lineare, mentre tra le altre coppie di classi è quadratico.

4.3 Modelli Discriminativi Probabilistici

Per il problema della classificazione a due classi, abbiamo visto che la probabilità posteriore della classe \mathcal{C}_1 può essere scritta come una sigmoide logistica che agisce su una funzione lineare di \vec{x} .

Analogamente, per il caso multiclass, la probabilità posteriore della classe \mathcal{C}_k può essere ottenuta da una trasformazione softmax di una funzione lineare di \vec{x} .

In particolare, per specifiche scelte delle distribuzioni condizionali delle classi $p(\vec{x} | \mathcal{C}_k)$ possiamo usare la massima verosimiglianza per determinare i parametri delle densità e i prior $p(\mathcal{C}_k)$ e poi applicare il teorema di Bayes per trovare le probabilità posteriori delle classi.

Questi modelli vengono chiamati modelli generativi perché permettono di generare campioni \vec{x} estraendo i valori dalla distribuzione marginale $p(\vec{x})$.

Tuttavia, un approccio alternativo è quello di utilizzare la forma funzionale del modello lineare generalizzato in modo esplicito e determinare i suoi parametri direttamente con la massima verosimiglianza.

In questo approccio diretto stiamo massimizzando una funzione di verosimiglianza definita attraverso la distribuzione condizionale $p(\mathcal{C}_k | \vec{x})$ che rappresenta una forma di training discriminativa. Un vantaggio dell'approccio discriminativo è che in genere i parametri di adattamento da determinare sono meno numerosi. Questo può portare un miglioramento delle prestazioni predittive, in particolare quando le ipotesi di densità condizionale della classe forniscono una scarsa approssimazione alle distribuzioni reali.

Funzioni base fisse

Finora abbiamo considerato modelli di classificazione che operano direttamente con il vettore di input \vec{x} originale. Tuttavia, tutti gli algoritmi analizzati sono ugualmente applicabili se prima si esegue una trasformazione non lineare fissa degli input usando un vettore di funzioni base $\phi(\vec{x})$. I confini decisionali risultanti saranno lineari nello spazio delle feature ϕ e corrispondono a confini decisionali non lineari nello spazio \vec{x} originale. Perciò le classi che sono linearmente separabili nello spazio delle feature $\phi(\vec{x})$ non devono necessariamente essere linearmente separabili nello spazio di osservazione originale \vec{x} .

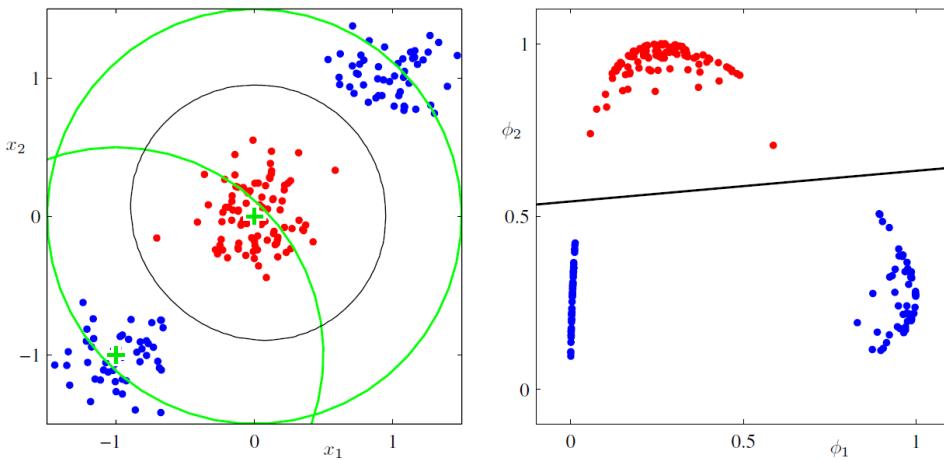


Figura 4.10: Illustrazione del ruolo delle funzioni base non lineari $\phi(\vec{x})$ nei modelli di classificazione lineare.

Il grafico di sinistra mostra lo spazio di input originale (\vec{x}_1, \vec{x}_2) insieme ai punti dati di due classi etichettate come rosso e blu. In questo spazio vengono definite due funzioni base Gaussiane $\phi_1(\vec{x})$ e $\phi_2(\vec{x})$ i cui centri sono indicati dalle croci verdi e i contorni dai cerchi verdi.

Il grafico di destra mostra il corrispondente spazio delle feature (ϕ_1, ϕ_2) insieme al confine decisionale lineare (in nero) ottenuto da un modello di regressione logistica. Questo corrisponde a un confine decisionale non lineare nello spazio di input originale (rappresentato dalla curva nera nel grafico di sinistra).

4.3.1 Regressione logistica

Considerando il problema della classificazione a due classi ($K = 2$). Nella discussione degli approcci generativi abbiamo visto che, sotto ipotesi piuttosto generali, la probabilità posterior della classe \mathcal{C}_1 può essere scritta come una sigmoide logistica che agisce su una funzione lineare del vettore di feature ϕ in modo che $p(\mathcal{C}_1 | \phi) = \sigma(\vec{w}^T \phi)$. Tale modello è noto come *regressione logistica* (**logistic regression**), anche se dobbiamo sottolineare che si tratta di un modello di classificazione piuttosto che di regressione.

Per uno spazio di feature ϕ di dimensione M questo modello ha M parametri regolabili. **N.B:** al contrario, se avessimo adattato le densità condizionali Gaussiane delle classi usando la massima verosimiglianza avremmo usato $2M$ parametri per le medie e $\frac{M(M+1)}{2}$ parametri per la matrice di covarianza condivisa. Quindi, insieme al prior della classe $p(\mathcal{C}_1)$, si ottiene un totale di $\frac{M(M+5)}{2} + 1$ parametri, che cresce quadraticamente con M .

Utilizziamo ora la massima verosimiglianza per determinare i parametri del modello di regressione logistica. A tal fine utilizzeremo la derivata della funzione sigmoide logistica, che può essere comodamente espressa in termini della funzione sigmoide stessa, ovvero:

$$\frac{d}{da} \sigma(a) = \sigma(1 - \sigma)$$

Consideriamo un dataset $\mathcal{D} = \{\phi_n, t_n\}$ dove $t_n \in \{0, 1\}$ rappresenta il target e $\phi_n = \phi(\vec{x}_n)$ con $n = 1, \dots, N$. Allora la likelihood può essere scritta come:

$$p(\vec{t} | \vec{w}) = \prod_{n=1}^N y_n^{t_n} \{1 - y_n\}^{1-t_n}$$

dove $\vec{t} = (t_1, \dots, t_N)^T$ ed $y_n = p(\mathcal{C}_1 | \phi_n) = \sigma(\vec{w}^T \phi_n)$.

Possiamo definire una funzione di errore considerando il logaritmo negativo della likelihood (tale funzione di errore è detta *crossentropy*). In particolare abbiamo:

$$E(\vec{w}) = -\ln p(\vec{t} | \vec{w}) = -\sum_{n=1}^N \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\}$$

dove $y_n = \sigma(\vec{w}^T \phi_n)$.

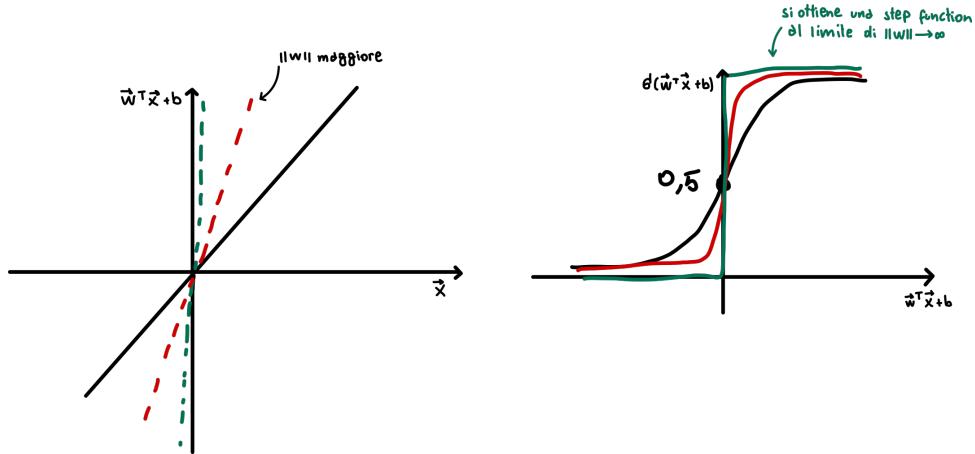
In particolare, utilizzando la forma della derivata della funzione sigmoide logistica osservata (ovvero $\frac{d}{da} \sigma(a) = \sigma(1 - \sigma)$), possiamo calcolare il gradiente della funzione di errore rispetto a \vec{w} ottenendo:

$$\nabla_{\vec{w}} E(\vec{w}) = \sum_{n=1}^N (y_n - t_n) \phi_n$$

Vediamo che il contributo al gradiente del punto dati n è dato dall'errore $y_n - t_n$ tra il valore target e la previsione del modello, moltiplicato per il vettore della funzione base ϕ_n . Inoltre osserviamo che il risultato ottenuto assume esattamente la stessa forma del gradiente della funzione di errore della somma dei quadrati per il modello di regressione lineare (per questo motivo tale modello è detto regressione logistica anche se è un modello di classificazione).

N.B.: questa soluzione di massima verosimiglianza è altamente incline all'overfitting quando \mathcal{C}_1 e \mathcal{C}_2 sono linearmente separabili.

Questo perché la soluzione di massima verosimiglianza si verifica quando l'iperpiano corrispondente a $\sigma = 0.5$ (equivalente a $\vec{w}^T \phi = 0$) separa le due classi e la grandezza di \vec{w} (ovvero $\|\vec{w}\|_2$) tende all'infinito. In questo caso, la funzione sigmoide logistica diventa infinitamente ripida nello spazio delle feature corrispondendo a una funzione a gradini tale che ad ogni punto di training di ciascuna classe k venga assegnata una probabilità posterior $p(\mathcal{C}_k | \vec{x}) = 1$.



4.3.2 Regressione logistica Bayesiana

Passiamo ora a una trattazione Bayesiana della regressione logistica.

In precedenza, durante l'analisi dei modelli di regressione, abbiamo sviluppato un approccio per l'apprendimento Bayesiano completo.

Proviamo ad applicarlo al problema di classificazione. In particolare dobbiamo:

1. Decidere un prior $p(\vec{w})$.
2. Massimizzare il posterior risultante per ottenere una distribuzione dei parametri $p(\vec{w} | \vec{t})$.
3. Derivare la distribuzione predittiva $p(\mathcal{C}_k | \vec{\Phi}, \vec{t})$ da utilizzare su nuovi dati $\phi(\vec{x})$.

Osserviamo che il passo 1 risulta essere pittosto facile, anche se è una delle critiche principali all'apprendimento Bayesiano. Mentre i passi 2 e 3 sono purtroppo intrattabili poiché la distribuzione posterior dei parametri non è più Gaussiana.

Per ovviare a tale problema consideriamo l'applicazione dell'approssimazione di Laplace.

Approssimazione di Laplace

L'approssimazione di Laplace si ottiene trovando la modalità della distribuzione del posterior e adattando una Gaussiana centrata su tale modalità. Ciò richiede la valutazione delle derivate seconde del logaritmo del posterior.

Innanzitutto assumiamo di considerare un prior Gaussiano, ovvero della forma $p(\vec{w}) = \mathcal{N}(\vec{w} | \vec{m}_0, S_0)$ dove \vec{m}_0 ed S_0 sono degli iperparametri fissati. Questo perché stiamo cercando di definire una rappresentazione Gaussiana per la distribuzione posterior.

Per ottenere un'approssimazione Gaussiana della distribuzione posterior, dobbiamo massimizzare la distribuzione posterior in modo da ricavare la soluzione **MAP** (*Maximum Posterior*) \vec{w}_{MAP} , che definisce la media della Gaussiana. La covarianza invece è data dall'inverso della matrice delle derivate seconde del logaritmo negativo della likelihood.

Tale approssimazione Gaussiana per la distribuzione posterior assume quindi la seguente forma:

$$q(\vec{w}) = \mathcal{N}(\vec{w} | \vec{w}_{\text{MAP}}, S_N)$$

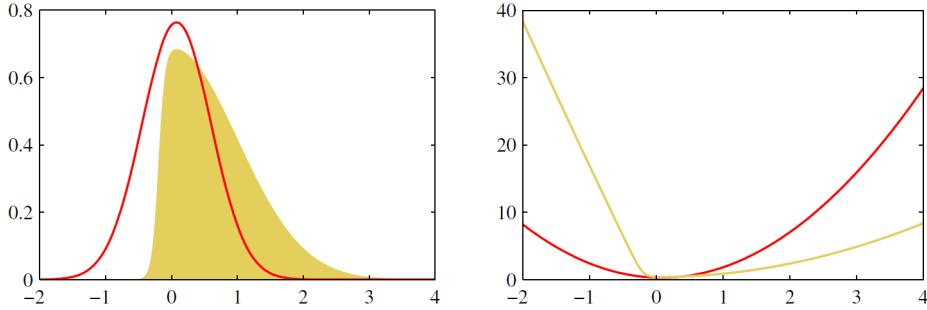


Figura 4.11: Illustrazione dell'approssimazione di Laplace applicata alla distribuzione $p(z) \propto \exp(-\frac{z^2}{2})\sigma(20z + 4)$, dove $\sigma(z)$ è la funzione sigmoide logistica definita da $\sigma(z) = (1 + e^{-z})^{-1}$. Il grafico di sinistra mostra la distribuzione normalizzata $p(z)$ (in giallo), insieme all'approssimazione di Laplace centrata sulla modalità z_0 di $p(z)$ (in rosso). Il grafico di destra mostra i logaritmi negativi delle curve corrispondenti.

Avendo ottenuto un'approssimazione Gaussiana alla distribuzione posterior, rimane il compito di marginalizzare rispetto a questa distribuzione per poter fare delle previsioni.

Distribuzione predittiva

Una volta definita l'approssimazione di Laplace allora possiamo scrivere la distribuzione predittiva (approssimata). In particolare, dato un nuovo vettore di feature $\phi(\vec{x})$, la distribuzione predittiva per la classe \mathcal{C}_1 si ottiene marginalizzando rispetto alla distribuzione posterior $p(\vec{w} | \vec{t})$ a sua volta approssimata da una distribuzione Gaussiana $q(\vec{w})$, ovvero:

$$p(\mathcal{C}_1 | \phi, \vec{t}) = \int p(\mathcal{C}_1 | \phi, \vec{w}) p(\vec{w} | \vec{t}) d\vec{w} \approx \int \sigma(\vec{w}^T \phi) q(\vec{w}) d\vec{w}$$

Tale relazione si tratta della convoluzione tra una sigmoide logistica ed una Gaussiana, che può essere approssimata (dopo una lunga derivazione) come:

$$p(\mathcal{C}_1 | \phi, \vec{t}) \approx \sigma(k(\sigma_a^2)\mu_a)$$

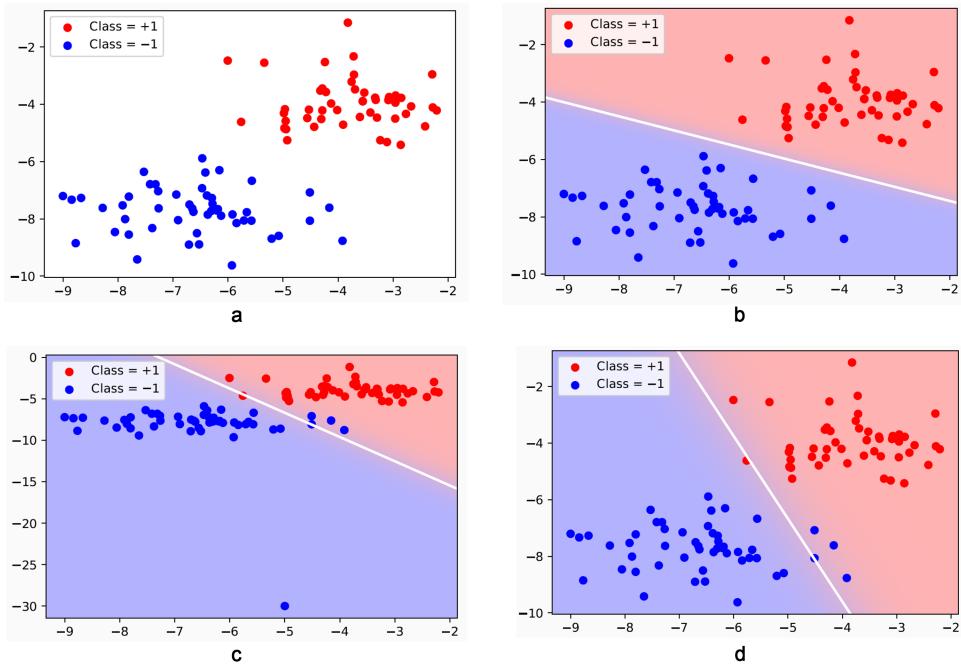
dove $k(\sigma^2) = (1 + \frac{\pi\sigma_a^2}{8})^{-1/2}$, $\sigma_a^2 = \text{Var}[a] = \phi^T S_N \phi$ ed $\mu_a = \vec{w}_{\text{MAP}}^T \phi$.

N.B.: la corrispondente probabilità per la classe \mathcal{C}_2 sarà data da $p(\mathcal{C}_2 | \phi, \vec{t}) = 1 - p(\mathcal{C}_1 | \phi, \vec{t})$.

Capitolo 5

KERNEL MACHINES

Consideriamo un semplice problema di classificazione linearmente separabile (a). Abbiamo analizzato degli strumenti adatti per questi problemi, ad esempio un discriminante lineare generativo (b). Un problema di molti approcci probabilistici però è la sensibilità agli outlier, poiché questi spostano di molto la media della classe modificando l'iperpiano (c). Perciò, i metodi che trattano tutti i campioni allo stesso modo (ovvero assegnano lo stesso peso a tutti i campioni) possono degradarsi rapidamente (d). Vogliamo quindi riformulare un problema di classificazione analizzando il margine.



5.1 Maximum Margin Classifiers

Un modo utile per pensare ad un problema di classificazione è quello di:

1. Rappresentare i dati in \mathbb{R}^D .
2. Partizionare \mathbb{R}^D in modo tale che solo i campioni con la stessa etichetta rientrino nella stessa partizione.

Consideriamo una partizione conveniente, ovvero quella che separa lo spazio \mathbb{R}^D a metà utilizzando un iperpiano di separazione H . A tale scopo consideriamo una funzione $f : \mathbb{R}^D \rightarrow \mathbb{R}$ definita come:

$$f(\vec{x}; \vec{w}, b) = \langle \vec{w}, \vec{x} \rangle + b = \vec{w}^T \vec{x} + b$$

N.B: ricordiamo che $\langle \vec{x}, \vec{y} \rangle$ rappresenta il prodotto interno tra i vettori \vec{x} ed \vec{y} .

Quindi definiamo l'iperpiano di separazione H , che suddivide lo spazio utilizzando f , come:

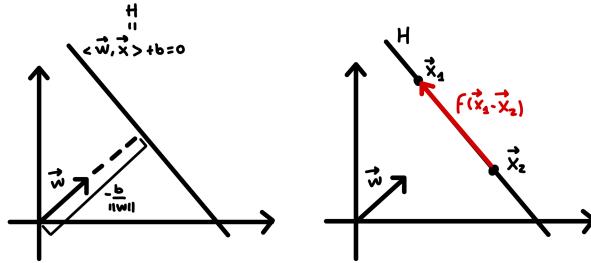
$$H = \{\vec{x} \mid f(\vec{x}; \vec{w}, b) = \langle \vec{w}, \vec{x} \rangle + b = 0\}$$

L'iperpiano definito da \vec{w} e b è perpendicolare a \vec{w} .

Per verificare tale affermazione scegliamo un qualsiasi \vec{x}_1 ed \vec{x}_2 sull'iperpiano H e consideriamo:

$$f(\vec{x}_1) - f(\vec{x}_2) = \langle \vec{w}, \vec{x}_1 \rangle + b - \langle \vec{w}, \vec{x}_2 \rangle - b = \langle \vec{w}, \vec{x}_1 - \vec{x}_2 \rangle$$

In particolare osserviamo che se $f(\vec{x}_1) = 0$ ed $f(\vec{x}_2) = 0$, allora $\langle \vec{w}, \vec{x}_1 - \vec{x}_2 \rangle = 0$ e questo implica che $(\vec{x}_1 - \vec{x}_2) \perp \vec{w}$.



Quindi considerando un dataset di training che comprende N vettori di input $\vec{x}_1, \dots, \vec{x}_N$, con i corrispondenti valori target t_1, \dots, t_N dove $t_i \in \{-1, 1\}$, allora un nuovo punto \vec{x} viene classificato in base al segno che assume $y(\vec{x})$. In particolare, quando ci viene presentato un nuovo campione \vec{x} , questo viene classificato in base alla porzione di piano (suddiviso dall'iperpiano H) in cui tale campione si trova, ovvero:

$$\text{class}(\vec{x}) = \begin{cases} +1 & \text{se } f(\vec{x}; \vec{w}, b) \geq 0 \\ -1 & \text{se } f(\vec{x}; \vec{w}, b) < 0 \end{cases}$$

Perciò, quando si addestra il modello sui dati $\{(\vec{x}_i, y_i) \mid i = 1, \dots, N\}$ si cercano \vec{w} e b in modo tale da posizionare tutti i campioni sul lato corretto dell'iperpiano, ovvero:

$$\begin{aligned} \langle \vec{w}, \vec{x}_i \rangle + b &\geq 0 && \text{quando } y_i = +1 \\ \langle \vec{w}, \vec{x}_i \rangle + b &< 0 && \text{quando } y_i = -1 \end{aligned}$$

N.B: queste condizioni possono essere combinate nella forma più compatta $y_i(\langle \vec{w}, \vec{x}_i \rangle + b) \geq 0$.

Possiamo definire il **margine** come la distanza tra un iperpiano di separazione e il punto più vicino ad esso. Ovvero il margine rappresenta la distanza perpendicolare tra il confine decisionale e i campioni più vicini.

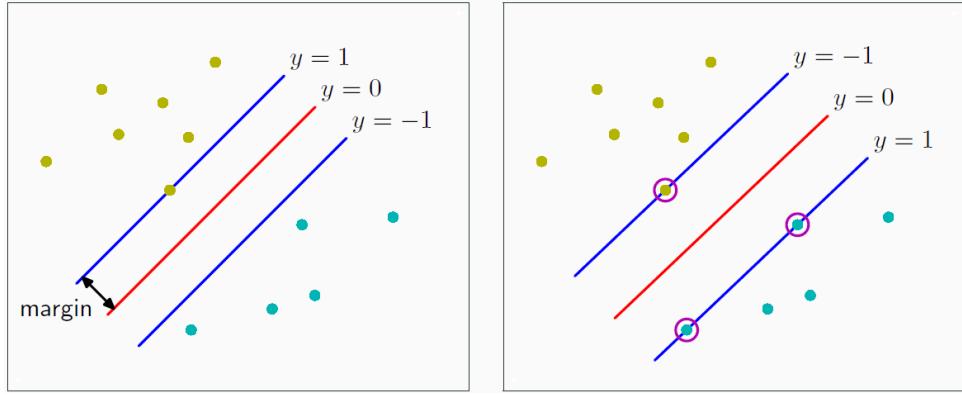


Figura 5.1: Il margine è definito come la distanza perpendicolare tra il confine decisionale e i punti dati più vicini, come mostrato nella figura a sinistra. La massimizzazione del margine porta a una particolare scelta del confine decisionale, come mostrato nella figura di destra. La posizione di questo confine decisionale è determinata da un sottoinsieme di punti dati, noti come *vettori di supporto* (**support vectors**), indicati dai cerchi.

L’obiettivo è quello di massimizzare tale distanza. Vediamo come ottenere tale massimizzazione.

La distanza perpendicolare di un punto \vec{x} da un iperpiano definito da $y(\vec{x}) = \langle \vec{w}, \vec{x} \rangle + b = 0$ è caratterizzata da:

$$\frac{y_n(\langle \vec{w}, \vec{x}_n \rangle + b)}{\|\vec{w}\|}$$

Vogliamo quindi ottimizzare i parametri \vec{w} e b per massimizzare questa distanza (vogliamo massimizzare la distanza minima). La soluzione con il massimo margine si trova quindi risolvendo:

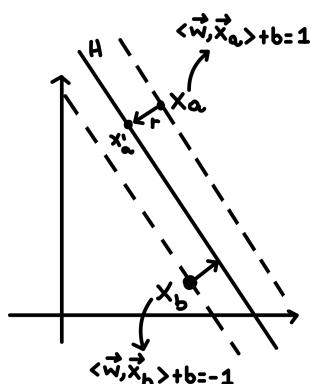
$$\arg \max_{\vec{w}, b} \left\{ \frac{1}{\|\vec{w}\|} \min_n [y_n(\langle \vec{w}, \vec{x}_n \rangle + b)] \right\}$$

soggetto al vincolo $y_n(\langle \vec{w}, \vec{x}_n \rangle + b) \geq 0$.

La soluzione diretta di questo problema di ottimizzazione sarebbe molto complessa, per cui lo convertiamo in un problema equivalente di risoluzione più facile.

N.B.: per fare questo possiamo scalare liberamente $\vec{w} \rightarrow \lambda \vec{w}$ e $b \rightarrow \lambda b$ senza modificare la distanza tra i punti e l’iperpiano. In questo modo possiamo scalare $y_n(\langle \vec{w}, \vec{x}_n \rangle + b) = 1$ per il punto \vec{x}_n più vicino all’iperpiano.

Sia r la distanza ortogonale da \vec{x}_n all’iperpiano. Allora la proiezione ortogonale di \vec{x}_n sull’iperpiano è data da $\vec{x}'_n = \vec{x}_n - r \frac{\vec{w}}{\|\vec{w}\|}$.



Siccome x'_a giace sull'iperpiano per costruzione, allora possiamo sostituire l'ultima relazione nell'equazione dell'iperpiano. Ovvero abbiamo che:

$$\left\langle \vec{w}, \vec{x}_a - r \frac{\vec{w}}{\|\vec{w}\|} \right\rangle + b = 0$$

Da cui, sfruttando la bilinearità del prodotto interno, otteniamo:

$$\langle \vec{w}, \vec{x}_a \rangle + b - r \frac{\langle \vec{w}, \vec{w} \rangle}{\|\vec{w}\|} = 0$$

Ricordando che \vec{x}_a appartiene al margine per ipotesi (ovvero \vec{x}_a giace sul margine), possiamo concludere che:

$$r = \frac{1}{\|\vec{w}\|}$$

Da cui otteniamo la massimizzazione del margine:

$$\max_{\vec{w}, b} \frac{1}{\|\vec{w}\|}$$

soggetto al vincolo $y_n(\langle \vec{w}, \vec{x}_n \rangle + b) \geq 1 \forall n = 1, \dots, N$ (detta rappresentazione canonica dell'iperpiano).

Per definizione, ci sarà sempre almeno un vincolo attivo dato dal fatto che esisterà sempre un punto vicino all'iperpiano. In particolare, una volta massimizzato il margine ci saranno almeno due vincoli attivi (uno per ogni classe).

Quindi il problema di ottimizzazione richiede semplicemente di massimizzare $\|\vec{w}\|^{-1}$ che equivale a minimizzare $\|\vec{w}\|^2$. Quindi dobbiamo risolvere il seguente problema di ottimizzazione (detto rappresentazione canonica **Hard Margin SVM**):

$$\min_{\vec{w}, b} \frac{1}{2} \|\vec{w}\|^2$$

soggetto ancora al vincolo $y_n(\langle \vec{w}, \vec{x}_n \rangle + b) \geq 1 \forall n = 1, \dots, N$.

N.B.: il fattore $\frac{1}{2}$ è stato aggiunto per convenienza.

Questo è un esempio di un problema di **programmazione quadratica** in cui si cerca di minimizzare una funzione quadratica soggetta a una serie di vincoli di diseguaglianza lineari.

N.B.: sembra che il parametro bias b sia scomparso. Tuttavia, esso è determinato implicitamente attraverso i vincoli, perché questi richiedono che le variazioni di \vec{w} siano compensate dalle variazioni di b .

Per risolvere questo problema di ottimizzazione vincolata, introduciamo dei moltiplicatori di Lagrange $a_n \geq 0$ (un moltiplicatore a_n per ciascun vincolo) che restituiscono la seguente funzione **Lagrangiana**:

$$L(\vec{w}, b, \vec{a}) = \frac{1}{2} \|\vec{w}\|^2 - \sum_{n=1}^N a_n \{y_n(\langle \vec{w}, \vec{x}_n \rangle + b) - 1\}$$

dove $\vec{a} = (a_1, \dots, a_N)^T$ è il vettore dei moltiplicatori di Lagrange.

Ponendo le derivate di $L(\vec{w}, b, \vec{a})$ rispetto a \vec{w} e b uguali a zero (ovvero $\frac{\partial}{\partial \vec{w}} L = 0$ ed $\frac{\partial}{\partial b} L = 0$), si ottengono le seguenti condizioni:

$$\begin{aligned}\vec{w} &= \sum_{n=1}^N a_n y_n \vec{x}_n \\ 0 &= \sum_{n=1}^N a_n y_n\end{aligned}$$

Sostituendo le due condizioni nella funzione Lagrangiana otteniamo la seguente massimizzazione:

$$\max_{\vec{a}} \left\{ \sum_{n=1}^N a_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N a_n a_m y_n y_m \langle \vec{x}_n, \vec{x}_m \rangle \right\}$$

soggetto ai vincoli $a_n \geq 0 \forall n = 1, \dots, N$ ed $\sum_{n=1}^N a_n y_n = 0$.

Tale soluzione esprime la rappresentazione duale del problema del massimo margine (ovvero rappresentazione duale di Hard Margin SVM) che descrive un nuovo problema di programmazione quadratica ma in N variabili.

La soluzione di un problema di programmazione quadratica in M variabili in generale ha una complessità computazionale pari a $O(M^3)$. Passando alla formulazione duale abbiamo trasformato il problema di ottimizzazione originale (che prevedeva la minimizzazione su M variabili) nel problema duale (che ha N variabili).

Per un insieme fisso di M funzioni base, il cui numero M è inferiore al numero N di punti dati, il passaggio al problema duale appare svantaggioso. Tuttavia, consente di riformulare il modello utilizzando i kernel e quindi il classificatore a margine massimo può essere applicato in modo efficiente a spazi di features la cui dimensionalità supera il numero di punti dati.

Per classificare i nuovi punti di dati utilizzando il modello addestrato, possiamo valutare il segno della funzione di decisione $f(\vec{x}; \vec{w}, b) = \vec{w}^T \vec{x} + b$. In particolare possiamo esprimere tale funzione in termini dei parametri $\{a_n\}$ sostituendo \vec{w} nella funzione di decisione, ottenendo:

$$f(\vec{x}) = \sum_{n=1}^N a_n y_n \langle \vec{x}, \vec{x}_n \rangle + b$$

In particolare un problema di ottimizzazione vincolata di questa forma soddisfa le seguenti tre condizioni di *Karush-Kuhn-Tucker (KKT)*:

$$\begin{aligned}a_n &\geq 0 \\ y_n f(\vec{x}_n) - 1 &\geq 0 \\ a_n \{y_n f(\vec{x}_n) - 1\} &= 0\end{aligned}$$

Di conseguenza per ogni punto dati \vec{x}_n abbiamo che $a_n = 0$ oppure $y_n f(\vec{x}_n) = 1$.

In particolare, ogni punto dati per cui $a_n = 0$ non comparirà nella sommatoria di $f(\vec{x})$ e quindi non contribuisce alla soluzione (cioè non ha alcun ruolo nel fare previsioni di nuovi punti dati).

I rimanenti punti dati \vec{x}_n per i quali $a_n > 0$ e $y_n f(\vec{x}_n) = 1$ sono detti **vettori di supporto**. In particolare tali punti \vec{x}_n per cui $a_n > 0$, poiché soddisfano $y_n f(\vec{x}_n) = 1$, corrispondono ai punti che si trovano sugli iperpiani del massimo margine nello spazio delle features.

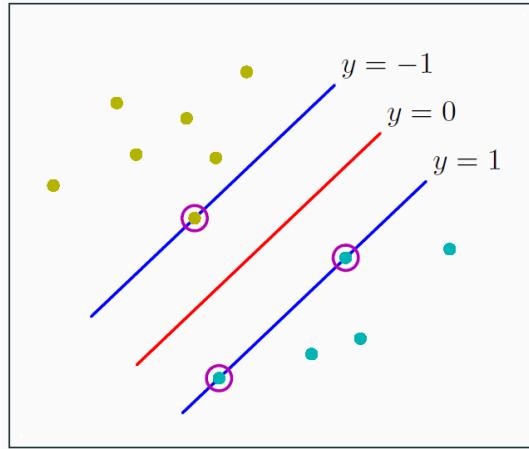


Figura 5.2: I support vectors sono i punti dati che si trovano sugli iperpiani del massimo margine (cioè sono i punti dati cerchiati).

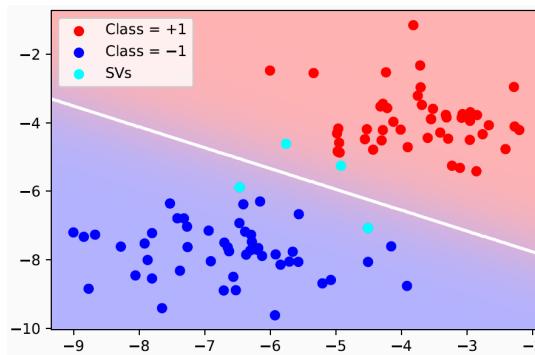
Osserviamo analiticamente che solo i support vectors (**SV**) contribuiscono alla classificazione. Infatti, poiché per i punti dati che non sono support vectors abbiamo $a_n = 0$, allora otteniamo:

$$f(\vec{x}) = \sum_{n=1}^N a_n y_n \langle \vec{x}, \vec{x}_n \rangle + b = \sum_{m \in \text{SV}} a_m y_m \langle \vec{x}, \vec{x}_m \rangle + b$$

Questa proprietà è fondamentale per l'applicabilità delle *Support Vector Machines (SVM)*.

In questo particolare caso le SVM sono anche più generalmente note come **Sparse Kernel Machines**.

Quindi una volta addestrato il modello, è possibile scartare una parte significativa dei punti dati e conservare solo i vettori di supporto. In questo modo otteniamo un classificatore lineare che risulta essere robusto agli outliers.



5.2 Soft Margin Classifier

Finora abbiamo ipotizzato che i punti dei dati di addestramento siano linearmente separabili (quindi che il problema di classificazione sia linearmente separabile) perciò la SVM risultante fornirà una separazione esatta dei dati di addestramento.

Però questa particolare situazione non si verifica quasi mai. Infatti spesso è possibile trovare problemi non linearmente separabili per i quali non è possibile definire un'iperpiano di separazione attraverso un discriminante lineare.

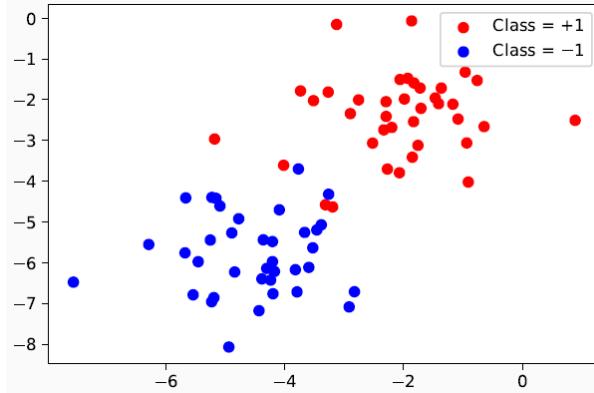


Figura 5.3: Esempio di un dataset non linearmente separabile.

Abbiamo quindi bisogno di un modo per modificare la SVM in modo da permettere che alcuni dei punti di addestramento vengano classificati in modo errato. Per fare questo introduciamo per ogni punto di addestramento \vec{x}_n delle variabili ξ_n (con $n = 1, \dots, N$), dette **varibili slack**, tali che:

$$\xi_n = \begin{cases} 0 & \text{se } \vec{x}_n \text{ si trova sul lato corretto del margine} \\ |y_n - \langle \vec{w}, \vec{x}_n \rangle + b| & \text{altrimenti} \end{cases}$$

N.B.: i punti con $\xi_n = 0$ sono classificati correttamente e si trovano sul margine oppure sul lato corretto del margine e quindi sul lato corretto del confine decisionale. Invece i punti con $0 < \xi_n \leq 1$ si trovano su lato corretto del confine decisionale ma sono localizzati all'interno del margine. Infine i punti con $\xi_n > 1$ si trovano sul lato sbagliato del confine decisionale e saranno classificati in modo errato.

N.B.: i punti che si trovano sul confine decisionale avranno $\xi_n = 1$.

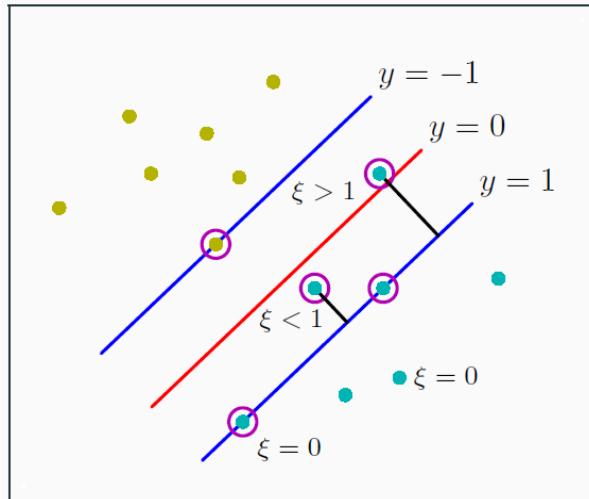


Figura 5.4: Illustrazione delle variabili slack $\xi_n \geq 0$. I punti cerchiati rappresentano i vettori di supporto.

Questo procedimento viene descritto come un rilassamento sul vincolo dell'Hard Margin SVM e definisce un **Soft Margin SVM** che consente la classificazione errata di alcuni punti del dataset di addestramento.

N.B.: questa soluzione permette la sovrapposizione delle distribuzioni delle classi (ovvero permette che le classi non siano linearmente separabili), però comporta una maggiore sensibilità agli outliers perché l'errore per questi valori aumenta linearmente con ξ .

Quindi il nostro obiettivo diventa quello di massimizzare il margine penalizzando in modo attenuato i punti che si trovano sul lato sbagliato del confine del margine. Perciò il problema di ottimizzazione sarà:

$$\min_{\vec{w}, b} \frac{1}{2} \|\vec{w}\|^2 + C \sum_{n=1}^N \xi_n$$

soggetto ai vincoli $y_n(\langle \vec{w}, \vec{x}_n \rangle + b) \geq 1 - \xi_n$ per ogni $n = 1, \dots, N$.

Dove il parametro $C > 0$ controlla il compromesso tra la penalità data dalla variabile slack ed il margine. Tale parametro è analogo al coefficiente di regolarizzazione (infatti quest'ultimo controlla il compromesso tra la minimizzazione degli errori e la complessità del modello).

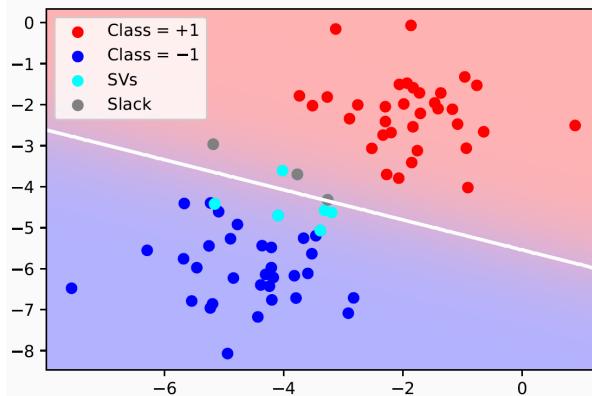
N.B.: se $C \rightarrow \infty$ otteniamo la precedente formulazione di SVM per dati linearmente separabili.

Come abbiamo fatto per Hard Margin SVM possiamo definire la rappresentazione duale di Soft Margin SVM:

$$\max_{\vec{a}} \left\{ \sum_{n=1}^N a_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N a_n a_m y_n y_m \langle \vec{x}_n, \vec{x}_m \rangle \right\}$$

soggetto ai vincoli $0 \leq a_n \leq C \ \forall n = 1, \dots, N$ ed $\sum_{n=1}^N a_n y_n = 0$.

N.B.: abbiamo una rappresentazione duale di Soft Margin SVM identica a quella di Hard Margin SVM tranne che per il primo vincolo (ovvero il vincolo sui possibili valori del moltiplicatore Lagrangiano a_n).



5.3 SVM per problemi non lineari

Vogliamo vedere se è possibile estendere le osservazioni effettuate per problemi con discriminanti lineari a problemi i cui confini di decisione risultano essere non lineari.

5.3.1 Modo alternativo di ottenere la SVM primale

Come per il caso linearmente separabile, possiamo riformulare la SVM per distribuzioni non linearmente separabili in termini di minimizzazione di una funzione di errore regolarizzata.

Quindi possiamo derivare la Support Vector Machine elaborando un rischio empirico invece di analizzare il margine. Per fare questo consideriamo la solita classe di ipotesi \mathcal{H} contenente funzioni discriminanti lineari del tipo $f(\vec{x}) = \langle \vec{w}, \vec{x} \rangle + b$.

Vogliamo determinare una funzione loss per \mathcal{H} e per il dataset

$$\mathcal{D} = \{(\vec{x}_n, y_n) \mid n = 1, \dots, N\}.$$

Ricordiamo che la funzione loss ideale sarebbe la **loss zero-uno**, la quale assegna il valore 1 nel caso in cui la predizione sia errata e assegna il valore 0 altrimenti. Ovvero:

$$\mathcal{L}(\vec{w}, b; \mathcal{D}) = \sum_{n=1}^N 1_{\{f(\vec{x}_n) \neq y_n\}}$$

Sfortunatamente la valutazione di questa funzione loss risulta essere un problema di ottimizzazione combinatoria che è un problema NP-hard.

Quindi dobbiamo considerare una loss alternativa che permetta di ottenere come risultato la stessa SVM studiata.

A tale scopo consideriamo gli errori commessi da una SVM. Tali errori crescono in modo lineare rispetto alla loro distanza dal lato corretto del margine. Quindi la funzione loss che di cui necessitiamo è nota come **hinge loss** (*loss a cerniera*) ed è definita nel seguente modo:

$$\mathcal{L}(\vec{w}, b; \mathcal{D}) = \sum_{n=1}^N \max\{0, 1 - y_n(\langle \vec{w}, \vec{x}_n \rangle + b)\} = \sum_{n=1}^N \ell(\vec{x}_n, y_n)$$

dove:

$$\ell(\vec{x}, y) = \begin{cases} 0 & \text{se } y(\langle \vec{w}, \vec{x} \rangle + b) \geq 1 \\ 1 - y(\langle \vec{w}, \vec{x} \rangle + b) & \text{se } y(\langle \vec{w}, \vec{x} \rangle + b) < 1 \end{cases}$$

Tale funzione di errore hinge può essere vista come un'approssimazione dell'errore di classificazione (ovvero la funzione che vogliamo minimizzare) e viene rappresentata nel seguente modo:

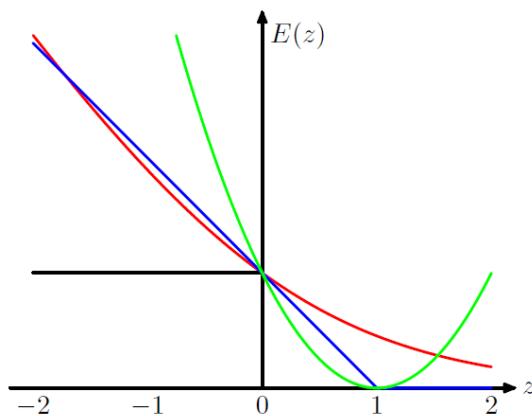


Figura 5.5: Grafico della funzione di errore hinge usata nelle SVM (rappresentata in blu), insieme alla funzione di errore per la regressione logistica (rappresentata in rosso), ridimensionata di un fattore $\frac{1}{\ln(2)}$ in modo che passi per il punto $(0, 1)$.

Sono mostrati anche l'errore di classificazione (in nero) e l'errore quadratico (in verde).

A questo punto dobbiamo determinare il controllo della complessità del modello. A tale scopo, ricordando il procedimento analizzato durante la discussione sulla regressione lineare, otteniamo:

$$(\vec{w}^*, b^*) = \arg \min_{\vec{w}, b} \frac{1}{2} \|\vec{w}\|^2 + C \sum_{n=1}^N \max\{0, 1 - y_n(\langle \vec{w}, \vec{x}_n \rangle + b)\}$$

Abbiamo così ottenuto un problema di ottimizzazione non vincolato (ma comunque convesso) espresso in termini di loss e di regolarizzatore.

Quindi il problema della massimizzazione del margine può essere visto come una regolarizzazione.

N.B: al contrario di λ , l'iperparametro C viene utilizzato per pesare la loss invece che per il regolatore.

Il risultato rappresenta però solamente una forma primale alternativa per una SVM.

5.3.2 Sguardo più approfondito alla forma duale della SVM

Torniamo alla forma duale di una SVM analizzata in precedenza:

$$\max_{\vec{a}} \left\{ \sum_{n=1}^N a_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N a_n a_m y_n y_m \langle \vec{x}_n, \vec{x}_m \rangle \right\}$$

soggetto ai vincoli $0 \leq a_n \leq C \ \forall n = 1, \dots, N$ ed $\sum_{n=1}^N a_n y_n = 0$.

Osserviamo che l'unico utilizzo dei campioni di input è nel prodotto interno $\langle \vec{x}_n, \vec{x}_m \rangle$. Questo significa che per apprendere è sufficiente calcolare i prodotti interni fra i campioni di input \vec{x}_n .

Perciò potremmo utilizzare una combinazione $\phi : \mathbb{R}^D \rightarrow \mathcal{H}$ sui dati (se $(\mathcal{H}, \langle \cdot, \cdot \rangle_{\mathcal{H}})$ è uno spazio del prodotto interno). In questo modo otteniamo un nuovo problema di ottimizzazione

$$\max_{\vec{a}} \left\{ \sum_{n=1}^N a_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N a_n a_m y_n y_m \langle \phi(\vec{x}_n), \phi(\vec{x}_m) \rangle_{\mathcal{H}} \right\}$$

soggetto ai vincoli $0 \leq a_n \leq C \ \forall n = 1, \dots, N$ ed $\sum_{n=1}^N a_n y_n = 0$.

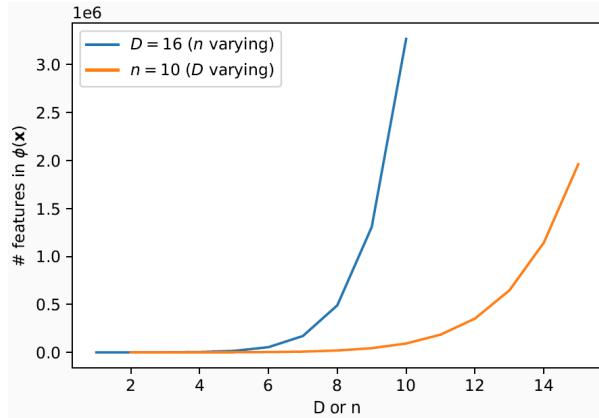
Le combinazioni ϕ di questo tipo sono solitamente chiamate **feature maps**, poiché mappano la rappresentazione di una feature in un particolare spazio ad un'altra rappresentazione della stessa feature in un nuovo spazio.

Notiamo che non ci sono limitazioni sulla forma della feature map.

Il vantaggio principale di questa mappatura esplicita delle feature è che in questo modo possiamo utilizzare combinazioni non lineari delle feature in input.

Il classificatore risultante sarà quindi lineare nel nuovo spazio delle feature, ma non lineare in quello originale.

Però non sempre funziona tutto correttamente. Infatti consideriamo una feature map polinomiale di grado n tale che $\phi : \mathbb{R}^D \rightarrow \mathbb{R}^K$. Allora osserviamo che la complessità dello spazio K delle variabili di output rischia di esplodere.



Infatti abbiamo che il numero di monomi di grado $\leq n$ contenuti nello spazio D delle variabili di input è dato da:

$$\binom{D+n-1}{n} = \frac{1}{(D-1)!} (n+1)^{\overline{D-1}}$$

dove $\overline{D-1}$ rappresenta la funzione di fattoriale incrementale.

Per risolvere questo problema adottiamo una risoluzione che include l'utilizzo della funzione **kernel**.

5.3.3 Kernel trick

Torniamo alla precedente rappresentazione duale per una SVM:

$$\max_{\vec{a}} \left\{ \sum_{n=1}^N a_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N a_n a_m y_n y_m \langle \phi(\vec{x}_n), \phi(\vec{x}_m) \rangle_{\mathcal{H}} \right\}$$

soggetto ai vincoli $0 \leq a_n \leq C \ \forall n = 1, \dots, N$ ed $\sum_{n=1}^N a_n y_n = 0$.

Osserviamo che non abbiamo bisogno delle singole combinazioni $\phi(\vec{x}_n)$ e $\phi(\vec{x}_m)$. Tutto ciò che necessitiamo è il prodotto interno $\langle \vec{x}_n, \vec{x}_m \rangle$.

Definiamo:

- **Funzione kernel** $k(\vec{x}, \vec{z}) = \langle \phi(\vec{x}), \phi(\vec{z}) \rangle_{\mathcal{H}}$.
- **Matrice kernel** (oppure **matrice Gram**)
 $K[n, m] = \langle \phi(\vec{x}_n), \phi(\vec{x}_m) \rangle_{\mathcal{H}} = k(\vec{x}_n, \vec{x}_m)$.

N.B.: la simmetria del prodotto interno implica che k e K sono simmetrici, mentre la definitività positiva del prodotto interno implica che K è una matrice semidefinita positiva (cioè $\vec{x}^T K \vec{x} \geq 0 \ \forall \vec{x}$).

La funzione kernel più semplice è il **kernel lineare** definito nel seguente modo:

$$k(\vec{x}, \vec{z}) = \vec{x}^T \vec{z}$$

corrispondente alla feature map lineare $\phi(\vec{x}) = \vec{x}$.

Quindi molti modelli lineari possono essere riformulati in una rappresentazione duale equivalente in cui le previsioni si basano su combinazioni lineari di una funzione kernel valutata in base ai punti dati di addestramento.

Il **kernel trick** (detto anche sostituzione del kernel) si riferisce all'utilizzo di funzioni kernel per sostituire i prodotti interni nei modelli non lineari.

Per sfruttare la sostituzione dei kernel, dobbiamo essere in grado di costruire funzioni kernel valide. Un approccio è quello di scegliere una mappatura dello spazio delle feature $\phi(\vec{x})$ e usarla per trovare il kernel corrispondente $k(\vec{x}, \vec{z}) = \langle \phi(\vec{x}), \phi(\vec{z}) \rangle$.

Un approccio alternativo consiste nel costruire direttamente le funzioni kernel. In questo caso dobbiamo assicurarc che la funzione scelta sia un kernel valido, ovvero che corrisponda a un prodotto scalare in uno spazio di feature.



ES:

Consideriamo il caso particolare di uno spazio di input bidimensionale $\vec{x} = (x_1, x_2)$, cioè $V = \mathbb{R}^2$. In questo caso possiamo espandere i termini e quindi identificare la corrispondente mappatura non lineare delle feature. Ovvero:

$$\begin{aligned} k(\vec{x}, \vec{z}) &= \langle \vec{x}, \vec{z} \rangle^2 = (\vec{x}^T \vec{z})^2 = (x_1 z_1 + x_2 z_2)^2 \\ &= x_1^2 z_1^2 + 2x_1 z_1 x_2 z_2 + x_2^2 z_2^2 = (x_1^2, \sqrt{2}x_1 x_2, x_2^2)(z_1^2, \sqrt{2}z_1 z_2, z_2^2) \\ &= \phi(\vec{x})^T \phi(\vec{z}) \end{aligned}$$

Quindi $k(\vec{x}, \vec{z}) = (\vec{x}^T \vec{z})^2$ corrisponde alla combinazione polinomiale $\phi(\vec{x}) = (x_1^2, \sqrt{2}x_1 x_2, x_2^2)$.

Abbiamo bisogno di un modo semplice per verificare se una funzione costituisce un kernel valido senza dover costruire esplicitamente la funzione $\phi(x)$. In particolare, la condizione necessaria e sufficiente affinché una funzione $k(\vec{x}, \vec{z})$ sia un kernel valido è che la matrice di Gram K deve essere semidefinita positiva per tutte le possibili scelte dell'insieme $\{\vec{x}_n\}$.

Una tecnica potente per la costruzione di nuovi kernel è quella di costruirli a partire da kernel più semplici (come se fossero dei blocchi di costruzione). A tale scopo sono state definite alcune proprietà.

In particolare, dati i kernel validi $k_1(x, z)$ e $k_2(x, z)$ allora anche i seguenti nuovi kernel (ottenuti a partire dai kernel dati) risultano essere validi:

$$\begin{aligned} k(x, z) &= xk_1(x, z) \\ k(x, z) &= f(x)k_1(x, z)f(z) \\ k(x, z) &= q(k_1(x, z)) \\ k(x, z) &= \exp(k_1(x, z)) \\ k(x, z) &= k_1(x, z) + k_2(x, z) \\ k(x, z) &= k_1(x, z)k_2(x, z) \\ k(x, z) &= k_3(\phi(x), \phi(z)) \\ k(x, z) &= \vec{x}^T A \vec{z} \\ k(x, z) &= k_a(x_a, z_a) + k_b(x_b, z_b) \\ k(x, z) &= k_a(x_a, z_a)k_b(x_b, z_b) \end{aligned}$$

dove $c > 0$ è una costante, $f(\cdot)$ è una qualsiasi funzione, $q(\cdot)$ è un polinomio con coefficienti non negativi, $\phi(x) : x \rightarrow \mathbb{R}^M$ è una funzione, $k_3(\cdot, \cdot)$ è un kernel valido in

\mathbb{R}^M , A è una matrice simmetrica semidefinita positiva, x_a ed x_b sono variabili (non necessariamente disgiunte) con $x = (x_a, x_b)$ ed k_a e k_b sono kernel validi sui rispettivi spazi.

Un kernel comunemente utilizzato è quello Gaussiano:

$$k(\vec{x}, \vec{z}) = \exp \left\{ -\frac{\|\vec{x} - \vec{z}\|^2}{2\sigma^2} \right\}$$

Possiamo vedere che questo è un kernel valido espandendo il quadrato all'interno dell'esponenziale:

$$\|\vec{x} - \vec{z}\|^2 = \vec{x}^T \vec{x} - 2\vec{x}^T \vec{z} + \vec{z}^T \vec{z}$$

Da cui otteniamo:

$$k(\vec{x}, \vec{z}) = \exp \left(-\frac{\vec{x}^T \vec{x}}{2\sigma^2} \right) \exp \left(\frac{\vec{x}^T \vec{z}}{\sigma^2} \right) \exp \left(-\frac{\vec{z}^T \vec{z}}{2\sigma^2} \right)$$

Notiamo che il vettore delle feature ϕ che corrisponde al kernel Gaussiano ha una dimensionalità infinita.

Un altro kernel molto utilizzato è conosciuto come **radial basis function** che dipende esclusivamente dalla grandezza della distanza tra gli argomenti, ovvero:

$$k(\vec{x}, \vec{z}) = (\|\vec{x} - \vec{z}\|)$$

Sintetizzando, il concetto di kernel, formulato come prodotto interno in uno spazio di feature, ci permette di costruire estensioni di algoritmi noti facendo uso del trucco del kernel. L'idea generale è che, se abbiamo un algoritmo formulato in modo tale che il vettore di input \vec{x} entri solo sotto forma di prodotti scalari, allora possiamo sostituire il prodotto scalare con un'altra scelta di kernel.

A questo punto non ci resta che aggiustare il problema di apprendimento:

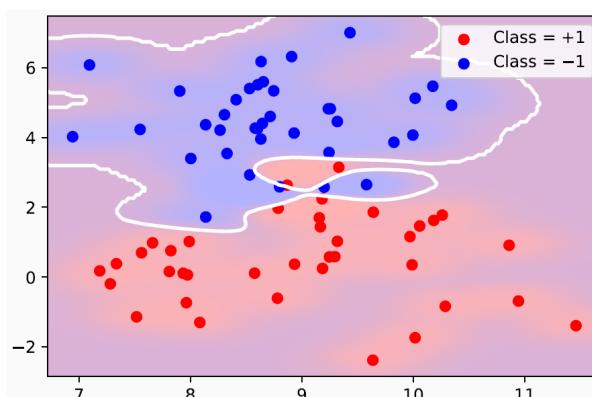
$$\max_{\vec{a}} \left\{ \sum_{n=1}^N a_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N a_n a_m y_n y_m k(\vec{x}_n, \vec{x}_m) \right\}$$

soggetto ai vincoli $0 \leq a_n \leq C \forall n = 1, \dots, N$ ed $\sum_{n=1}^N a_n y_n = 0$.

Mentre per il classificatore avremo:

$$f(\vec{x}) = \sum_{n=1}^N a_n y_n k(\vec{x}, \vec{x}_n) + b = \sum_{\vec{z} \in SV} a_n y_n k(\vec{x}, \vec{z}) + b$$

Il trucco del kernel è un altro modo per affrontare il caso della separazione non lineare. Inoltre risulta essere il modo principale per aumentare la complessità del modello senza cambiare la formulazione del modello sottostante.



5.4 SVM nella pratica

La SVM è fondamentalmente un classificatore a due classi. Nella pratica però ci troviamo spesso a dover affrontare problemi multiclassi che coinvolgono $K > 2$ classi. Sono stati quindi proposti vari metodi per combinare più SVM a due classi al fine di per costruire un classificatore multiclassi.

Un approccio comunemente utilizzato è quello del classificatore *uno-contro-tutti* che consiste nel costruire K SVM separate, per le quali il k -esimo modello $y_k(x)$ viene addestrato utilizzando i dati della classe C_k come esempi positivi e i dati delle restanti $K - 1$ classi come esempi negativi.

Tuttavia abbiamo visto che l'utilizzo di questi tipi di classificatori può portare a risultati incoerenti in cui un input viene assegnato a più classi contemporaneamente. Perciò utilizzeremo come regola di classificazione il massimo valore di output (ovvero si assegna la classe che appartiene al classificatore che genera il valore in uscita più elevato).

La complessità temporale e spaziale dei risolutori SVM può essere imprevedibile e varia a seconda del solutore utilizzato.

Fortunatamente, sono disponibili molti pacchetti solidi per l'addestramento di SVM con varie tecniche.

LibLinear

LibLinear è un risolutore SVM robusto e veloce il cui obiettivo principale è la risoluzione della formulazione primale dell'obiettivo.

Può passare a una formulazione duale per problemi su larga scala.

Inoltre può sfruttare macchine multi-core (utili per il caso multiclassi).

```
class sklearn.svm.LinearSVC(
    penalty='l2', loss='squared_hinge',
    dual=True, tol=0.0001, C=1.0,
    multi_class='ovr', fit_intercept=True,
    intercept_scaling=1,
    class_weight=None, verbose=0,
    random_state=None, max_iter=1000
)
```

LibSVM

LibSVM è un risolutore SVM duale il quale, sebbene offra una certa flessibilità grazie al trucco del kernel, può scalare male nello spazio (o nel tempo, se K non è precalcolato).

Supporta un'enorme varietà di formulazioni alternative dell'obiettivo di SVM.

```
class sklearn.svm.SVC(
    C=1.0, kernel='rbf',
    degree=3, gamma='scale', coef0=0.0,
    shrinking=True, probability=False, tol=0.001,
    cache_size=200, class_weight=None, verbose=False,
    max_iter=-1, decision_function_shape='ovr',
    break_ties=False, random_state=None
)
```

N.B.: per i risolutori duali, la selezione del kernel è fondamentale.

```
kernel{'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'}, default='rbf'

Specifies the kernel type to be used in the algorithm. It must be one
of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable.
If none is given, 'rbf' will be used. If a callable is given it is
used to pre-compute the kernel matrix from data matrices; that
matrix should be an array of shape (n_samples, n_samples).
```

Stochastic Gradient Solvers

Per i dataset molto grandi, l'opzione migliore è spesso quella di risolvere direttamente la forma primale utilizzando l'**Empirical Risk Minimization** (*minimizzazione empirica del rischio*).

La **Stochastic Gradient Solvers** (*discesa stocastica del gradiente*) implementa questo metodo in modo efficiente e sequenziale.

```
class sklearn.linear_model.SGDClassifier(
    loss='hinge', *, penalty='l2', alpha=0.0001,
    l1_ratio=0.15, fit_intercept=True, max_iter=1000,
    tol=0.001, shuffle=True, verbose=0, epsilon=0.1,
    n_jobs=None, random_state=None, learning_rate='optimal',
    eta0=0.0, power_t=0.5, early_stopping=False,
    validation_fraction=0.1, n_iter_no_change=5,
    class_weight=None, warm_start=False, average=False
)
```

Capitolo 6

MODELLI NON PARAMETRICI

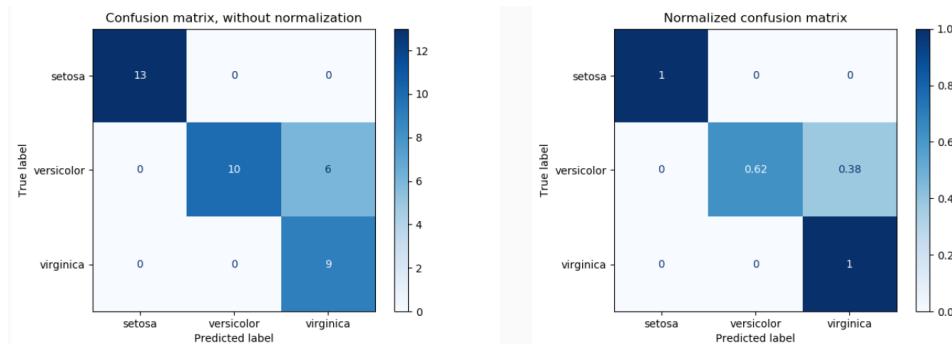
6.1 Tecniche di valutazione del classificatore

Vediamo le modalità per misurare le prestazioni di un classificatore dopo essere stato addestrato.

Se si ha un problema di classificazione *bilanciato* (ovvero per il quale i campioni sono equamente distribuiti per tutte le classi), allora possiamo utilizzare la misura dell'**accuracy** definita come:

$$\text{accuracy} := \frac{\text{numero dei campioni di test correttamente classificati}}{\text{numero dei campioni di test totali}}$$

Un buon strumento per avere una visione dei tipi di errori che un classificatore sta commettendo è la **matrice di confusione**. Ogni colonna della matrice rappresenta i valori predetti, mentre ogni riga rappresenta i valori reali. Quindi l'elemento sulla riga i e sulla colonna j (cioè l'elemento c_{ij}) rappresenta il numero di casi in cui il classificatore ha classificato la classe i come classe j . In particolare, se $j = i$ allora indica il numero di classificazioni corrette per la classe i , altrimenti indica il numero di classificazioni errate.



Possiamo suddividere le classificazioni in quattro diversi tipi:

- **True Positive** (*Veri Positivi*, **TP**), rappresentano le previsioni di dati appartenenti alla classe positiva correttamente classificati come positivi.
- **True Negative** (*Veri Negativi*, **TN**): rappresentano le previsioni di dati appartenenti alla classe negativa correttamente classificati come negativi.
- **False Positive** (*Falsi Positivi*, **FP**): rappresentano le previsioni di dati appartenenti alla classe negativa erroneamente classificati come positivi. Sono noti anche come **errori di tipo I**.

- **False Negative** (*Falsi Negativi, FN*): rappresentano le previsioni di dati appartenenti alla classe positiva erroneamente classificati come negativi. Sono noti anche come **errori di tipo II**.

		Predicted Class		
		Positive	Negative	
Actual Class	Positive	True Positive (TP)	False Negative (FN) Type II Error	Sensitivity $\frac{TP}{TP + FN}$
	Negative	False Positive (FP) Type I Error	True Negative (TN)	Specificity $\frac{TN}{TN + FP}$
		Precision $\frac{TP}{TP + FP}$	Negative Predictive Value $\frac{TN}{TN + FN}$	Accuracy $\frac{TP + TN}{TP + TN + FP + FN}$

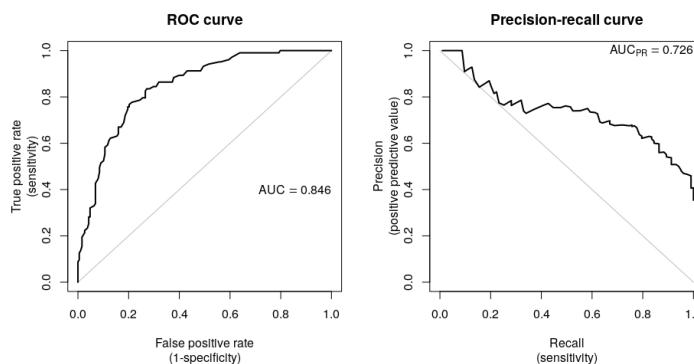
Possiamo quindi definire alcune metriche utili per un problema di classificazione *non bilanciato*:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{F1} = 2 \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Possiamo visualizzare le misure di *precision* e di *recall* utilizzando la curva **Receiver-Operating Characteristic (ROC)** o la curva **Precision-Recall (PR)** che mostrano come varia la *precision* in funzione della *recall* per diversi valori di soglia.



La metrica di valutazione da utilizzare dipende spesso dal tipo di problema (classificazione o regressione) e dalla distribuzione dei dati di addestramento (bilanciati o sbilanciati). Ottenerne una stima affidabile di un classificatore può essere complicato, poiché dipende chiaramente dalla suddivisione dei dati di training e di testing.

Questo diventa il problema centrale quando si esegue la selezione del modello tramite cross-validation.

N.B: queste metriche di valutazione sono definite in `sklearn.metrics`.

6.2 Istogrammi

Fino ad ora ci siamo concentrati sull'utilizzo di distribuzioni di probabilità governate da un numero di parametri i cui valori devono essere determinati da un dataset. Questo approccio è detto **parametrico**, poiché dobbiamo stimare i parametri del modello. Un limite importante di questo approccio è che la densità scelta potrebbe essere un modello inadeguato della distribuzione che genera i dati, ottenendo scarse prestazioni predittive.

Vogliamo quindi vedere se è possibile realizzare modelli che non necessitano dei parametri per la stima della densità. Tale approccio è detto **non parametrico**.

Definiamo innanzitutto il metodo dell'istogramma per la stima delle densità.

Un **istogramma** suddivide il dominio in **bins** distinti di ampiezza Δ_i e poi conta il numero n_i di osservazioni di x che cadono nel bin i . Per trasformare questo conteggio in una densità di probabilità normalizzata, è sufficiente dividere per il numero totale N di osservazioni e per l'ampiezza Δ_i dei bins, cioè:

$$p_i = \frac{n_i}{N\Delta_i}$$

Si ottiene così un modello per la densità $p(x)$ che è costante sull'ampiezza di ogni bin.

N.B.: spesso i bin sono scelti in modo da avere la stessa ampiezza $\Delta_i = \Delta$.

In particolare si ottiene uno stimatore di densità con un singolo iperparametro Δ .

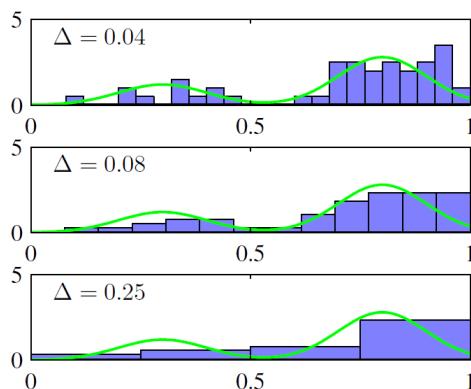


Figura 6.1: Illustrazione dell'approccio a istogrammi alla stima della densità, in cui un dataset viene generato dalla distribuzione bimodale (ovvero una distribuzione con due massimi) mostrata dalla curva verde.

Vengono mostrate le stime di densità dell'istogramma (con una larghezza di bin comune Δ) per diversi valori di Δ .

La precedente figura mostra un esempio di stima della densità con l'utilizzo dell'istogramma. Qui i dati vengono estratti dalla distribuzione, corrispondente alla curva verde (formata da un'insieme di due Gaussiane). In particolare vengono rappresentati tre diversi esempi di stima della densità corrispondente a tre scelte diverse per l'ampiezza del bin Δ .

Si nota che quando Δ è molto piccolo (figura in alto), il modello di densità risulta essere molto spigoloso (*spiky*), con una struttura che non approssima la distribuzione che ha generato il dataset.

Al contrario, se Δ è troppo grande (figura in basso), il modello di densità risulta essere troppo omogeneo (*smooth*), con una struttura che non approssima la distribuzione che ha generato il dataset.

I risultati migliori si ottengono per un valore intermedio di Δ (figura centrale). Quindi la selezione di Δ è critica, infatti una scelta di Δ troppo piccola comporta uno stimatore con una varianza elevata, mentre una scelta di Δ troppo elevata comporta uno stimatore con un bias elevato.

Si noti che il metodo dell'istogramma ha la proprietà che, una volta calcolato l'istogramma, l'insieme dei dati stesso può essere scartato, il che può essere vantaggioso se il dataset è di grandi dimensioni.

Un problema evidente è che la densità stimata presenta discontinuità ai bordi dei bin. Un'altra limitazione importante dell'approccio a histogrammi è la dimensionalità. Infatti se dividiamo ogni variabile in uno spazio D -dimensionale in M bins, allora il numero totale di bins sarà M^D (scalatura esponenziale).

N.B: in uno spazio ad alta dimensionalità, la quantità di dati necessari per fornire stime significative della densità di probabilità locale risulta essere proibitiva.

Nonostante le loro limitazioni, gli histogrammi ci permettono di effettuare due importanti considerazioni sulla stima della densità.

In primo luogo, per stimare la densità di probabilità in un determinato punto x del dominio dobbiamo esaminare i punti dati che si trovano vicino ad x .

N.B: il concetto di prossimità richiede che si assuma una qualche forma di misura della distanza (in questo caso abbiamo assunto la distanza Euclidea).

In secondo luogo, il valore del parametro Δ non deve essere né troppo grande né troppo piccolo per ottenere buoni risultati. Questo ricorda la scelta della complessità del modello nell'adattamento di curve polinomiali (con grado del polinomio M), o alternativamente la scelta del parametro di regolarizzazione λ .

Considerate tali premesse, analizziamo alcuni metodi non parametrici utilizzati per la stima della densità.

Per motivare come estendere l'idea di un'istogramma ad uno stimatore puramente locale, dobbiamo considerare la distribuzione Binomiale.

La distribuzione Binomiale rappresenta la probabilità di vedere m successi in una sequenza di N prove di Bernoulli, cioè:

$$\text{Bin}(m \mid N, \mu) = \binom{N}{m} \mu^m (1 - \mu)^{(N-m)}$$

$$\mathbb{E}(m) = \sum_{m=0}^N m \cdot \text{Bin}(m \mid N, \mu) = N\mu$$

$$\text{Var}(m) = \sum_{m=0}^N (m - \mathbb{E}[m])^2 \cdot \text{Bin}(m \mid N, \mu) = N\mu(1 - \mu)$$

dove μ rappresenta la probabilità di avere successo ed $\binom{N}{m} = \frac{N!}{(N-m)!m!}$ è il coefficiente binomiale e rappresenta il numero di sottoinsiemi di m elementi estratti da un insieme di N elementi.

La distribuzione Binomiale è unimodale (cioè possiede un unico modo, ovvero esiste un solo valore massimo) e in qualche modo simile alla Gaussiana.

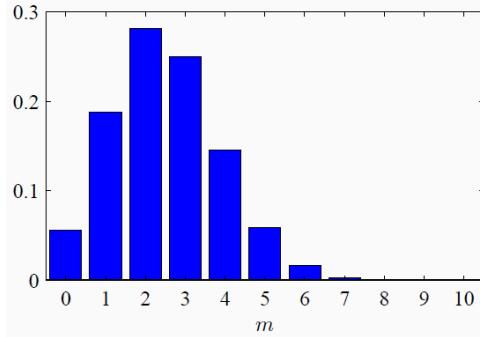


Figura 6.2: Grafico dell’istogramma della distribuzione binomiale in funzione di m per $N = 10$ ed $\mu = 0.25$.

Assumiamo quindi che le osservazioni vengano estratte da una densità di probabilità sconosciuta $p(\vec{x})$ in uno spazio D -dimensionale Euclideo. L’obiettivo quindi è quello di stimare il valore di $p(\vec{x})$.

Così come abbiamo osservato per la località degli istogrammi, consideriamo una piccola regione \mathcal{R} situata intorno ad \vec{x} (quindi contenente \vec{x}).

La massa di probabilità associata a tale regione è data da:

$$P = \int_{\mathcal{R}} p(\vec{x}) d\vec{x}$$

Supponiamo ora di avere N osservazioni estratte da $p(\vec{x})$, tale che ogni osservazione (ovvero ogni punto dati) abbia una probabilità P di trovarsi all’interno di \mathcal{R} .

Allora il numero totale K di punti che rientrano in \mathcal{R} sarà distribuito secondo la distribuzione Binomiale, ovvero:

$$\text{Bin}(K | N, P) = \binom{N}{K} P^K (1 - P)^{(N-K)}$$

Utilizzando la formula della media per la distribuzione Binomiale possiamo osservare che la frazione dei dati localizzati all’interno della regione \mathcal{R} sarà $\mathbb{E}\left[\frac{K}{N}\right] = P$ e, analogamente, utilizzando la formula della varianza per la distribuzione Binomiale possiamo visualizzare che sarà $\text{Var}\left[\frac{K}{N}\right] = \frac{P(1-P)}{N}$.

Per grandi valori di N , tale distribuzione avrà un picco intorno alla media e quindi sarà concentrata intorno a P . Perciò avremo $K \approx NP$.

Se, tuttavia, assumiamo anche che la regione \mathcal{R} sia sufficientemente piccola, tale da considerare la densità di probabilità $p(\vec{x})$ costante su tutta la regione (ovvero assumiamo che il volume di \mathcal{R} sia sufficientemente piccolo e quindi che $p(\vec{x})$ sia costante), allora avremo che $P \simeq p(\vec{x})V$, dove V rappresenta il volume di \mathcal{R} .

Combinando queste due osservazioni otteniamo la seguente soluzione per la stima della densità:

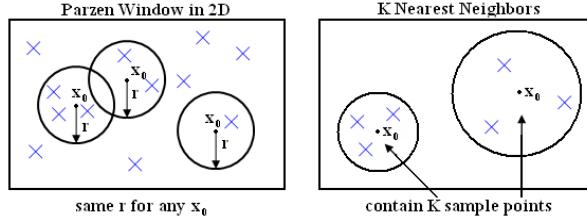
$$p(\vec{x}) = \frac{K}{NV}$$

Notiamo che la stima di $p(\vec{x})$ dipende da due ipotesi contraddittorie, ovvero che:

- La regione \mathcal{R} sia sufficientemente piccola da far sì che la distribuzione di densità $p(\vec{x})$ sia costante su tutta la regione.
- La regione \mathcal{R} sia sufficientemente grande da far sì che il numero K di punti che ricadono all’interno della regione sia sufficiente per raggiungere il picco della distribuzione Binomiale.

Possiamo quindi sfruttare il risultato per la stima della densità ottenuto in due modi diversi:

1. Possiamo fissare V e determinare K dai dati (tecnica del **Kernel Density Estimator**).
2. Possiamo fissare K e determinare il valore di V dai dati (tecnica del **K-Nearest Neighbor**).



N.B.: possiamo dimostrare che entrambi gli estimatori convergono alla vera densità di probabilità nel limite $N \rightarrow \infty$ a condizione che V si restrinja adeguatamente con N e che K cresca adeguatamente con N .

6.3 Kernel Density Estimator

Consideriamo la regione \mathcal{R} come un piccolo ipercubo unitario centrato sul punto \vec{x} in cui vogliamo determinare la densità di probabilità. Per contare il numero K di punti situati all'interno di questa regione è conveniente definire la seguente funzione kernel (detta funzione **Parzen window**):

$$k(\vec{u}) = \begin{cases} 1 & \text{se } |u_i| \leq \frac{1}{2}, \text{ per } i = 1, \dots, D \\ 0 & \text{altrimenti} \end{cases}$$

che rappresenta un cubo unitario centrato sull'origine \vec{x} .

Quindi, per ogni punto \vec{x} , la quantità $k(\frac{\vec{x}-\vec{x}_n}{h})$ sarà pari ad uno (cioè $k(\frac{\vec{x}-\vec{x}_n}{h}) = 1$) se il punto dati \vec{x}_n si trova all'interno di un ipercubo di lato h e centrato \vec{x} , altrimenti sarà zero.

Il numero totale K di punti dati che giacciono all'interno di questo cubo (intorno ad \vec{x}) sarà quindi:

$$K = \sum_{n=1}^N k\left(\frac{\vec{x} - \vec{x}_n}{h}\right)$$

Sostituendo questa espressione nella relazione per la stima della densità (cioè $p(\vec{x}) = \frac{K}{NV}$) si ottiene il seguente risultato:

$$p(\vec{x}) = \frac{1}{N} \sum_{n=1}^N \frac{1}{h^D} k\left(\frac{\vec{x} - \vec{x}_n}{h}\right)$$

dove abbiamo usato $V = h^D$ per il volume di un ipercubo di lato h in D dimensioni.

Questo risultato però soffre ancora del problema della discontinuità dei confini che abbiamo osservato con il modello degli istogrammi. In questo caso particolare si tratta di una discontinuità ai confini degli ipercubi.

Possiamo però ottenere un modello di densità più omogeneo se scegliamo una funzione kernel più omogenea. Una scelta comune è data dalla funzione kernel Gaussiana che comporta il seguente modello di densità:

$$p(\vec{x}) = \frac{1}{N} \sum_{n=1}^N \frac{1}{(2\pi h^2)^{D/2}} \exp \left\{ \frac{\|\vec{x} - \vec{x}_n\|^2}{2h^2} \right\}$$

dove h rappresenta la deviazione standard (cioè σ) delle componenti Gaussiane. Pertanto, il modello di densità si ottiene ponendo una Gaussiana attorno ad ogni punto dati e sommando i contributi sull'intero dataset. Quindi, dividendo per N otteniamo una densità correttamente normalizzata.

Questo risultato definisce il **Kernel Density Estimator** (o **Parzen Density Estimator**).

L'iperparametro h è chiamato *larghezza di banda del kernel* (**bandwidth kernel**) e controlla la quantità di smoothing dell'istogramma.

Ancora una volta, l'ottimizzazione di h è un problema di complessità del modello, analogo alla scelta dell'ampiezza dei bin nella stima della densità dell'istogramma o del grado del polinomio utilizzato nell'adattamento della curva. Infatti necessitiamo di un compromesso tra la sensibilità al rumore per un valore di h piccolo, e l'eccessivo smoothing per h grande.

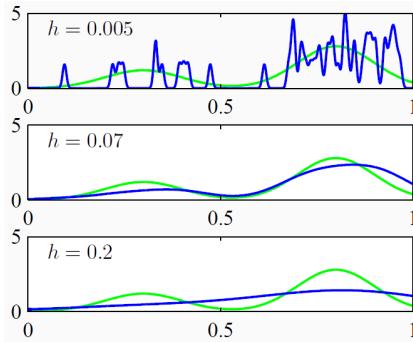


Figura 6.3: Illustrazione del modello Kernel Density Estimator applicato allo stesso dataset usato per dimostrare l'approccio a istogrammi (cioè i dati vengono generati attraverso la curva bimodale, costruita con due Gaussiane, rappresentata in verde).

Notiamo che h agisce come un parametro di smoothing. In particolare se viene scelto un valore di h troppo piccolo (figura superiore), il risultato è un modello di densità molto rumoroso. Al contrario, se viene scelto un valore di h troppo grande (figura inferiore), allora non si riesce a catturare la natura della distribuzione da cui sono stati generati i dati.

Il modello di densità migliore si ottiene per un valore intermedio di h (figura centrale).

La funzione kernel deve soddisfare le seguenti due condizioni:

$$\begin{aligned} k(\vec{u}) &\geq 0 \\ \int k(\vec{u}) d\vec{u} &= 1 \end{aligned}$$

Tali condizioni assicurano che la distribuzione di probabilità risultante sia ovunque non negativa e che il valore dell'integrale sia unitario (questo garantisce un ipercubo unitario).

Questo tipo di approcci sono spesso chiamati **metodi locali** poiché per stimare la distribuzione di densità $p(\vec{x})$ si considerano i punti dati vicini a \vec{x} .

6.3.1 Nadaraya-Watson (o Kernel Regression)

Possiamo applicare questo tipo di metodo locale ai problemi di regressione.

Ricordiamo che l'espressione per la funzione di regressione $y(\vec{x})$ (ovvero per la funzione di predizione ottima) corrisponde al valore medio della variabile target condizionata dalla variabile input, ovvero:

$$y(\vec{x}) = \mathbb{E}[t \mid \vec{x}] = \int t p(t \mid \vec{x}) dt = \int t \frac{p(\vec{x}, t)}{p(\vec{x})} dt$$

Adesso applichiamo il modello Kernel Density Estimator per approssimare le densità $p(\vec{x}, t)$ ed $p(\vec{x})$, ovvero:

$$\begin{aligned}\hat{p}(\vec{x}, t) &= \frac{1}{N} \sum_{n=1}^N k_h(\vec{x} - \vec{x}_n) k_h(t - t_n) \\ \hat{p}(\vec{x}) &= \frac{1}{N} \sum_{n=1}^N k_h(\vec{x} - \vec{x}_n)\end{aligned}$$

Allora l'approssimazione per la funzione di predizione ottima sarà:

$$\begin{aligned}\mathbb{E}[t \mid \vec{x}] &= \int t \frac{p(\vec{x}, t)}{p(\vec{x})} dt = \int t \frac{\frac{1}{N} \sum_{n=1}^N k_h(\vec{x} - \vec{x}_n) k_h(t - t_n)}{\frac{1}{N} \sum_{n=1}^N k_h(\vec{x} - \vec{x}_n)} dt \\ &= \frac{\sum_{n=1}^N k_h(\vec{x} - \vec{x}_n) \int t k_h(t - t_n) dt}{\sum_{n=1}^N k_h(\vec{x} - \vec{x}_n)}\end{aligned}$$

dove $\int t k_h(t - t_n) dt = t_n$ (poiché $\int t k_h(t) dt = 0$ quindi $\int t k_h(t - 0) dt = 0$ da cui $\int t k_h(t - t_n) dt = t_n$).

Quindi:

$$\mathbb{E}[t \mid \vec{x}] = \frac{\sum_{n=1}^N k_h(\vec{x} - \vec{x}_n) t_n}{\sum_{n=1}^N k_h(\vec{x} - \vec{x}_n)}$$

Il risultato è noto come modello **Nadaraya-Watson** o **Kernel Regression (regressione kernel)**.

Per una funzione kernel localizzata, essa ha la proprietà di attribuire un peso maggiore ai punti dati \vec{x}_n che sono vicini a \vec{x} .

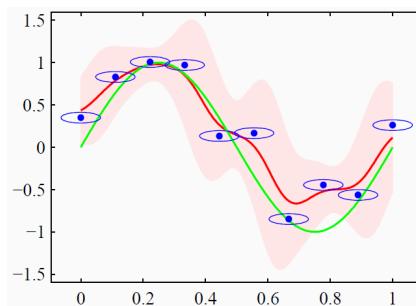


Figura 6.4: Illustrazione del problema di regressione per un dataset sinusoidale con l'utilizzo del modello Nadaraya-Watson, applicando kernel Gaussiani isotropi. La funzione sinusoidale originale è rappresentata dalla curva verde, mentre i punti dati sono indicati in blu. In particolare, ogni punto rappresenta il centro di un kernel Gaussiano isotropo.

La funzione di regressione risultante, data dalla media condizionale $\mathbb{E}[t \mid \vec{x}]$, è indicata dalla linea rossa (insieme alla regione delle deviazioni standard per la distribuzione condizionale $p(t|\vec{x})$ indicata dall'ombreggiatura rossa). L'ellisse blu attorno a ciascun punto dati mostra il contorno di una deviazione standard per il kernel corrispondente. Notiamo che tali kernel appaiono non circolari a causa delle diverse scale sugli assi orizzontale e verticale.

6.4 K-Nearest Neighbors

Una delle difficoltà incontrate con l'approccio Kernel Density Estimator per la stima della densità è che il parametro h (detto *bandwidth kernel*) che regola l'ampiezza del kernel è fisso per tutti i kernel.

Nelle regioni ad alta densità dei dati (ovvero regioni che presentano molti campioni), un valore elevato di h può portare a un eccessivo smoothing e ad una conseguente perdita dei dettagli della struttura che potrebbe essere estratta dai dati.

Al contrario, nelle regioni a bassa densità dei dati (ovvero regioni che presentano pochi campioni) la riduzione di h può portare a stime rumorose.

Pertanto la scelta ottimale di h può dipendere dalla posizione all'interno dello spazio dei dati.

Questo problema è affrontato dal metodo **K-Nearest Neighbor (KNN)** per la stima della densità.

Quindi a partire dall'espressione generale della densità $p(\vec{x}) = \frac{K}{NV}$, invece di fissare V (tramite la larghezza di banda h) e ricavare dai dati un'espressione per K , vogliamo fissare K e attraverso i dati cercare una soluzione appropriata per V .

Per fare questo, consideriamo una piccola ipersfera centrata sul punto \vec{x} in corrispondenza del quale vogliamo stimare il valore della densità $p(\vec{x})$ e facciamo crescere il raggio dell'ipersfera fino a che questa non contenga esattamente K punti dati (appartenenti al dataset di training).

Quindi la stima della densità è data da $p(\vec{x}) = \frac{K}{NV}$ con V uguale al volume della sfera corrispondente.

In questo caso il grado di smoothing del modello è governato dall'iperparametro K la quale scelta ottimale risulta essere un valore intermedio.

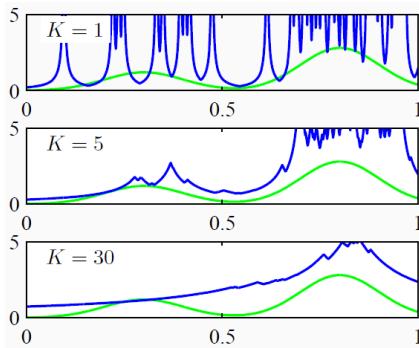


Figura 6.5: Illustrazione del modello K-Nearest Neighbor applicato allo stesso dataset dei precedenti approcci analizzati.

Vediamo che il parametro K governa il grado di di smoothing, per cui un piccolo valore di K (figura superiore) porta a un modello di densità molto rumoroso, mentre un valore elevato di K (figura inferiore) attenua la natura bimodale della distribuzione originale (mostrata dalla verde) da cui è stato generato il datset.

6.4.1 K-Nearest Neighbors Classification

Mostriamo come possiamo estendere la tecnica del K-Nearest Neighbors per la stima della densità al problema della classificazione.

Idealmente vogliamo applicare separatamente ad ogni classe l'approccio K-Nearest Neighbors per stimarne la densità e poi utilizzarne il teorema di Bayes.

Supponiamo di avere un dataset di dimensione N , contenente N_k esempi per ogni classe C_k (cioè tale che $\sum_k N_k = N$).

Se vogliamo classificare un nuovo punto \vec{x} , allora utilizziamo il trucco dell'ipersfera disegnando una sfera centrata su \vec{x} contenente esattamente K punti indipendentemente dalla loro classe.

Supponiamo che questa sfera abbia volume $V_{\vec{x}}$ e che contenga K_k punti della classe C_k . Allora possiamo facilmente stimare la densità associata a ciascuna classe come:

$$p(\vec{x} | C_k) = \frac{K_k}{N_k V_{\vec{x}}}$$

In modo simile possiamo stimare la densità marginale $p(\vec{x})$ e la distribuzione prior $P(C_k)$ come:

$$p(\vec{x}) = \frac{K}{NV_{\vec{x}}}$$

$$p(C_k) = \frac{N_k}{N}$$

Possiamo quindi combinare questi risultati utilizzando il teorema di Bayes per ottenere la distribuzione posterior relativa alla classe C_k :

$$p(C_k | \vec{x}) = \frac{p(\vec{x} | C_k)p(C_k)}{p(\vec{x})} = \frac{K_k}{N_k V_{\vec{x}}} \cdot \frac{N_k}{N} \cdot \frac{V_{\vec{x}}}{K} = \frac{K_k}{K}$$

Perciò si assegna ad un nuovo punto di test \vec{x} la classe con la maggiore probabilità posterior, corrispondente al valore maggiore di $\frac{K_k}{K}$.

Quindi, per classificare un nuovo punto, si identificano i K punti più vicini del dataset di training e assegniamo al nuovo punto la classe che ha il maggior numero di rappresentanti di questo insieme.

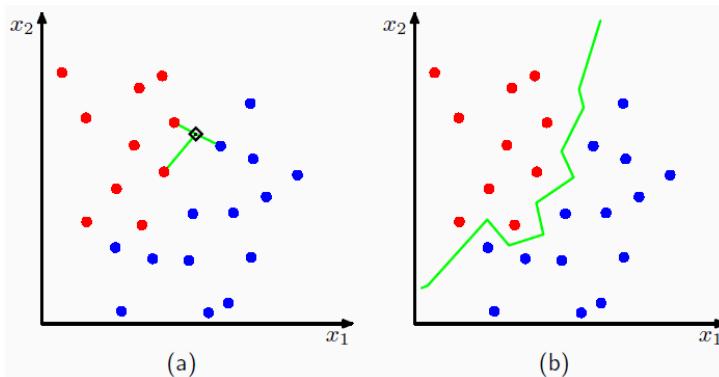


Figura 6.6: (a) un nuovo punto, indicato dal diamante nero, viene classificato in base all'appartenenza alla classe maggioritaria dei K punti dati di training più vicini. In questo caso si considerano $K = 3$ punti vicini ed osserviamo che la classe maggioritaria dei vicini del nuovo punto risulta essere quella rossa (perciò il nuovo punto sarà classificato come appartenente alla classe rossa).

(b) il confine decisionale risultante è composto da iperpiani che formano bisettrici perpendicolari di coppie di punti appartenenti a classi diverse.

Capitolo 7

UNSUPERVISED LEARNING

I modelli di *apprendimento non supervisionato* (**unsupervised learning**) sono un tipo di algoritmi di Machine Learning in cui l'apprendimento viene effettuato a partire da dati non etichettati. L'obiettivo è quello di costruire una rappresentazione dei dati e quindi trovarne i pattern che li caratterizzano.

7.1 K-Means Clustering

Il **Clustering** si riferisce a tecniche di apprendimento che acquisiscono informazioni a partire da gruppi (detti *cluster*) di punti dati correlati nello spazio delle features.

Il Clustering è una tecnica robusta, nel senso che non fallisce mai. La sua utilità dipende dalla distribuzione dei dati e dal tipo di algoritmo di clustering che si esegue.

K-Means è un algoritmo di clustering molto semplice che associa ad ogni punto una delle k medie (dette **centroidi**) nello spazio dei dati.

In particolare, dato un insieme iniziale di k centroidi $\vec{m}_1^1, \dots, \vec{m}_k^1$, allora si eseguono due fasi alternate:

- *Assegnamento (Assignment)*: $S_i^t = \{\vec{x}_p \mid \|\vec{x}_p - \vec{m}_i^t\| \leq \|\vec{x}_p - \vec{m}_j^t\| \forall j\}$
- *Aggiornamento (Update)*: $\vec{m}_i^{t+1} = \frac{1}{|S_i^t|} \sum_{\vec{x}_i \in S_i^t} \vec{x}_j$

Tale algoritmo converge quando gli assegnamenti dei cluster non cambiano più.

N.B.: non è garantito che questo algoritmo trovi i cluster ottimali globali.

Iniziamo quindi a considerare il problema dell'identificazione di gruppi (o cluster) di punti dati in uno spazio multidimensionale. Supponiamo di avere un dataset $\mathcal{D} = \{\vec{x}_1, \dots, \vec{X}_N\}$ costituito da N campioni di una variabile \vec{x} Euclidea D -dimensionale. L'obiettivo è quello di partizionare il dataset in un certo numero K di cluster, dove per il momento supponiamo che il valore di K sia noto.

Intuitivamente, possiamo pensare ad un cluster come ad un sottoinsieme del dataset \mathcal{D} in cui la somma delle distanze tra i punti contenuti in esso è piccola rispetto alla somma delle distanze dei punti esterni al cluster.

Possiamo formalizzare questa nozione introducendo innanzitutto un insieme di vettori prototipi D -dimensionali $\vec{\mu}_k$, dove $k = 1, \dots, K$ (in particolare $\vec{\mu}_k$ è un prototipo associato al k -esimo cluster).

Possiamo quindi pensare tali $\vec{\mu}_k$ come i centri dei cluster. Il nostro obiettivo perciò sarà quello di trovare un assegnamento dei punti ai cluster tale che la somma dei quadrati delle distanze di ciascun punto dal vettore $\vec{\mu}_k$ più vicino sia minima.

A questo punto è opportuno definire una notazione per descrivere l'assegnamento dei punti ai cluster. In particolare, per ogni punto \vec{x}_n introduciamo un insieme

corrispondente di variabili binarie indicatori $r_{nk} \in \{0, 1\}$, con $k = 1, \dots, K$, che descrivono a quale dei K cluster è stato assegnato il punto \vec{x}_n .

Perciò se il punto dati \vec{x}_n è assegnato a cluster k allora $r_{nk} = 1$ altrimenti $r_{nj} = 0$ per $j \neq k$ (noto come schema di codifica 1-di- K).

Formalmente:

$$r_{nk} = \begin{cases} 1 & \text{se } \vec{x}_n \text{ è stato assegnato al cluster } k \\ 0 & \text{altrimenti} \end{cases}$$

Possiamo quindi definire una funzione obiettivo, detta *misura di distorsione (distortion measure)*, come:

$$J = \sum_{n=1}^N \sum_{k=1}^K r_{nk} \|\vec{x}_n - \vec{\mu}_k\|^2$$

Questa funzione rappresenta la somma dei quadrati delle distanze di ogni punto dal suo vettore $\vec{\mu}_k$ assegnato. Il nostro obiettivo è quello di trovare i valori di r_{nk} e $\vec{\mu}_k$ in modo da minimizzare J .

Per fare ciò definiamo una procedura iterativa, detta algoritmo **EM**, in cui ogni iterazione prevede due passi corrispondenti rispettivamente a successive ottimizzazioni di r_{nk} e $\vec{\mu}_k$.

Vediamo come si implementa tale algoritmo.

Per prima cosa scegliamo alcuni valori iniziali per i $\vec{\mu}_k$. Quindi si eseguono in modo iterativo i seguenti due passaggi:

1. **E-Step (Expectation):** l'obiettivo è quello di minimizzare J rispetto ad r_{nk} mantenendo $\vec{\mu}_k$ fisso.

Essendo J è una funzione lineare di r_{nk} (come possiamo osservare dalla funzione obiettivo), allora questa fase di ottimizzazione può essere eseguita in modo indipendente per ognuno degli n punti e permette di ottenere una soluzione in forma chiusa.

In particolare possiamo assegnare l' n -esimo punto al centroide $\vec{\mu}_k$ del cluster più vicino (cioè ad ogni punto \vec{x}_n assegniamo $r_{nk} = 1$ per il cluster k che comporta il minimo valore di $\|\vec{x}_n - \vec{\mu}_k\|^2$). Formalmente:

$$r_{nk} = \begin{cases} 1 & \text{se } k = \arg \min_j \|\vec{x}_n - \vec{\mu}_j\|^2 \\ 0 & \text{altrimenti} \end{cases}$$

2. **M-Step (Maximization):** l'obiettivo è quello di minimizzare J rispetto ad $\vec{\mu}_k$ mantenendo r_{nk} fisso.

Essendo J una funzione quadratica di $\vec{\mu}_k$ (come possiamo osservare dalla funzione obiettivo), allora J può essere minimizzata ponendo a zero la sua derivata rispetto a $\vec{\mu}_k$. In particolare otteniamo:

$$2 \sum_{n=1}^N r_{nk} (\vec{x}_n - \vec{\mu}_k) = 0$$

Tale equazione può essere facilmente risolta rispetto a $\vec{\mu}_k$. In particolare otteniamo:

$$\vec{\mu}_k = \frac{\sum_{n=1}^N r_{nk} \vec{x}_n}{\sum_{n=1}^N r_{nk}}$$

Il denominatore rappresenta il numero di punti assegnati al cluster k e quindi l'obiettivo di tale espressione è quello di determinare $\vec{\mu}_k$ come la media di tutti i punti \vec{x}_n assegnati al cluster k .

Per questo motivo, la procedura è nota come algoritmo K-Means.

Questa procedura di ottimizzazione viene ripetuta fino alla convergenza. Ovvero queste due fasi di assegnazione dei punti ai cluster (*E-Step*) e di calcolo delle medie dei cluster (*M-Step*) vengono ripetute in modo alternato fino a quando non si verificano ulteriori cambiamenti nelle assegnazioni (ovvero fino a quando le assegnazioni non cambiano durante la fase *E-Step*), oppure fino a quando non viene superato un numero massimo di iterazioni.

N.B.: poiché ogni fase riduce il valore della funzione obiettivo J , allora la convergenza dell'algoritmo è assicurata. Tuttavia, potrebbe convergere ad un minimo locale.

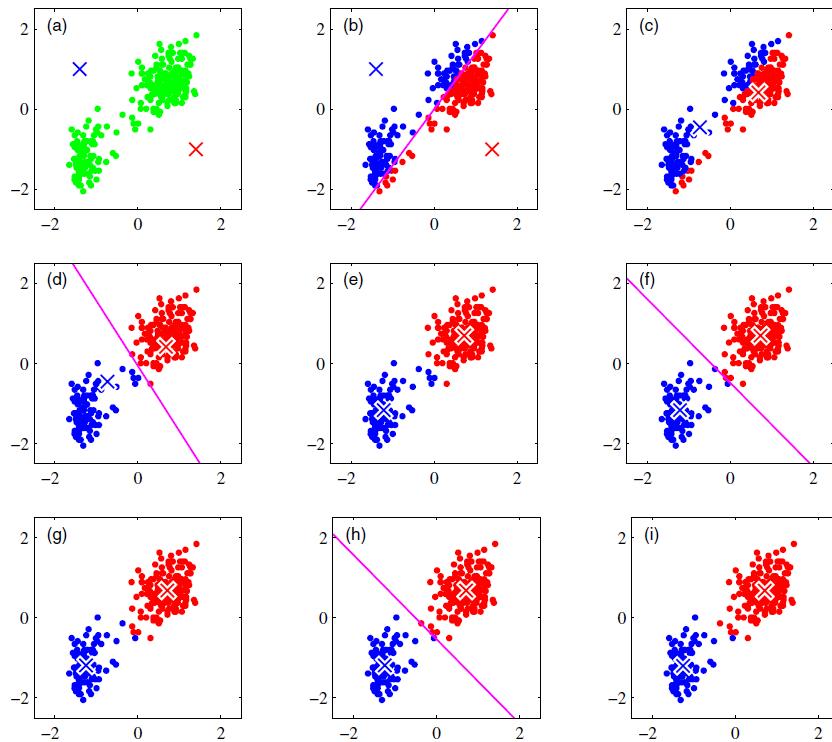


Figura 7.1: Illustrazione dell'algoritmo K-means (con $K = 2$). (a) I punti verdi rappresentano il dataset in uno spazio Euclideo bidimensionale. Le scelte iniziali dei centri $\vec{\mu}_1$ e $\vec{\mu}_2$ sono indicate rispettivamente dalle croci rosse e blu. (b) Nella fase iniziale dell'*E-Step*, ogni punto viene assegnato al cluster rosso o al cluster blu, a seconda del centro del cluster più vicino. Essendo $K = 2$, questo equivale a classificare i punti in base a quale lato della bisettrice perpendicolare ai due centri dei cluster (indicata dalla linea magenta), si trovano. (c) Nella successiva fase *M-Step*, ogni centro del cluster viene ricalcolato come media dei punti assegnati al cluster corrispondente. (d)-(i) mostrano le successive fasi E e M eseguite in modo alternato fino alla convergenza finale dell'algoritmo.

7.2 Gaussian Mixture Models

Sebbene la distribuzione con una singola Gaussiana abbia alcune importanti proprietà analitiche, soffre di notevoli limitazioni quando si tratta di modellare dataset reali.



ES:

Consideriamo il seguente dataset conosciuto come “Old Faithful” che comprende 272 misurazioni dell’eruzione del geyser Old Faithful.

Le misurazioni comprendono la durata dell’eruzione in minuti (asse orizzontale) ed il tempo in minuti fino all’eruzione successiva (asse verticale).

Notiamo che il dataset forma due gruppi dominanti per i quali una semplice distribuzione Gaussiana non è in grado di catturarne la struttura. Al contrario una sovrapposizione lineare di due Gaussiane fornisce una migliore caratterizzazione del dataset.

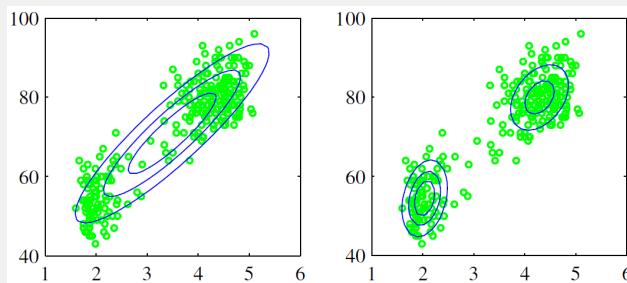


Figura 7.2: Illustrazione del dataset “Old Faithful” in cui le curve blu rappresentano i contorni di densità di probabilità costante.

In particolare, a sinistra è riportata una singola distribuzione Gaussiana che è stata adattata ai dati utilizzando la massima verosimiglianza. Si noti che questa distribuzione non riesce a catturare i due gruppi di dati e colloca gran parte della sua massa di probabilità nella regione centrale tra i gruppi, dove i dati sono relativamente scarsi.

A destra, invece, la distribuzione è data da una combinazione lineare di due Gaussiane che è stata adattata ai dati con la massima verosimiglianza. Questa soluzione fornisce una migliore rappresentazione dei dati.

In particolare osserviamo che una combinazione lineare di più Gaussiane può dare origine a densità più complesse. In particolare, utilizzando un numero sufficiente di Gaussiane e regolando le loro medie, le loro covarianze e i coefficienti della combinazione lineare, allora possiamo approssimare qualsiasi densità continua con una precisione arbitraria.

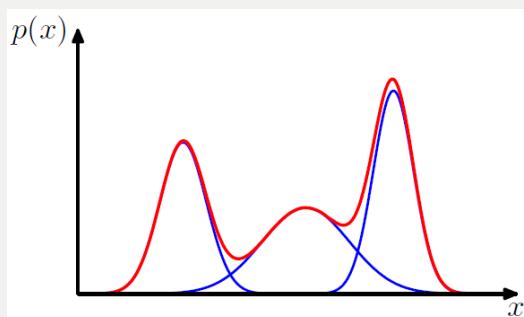


Figura 7.3: Esempio di distribuzione di una *mixture* Gaussiana (indicata in rosso) ottenuta dalla combinazione di tre Gaussiane (rappresentate in blu).

Quindi possiamo realizzare combinazioni lineari di distribuzioni più elementari (come le Gaussiane), in modo da definire modelli probabilistici conosciuti come *distribuzioni miste (mixture distributions)*.

Consideriamo quindi una sovrapposizione di K densità Gaussiane (ognuna delle quali è

detta componente della *mixture*) della forma:

$$p(\vec{x}; \vec{\theta}) = \sum_{k=1}^K \pi_k \mathcal{N}(\vec{x} | \vec{\mu}_k, \Sigma_k)$$

detta **mixture of Gaussians** (*miscela di Gaussiane*). In particolare, come abbiamo osservato, ogni densità Gaussiana $\mathcal{N}(\vec{x} | \vec{\mu}_k, \Sigma_k)$ è detta **componente** della miscela con media $\vec{\mu}_k$ e covarianza Σ_k . Inoltre il parametro π_k è detto **mixing coefficients** (*coefficienti di miscelazione*).

N.B.: da tale formulazione possiamo definire un **Gaussian Mixture Model (GMM)**.

N.B.: $\vec{\theta} = [\mu_1, \dots, \mu_k, \Sigma_1, \dots, \Sigma_k, \pi_1, \dots, \pi_k]$ rappresenta il vettore contenente tutti i parametri del modello.

Se integriamo entrambi i membri della miscela rispetto ad \vec{x} ed osserviamo che sia $p(\vec{x}; \vec{\theta})$ che le singole componenti Gaussiane sono normalizzate, allora otteniamo la condizione $\sum_{k=1}^K \pi_k = 1$.

Inoltre, il requisito che $p(\vec{x}; \vec{\theta}) \geq 0$ insieme a $\mathcal{N}(\vec{x} | \vec{\mu}_k, \Sigma_k) \geq 0$, implica la condizione $\pi_k \geq 0$ per tutti i k .

Combinando queste due condizioni, otteniamo:

$$\begin{aligned} 0 &\leq \pi_k \leq 1 \\ \sum_{k=1}^K \pi_k &= 1 \end{aligned}$$

N.B.: i coefficienti di miscelazione π_k rappresentano delle probabilità.

La forma della distribuzione della miscela di Gaussiana è regolata dai parametri π_k , $\vec{\mu}_k$ e Σ_k . Un modo per stimare i valori di questi parametri potrebbe essere quello di utilizzare la massima verosimiglianza (Maximum Likelihood):

$$p(\mathcal{D}; \vec{\theta}) = \prod_{n=1}^N p(\vec{x}_n | \vec{\theta}) = \prod_{n=1}^N \sum_{k=1}^K \pi_k \mathcal{N}(\vec{x}_n | \vec{\mu}_k, \Sigma_k)$$

Applicando come al solito il logaritmo alla distribuzione per il calcolo del massimo (poiché la funzione logaritmica non cambia il massimo) otteniamo:

$$\ln p(\mathcal{D}; \vec{\theta}) = \sum_{n=1}^N \ln \left\{ \sum_{k=1}^K \pi_k \mathcal{N}(\vec{x}_n | \vec{\mu}_k, \Sigma_k) \right\}$$

Osserviamo che la soluzione è diventata più complessa rispetto a quella ottenuta con una singola Gaussiana, a causa della presenza della sommatoria su k all'interno del logaritmo.

Di conseguenza, la soluzione di massima verosimiglianza per i parametri non ha più una soluzione analitica in forma chiusa.

Perciò dobbiamo definire approcci diversi per massimizzare la distribuzione likelihood.

Passiamo quindi ad una formulazione delle miscele Gaussiane in termini di **variabili latenti** discrete, ovvero variabili che non vengono mai osservate ma vengono introdotte per semplificare il problema di ottimizzazione. A tale scopo introduciamo una variabile casuale binaria K -dimensionale \vec{z} con rappresentazione 1-di- K (perciò un particolare

elemento z_k è uguale a 1 e tutti gli altri elementi sono uguali a 0). Quindi i valori di $z_k \in \{0, 1\}$ con la condizione che $\sum_k z_k = 1$.

N.B.: abbiamo K possibili stati per il vettore \vec{z} a seconda di quale elemento è non nullo.

Quindi definiamo la distribuzione congiunta $p(\vec{x}, \vec{z})$ in termini della distribuzione marginale $p(\vec{z})$ e della distribuzione condizionale $p(\vec{x} | \vec{z})$.

In particolare la distribuzione marginale $p(\vec{z})$ è specificata in termini dei coefficienti di miscelazione π_k . Ovvero, considerando un particolare valore z_k , la distribuzione marginale è definita come:

$$p(z_k = 1) = \pi_k$$

dove i coefficienti π_k devono soddisfare le condizioni $0 \leq \pi_k \leq 1$ ed $\sum_{k=1}^K \pi_k = 1$.

Poiché \vec{z} utilizza una rappresentazione 1-di- K allora possiamo riscrivere questa distribuzione marginale come:

$$p(\vec{z}) = \prod_{k=1}^K \pi_k^{z_k}$$

Analogamente la distribuzione condizionale di \vec{x} dato un particolare valore z_k è definita da una Gaussiana:

$$p(\vec{x} | z_k = 1) = \mathcal{N}(\vec{x} | \vec{\mu}_k, \Sigma_k)$$

Quindi, poiché \vec{z} utilizza una rappresentazione 1-di- K , allora possiamo riscrivere questa distribuzione condizione come:

$$p(\vec{x} | \vec{z}) = \prod_{k=1}^K \mathcal{N}(\vec{x} | \vec{\mu}_k, \Sigma_k)^{z_k}$$

Perciò la distribuzione congiunta è data da $p(\vec{x}, \vec{z}) = p(\vec{z})p(\vec{x} | \vec{z})$, da cui possiamo ricavare la distribuzione marginale di \vec{x} sommando la distribuzione congiunta su tutti i possibili stati di \vec{z} (cioè si elimina la dipendenza da \vec{z} nella distribuzione congiunta con la proprietà della somma, ovvero $p(\vec{x}) = \sum_{\vec{z}} p(\vec{x}, \vec{z})$):

$$p(\vec{x}) = \sum_{\vec{z}} p(\vec{z})p(\vec{x} | \vec{z}) = \sum_{k=1}^K \pi_k \mathcal{N}(\vec{x} | \vec{\mu}_k, \Sigma_k)$$

Pertanto, la distribuzione marginale di \vec{x} è una Gaussian mixture.

Grazie a questa rappresentazione di $p(\vec{x})$ ne consegue che, se abbiamo N osservazioni $\vec{x}_1, \dots, \vec{x}_N$, allora per ogni punto osservato \vec{x}_n esiste una corrispondente variabile latente \vec{z}_n .

Abbiamo quindi trovato una formulazione equivalente della Gaussian mixture che coinvolge una variabile latente esplicita. In questo modo siamo in grado di sfruttare la distribuzione congiunta $p(\vec{x}, \vec{z})$ invece che la distribuzione marginale $p(\vec{x})$. Questo comporterà notevoli semplificazioni, in particolare attraverso l'introduzione dell'algoritmo Expectation-Maximization (EM).

Un'altra importante grandezza è la probabilità condizionale di \vec{z} dato \vec{x} . In particolare utilizzeremo $\gamma(z_k)$ per indicare la probabilità condizionale $p(z_k = 1 | \vec{x})$, il quale valore può essere trovato con il teorema di Bayes:

$$\gamma(z_k) \equiv p(z_k = 1 | \vec{x}) = \frac{p(z_k = 1)p(\vec{x} | z_k = 1)}{\sum_{j=1}^K p(z_j = 1)p(\vec{x} | z_j = 1)}$$

$$= \frac{\pi_k \mathcal{N}(\vec{x} \mid \vec{\mu}_k, \Sigma_k)}{\sum_{j=1}^K \pi_j \mathcal{N}(\vec{x} \mid \vec{\mu}_j, \Sigma_k)}$$

In particolare, π_k rappresenta la probabilità prior di $z_k = 1$ e la quantità $\gamma(z_k)$ rappresenta la corrispondente probabilità posterior (dopo avere osservato \vec{x}).

Inoltre possiamo vedere $\gamma(z_k)$ come la *responsabilità* (**responsibility**) assunta dalla componente k per “giustificare” l’osservazione \vec{x} .

Supponiamo di avere un dataset $\mathcal{D} = \{\vec{x}_1, \dots, \vec{x}_N\}$ che vogliamo modellare utilizzando una mixture di Gaussiane (GMM). Possiamo rappresentare questo dataset come una matrice X di dimensione $N \times D$, la cui n -esima riga è definita da \vec{x}_n^T .

Allo stesso modo le corrispondenti variabili latenti sono denotate da una matrice Z di dimensione $N \times K$, la cui n -esima riga è definita da \vec{z}_n^T .

Assumiamo che i dati \vec{x}_n siano indipendenti ed identicamente distribuiti (i.i.d.). Allora vogliamo massimizzare:

$$\ln p(X \mid \vec{\pi}, \vec{\mu}, \Sigma) = \sum_{n=1}^N \ln \left\{ \sum_{k=1}^K \pi_k \mathcal{N}(\vec{x}_n \mid \vec{\mu}_k, \Sigma_k) \right\}$$

che rappresenta il logaritmo della likelihood. Per massimizzare tale funzione dobbiamo calcolare le derivate rispetto a $\vec{\pi}$, $\vec{\mu}$ e Σ e porle a 0.

Innanzitutto calcoliamo il gradiente rispetto alla media $\vec{\mu}_k$:

$$\begin{aligned} 0 &= - \sum_{n=1}^N \frac{\pi_k \mathcal{N}(\vec{x}_n \mid \vec{\mu}_k, \Sigma_k)}{\sum_j \pi_j \mathcal{N}(\vec{x}_n \mid \vec{\mu}_j, \Sigma_j)} \Sigma_k (\vec{x}_n - \vec{\mu}_k) \\ \Rightarrow 0 &= \sum_{n=1}^N \gamma(z_{nk}) \Sigma_k (\vec{x}_n - \vec{\mu}_k) \end{aligned}$$

Moltiplicando per Σ_k^{-1} e chiamando $N_k = \sum_{n=1}^N \gamma(z_{nk})$ otteniamo:

$$\vec{\mu}_k = \frac{1}{N_k} \sum_{n=1}^N \gamma(z_{nk}) \vec{x}_n$$

N.B.: possiamo interpretare N_k come il numero effettivo di punti assegnati al cluster k . Osserviamo la forma di questa soluzione. Vediamo che la media $\vec{\mu}_k$ per la k -esima componente Gaussiana si ottiene a partire dalla media pesata su tutti i punti del dataset in cui il peso del punto \vec{x}_n è dato dalla probabilità posterior $\gamma(z_{nk})$ che la componente k si assuma la responsabilità di generare \vec{x}_n .

Calcoliamo il gradiente rispetto alla matrice di covarianza Σ_k seguendo lo stesso ragionamento. In particolare otteniamo:

$$\Sigma_k = \frac{1}{N_k} \sum_{n=1}^N \gamma(z_{nk}) (\vec{x}_n - \vec{\mu}_k)(\vec{x}_n - \vec{\mu}_k)^T$$

che ha la stessa forma del risultato corrispondente ad una singola Gaussiana applicata all’intero dataset, ma pesando ogni punto per la corrispondente probabilità posterior e

con il denominatore dato dal numero effettivo di punti associati alla corrispondente componente.

Infine calcoliamo il gradiente rispetto ai coefficienti di miscelazione π_k . In questo caso dobbiamo tenere conto del vincolo $\sum_{k=1}^K \pi_k = 1$. Questo vincolo può essere rispettato utilizzando un moltiplicatore di Lagrange. Quindi la funzione da massimizzare diventa:

$$\ln p(X | \vec{\pi}, \vec{\mu}, \Sigma) + \lambda \left(\sum_{k=1}^K \pi_k - 1 \right)$$

In particolare, derivando rispetto a π_k e ponendo la derivata a 0 otteniamo:

$$\begin{aligned} 0 &= \sum_{n=1}^N \frac{\mathcal{N}(\vec{x}_n | \vec{\mu}_k, \Sigma_k)}{\sum_j \mathcal{N}(\vec{x}_n | \vec{\mu}_j, \Sigma_j)} + \lambda \\ \Rightarrow 0 &= \frac{1}{\pi_k} \sum_{n=1}^N \frac{\pi_k \mathcal{N}(\vec{x}_n | \vec{\mu}_k, \Sigma_k)}{\sum_j \pi_j \mathcal{N}(\vec{x}_n | \vec{\mu}_j, \Sigma_j)} + \lambda \\ \Rightarrow 0 &= \frac{N_k}{\pi_k} + \lambda \end{aligned}$$

dove ancora una volta possiamo osservare l'apparizione delle responsabilità. La derivata parziale rispetto al moltiplicatore di Lagrange λ sarà:

$$0 = \sum_{k=1}^K \pi_k - 1 \Rightarrow \sum_{k=1}^K \pi_k = 1$$

Sostituendo questo risultato con quello ottenuto per π_k ricaviamo:

$$\pi_k = \frac{N_k}{N}$$

Otteniamo quindi che il coefficiente di miscelazione per la k -esima componente è dato dalla responsabilità media che quella componente si assume per “giustificare” i dati.

N.B.: i risultati ottenuti per i parametri $\vec{\pi}$, $\vec{\mu}$ e Σ non costituiscono una soluzione in forma chiusa per i parametri del modello perché le responsabilità $\gamma(z_{nk})$ dipende da tali parametri in modo complesso.

Tuttavia questi risultati definiscono uno schema iterativo per trovare una soluzione al problema della massima verosimiglianza. In particolare tale problema può essere implementato attraverso l'algoritmo EM applicato al caso particolare del Gaussian Mixture Model. Vediamo come implementare tale algoritmo.

Dato un GMM (Gaussian Mixture Model), l'obiettivo è quello massimizzare la funzione di verosimiglianza rispetto ai parametri (ovvero medie e covarianze delle componenti e coefficienti di miscelazione). Quindi iteriamo i seguenti passaggi:

1. Inizializziamo i coefficienti di miscelazione $\vec{\pi}$, la media $\vec{\mu}$ e la covarianza Σ e valutiamo il valore iniziale del logaritmo della likelihood.
2. **E-Step.** Valutiamo le responsabilità utilizzando gli attuali valori dei parametri.

$$\gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(\vec{x}_n | \vec{\mu}_k, \Sigma_k)}{\sum_{j=1}^K \pi_j \mathcal{N}(\vec{x}_n | \vec{\mu}_j, \Sigma_j)}$$

3. **M-Step.** Stimiamo nuovamente i parametri utilizzando le responsabilità correnti.

$$\begin{aligned}\mu_k^{\text{new}} &= \frac{1}{N_k} \sum_{n=1}^N \gamma(z_{nk}) \vec{x}_n \\ \Sigma_k^{\text{new}} &= \frac{1}{N_k} \sum_{n=1}^N \gamma(z_{nk})(\vec{x}_n - \mu_k^{\text{new}})(\vec{x}_n - \mu_k^{\text{new}})^T \\ \pi_k^{\text{new}} &= \frac{N_k}{N} \\ \text{con } N_k &= \sum_{n=1}^N \gamma(z_{nk}).\end{aligned}$$

4. Valutiamo il logaritmo della likelihood:

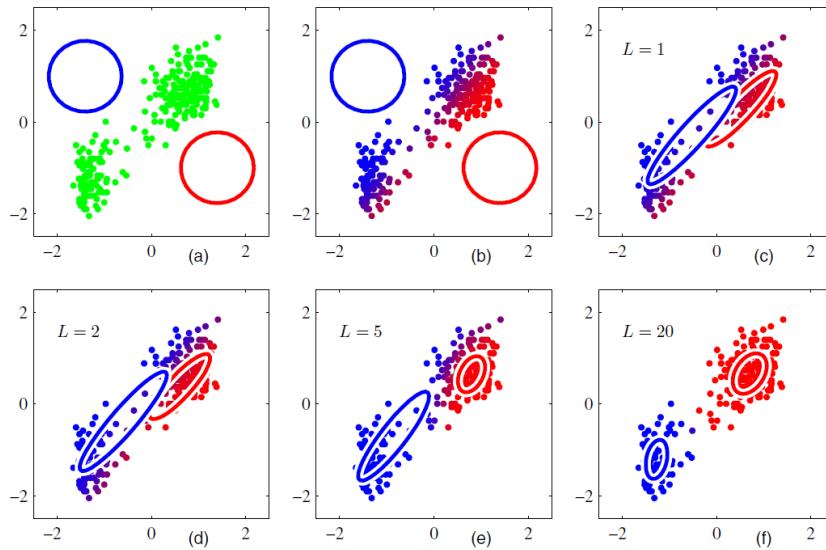
$$\ln p(X \mid \vec{\pi}, \vec{\mu}, \Sigma) = \sum_{n=1}^N \ln \left\{ \sum_{k=1}^K \pi_k \mathcal{N}(\vec{x}_n \mid \vec{\mu}_k, \Sigma_k) \right\}$$

Verifichiamo la convergenza dei parametri o del logaritmo della likelihood. Se il criterio di convergenza non è soddisfatto torniamo al punto 2.

Innanzitutto scegliamo i valori iniziali per le medie, le covarianze e i coefficienti di miscelazione. Quindi alterniamo i due passaggi E-Step ed M-Step.

Nell'E-Step utilizziamo i valori correnti dei parametri per valutare le probabilità posteriori (o responsabilità). Perciò utilizziamo queste probabilità nell'M-Step, per stimare nuovamente le medie, le covarianze e i coefficienti di miscelazione.

Consideriamo l'algoritmo convergente quando la variazione della funzione log likelihood, o in alternativa dei parametri, scende al di sotto di una certa soglia.



La figura mostra l'implementazione dell'algoritmo EM per il dataset “Old Faithful”.

In questo caso si utilizza una miscela di due Gaussiane, con i centri inizializzati utilizzando gli stessi valori dell'algoritmo K-Means e con le matrici di precisione inizializzate in modo da essere proporzionali alla matrice unitaria.

Il grafico (a) mostra i punti in verde, insieme alla configurazione iniziale del modello di miscela in cui i contorni di una deviazione standard per le due componenti Gaussiane sono rappresentati dai cerchi rosso e blu.

Il grafico (b) mostra il risultato della fase iniziale del passo E, in cui ogni punto è rappresentato utilizzando una proporzione di colore blu pari alla probabilità posteriori di essere stato generato dalla componente blu e una corrispondente proporzione di colore rosso data dalla probabilità posteriori di essere stato generato dalla componente rossa. **N.B:** i punti in viola rappresentano i punti di indecisione che possono appartenere ad entrambe le classi.

Il grafico (c) mostra il risultato ottenuto dopo il primo passo M, in cui la media della Gaussiana blu si è spostata verso il centro di massa del colore blu.

Analogamente, la media della Gaussiana rossa si è spostata verso il centro di massa del colore rosso.

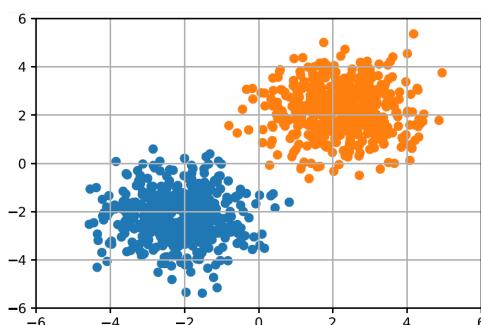
I grafici (d), (e) ed (f) mostrano i risultati dopo 2, 5 e 20 iterazioni complete dell'algoritmo EM. In particolare, nel grafico (f) l'algoritmo è vicino alla convergenza.

Osserviamo che l'algoritmo EM richiede molte più iterazioni per raggiungere la convergenza (approssimata) rispetto all'algoritmo K-Means e che ogni ciclo richiede significativamente più calcoli.

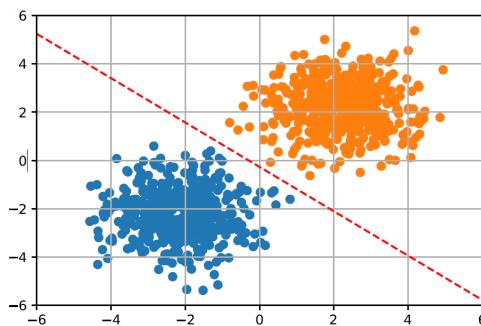
N.B: il GMM può essere pensato come una sorta di Soft K-Means.

7.3 Principal Component Analysis

Supponiamo di avere un problema di classificazione per un dataset i cui dati sono distribuiti nel seguente modo:

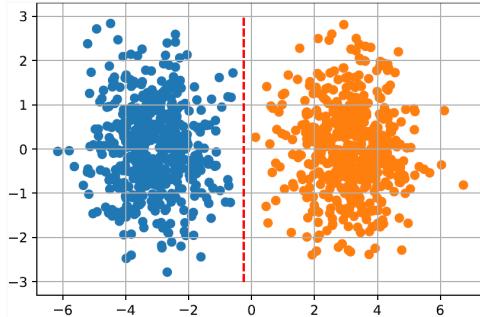


In precedenza abbiamo analizzato e compreso il modo per addestrare un classificatore per dataset linearmente separabili:

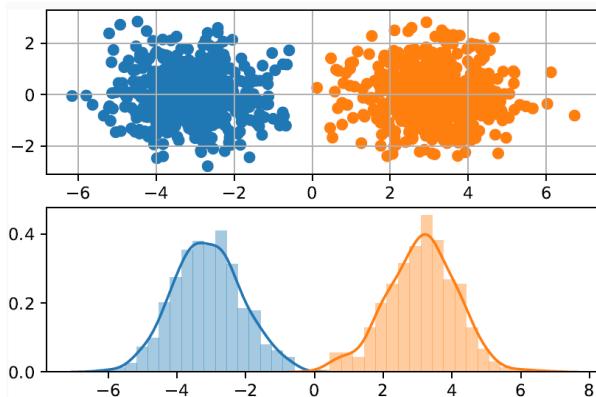


In particolare, nello spazio delle feature originale necessitiamo di entrambe le caratteristiche per definire il discriminante che divide le due classi.

Quindi vogliamo vedere se possiamo trasformare questo spazio in modo che le feature siano decorrelate. Ad esempio potremmo ruotare lo spazio delle feature in modo tale da far allineare la direzione dei dati con gli assi. In questo modo otteniamo un problema più semplice da risolvere, poiché la funzione discriminante necessita di una sola caratteristica:



Abbiamo quindi trasformato un problema bidimensionale in un problema monodimensionale.



Vogliamo quindi definire delle tecniche per proiettare i dati su sottospazi dello spazio originale accuratamente scelti per preservare le qualità desiderate.

La **Principal Component Analysis (PCA)** è una tecnica utilizzata per applicazioni quali la riduzione della dimensionalità, la compressione dei dati con perdita, l'estrazione di feature e la visualizzazione dei dati. Tale tecnica è anche conosciuta come *Karhunen-Loëve transform*.

L'obiettivo della PCA è quello di trovare una proiezione ortogonale su un sottospazio dimensionale inferiore. In particolare esistono due definizioni equivalenti di PCA che generano lo stesso algoritmo:

1. La PCA può essere definita come la proiezione ortogonale dei dati su uno spazio lineare dimensionale inferiore, detto *sottospazio principale (principal subspace)*, tale che la varianza dei dati proiettati sia massimizzata.
2. La PCA può essere definita come la proiezione lineare che minimizza la distanza quadratica media tra i punti e le loro proiezioni, ovvero minimizza la somma dei quadrati degli errori di proiezione.

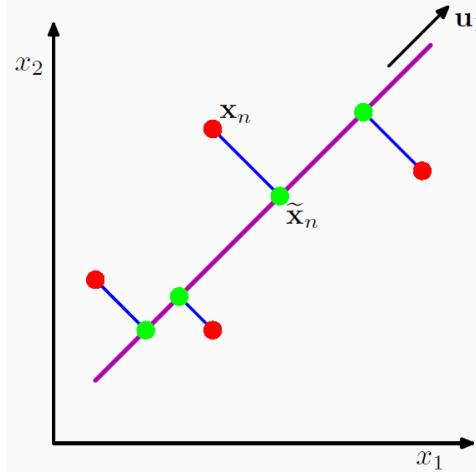


Figura 7.4: L’obiettivo della PCA è quello di trovare un sottospazio dimensionale di minore dimensionalità, noto come sottospazio principale (indicato dalla linea viola), tale che la proiezione ortogonale dei dati (indicati con i punti rossi) sul sottospazio massimizzi la varianza dei punti proiettati (indicati con i punti verdi).

Una definizione alternativa di PCA si basa sulla minimizzazione della somma dei quadrati degli errori di proiezione (indicata dalle linee blu).

Analizziamo singolarmente queste definizioni.

7.3.1 Formulazione della massima varianza

Consideriamo un dataset di osservazioni non etichettate $\mathcal{D} = \{\vec{x}_1, \dots, \vec{x}_N\}$ tale che $\vec{x}_n \in \mathbb{R}^D$ (ovvero sono variabili Euclidee con dimensionalità D).

Il nostro obiettivo è quello di trovare un sottospazio di dimensionalità $M < D$ in cui massimizziamo la varianza dei dati proiettati.

N.B: per adesso assumiamo che il valore M sia noto.

Innanzitutto consideriamo la proiezione su un sottospazio unidimensionale ($M = 1$).

Possiamo definire la direzione di questo sottospazio utilizzando un vettore

D -dimensionale \vec{u}_1 che assumiamo abbia norma unitaria (ovvero $\vec{u}_1^T \vec{u}_1 = 1$). Questo perché siamo interessati solo alla direzione di \vec{u}_1 e non alla sua grandezza.

Proiettiamo ogni punto \vec{x}_n su un valore scalare $\vec{u}_1^T \vec{x}_n$. Allora la media dei dati proiettati sarà $\vec{u}_1^T \bar{x}$, dove \bar{x} rappresenta la media del dataset ed è data da:

$$\bar{x} = \frac{1}{N} \sum_{n=1}^N \vec{x}_n$$

Quindi possiamo riscrivere la media dei dati proiettati come:

$$\vec{u}_1^T \bar{x} = \vec{u}_1^T \left(\frac{1}{N} \sum_{n=1}^N \vec{x}_n \right)$$

Inoltre, la varianza dei dati proiettati sarà:

$$\frac{1}{N} \sum_{n=1}^N (\vec{u}_1^T \vec{x}_n - \vec{u}_1^T \bar{x})^2 = \vec{u}_1^T S \vec{u}_1$$

dove S rappresenta la matrice di covarianza definita come:

$$S = \frac{1}{N} \sum_{n=1}^N (\vec{x}_n - \bar{x})(\vec{x}_n - \bar{x})^T$$

Perciò massimizziamo la varianza proiettata $\vec{u}_1^T S \vec{u}_1$ rispetto ad \vec{u}_1 .

N.B: tale massimizzazione deve essere vincolata per evitare che $\|\vec{u}_1\| \rightarrow \infty$. Questo vincolo deriva dalla condizione di normalizzazione $\vec{u}^T \vec{u} = 1$.

Per far rispettare questo vincolo, costruiamo un Lagrangiano ed introduciamo un moltiplicatore di Lagrange λ . Perciò il problema di ottimizzazione diventa quello di massimizzare:

$$\vec{u}_1^T S \vec{u}_1 + \lambda(1 - \vec{u}_1^T \vec{u}_1)$$

Ponendo la derivata rispetto a \vec{u}_1 uguale a zero otteniamo:

$$S \vec{u}_1 = \lambda \vec{u}_1$$

In particolare \vec{u}_1 rappresenta un autovettore di S .

Se moltiplichiamo entrambi i membri per \vec{u}_1^T ed utilizziamo l'ipotesi $\vec{u}_1^T \vec{u}_1 = 1$, si vede che la varianza proiettata è data da:

$$\vec{u}_1^T S \vec{u}_1 = \lambda_1$$

Ovvero otteniamo che la varianza è massima quando \vec{u}_1 è uguale all'autovettore che possiede l'autovalore λ_1 maggiore. Quindi per ottenere il primo sottospazio principale dobbiamo trovare l'autovettore corrispondente al massimo autovalore λ_1 .

Questo autovettore è noto come *prima componente principale*.

Se vogliamo un ulteriore sottospazio principale, dobbiamo trovare \vec{u}_2 che massimizza la varianza tra tutti gli \vec{u}_2 ortogonali ad \vec{u}_1 . Tale autovettore è detto *seconda componente principale*.

In generale possiamo definire ulteriori componenti principali, scegliendo per ogni nuova direzione quella che massimizza la varianza proiettata tra tutte le possibili direzioni ortogonali a quelle già considerate.

Quindi, se consideriamo il caso generale di un sottospazio di proiezione M -dimensionale, allora la proiezione lineare ottimale per la quale la varianza dei dati proiettati è massimizzata è definita dagli autovettori $\vec{u}_1, \dots, \vec{u}_M$ della matrice di covarianza dei dati S corrispondenti agli M autovalori maggiori $\lambda_1, \dots, \lambda_M$.

Perciò per trovare il sottospazio principale M -dimensionale dobbiamo:

1. Calcolare la media \bar{x} e la matrice di covarianza S sul dataset \mathcal{D} .
2. Trovare la matrice $U \in \mathbb{R}^{D \times M}$ le cui colonne sono gli M autovettori corrispondenti agli M autovalori più grandi di S .
3. Proiettare i dati utilizzando $U^T(X - \bar{x})$.

7.3.2 Formulazione del minimo errore

Analizziamo una formulazione della PCA alternativa basata sulla minimizzazione dell'errore di proiezione. A tale scopo introduciamo un insieme ortonormale completo di vettori base D -dimensionali $\{\vec{u}_1, \dots, \vec{u}_D\}$ tali che:

$$\vec{u}_i^T \vec{u}_j = \begin{cases} 1 & \text{se } i = j \\ 0 & \text{altrimenti} \end{cases}$$

Poiché questa base è completa, allora ogni dato \vec{x}_n può essere rappresentato esattamente da una combinazione lineare dei vettori base, ovvero:

$$\vec{x}_n = \sum_{i=1}^D \alpha_{ni} \vec{u}_i$$

dove i coefficienti α_{ni} saranno diversi per ogni dato \vec{x}_n .

Effettuiamo il prodotto interno con \vec{u}_j su entrambi i lati e, facendo uso della proprietà di ortonormalità, otteniamo che $\alpha_{nj} = \vec{x}_n^T \vec{u}_j$. Quindi possiamo scrivere:

$$\vec{x}_n = \sum_{i=1}^D (\vec{x}_n^T \vec{u}_i) \vec{u}_i$$

Il nostro obiettivo è quello di approssimare ogni punto dati \vec{x}_n utilizzando solamente M vettori base (ovvero vogliamo realizzare una rappresentazione con un numero ristretto $M < D$ di variabili corrispondenti ad una proiezione su un sottospazio di dimensionalità inferiore). In particolare possiamo approssimare ogni punto dati \vec{x}_n come:

$$\tilde{\vec{x}}_n = \sum_{i=1}^M z_{ni} \vec{u}_i + \sum_{i=M+1}^D b_i \vec{u}_i$$

dove gli $\{z_{ni}\}$ dipendono da un particolare punto dati, mentre i $\{b_i\}$ sono costanti uguali per tutti i punti dati.

Possiamo quindi scegliere $\{\vec{u}_i\}$, $\{z_{ni}\}$ e $\{b_i\}$ in modo da minimizzare la distorsione introdotta dalla riduzione della dimensionalità. Come misura di distorsione utilizziamo la distanza al quadrato tra il punto dati originale \vec{x}_n e la sua approssimazione $\tilde{\vec{x}}_n$ sul dataset.

Quindi il nostro obiettivo risulta essere quello di minimizzare:

$$J = \frac{1}{N} \sum_{n=1}^N \|\vec{x}_n - \tilde{\vec{x}}_n\|^2$$

Consideriamo innanzitutto la minimizzazione rispetto alle quantità $\{z_{ni}\}$.

Sostituendo per $\tilde{\vec{x}}_n$, ponendo a zero la derivata rispetto a z_{nj} e facendo uso delle condizioni di ortonormalità otteniamo:

$$z_{nj} = \vec{x}_n^T \vec{u}_j \quad j = 1, \dots, M$$

Analogamente, minimizziamo rispetto alla quantità $\{b_i\}$ ponendo a zero la derivata di J rispetto a b_j e facendo nuovamente uso delle relazioni di ortonormalità ottenendo:

$$b_j = \vec{x}^T \vec{u}_j \quad j = M+1, \dots, D$$

Sostituendo z_{ni} e b_i nell'espansione della base completa ed utilizzando $\alpha_{nj} = \vec{x}_n^T \vec{u}_j$ otteniamo:

$$\vec{x}_n - \tilde{\vec{x}}_n = \sum_{i=M+1}^D \{(\vec{x}_n - \vec{x})^T \vec{u}_i\} \vec{u}_i$$

da cui si vede che il vettore spostamento da \vec{x}_n ad $\tilde{\vec{x}}_n$ giace nello spazio ortogonale al sottospazio principale (poiché, come possiamo osservare nella figura dell'esempio iniziale, tale vettore spostamento è una combinazione lineare di \vec{u}_i per $i = M+1, \dots, D$).

Otteniamo quindi un'espressione per la misura di distorsione J come funzione puramente di \vec{u}_i nella seguente forma:

$$J = \frac{1}{N} \sum_{n=1}^N \sum_{i=M+1}^D (\vec{x}_n \vec{u}_i - \bar{x}^T \vec{u}_i)^2 = \sum_{i=M+1}^D \vec{u}_i^T S \vec{u}_i$$

Rimane il compito di minimizzare J rispetto a $\{\vec{u}_i\}$, che deve essere una minimizzazione vincolata (altrimenti otterremmo un risultato $\vec{u}_i = 0$). I vincoli derivano dalle condizioni di ortonormalità e la soluzione è espressa in termini di espansione degli autovettori della matrice di covarianza.

Per semplicità consideriamo innanzitutto il caso di uno spazio dati bidimensionale $D = 2$ e di un sottospazio principale unidimensionale $M = 1$. Allora dobbiamo scegliere una direzione \vec{u}_2 in modo da minimizzare $J = \vec{u}_2^T S \vec{u}_2$, soggetta al vincolo di normalizzazione $\vec{u}_2^T \vec{u}_2 = 1$. Utilizziamo un moltiplicatore di Lagrange λ_2 per imporre tale vincolo. Allora otteniamo il seguente problema di minimizzazione:

$$\tilde{J} = \vec{u}_2^T S \vec{u}_2 + \lambda_2 (1 - \vec{u}_2^T \vec{u}_2)$$

Ponendo a zero la derivata rispetto a \vec{u}_2 si ottiene $S \vec{u}_2 = \lambda_2 \vec{u}_2$ e quindi \vec{u}_2 è un autovettore di S con autovalore λ_2 . Pertanto, qualsiasi autovettore definisce un punto stazionario della misura di distorsione.

Per trovare il valore minimo di J sostituiamo la soluzione di \vec{u}_2 nella misura di distorsione, ottenendo $J = \lambda_2$.

Si ottiene quindi il valore minimo di J scegliendo \vec{u}_2 come autovalore corrispondente al più piccolo dei due autovalori. Pertanto, dovremmo scegliere il sottospazio principale in modo che sia allineato con l'autovettore che ha l'autovalore maggiore. Quindi per minimizzare la distanza quadratica media di proiezione, dovremmo scegliere il sottospazio delle componenti principali in modo tale che passi per la media dei punti e che sia allineato con le direzioni della varianza massima.

La soluzione generale alla minimizzazione di J per un D arbitrario ed un $M < D$ arbitrario si ottiene scegliendo gli $\{\vec{u}_i\}$ come autovettori della matrice di covarianza data da:

$$S \vec{u}_i = \lambda_i \vec{u}_i \quad i = 1, \dots, D$$

dove gli autovettori \vec{u}_i sono scelti come ortonormali.

Il valore corrispondente della misura di distorsione è quindi dato da:

$$J = \sum_{i=M+1}^D \lambda_i$$

che rappresenta semplicemente la sommatoria tra gli autovalori degli autovettori ortogonali al sottospazio principale.

Quindi otteniamo nuovamente che il migliore sottospazio principale in M dimensioni corrisponde agli M autovettori di S più piccoli.

7.3.3 Applicazioni della PCA

Compressione dei dati

Illustriamo l'utilizzo della PCA per la compressione dei dati considerando il dataset del riconoscimento delle cifre.

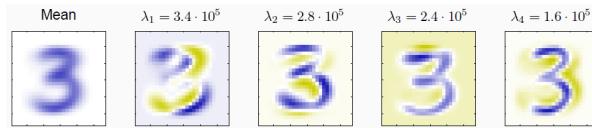


Figura 7.5: Illustrazione del vettore media \bar{x} del dataset di riconoscimento delle cifre con i primi quattro autovettori PCA $\vec{u}_1, \dots, \vec{u}_4$ insieme ai corrispondenti autovalori.

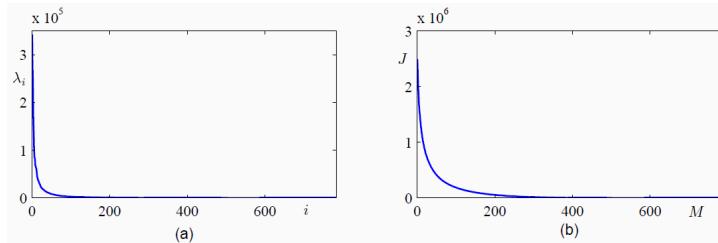


Figura 7.6: (a) Grafico dello spettro degli autovalori relativo al dataset del riconoscimento delle cifre. (b) Grafico della somma degli autovalori scartati, che rappresenta la distorsione J della somma dei quadrati introdotta dalla proiezione dei dati su un sottospazio delle componenti principali di dimensionalità M .

La misura di distorsione J associata alla scelta di un particolare valore di M è data dalla somma degli autovalori da $M+1$ fino a D .

Utilizzando $\bar{x} = \sum_{i=1}^D (\bar{x}^T \vec{u}_i) \vec{u}_i$ possiamo scrivere l'approssimazione PCA di \vec{x}_n come:

$$\tilde{x}_n = \sum_{i=1}^M (\tilde{x}_n^T \vec{u}_i) \vec{u}_i + \sum_{i=M+1}^D (\tilde{x}_n^T \vec{u}_i) \vec{u}_i = \bar{x} + \sum_{i=1}^M (\tilde{x}_n^T \vec{u}_i - \bar{x}^T \vec{u}_i) \vec{u}_i$$

Questa approssimazione rappresenta una compressione del dataset poiché per ogni punto dati abbiamo sostituito il vettore \vec{x}_n D -dimensionale con un vettore M -dimensionale avente componenti $(\tilde{x}_n^T \vec{u}_i - \bar{x}^T \vec{u}_i)$.

N.B.: più piccolo è il valore di M e maggiore sarà il grado di compressione.

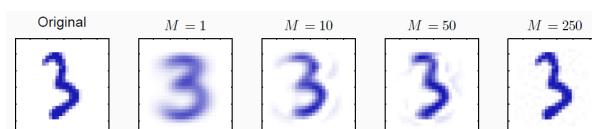


Figura 7.7: Esempio di ricostruzione PCA dei punti dati per il dataset di riconoscimento delle cifre. In particolare la ricostruzione PCA è ottenuta mantenendo M componenti principali per vari valori di M . Osserviamo che all'aumentare di M la ricostruzione diventa più accurata. In particolare, la ricostruzione diventa perfetta se $M = D = 28 \times 28 = 768$.

Data whitening

Un'altra comune applicazione della PCA riguarda il pre-processing dei dati. In questo caso l'obiettivo non è la riduzione della dimensionalità ma piuttosto la trasformazione del dataset in modo tale da standardizzare alcune delle sue proprietà. Questo permette di applicare con successo gli algoritmi di riconoscimento del pattern al dataset.

Tipicamente si svolge questa operazione di standardizzazione dei dati durante il preprocessing quando le variabili originali possiedono differenti unità di misura o presentano una varianza significativa.

Questa operazione di standardizzazione prevede di centrare i dati, sottraendone la media (ovvero si traslano i dati sottraendo la media ad ogni punto), e di dividere ogni dimensione originale per la sua deviazione standard (in modo da ruotare i dati).

In particolare le componenti della matrice di covarianza per i dati standardizzati sono definite come:

$$\rho_{ij} = \frac{1}{N} \sum_{n=1}^N \frac{(x_{ni} - \bar{x}_i)}{\sigma_i} \frac{(x_{nj} - \bar{x}_j)}{\sigma_j}$$

dove σ_i rappresenta la varianza di x_i .

Tale matrice di covarianza è detta **matrice di correlazione** dei dati originali.

N.B.: $\rho_{ij} = 1$ se due componenti x_i ed x_j di un dato sono perfettamente correlate, mentre $\rho_{ij} = 0$ se non sono correlate.

Questo risultato è utile per tenere conto di dimensioni di input diversamente scalate. Tuttavia, la rappresentazione risultante ha ancora dimensioni correlate.

Utilizzando la PCA possiamo realizzare una normalizzazione più sostanziale dei dati, assegnandone una media nulla ed una covarianza unitaria, così da rendere la variabili non correlate.

A tale scopo riscriviamo l'equazione dell'autovettore $S\vec{u}_i = \lambda_i \vec{u}_i$ nella seguente forma:

$$SU = UL$$

dove L rappresenta una matrice diagonale di dimensione $D \times D$ con elementi λ_i ed U rappresenta una matrice ortogonale di dimensione $D \times D$ le cui colonne sono date dagli elementi \vec{u}_i .

Quindi per ogni punto \vec{x}_n possiamo definire la seguente trasformazione (che permette di scalare e ruotare i dati centrati):

$$y_n = L^{-1/2} U^T (\vec{x}_n - \bar{x})$$

Questa operazione è conosciuta come **data whitening** o **sphering**.

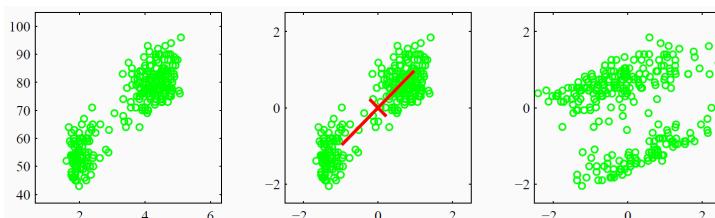


Figura 7.8: Illustrazione degli effetti del pre-processing lineare applicato al dataset Old Faithful. Il grafico a sinistra mostra i dati originali. Il grafico al centro mostra il risultato della standardizzazione delle singole variabili a media nulla e varianza unitaria. Il grafico a destra mostra il risultato della *data whitening* per ottenere media zero e covarianza unitaria.

Osserviamo che gli assi principali per il secondo e terzo grafico risultano essere normalizzati su un'intervalllo $\pm \lambda_i^{-1/2}$.

È interessante confrontare la PCA con il discriminante lineare di Fisher. Entrambi i metodi possono essere considerati come tecniche di riduzione della dimensionalità. Tuttavia, la PCA non è supervisionata e dipende soltanto dai valori \vec{x}_n , mentre il discriminante lineare di Fisher utilizza anche informazioni sull'etichettatura dei dati. Questa differenza è evidenziata nel seguente grafico:

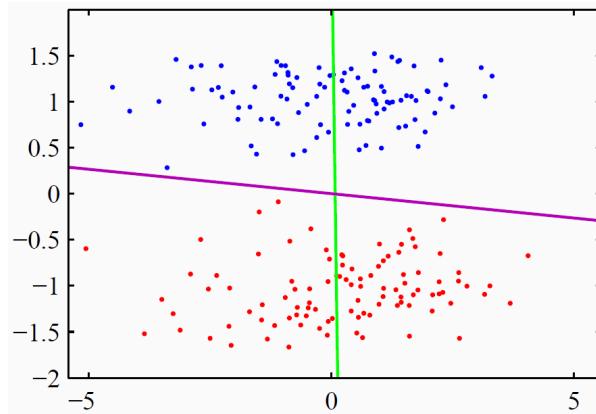


Figura 7.9: Comparazione tra la PCA ed il discriminante lineare di Fisher per la riduzione lineare della dimensionalità. In questo caso i dati in due dimensioni, appartenenti a due classi indicate in rosso e in blu, devono essere proiettati su un'unica dimensione.

PCA sceglie la direzione della massima varianza (mostrata dalla curva viola) che porta ad una forte sovrapposizione delle classi, mentre il discriminante lineare di Fisher tiene conto delle etichette delle classi e porta a una proiezione sulla curva verde che offre una migliore separazione delle classi.

N.B.: un'altra applicazione comune della PCA è la visualizzazione dei dati. In questo caso ogni punto viene proiettato su un sottospazio principale bidimensionale ($M = 2$) in modo tale che ogni punto \vec{x}_n possa essere rappresentato su un piano cartesiano. Quindi le coordinate di \vec{x}_n vengono trasformate in $\vec{x}_n^T \vec{u}_1$ ed $\vec{x}_n^T \vec{u}_2$, dove \vec{u}_1 ed \vec{u}_2 sono gli autovettori corrispondenti ai primi due autovalori maggiori.

N.B.: il costo computazionale della decomposizione dell'intero autovettore per una matrice di dimensioni $D \times D$ è $O(D^3)$. Questo garantisce un'ottima scalabilità della PCA.

Esistono numerose varianti della PCA come la *PCA probabilistica* (**PPCA**), che rappresenta un modello applicato alle variabili latenti continue che inserisce la PCA in un quadro Bayesiano, o il *Kernel PCA* (**KPCA**), che rappresenta una versione non lineare della PCA permettendo di analizzare sottospazi principali non lineari dei dati.

Capitolo 8

ENSEMBLE MODELS

I modelli di regressione e di classificazione che abbiamo analizzato risultano essere modelli singolari, cioè adattiamo un modello e poi facciamo previsioni utilizzando un solo modello.

Spesso possiamo ottenere prestazioni migliori combinando insieme più modelli, invece di utilizzare un singolo modello.

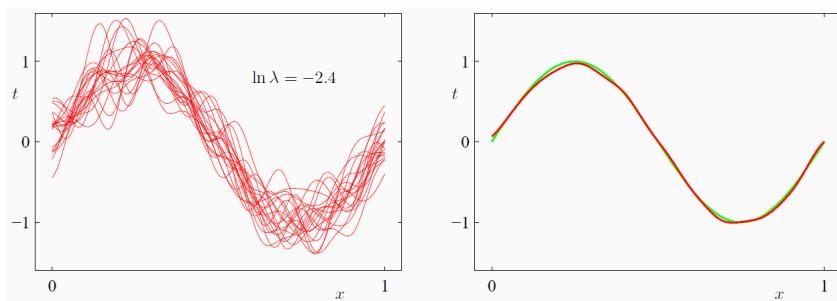
Ad esempio potremmo addestrare M modelli diversi e poi fare le previsioni utilizzando la media delle previsioni fatte da ciascun modello, oppure potremmo addestrare i modelli in modo sequenziale e incoraggiare i modelli successivi a compensare gli errori commessi dai modelli precedenti, o ancora potremmo addestrare più modelli e poi selezionare il migliore per le fare previsioni di un dato \vec{x} .

Questi modelli sono solitamente chiamati *modelli d'insieme* (**Ensemble Models**) poiché fanno previsioni basate su un insieme di modelli addestrati invece che su un solo modello.

8.1 Committees e Bagging

Il modo più semplice per costruire un **committees** (*comitato*) è quello di calcolare la media delle previsioni di un insieme di modelli. Tale procedura può essere motivata a partire dalle considerazioni fatte sull'importanza del compromesso tra bias e varianza, che scomponete l'errore dovuto a un modello in una componente di bias, che deriva dalle differenze tra il modello e la funzione reale da prevedere, e nella componente di varianza, che rappresenta la sensibilità del modello ai singoli dati.

Dalla seguente figura ricordiamo che addestrando più polinomi e facendo la media delle funzioni risultanti, il contributo derivante dalla varianza tende ad annullarsi portando ad un miglioramento delle previsioni. In particolare, facendo la media di un insieme di modelli con basso bias (corrispondenti a polinomi di ordine superiore), otteniamo previsioni accurate per la funzione sinusoidale da cui sono stati generati i dati.



In pratica, però, disponiamo di un solo dataset e quindi dobbiamo trovare un modo per

introdurre la variabilità tra i diversi modelli all'interno del committee. Un approccio è quello di utilizzare i dataset *bootstrap*.

In particolare il metodo bootstrap prevede di creare un nuovo dataset \mathcal{D}_B estraendo N punti a caso dal dataset di partenza \mathcal{D} con sostituzione, ovvero in modo che alcuni punti di \mathcal{D} possono essere replicati in \mathcal{D}_B mentre altri punti di \mathcal{D} possono essere assenti in \mathcal{D}_B . Questo processo può essere ripetuto M volte per generare M dataset, ciascuno di dimensione N ottenuto campionando dal dataset originale \mathcal{D} . Ognuno di questi dataset quindi riflette la distribuzione di \mathcal{D} , ma risulta essere incompleto.

Applichiamo quindi tale metodo bootstrap. Consideriamo un problema di regressione in cui cerchiamo di predire il valore di un singolo target continuo e supponiamo di generare M dataset bootstrap. Quindi addestriamo un modello $y_m(\vec{x})$, con $m = 1, \dots, M$, per ognuno di questi dataset bootstrap.

Allora il modello committee può essere ottenuto facendo la media di questi M modelli base:

$$y_{\text{COM}}(\vec{x}) = \frac{1}{M} \sum_{m=1}^M y_m(\vec{x})$$

Questa procedura è nota come aggregazione bootstrap o **bagging**.

Chiamiamo con $h(\vec{x})$ la vera funzione di regressione che genera il dataset \mathcal{D} che stiamo cercando di prevedere. Allora l'output di ciascuno dei modelli può essere scritto come il valore di tale funzione più un errore:

$$y_m(\vec{x}) = h(\vec{x}) + \varepsilon_m(\vec{x})$$

L'errore quadratico medio di ogni modello assume quindi la seguente forma:

$$\mathbb{E}_{\vec{x}}[\{y_m(\vec{x}) - h(\vec{x})\}^2] = \mathbb{E}_{\vec{x}}[\varepsilon_m(\vec{x})^2]$$

L'errore medio commesso dai singoli modelli è dato da:

$$E_{\text{AV}} = \frac{1}{M} \sum_{m=1}^M \mathbb{E}_{\vec{x}}[\varepsilon_m(\vec{x})^2]$$

Analogamente, l'errore atteso dal committee è dato da:

$$\begin{aligned} E_{\text{COM}} &= \mathbb{E}_{\vec{x}} \left[\left\{ \frac{1}{M} \sum_{m=1}^M [y_m(\vec{x}) - h(\vec{x})] \right\}^2 \right] \\ &= \mathbb{E}_{\vec{x}} \left[\left\{ \frac{1}{M} \sum_{m=1}^M \varepsilon_m(\vec{x}) \right\}^2 \right] \end{aligned}$$

N.B.: il quadrato della sommatoria risulta essere un calcolo abbastanza complesso, perciò dobbiamo fare delle assunzioni per semplificare.

Ipotizziamo che gli errori abbiano media nulla e che non siano correlati, ovvero che:

$$\begin{aligned} \mathbb{E}_{\vec{x}}[\varepsilon_m(\vec{x})] &= 0 \\ \mathbb{E}_{\vec{x}}[\varepsilon_m(\vec{x}) \varepsilon_l(\vec{x})] &= 0 \quad m \neq l \end{aligned}$$

Allora otteniamo:

$$E_{\text{COM}} = \frac{1}{M} E_{\text{AV}}$$

Questo risultato suggerisce che possiamo ridurre l'errore medio dei singoli modelli di un fattore $\frac{1}{M}$ semplicemente facendo la media degli M modelli.

Purtroppo questo risultato dipende dal presupposto fondamentale che gli errori dovuti ai singoli modelli siano non correlati. In genere però, gli errori risultano essere molto correlati e la riduzione dell'errore complessivo è generalmente piccola.

Tuttavia è possibile dimostrare che l'errore del committee non supera l'errore del modello atteso, ovvero $E_{\text{COM}} \leq E_{\text{AV}}$.

Al fine di ottenere miglioramenti più significativi, si ricorre a una tecnica più sofisticata per la costruzione di committee, nota come **boosting**.

8.2 Boosting

Il **boosting** è una tecnica potente che permette di combinare più classificatori base per produrre una forma di committee con prestazioni significativamente migliori.

In particolare descriviamo *AdaBoost* che rappresenta la forma più diffusa di algoritmo di boosting. La tecnica di boosting è stata originariamente progettata per risolvere problemi di classificazione, ma può essere estesa anche ai problemi di regressione.

La differenza principale tra il boosting e i metodi committee (come il bagging) è che in questo caso i classificatori di base vengono addestrati in sequenza ed ogni classificatore di base viene addestrato utilizzando una forma pesata del dataset, in cui il peso associato a ciascun dato dipende dalle prestazioni dei classificatori precedenti.

In particolare, i punti che sono stati classificati in modo errato da uno dei classificatori di base, vengono utilizzati per addestrare il classificatore successivo della sequenza.

Una volta che tutti i classificatori sono stati addestrati, le loro predizioni vengono combinate attraverso uno schema di voto a maggioranza pesata.

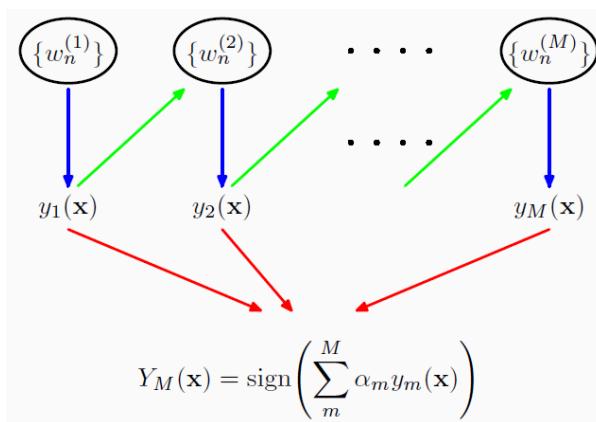


Figura 8.1: Illustrazione schematica del boosting. Ogni classificatore di base $y_m(\vec{x})$ viene addestrato su una forma pesata del dataset (frecce blu) in cui i pesi $w_n^{(m)}$ dipendono dalla performance del precedente classificatore di base $y_{m-1}(\vec{x})$ (frecce verdi). Una volta addestrati tutti i classificatori di base, questi vengono combinati per realizzare il classificatore finale $Y_M(\vec{x})$ (frecce rosse).

Descriviamo l'idea dell'algoritmo di AdaBoost.

Consideriamo un problema di classificazione binario per un dataset

$$\mathcal{D} = \{(\vec{x}_1, t_1), \dots, (\vec{x}_N, t_N)\} \text{ con } t_n \in \{-1, 1\}.$$

Ad ogni punto \vec{x}_n associamo un peso w_n inizializzato ad $\frac{1}{N}$. Supponiamo di avere una procedura per addestrare un classificatore di base con campioni pesati per generare una funzione $y(\vec{x}) \in \{-1, 1\}$.

Ad ogni iterazione, AdaBoost addestra un nuovo classificatore utilizzando un dataset i cui pesi vengono aggiustati in base alle prestazioni del classificatore precedentemente addestrato, in modo da dare maggior peso ai punti erroneamente classificati.

Una volta addestrati M classificatori di base, questi vengono combinati per formare un committee con coefficienti che attribuiscono pesi differenti ad ogni classificatore.

N.B: nel boosting i classificatori di base sono spesso chiamati **weak learners**.

Formalizziamo questo algoritmo:

1. Inizializziamo i coefficienti di ponderazione (**weighting coefficients**) dei dati $\{w_n\}$ ponendo $w_n^{(1)} = \frac{1}{N}$ per $n = 1, \dots, N$.
2. Per $m = 2, \dots, M$:
 - a. Adattiamo un classificatore $y_m(\vec{x})$ ai dati di addestramento minimizzando la seguente funzione d'errore pesato:

$$J_m = \sum_{n=1}^N w_n^{(m)} I(y_m(\vec{x}_n) \neq t_n)$$

dove $I(y_m(\vec{x}_n) \neq t_n)$ rappresenta la funzione indicatrice che vale 1 quando $y_m(\vec{x}_n) \neq t_n$ e 0 altrimenti.

- b. Valutiamo le quantità:

$$\varepsilon_m = \frac{\sum_{n=1}^N w_n^{(m)} I(y_m(\vec{x}_n) \neq t_n)}{\sum_{n=1}^N w_n^{(m)}}$$

dove ε_m rappresenta il tasso di errore (ponderato) del classificatore base y_m . Usiamo tali quantità per valutare:

$$\alpha_m = \ln \left\{ \frac{1 - \varepsilon_m}{\varepsilon_m} \right\}$$

- c. Aggiorniamo i coefficienti di ponderazione dei dati:

$$w_n^{(m+1)} = w_n^{(m)} \exp\{\alpha_m I(y_m(\vec{x}_n) \neq t_n)\}$$

N.B: i coefficienti di ponderazione $w_n^{(m)}$ incrementano per i punti classificati in modo errato e diminuiscono per i punti classificati correttamente. I classificatori successivi sono quindi costretti a porre maggiore enfasi sui punti che sono stati classificati in modo errato dai classificatori precedenti.

3. Facciamo le previsioni utilizzando il modello finale definito da:

$$Y_M(\vec{x}) = \text{sign} \left(\sum_{m=1}^M \alpha_m y_m(\vec{x}) \right)$$

Vediamo che il primo classificatore di base $y_1(\vec{x})$ viene addestrato utilizzando i coefficienti di ponderazione $w_n^{(1)}$ che sono tutti uguali. Questo procedura permette di addestrare un singolo classificatore.

N.B.: i coefficienti di ponderazione α_m assegnano un peso maggiore ai classificatori più accurati nel calcolo del classificatore finale.

N.B.: AdaBoost funziona molto bene nella pratica, soprattutto con molti weak learners.

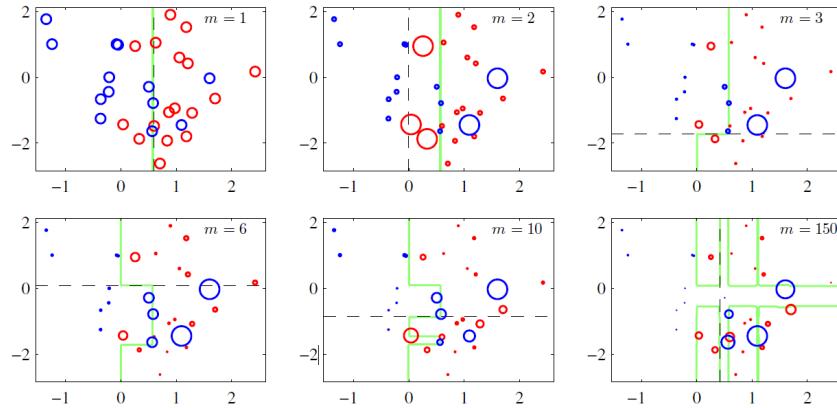


Figura 8.2: Illustrazione del *boosting* in cui i classificatori base consistono in semplici soglie applicate ad una delle variabili di input (rappresentate sugli assi).

Pertanto ogni classificatore base classifica un input in base al superamento della soglia di una delle caratteristiche di input e quindi si limita a suddividere lo spazio in due regioni separate da una superficie decisionale parallela a uno degli assi.

Ogni figura mostra il numero m di classificatori base addestrati finora, insieme al confine decisionale del classificatore base più recente (rappresentato dalla linea nera tratteggiata) e al confine decisionale combinato (rappresentato dalla linea verde).

Ogni punto è rappresentato da un cerchio il cui raggio indica il peso assegnato a quel punto nell'addestramento dell'ultimo classificatore base.

Ad esempio, vediamo che i punti classificati in modo errato dal classificatore base $m = 1$ hanno un peso maggiore durante l'addestramento del classificatore base $m = 2$.

8.3 Tree Models

I *modelli ad albero* (**Tree Models**) sono modelli semplici che funzionano per partizione, ovvero suddividono lo spazio di input in cuboidi allineati agli assi ed assegnano ad ogni partizione un singolo modello responsabile per quella regione.

Quindi possiamo considerare questa tecnica come una combinazione di modelli (Ensemble Models) in cui a turno un solo modello è responsabile per fare la previsione. Dato un punto \vec{x} , il processo di selezione di uno specifico modello può essere descritto da un processo decisionale sequenziale corrispondente alla traversata di un albero binario (che si divide in due rami ad ogni nodo). Quindi per selezionare il modello si attraversa l'albero in un processo decisionale sequenziale.

Concentriamoci su una particolare struttura ad albero detta **Classification and Regression Tree (CART)**.



ES:

Consideriamo il seguente partizionamento binario ricorsivo dello spazio di input insieme alla corrispondente struttura ad albero:

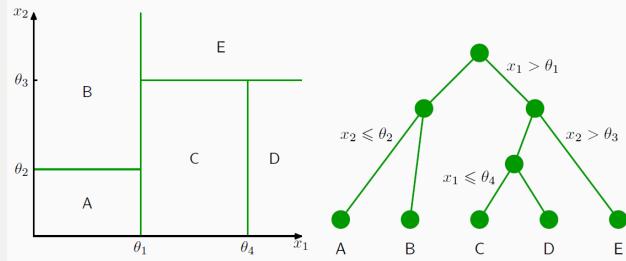


Figura 8.3: Nella figura di sinistra è illustrato uno spazio di input bidimensionale partizionato in cinque regioni utilizzando confini allineati agli assi.

Nella figura di destra è rappresentato l'albero binario corrispondente al partizionamento dello spazio di input.

In questo esempio, il primo passo divide l'intero spazio di input in due regioni a seconda che $x_1 \leq \theta_1$ o $x_1 > \theta_1$, dove θ_1 rappresenta un parametro del modello. In questo modo si creano due sottoregioni, ognuna delle quali può essere successivamente suddivisa in modo indipendente. Ad esempio, la regione $x_1 \leq \theta_1$ viene ulteriormente suddivisa a seconda che $x_2 \leq \theta_2$ o $x_2 > \theta_2$, dando origine alle regioni denominate *A* e *B*.

La suddivisione ricorsiva può essere descritta dall'attraversamento dell'albero binario mostrato in figura (a destra).

Per ogni nuovo input \vec{x} , determiniamo la regione di appartenenza seguendo un percorso che inizia dalla radice dell'albero e termina in una specifica foglia (che ne determina la regione), in base ai criteri decisionali di ciascun nodo.

All'interno di ogni regione, c'è un modello separato per prevedere la variabile target.

N.B.: una proprietà chiave dei modelli ad albero è che sono facilmente interpretabili dall'uomo perché corrispondono a una sequenza di decisioni binarie applicate alle singole variabili di input.

Per apprendere un modello di questo tipo da un dataset di addestramento dobbiamo determinare la struttura dell'albero, inclusa la variabile di input scelta ad ogni nodo per formare il criterio di divisione ed il valore del parametro di soglia θ_i per la divisione. Dobbiamo anche definire i valori delle variabili predittive all'interno di ogni regione.

Consideriamo innanzitutto un problema di regressione per un dataset $\mathcal{D} = \{(\vec{x}_1, t_1), \dots, (\vec{x}_N, t_N)\}$ di dimensione D , in cui l'obiettivo è prevedere una singola variabile target t continua.

Desideriamo partizionare lo spazio \mathbb{R}^D in modo tale che la stima del target in ogni partizione minimizzi l'errore quadratico. In particolare il valore ottimale della variabile predittiva all'interno di una determinata regione è dato semplicemente dalla media dei valori target t_n per tutti i punti \vec{x}_n appartenenti a quella regione.

Il vero problema riguarda come determinare la struttura dell'albero decisionale (ovvero come determinare le partizioni). In particolare il problema di determinare la struttura

ottimale per minimizzare l'errore della somma dei quadrati è di solito un problema computazionalmente intrattabile a causa del numero combinatorio di possibili soluzioni.

Possiamo invece utilizzare una politica ricorsiva *greedy* effettuata a partire da un singolo nodo radice, corrispondente all'intero spazio di input, ed aggiungendo un nodo alla volta per far crescere l'albero. Ad ogni passo avremo un certo numero di regioni candidate nello spazio di input che possono essere suddivise. Per ciascuna di queste regioni è possibile scegliere quale variabile di input D dividere, oltre al valore della soglia.

L'ottimizzazione della scelta della regione da suddividere e della scelta della variabile di input (insieme alla soglia) può essere fatta tenendo presente che per una data scelta della variabile di split e della soglia, la scelta ottimale della variabile predittiva è data dalla media locale dei dati.

Questa operazione viene ripetuta per tutte le possibili scelte della variabile da dividere e viene selezionata la variabile che genera il minore errore residuo della somma dei quadrati.

Possiamo schematizzare questa politica *greedy* nel seguente modo:

1. Partiamo da un singolo nodo radice corrispondente all'intero spazio di input.
2. Iteriamo su ciascuna delle D possibili variabili di divisione.
 - a. Consideriamo ogni possibile soglia per dividere i dati in due insiemi.
 - b. Selezionare la divisione che riduce al minimo la somma degli errori quadratici nelle suddivisioni.
3. Applichiamo ricorsivamente questa procedura ai figli del nodo.

Dobbiamo adesso decidere il criterio di arresto per questo algoritmo. La strategia comune consiste nel segmentare in modo eccessivo lo spazio di input così da costruire un albero di grandi dimensioni (ovvero vogliamo ottenere un albero profondo ed ampio) e successivamente sfoltire l'albero risultante per bilanciare la complessità e la minimizzazione dell'errore sui dati di addestramento.

Supponiamo di avere un albero T (ovvero una partizione) i cui nodi foglia sono indicizzati da $\tau = 1, \dots, |T|$, dove $|T|$ rappresenta il numero totale di nodi foglia. Indichiamo con \mathcal{R}_τ la partizione dello spazio di input corrispondente alla foglia τ . Allora la predizione ottimale per la regione \mathcal{R}_τ è data da:

$$y_\tau = \frac{1}{N_\tau} \sum_{\vec{x}_n \in \mathcal{R}_\tau} t_n$$

Mentre l'errore quadratico corrispondente è dato da:

$$Q_\tau(T) = \sum_{\vec{x}_n \in \mathcal{R}_\tau} \{t_n - y_\tau\}^2$$

Allora il criterio di potatura è dato da:

$$C(T) = \sum_{\tau=1}^{|T|} Q_\tau(T) + \lambda |T|$$

dove il parametro di regolarizzazione λ determina il compromesso tra l'errore residuo complessivo della somma dei quadrati e la complessità del modello, misurata dal numero $|T|$ di nodi foglia.

Consideriamo adesso un problema di classificazione. Allora il processo di crescita e di potatura dell'albero è simile a quello appena descritto, cambiando l'errore della somma dei quadrati.

Definiamo $p_{\tau k}$ la proporzione di punti nella regione \mathcal{R}_τ assegnati alla classe k , dove $k = 1, \dots, K$. Allora abbiamo due diverse scelte per l'errore:

- **Cross-Entropy:**

$$Q_\tau(T) = \sum_{k=1}^K p_{\tau k} \ln p_{\tau k}$$

- **Indice di Gini:**

$$Q_\tau(T) = \sum_{k=1}^K p_{\tau k} (1 - p_{\tau k})$$

La Cross-Entropy e l'indice di Gini sono misure migliori del tasso di misclassificazione per la crescita dell'albero, perché sono più sensibili alle probabilità dei nodi. Inoltre, a differenza del tasso di misclassificazione, sono differenziabili e quindi più adatti ai metodi di ottimizzazione basati sul gradiente.

L'interpretabilità umana di un modello ad albero è spesso considerata il suo principale punto di forza. Tuttavia, nella pratica si scopre che la particolare struttura ad albero appresa è molto sensibile ai dettagli del dataset, quindi una piccola modifica ai dati di addestramento può comportare un insieme di suddivisioni molto diverso.

8.4 Conditional Mixture Models

In precedenza abbiamo osservato come i Gaussian Mixture Models (GMM) possano catturare distribuzioni multimodali complesse dei dati in input. Se disponiamo di un qualsiasi modello probabilistico per problemi supervisionati, possiamo anche costruire mixture di questi.

Uno dei molti vantaggi di dare un'interpretazione probabilistica al modello di regressione lineare è che esso può essere utilizzato come componente di modelli probabilistici più complessi. Consideriamo un semplice esempio corrispondente a una mixture di modelli di regressione lineare. Consideriamo un dataset $\mathcal{D} = \{(\vec{x}_1, t_1), \dots, (\vec{x}_N, t_N)\}$ per variabili target t_n continue.

Allora sfruttiamo l'interpretazione probabilistica della regressione lineare e, invece di trovare una singola stima di Maximum Likelihood per un singolo modello, utilizziamo EM per tutti i parametri di una mixture di regressori probabilistici.

Possiamo definire la distribuzione condizionale della mixture come:

$$p(t | \vec{\theta}) = \sum_{k=1}^K \pi_k \mathcal{N}(t | \vec{w}_k^T \phi, \beta^{-1})$$

dove π_k rappresenta il coefficiente di miscelazione e $\vec{\theta}$ rappresenta l'insieme dei parametri del modello.

Dato un insieme di osservazioni $\{\phi_n, t_n\}$, allora la funzione logaritmo della Likelihood per questo modello diventa:

$$\ln p(\vec{t} \mid \vec{\theta}) = \sum_{n=1}^N \ln \left\{ \sum_{k=1}^K \pi_k \mathcal{N}(t_n \mid \vec{w}_k^T \phi_n, \beta^{-1}) \right\}$$

dove $\vec{t} = (t_1, \dots, t_n)^T$ rappresenta il vettore delle variabili target.

Possiamo introdurre un insieme $Z = \{\vec{z}_n\}$ di variabili binarie latenti, tali che $z_{nk} \in \{0, 1\}$ con rappresentazione 1-di- K (ovvero per ogni punto \vec{x}_n tutti gli elementi sono pari a zero, tranne un singolo valore uguale ad 1 che indica quale componente della mixture è responsabile della generazione di quel punto).

Allora, introducendo le variabili latenti Z , otteniamo che la funzione logaritmo della likelihood totale dei dati è definita come:

$$\ln p(\vec{t}, Z \mid \vec{\theta}) = \sum_{n=1}^N \sum_{k=1}^K \{z_{nk} \ln \mathcal{N}(t_n \mid \vec{w}_k^T \phi_n, \beta^{-1})\}$$

Per massimizzare questa funzione di verosimiglianza, possiamo ancora una volta utilizzare l'algoritmo EM. In questo modo possiamo trovare le stime MLE dei seguenti parametri:

- Responsabilità calcolata nell'E-Step:

$$\gamma_{nk} = \mathbb{E}[z_{nk}] = p(k \mid \phi_n, \theta^{\text{old}}) = \frac{\pi_k \mathcal{N}(t_n \mid \vec{w}_k^T \phi_n, \beta^{-1})}{\sum_j \pi_j \mathcal{N}(t_n \mid \vec{w}_j^T \phi_n, \beta^{-1})}$$

- Coefficienti di miscelazione calcolati nell'M-Step:

$$\pi_k = \frac{1}{N} \sum_{n=1}^N \gamma_{nk}$$

- Pesi calcolati nell'M-Step:

$$\vec{w}_k = (\Phi^T R_k \Phi) \Phi^T R_k \cdot \vec{t}$$

dove $R_k = \text{diag}(\gamma_{nk})$ è una matrice diagonale di dimensione $N \times N$.

- Precisione calcolata nell'M-Step:

$$\frac{1}{\beta} = \frac{1}{N} \sum_{n=1}^N \sum_{k=1}^K \gamma_{nk} (t_n - \vec{w}_k^T \phi_n)^2$$



ES:

Vediamo l'applicazione dell'algoritmo EM al seguente esempio di adattamento di una mixture di due rette per un dataset con una variabile di input x ed una variabile target t .

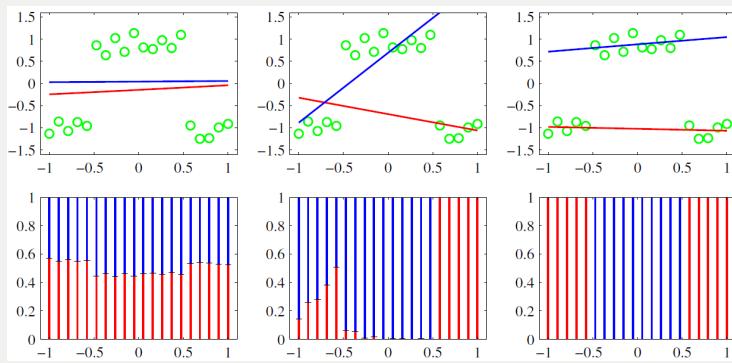


Figura 8.4: Esempio di dataset (rappresentato dai punti verdi), con una variabile di input x e una variabile target t , insieme ad una mixture di due modelli di regressione lineare le cui funzioni medie $y(x, \vec{w}_k)$, con $k \in \{1, 2\}$ sono rappresentate dalle linee blu e rosse. I tre grafici superiori mostrano la configurazione iniziale (a sinistra), il risultato dopo l'esecuzione di 30 iterazioni di EM (al centro) e il risultato dopo 50 iterazioni di EM (a destra). Qui β è stato inizializzato al reciproco della varianza reale dell'insieme dei valori target.

I tre grafici inferiori mostrano le corrispondenti responsabilità tracciate come una linea verticale per ogni punto dati, in cui la lunghezza del segmento blu rappresenta la probabilità posteriore della linea blu per quel punto di dati. Analogamente per il segmento rosso.

Mostriamo inoltre la densità predittiva utilizzando i valori dei parametri di convergenza ottenuti dall'algoritmo EM, corrispondenti ai grafici situati nella parte destra della precedente figura.

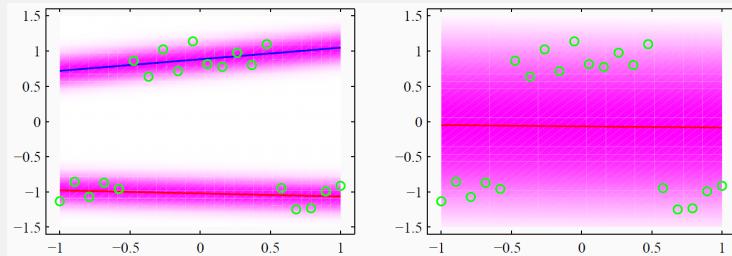


Figura 8.5: Il grafico di sinistra mostra la densità condizionale predittiva corrispondente alla soluzione convergente nella precedente figura (osservabile nel terzo grafico). Questo genera un valore per la funzione del logaritmo della likelihood uguale a -3.0 . La sezione verticale di uno di questi grafici per un particolare valore di x rappresenta la corrispondente distribuzione condizionale $p(t|x)$, che come vediamo è bimodale. Il grafico a destra mostra la densità predittiva di un singolo modello di regressione lineare adattato allo stesso dataset utilizzando la massima verosimiglianza. Questo modello genera un valore per la funzione del logaritmo della likelihood minore, pari a -27.6 .

Capitolo 9

DEEP LEARNING

9.1 Connectionism

Il *connessionismo (connectionism)* si riferisce ad una serie di tecniche ed algoritmi che impiegano *reti neurali artificiali (Artificial Neural Networks - ANN)* applicate nell'ambito dell'intelligenza artificiale con lo scopo di costruire macchine più intelligenti. Il connessionismo presenta una teoria basata sul comportamento del cervello umano e su come i neuroni si attivano secondo uno schema specifico e comunicano per creare una rete in grado di elaborare in fretta le informazioni e produrre risultati.

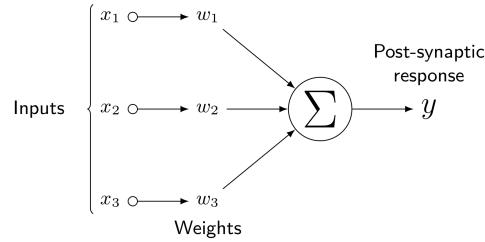
In una rete neurale artificiale (ANN) ogni nodo rappresenta la semplificazione di un neurone biologico. Un nodo tenta di simulare il comportamento dei neuroni con connessioni analoghe alle sinapsi di un neurone biologico e **funzioni di attivazione**, che stabiliscono quando il neurone invia un segnale. Nella forma più semplice, questa funzione di attivazione vale 1 se l'input sommato è più grande di un certo valore di soglia, oppure vale 0 se il segnale ricevuto rimane sotto la soglia della funzione di attivazione (inibizione del neurone).

Quindi una ANN costituisce un'enorme rete di neuroni raggruppati insieme attraverso meccanismi di inibizione e attivazione.

Regola di Hebb: se la cellula *A* contribuisce in modo consistente all'attività della cellula *B*, allora la sinapsi da *A* a *B* deve essere rafforzata.

Ovvero i neuroni che si attivano insieme vengono connessi attraverso una sinapsi, mentre i neuroni che si attivano in modo sfasato non riescono a connettersi tra loro. Questo concetto sta alla base della *propagazione dell'attivazione*, ovvero che l'attivazione di certi neuroni influenza l'attivazione di altri neuroni a cui sono questi connessi. In particolare tali neuroni possono attivarsi o inibirsi a seconda del valore della funzione di attivazione.

Il **Perceptron** rappresenta la più semplice **rete neurale feed-forward**, ovvero reti neurali artificiali in cui le informazioni si muovono secondo un'unica direzione (in avanti) rispetto ai nodi d'ingresso, attraverso una serie di nodi nascosti fino ai nodi d'uscita. Quindi in questa rete neurale le connessioni tra i nodi non formano cicli e non si tiene memoria degli input precedenti (perciò l'output è determinato unicamente dall'attuale input).



Una rete **Single Layer Perceptron (SLP)**, ovvero *perceptron a strato singolo*, è costituita da un unico strato in input seguito direttamente dall'output. In particolare ogni unità di ingresso è collegata tramite un peso ad ogni unità di uscita. Quindi questa rete neurale possiede un unico strato che effettua l'elaborazione dei dati e non presenta nodi nascosti.

The perceptron algorithm	
Input:	$D = \{(x_i, y_i)\}_{i=1}^N$ (training data)
Output:	learned weights w
$w_0 \leftarrow$	random initialization
$t \leftarrow 1$	
while not converged do	
for $(x, y) \in D$ do	
$\hat{y} = f(w^T x)$	
$w_t \leftarrow w_{t-1} + \eta(y - \hat{y})x$	
$t \leftarrow t + 1$	

In ogni nodo di uno strato successivo viene calcolata la somma dei prodotti dei pesi e degli input dello strato precedente. Se il valore di tale somma è al di sopra di una certa soglia allora il neurone si attiva e assume il valore attivato, altrimenti il neurone si inibisce e assume il valore disattivato. Solitamente la funzione di attivazione viene definita nel seguente modo:

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

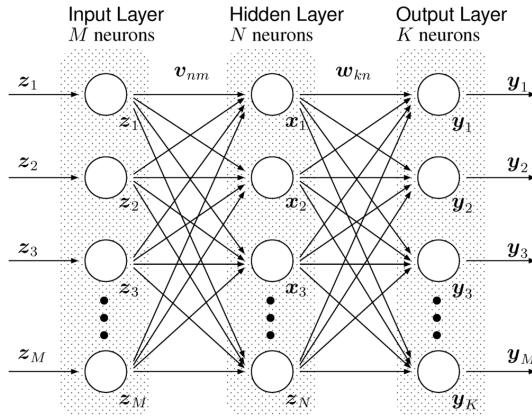
Con questa modalità il SLP può essere utilizzato per risolvere problemi di classificazione binaria.

Tuttavia il SLP garantisce una convergenza solo nel caso in cui il problema di classificazione risulti essere un problema linearmente separabile. In particolare non può risolvere problemi non linearmente separabili.

Per questo motivo si introduce una rete feed-forward più completa detta **Multilayer Perceptron (MLP)**. In questa rete lo strato di input e quello di output sono separati da uno o più strati nascosti, detti **hidden layer**. In particolare, ad ogni strato, ogni nodo è connesso con tutti i nodi dello strato precedente (con archi entranti) e si connette con tutti i nodi dello strato successivo (con archi uscenti).

Perciò la propagazione del segnale avviene in avanti senza cicli e senza connessioni trasversali.

Questo tipo di architettura fornisce alla rete una prospettiva globale in quanto aumentano le interazioni tra neuroni.



Per ogni layer si definisce una funzione di attivazione σ che determina le connessioni da attivare e quelle da inibire. Le funzioni di attivazioni più comuni sono le seguenti *funzioni sigmoidi*:

- **Funzione iperbolica:** $\sigma(\vec{x}) = \tanh(\vec{x})$
- **Funzione logistica:** $\sigma(\vec{x}) = (1 + e^{-\vec{x}})^{-1}$
- **Funzione softmax** (spesso usata nell'output della rete): $\sigma(\vec{x}) = \frac{\exp(\vec{x})}{\sum_i \exp(x_i)}$

N.B.: come abbiamo già analizzato le funzioni sigmoidi sono funzioni che hanno una curva con un tipico andamento a forma di "S" e che mappano l'intero spazio reale $[-\infty, \infty]$ in uno spazio probabilistico $[0, 1]$.

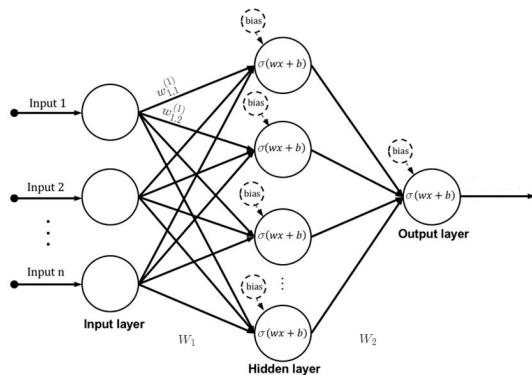
N.B.: l'importanza della funzione di attivazione sta nel far sì che un determinato modello apprenda ed esegua compiti difficili. Inoltre, una funzione di attivazione non lineare consente di sovrapporre più layer di neuroni per creare una Deep Neural Network, necessaria per apprendere serie di dati complessi con un'elevata precisione. Quindi dobbiamo calcolare:

Quindi la funzione di predizione (funzione di output) di un MLP con un unico hidden layer può essere definita come:

$$\hat{y}(\vec{x}) = \sigma(W_2^T \sigma(W_1^T \vec{x} + b_1) + b_2)$$

dove W_1 rappresenta la matrice dei pesi del primo layer, b_1 rappresenta il bias applicato al primo layer, W_2 rappresenta la matrice dei pesi del secondo layer, b_2 rappresenta il bias applicato al secondo layer e σ rappresentano le funzioni di attivazioni per ogni layer.

N.B.: $W_1^T \vec{x} + b_1 \in \mathbb{R}^N$ rappresenta la funzione lineare dell'hidden layer, mentre $\hat{y}(\vec{x}) \in \mathbb{R}^K$ rappresenta l'output della rete.



MLP esegue l'addestramento utilizzando una tecnica nota come **backpropagation**. In particolare tale tecnica valuta la minimizzazione di una funzione loss \mathcal{L} , calcolandone il gradiente rispetto ai pesi del modello e combinandola con qualche tecnica di ottimizzazione come la *discesa del gradiente*.

In questo modo l'algoritmo regola i pesi di ciascuna connessione per ridurre il valore della funzione loss di una piccola quantità. Dopo aver ripetuto questo processo per un numero adeguato di cicli, si ottiene una convergenza della rete in uno stato in cui l'errore risulta essere sufficientemente piccolo.

N.B: come abbiamo detto, per regolare correttamente i pesi si applica un metodo generale di ottimizzazione non lineare, chiamato discesa del gradiente. In particolare la rete calcola la derivata della funzione loss rispetto ai pesi della rete e modifica i pesi in modo tale che l'errore diminuisca (scendendo così sulla superficie della funzione loss). Per questo motivo è necessario che le funzioni di attivazione σ siano differenziabili.

Quindi consideriamo la seguente funzione loss Mean-Squared Error calcolata sul dataset \mathcal{D} rispetto ai parametri del modello θ :

$$\mathcal{L}(\mathcal{D}; \vec{\theta}) = \frac{1}{N} \sum_{(\vec{x}, y) \in \mathcal{D}} (y - f(\vec{x}; \vec{\theta}))^2$$

Eseguiamo la tecnica della discesa del gradiente rispetto a tutti i parametri del modello:

$$\begin{aligned} \vec{\theta}_{n+1} &= \vec{\theta}_n - \varepsilon \nabla_{\vec{\theta}} \mathcal{L}(\mathcal{D}; \vec{\theta}) \\ \Rightarrow \vec{\theta}_{n+1} &= \vec{\theta}_n - \varepsilon \frac{1}{N} \sum_{(\vec{x}, y) \in \mathcal{D}} \nabla_{\vec{\theta}} (y - f(\vec{x}; \vec{\theta}))^2 \end{aligned}$$

dove ε è detto **learning rate**. Calcoliamo il gradiente applicando la **chain rule** e ricordando che stiamo considerando una rete MLP con un unico hidden layer ed equazione di output $\hat{y}(\vec{x}) = \sigma(W_2^T \sigma(W_1^T \vec{x} + b_1) + b_2)$ con $\vec{\theta} = [W_1, b_1, W_2, b_2]$ e σ una qualche funzione di attivazione non lineare:

$$\begin{aligned} \nabla_{\vec{\theta}} (y - f(\vec{x}; \vec{\theta}))^2 &= \nabla_{\vec{\theta}} (y - \sigma(W_2^T \sigma(W_1^T \vec{x} + b_1) + b_2))^2 \\ &= -2(y - \sigma(W_2^T \sigma(W_1^T \vec{x} + b_1) + b_2)) \nabla_{\vec{\theta}} (\sigma(W_2^T \sigma(W_1^T \vec{x} + b_1) + b_2)) \end{aligned}$$

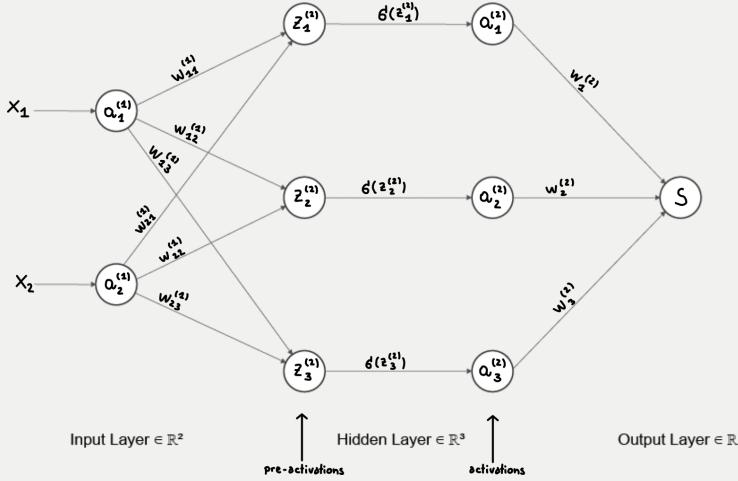
dove il primo termine rappresenta un termine di errore detto **signed error** ed il secondo termine può essere calcolato considerando le derivate parziali rispetto ai parametri.

Vediamo un esempio per intuire meglio il processo di backpropagation.



ES:

Consideriamo la seguente rete MLP costituita da un unico hidden layer:



Quindi consideriamo i seguenti vettori del modello:

$$\vec{a}^{(1)} = [a_1^{(1)} \ a_2^{(1)}]^T$$

$$\vec{z}^{(2)} = [z_1^{(2)} \ z_2^{(2)} \ z_3^{(2)}]^T$$

$$W^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \end{bmatrix}$$

$$W^{(2)} = \begin{bmatrix} w_1^{(2)} \\ w_2^{(2)} \\ w_3^{(2)} \end{bmatrix}$$

Allora possiamo scrivere:

$$z^{(2)} = W^{(1)}\vec{a}^{(1)} \quad \vec{a}^{(2)} = \sigma(W^{(1)}\vec{a}^{(1)})$$

Da cui ricaviamo l'output come:

$$s = W^{(2)}\vec{a}^{(2)} = W^{(2)}\sigma(W^{(1)}\vec{a}^{(1)})$$

Questa prima fase è detta **feedforward** in cui a partire dai dati in input (grezzi) si addestrano i successivi layer con l'obiettivo di calcolare una funzione di errore (funzione loss) posta nell'output del modello. Quindi il processo di feedforward prevede che ogni singola unità calcoli una funzione lineare dei suoi input (cioè $a_i = \sum_i w_{ji}z_i$). Tale somma viene quindi trasformata in una funzione di attivazione non lineare σ (cioè $z_j = \sigma(a_j)$).

N.B.: osserviamo che l'attivazione dell'input avviene attraverso una funzione identità del tipo $\vec{a}^{(1)} = \vec{x}$.

N.B.: l'algoritmo di **backpropagation** presuppone che siano state calcolate le attivazioni delle unità nascoste (*hidden unit*) e di uscita (*output unit*).

Quindi consideriamo un insieme di campioni $\mathcal{D} = \{(\vec{x}, y)\}$. L'obiettivo è quello di calcolare $\nabla_{\vec{\theta}}\mathcal{L}(\vec{x}, y; \vec{\theta})$ con $\vec{\theta} = [W^{(1)}, W^{(2)}]$, ovvero:

$$\left[\frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}} \ \cdots \ \frac{\partial \mathcal{L}}{\partial w_{23}^{(1)}} \ \frac{\partial \mathcal{L}}{\partial w_1^{(2)}} \ \frac{\partial \mathcal{L}}{\partial w_2^{(2)}} \ \frac{\partial \mathcal{L}}{\partial w_3^{(2)}} \right]$$

Consideriamo la seguente funzione loss:

$$\mathcal{L}(\vec{x}, y; \vec{\theta}) = \frac{1}{2}(W^{(2)}\sigma(W^{(1)}\vec{x}) - y)^2$$

Allora il gradiente rispetto ai parametri sarà dato da:

$$\nabla_{\vec{\theta}}\mathcal{L}(\vec{x}, y; \vec{\theta}) = \frac{1}{2}(W^{(2)}\sigma(W^{(1)}\vec{x}) - y) \cdot \nabla_{\vec{\theta}}(W^{(2)}\sigma(W^{(1)}\vec{x}) - y)$$

Osserviamo che y non dipende dai parametri del modello e quindi il gradiente di questo termine rispetto ai parametri $\vec{\theta}$ sarà nullo.

N.B.: il primo termine rappresenta il signed error.

Analizziamo il gradiente:

$$\begin{aligned} \nabla_{\vec{\theta}}(W^{(2)}\sigma(W^{(1)}\vec{x})) &= \nabla_{\vec{\theta}} \begin{bmatrix} w_1^{(2)} & w_2^{(2)} & w_3^{(2)} \end{bmatrix} \begin{bmatrix} \sigma(z_1^{(2)}) \\ \sigma(z_2^{(2)}) \\ \sigma(z_3^{(2)}) \end{bmatrix} \\ &= \nabla_{\vec{\theta}}(w_1^{(2)}\sigma(z_1^{(2)}) + w_2^{(2)}\sigma(z_2^{(2)}) + w_3^{(2)}\sigma(z_3^{(2)})) \end{aligned}$$

Quindi le derivate parziali della funzione loss sono date da:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w_1^{(2)}} &= \sigma(z_1^{(2)}) = a_1^{(2)} \\ \frac{\partial \mathcal{L}}{\partial w_2^{(2)}} &= \sigma(z_2^{(2)}) = a_2^{(2)} \\ \frac{\partial \mathcal{L}}{\partial w_3^{(2)}} &= \sigma(z_3^{(2)}) = a_3^{(2)} \end{aligned}$$

In questo modo abbiamo calcolato il gradiente rispetto ai pesi $W^{(2)}$ (ovvero $\nabla_{W^{(2)}}$). A questo punto possiamo calcolare il gradiente del risultato appena ottenuto rispetto ai pesi $W^{(1)}$ del modello. In particolare otteniamo:

$$\begin{aligned} \nabla_{W^{(1)}}(w_1^{(2)}\sigma(z_1^{(2)}) + w_2^{(2)}\sigma(z_2^{(2)}) + w_3^{(2)}\sigma(z_3^{(2)})) &= \\ w_1^{(2)}\nabla_{W^{(1)}}\sigma(z_1^{(2)}) + w_2^{(2)}\nabla_{W^{(1)}}\sigma(z_2^{(2)}) + w_3^{(2)}\nabla_{W^{(1)}}\sigma(z_3^{(2)}) & \end{aligned}$$

Questo perché i pesi di $W^{(2)}$ non dipendono dai pesi di $W^{(1)}$ e quindi possono essere portati fuori dal calcolo del gradiente. Invece le preattivazioni $\vec{z}^{(2)}$ dipendono dai pesi di $W^{(1)}$ (ovvero $z_1^{(2)}$ dipende da $[w_{11}^{(1)} \quad w_{12}^{(1)}]$, $z_2^{(2)}$ dipende da $[w_{21}^{(1)} \quad w_{22}^{(1)}]$ e $z_3^{(2)}$ dipende da $[w_{31}^{(1)} \quad w_{32}^{(1)}]$).

Vediamo come calcolare il primo gradiente rispetto ai pesi di $W^{(1)}$:

$$\nabla_{W^{(1)}}(\sigma(z_1^{(2)})) = \left[\sigma'(z_1^{(2)}) \frac{\partial}{\partial w_{11}^{(1)}} z_1^{(2)} \quad \sigma'(z_1^{(2)}) \frac{\partial}{\partial w_{12}^{(1)}} z_1^{(2)} \right]$$

$$\begin{aligned}
&= \left[\sigma'(z_1^{(2)}) \frac{\partial}{\partial w_{11}^{(1)}} (w_{11}^{(1)} a_1^{(1)} + \cancel{w_{12}^{(1)} a_1^{(1)}}) \right. \\
&\quad \left. \sigma'(z_1^{(2)}) \frac{\partial}{\partial w_{12}^{(1)}} (\cancel{w_{11}^{(1)} a_1^{(1)}} + w_{12}^{(1)} a_1^{(1)}) \right] \\
&= \left[\sigma'(z_1^{(2)}) a_1^{(1)} \quad \sigma'(z_1^{(2)}) a_2^{(1)} \right]
\end{aligned}$$

N.B.: le semplificazioni al secondo passaggio sono dovute al fatto che il termine semplificato non dipende dal peso su cui stiamo derivando (perciò la sua derivata parziale rispetto al peso è nulla).

Analogamente possiamo calcolare gli altri gradienti:

$$\nabla_{W^{(1)}} (\sigma(z_2^{(2)})) = [\sigma'(z_2^{(2)}) a_1^{(1)} \quad \sigma'(z_2^{(2)}) a_2^{(1)}]$$

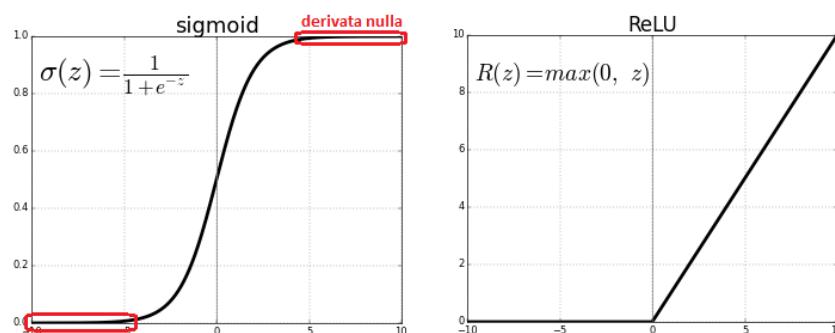
$$\nabla_{W^{(1)}} (\sigma(z_3^{(2)})) = [\sigma'(z_3^{(2)}) a_1^{(1)} \quad \sigma'(z_3^{(2)}) a_2^{(1)}]$$

In questo modo possiamo moltiplicare l'errore propagato all'indietro per il peso e quindi ricalcolare i parametri del modello.

L'algoritmo di backpropagation comporta però alcuni problemi:

- Unità saturate, ovvero molte funzioni di attivazioni risultano essere appiattite ai loro estremi e questo comporta avere dei gradienti quasi nulli in queste zone.
- Gradienti che svaniscono, ovvero l'algoritmo di backpropagation genera una lunga catena di gradienti moltiplicati, ognuno dei quali risulta essere molto piccolo. Questo significa che durante ogni iterazione dell'addestramento ciascuno dei pesi della rete riceve un aggiornamento proporzionale alla derivata parziale della funzione di errore rispetto al peso corrente. Il problema è che in alcuni casi il gradiente risulta essere decisamente piccolo, impedendo di fatto al peso di cambiare il suo valore. Nel peggiore dei casi ciò potrebbe impedire completamente alla rete di continuare ad addestrarsi.

Una possibile soluzione per risolvere questi due problemi potrebbe essere quella di utilizzare alcune funzioni di attivazioni non saturanti, come la **ReLU** (**Rectified Linear Unit**) che si calcola come $\max\{0, a\}$ (dove a rappresenta l'unità).



- Iperparametrizzazione, ovvero i numerosi parametri presenti nelle reti neurali possono causare un facile overfitting.

Una soluzione a questo problema potrebbe essere quella di utilizzare la regolarizzazione per controllare la grandezza (*magnitude*) dei pesi nella rete.

- N molto grande, ovvero avere un numero elevato di campioni comporta passi dell'algoritmo molto lenti. Infatti ad ogni iterazione della discesa del gradiente è necessario fare una media sull'intero dataset e questo comporta un grave rallentamento.

La soluzione a questo problema è quella di approssimare il vero gradiente, calcolato attraverso lo Stochastic Gradient Descent (SGD), con il gradiente di un singolo campione di addestramento (**Batch Gradient Descent**). In particolare possiamo definire un algoritmo detto **Online Stochastic Gradient Descent** il quale prevede di scegliere un vettore iniziale $\vec{\theta}$ di parametri ed un learning rate η . Quindi si ripetono i seguenti passaggi finché non si ottiene un valore minimo approssimativo:

1. Si mescolano casualmente i campioni di addestramento nel dataset \mathcal{D} .
 2. Per ogni campione $(\vec{x}, y) \in \mathcal{D}$ si aggiorna $\vec{\theta} := \vec{\theta} - \eta \nabla_{\vec{\theta}} \mathcal{L}(\{\vec{x}, y\}; \vec{\theta})$.
- La valutazione del gradiente su singoli esempi porta a passi dell'algoritmo molto rumorosi nello spazio dei parametri.

Una possibile soluzione per mitigare questo problema è quella di usare la tecnica del **momentum**, ovvero si deve mantenere una media mobile dei gradienti che viene aggiornata lentamente.

Un'altra possibile soluzione è l'utilizzo dei **mini-batch**, ovvero invece di considerare un singolo campione si calcola la media dei gradienti su un piccolo gruppo di campioni.

N.B.: comunemente si utilizza una combinazione di momentum e mini-batch per stabilizzare l'addestramento.

Riportiamo alcuni termini utili impiegati comunemente durante l'ottimizzazione nel Deep Learning:

- **epoca**: rappresenta un passaggio completo sui dati.
- **iterazione**: rappresenta un singolo passo del gradiente.
- N : indica il numero di campioni di addestramento.
- B : indica la dimensione del batch.

In particolare possiamo schematizzare con la seguente tabella il numero di iterazioni effettuate per ogni epoca su diversi algoritmi:

Algoritmo	Iterazioni Per Epoche	Commenti
Batch Gradient Descent	1	Viene effettuata un'unica iterazione ad ogni epoca (si valuta un singolo campione alla volta).
Stochastic Gradient Descent	N	Vengono effettuate N iterazioni per ogni epoca (si valutano tutti i campioni insieme).
Mini-Batch Gradient Descent	$\frac{N}{B}$	Vengono effettuate $\frac{N}{B}$ iterazioni per ogni epoca (si valutano gruppi di campioni).

Il Multilayer Perceptron possiede alcuni problemi:

- Dimensione del modello: il modello possiede molti parametri. Questo comporta problemi di memoria e di efficienza.
- Overfitting: i numerosi parametri ed i limitati dati di addestramento comportano un possibile eccessivo adattamento del modello al dataset di addestramento (lo stesso si può dire riguardo all'underfitting).
- Sottogeneralizzazione: la conseguenza dell'overfitting comporta una difficoltà nel generalizzare il modello a nuovi dati.
- Gradienti che svaniscono: problema causato dalla backpropagation (dovuto all'applicazione della regola della catena) che porta a valori del gradiente molto piccoli.
- Unità saturate: le funzioni di attivazione tradizionali possono portare a unità saturate (uscite vicine a 1 o 0 (oppure -1)) che hanno derivate prossime allo zero.

Questi problemi hanno portato la comunità a ignorare il potenziale di questi modelli per decenni.

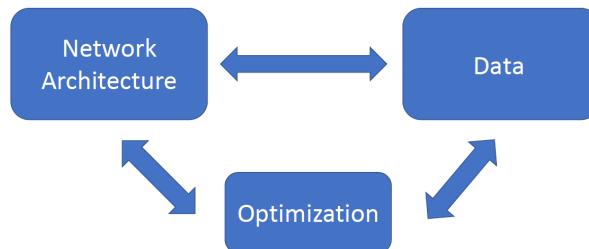
N.B.: tutti questi problemi (tranne il primo) devono essere monitorati durante il processo di training del modello (non basta valutare i risultati al termine dell'addestramento).

Tuttavia il MLP presenta una serie di caratteristiche estremamente interessanti. Innanzitutto rappresenta un modello end-to-end in cui possiamo addestrare tutti i componenti del modello utilizzando un singolo algoritmo di ottimizzazione. Inoltre MLP apprende rappresentazioni dell'input e del classificatore. Un MLP dovrebbe essere in grado di apprendere rappresentazioni di caratteristiche che a loro volta sono buone rappresentazioni per la classificazione.

9.2 Fondamenti del Deep Learning

Ogni sistema Deep Learning è costituito da tre componenti principali:

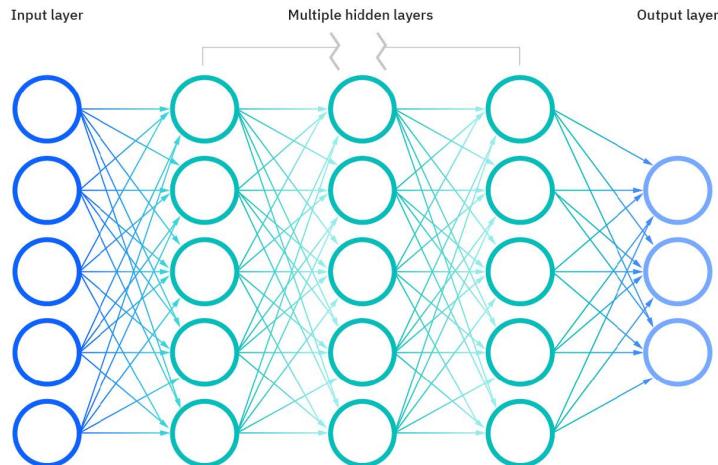
- **Architettura di rete.** Definisce la famiglia di funzioni che si vogliono utilizzare come modelli. Possiamo considerare un'architettura di rete come un grafo che definisce le connessioni tra i blocchi computazionali di base, dove tali blocchi sono detti **layers**. Quindi i layers rappresentano blocchi di costruzione riutilizzabili.
- **Ampio numero di dati.** I dati rappresentano l'elemento centrale del Deep Learning. In particolare si preferisce l'utilizzo di dati etichettati, ma possono essere impiegati anche dati non etichettati.
- **Ottimizzazione.** L'ottimizzazione ed il training permettono di ricavare informazioni sui dati. Si possono definire diversi algoritmi per gestire il processo di apprendimento (il 99% di questi algoritmi si basa sul calcolo del gradiente e sull'aggiornamento dei pesi). Inoltre vengono utilizzati una serie di trucchi per evitare il verificarsi di alcuni problemi.



In particolare queste componenti sono relazionate tra loro. Infatti:

- *Architettura - Dati:* le architetture sono fortemente correlate con i dati. Infatti a seconda dei dati a disposizione (ad esempio immagini, testo, suoni, ecc...) dobbiamo considerare famiglie di funzioni differenti. La dimensione dei dati e la qualità delle annotazioni possono determinare la dimensione della rete che possiamo addestrare.
- *Architettura - Ottimizzazione:* le diverse architetture adottano differenti strategie di addestramento. I problemi derivati dall'apprendimento possono essere moderati interagendo con l'architettura (ad esempio per ridurre l'overfitting).
- *Dati - Ottimizzazione:* a seconda dei dati disponibili possono essere impiegate diverse strategie di addestramento (ad esempio se i dati sono etichettati o se sono sufficienti).

Quindi l'architettura di una rete è costituita da un layer di input, che contiene i dati grezzi, uno o più hidden layer, che permettono di addestrare il modello, ed un layer di output, che permette di calcolare la funzione di predizione.



Perciò le principali componenti di un modello di Deep Learning sono:

1. Dati in input, possibilmente standardizzati (ad esempio $\vec{x} \in \mathbb{R}^{64}$).
2. Funzione loss (ad esempio Cross-Entropy).
3. Architettura della rete (ad esempio una rete con due hidden layer e larghezza 10).

Il Deep Learning riguarda una categoria di algoritmi di Machine Learning ispirati al funzionamento ed all’organizzazione del cervello e basati sull’apprendimento di più livelli di rappresentazione. Come sappiamo, le informazioni vengono elaborate nel cervello attraverso una connessione di un certo numero di neuroni attivi. In particolare la rappresentazione neurale è gerarchica ed il cervello elabora le informazioni in modo incrementale. Perciò l’idea è quella di realizzare rappresentazioni gerarchiche delle informazioni nei modelli computazionali.

Le rappresentazioni sono utili per gestire domini o per trasferire la conoscenza appresa. Le attività di elaborazione delle informazioni possono risultare più o meno facili a seconda di come vengono rappresentate le informazioni.

Il **Representation Learning** (*apprendimento delle rappresentazioni*) è particolarmente interessante perché fornisce un modo di eseguire l’apprendimento non supervisionato. Infatti spesso abbiamo a disposizione pochi dati di addestramento etichettati e quindi un addestramento con tecniche di apprendimento supervisionato sul sottoinsieme etichettato comporterebbe risultati con possibile overfitting.

L’apprendimento non supervisionato offre la possibilità di risolvere questo problema di overfitting imparando anche dai dati non etichettati. In particolare, possiamo apprendere buone rappresentazioni per i dati non etichettati e poi utilizzare queste rappresentazioni per risolvere l’attività di apprendimento supervisionato.



ES:

Si vogliono apprendere le rappresentazioni gerarchiche di immagini naturali. In particolare addestriamo un modello con due hidden layer attraverso un dataset di immagini naturali.

Il primo layer consiste in 24 basi (o gruppi) ognuna delle quali è un filtro di 10×10 pixel, mentre il secondo strato consiste in 100 basi, ognuna delle

quali è un filtro di 10×10 pixel.

Come mostrato nella figura in alto, le basi del primo layer apprese sono filtri di bordi (*edge*) orientati e localizzati.

Invece, come mostrato nella figura in basso, le basi del secondo layer apprese sono filtri che corrispondono ai contorni, agli angoli e ai contorni delle superfici (*surface boundaries*) nelle immagini.

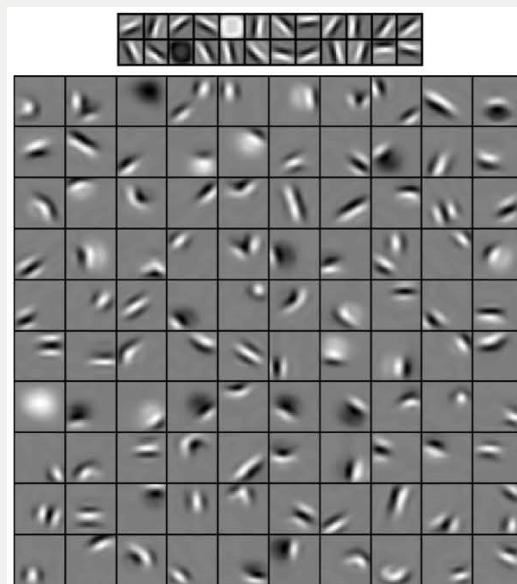


Figura 9.1: L'illustrazione rappresenta le basi del primo layer (in alto) e le basi del secondo layer (in basso) apprese da immagini naturali. Ogni base del secondo layer (filtro) è stata visualizzata come una combinazione lineare ponderata delle basi del primo layer.

L'obiettivo è quello di apprendere rappresentazioni gerarchiche di parti di oggetti in un contesto non supervisionato. Quindi a partire dalla rappresentazione del primo layer appresa da immagini naturali sono stati addestrati due ulteriori layer utilizzando immagini non etichettate di singole categorie di oggetti.

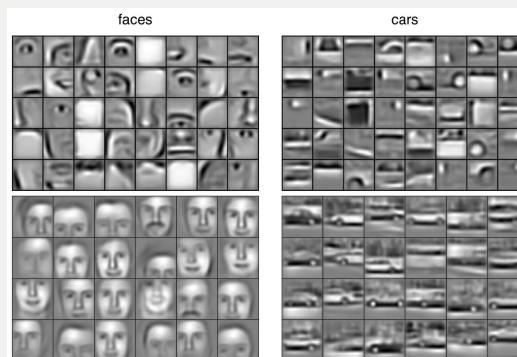


Figura 9.2: L'illustrazione rappresenta le basi del secondo layer (in alto) e del terzo layer (in basso) apprese da specifiche categorie di oggetti.

Come mostrato in figura, il secondo layer ha appreso le caratteristiche corrispondenti alle parti degli oggetti, anche se all'algoritmo non è stata fornita alcuna etichetta che specificasse la posizione degli oggetti o delle loro parti. Il terzo layer ha imparato a combinare le rappresentazioni delle parti del secondo layer in caratteristiche più complesse e di livello superiore.

Dati di input

Analogamente ad altri algoritmi di Machine Learning, i dati di input vengono rappresentati attraverso un tensore.

In particolare un **tensore** generalizza tutte le strutture definite in algebra lineare (ad esempio uno scalare viene rappresentato da un tensore di ordine 0, un vettore viene rappresentato da un tensore di ordine 1, un matrice viene rappresentata da un tensore di ordine 2, ecc...).

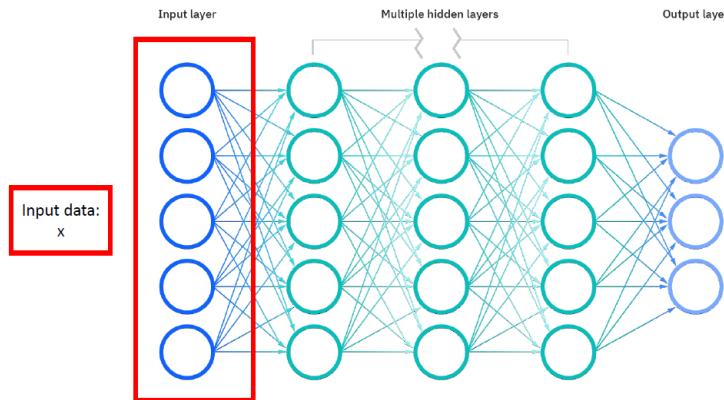
Diversamente da altri approcci però non cerchiamo di apprendere le features direttamente dai dati, ma vogliamo apprendere una gerarchia di rappresentazioni. Dobbiamo però valutare quanto rendere profonda (*deep*) e ampia (*wide*) questa gerarchia di rappresentazioni. Infatti una rappresentazione troppo profonda comporta un incremento della complessità e quindi un aumento dei parametri. Questo implicherebbe un maggiore rischio di overfitting del modello.



ES:

- Le immagini a colori di dimensione 128×128 pixel vengono rappresentate da un tensore di ordine 3. Quindi un'immagine a colori sarà un tensore $\vec{x} \in \mathbb{R}^{128 \times 128 \times 3}$.
- I segnali audio mono non sono altro che una sliding window di 100 campioni. Perciò un segnale audio mono sarà un tensore $\vec{x} \in \mathbb{R}^{100}$ (vettore).
- La codifica di un testo per un vocabolario V viene rappresentata da un tensore $\vec{x} \in \mathbb{R}^{|V|}$.
- Consideriamo un dataset $\mathcal{D} = \{(\vec{x}_n, y_n)\}_{n=1}^N$ di N campioni tale che $\vec{x}_n \in \mathbb{R}^{8 \times 8}$. Allora $\mathcal{D} \in \mathbb{R}^{N \times 8 \times 8}$.

N.B: una buona pratica è quella di normalizzare e scalare i dati di input. Ad esempio per le immagini si comprime il range dei possibili valori $[0, 255]$ in $[-1, 1]$.



Funzione Loss

Le reti neurali vengono addestrate utilizzando la discesa stocastica del gradiente e richiedono la scelta di una funzione loss nella fase di progettazione del modello. Esiste un ampio numero di funzioni loss che possono essere scelte per addestrare il modello. Perciò è fondamentale decidere quale funzione risulti essere più appropriata per un determinato modello. Vediamo alcune possibilità:

- **Binary Cross-Entropy:**

$$-\sum_{i=1}^{|N|} y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)$$

dove y_i rappresenta l'etichetta dell'esempio i ed \hat{y}_i rappresenta la previsione dell'esempio i fatta da un certo modello \mathcal{M} .

La scelta di questa funzione loss viene effettuata nel caso di problemi di **classificazione binaria**.

N.B.: in questo caso l'etichetta $y_i \in \{0, 1\}$.

In particolare, se la previsione è corretta avremo $y_i = \hat{y}_i$ e quindi la funzione loss assumerà il valore 0.



ES:

Consideriamo un dataset $\mathcal{D} = \{(\vec{x}_1, y_1)\}$ con un unico campione.

Supponiamo $y_1 = 1$ ed $\hat{y}_1 = 0.01$.

Allora:

$$\mathcal{L} = -(1 \cdot \log 0.01 + 0 \cdot \log 0.99)$$

- **Cross-Entropy:**

$$-\sum_{i=1}^{|N|} \left\{ \sum_{c=1}^{|C|} y_{i,c} \log \hat{y}_{i,c} \right\}$$

dove y_i rappresenta l'etichetta dell'esempio i e della classe c ed \hat{y}_i rappresenta la previsione dell'esempio i di appartenere alla classe c fatta da un certo modello \mathcal{M} .

La scelta di questa funzione loss viene effettuata nel caso di problemi di **classificazione multiclasse**.

N.B: in questo caso l'etichetta y viene codificata attraverso la rappresentazione **one-hot** che esprime per ogni elemento c del vettore y la probabilità che il campione in input \vec{x} appartenga alla classe c .



ES:

Consideriamo un dataset $\mathcal{D} = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2)\}$ con due campioni e ipotizziamo $K = 4$ classi.

Supponiamo $y_1 = [0 \ 0 \ 1 \ 0]$ (indica che il campione \vec{x}_1 appartiene alla classe 3) e $\hat{y}_1 = [0.1 \ 0.1 \ 0.75 \ 0.05]$.

Inoltre supponiamo $y_2 = [0 \ 1 \ 0 \ 0]$ (indica che il campione \vec{x}_2 appartiene alla classe 2) e $\hat{y}_1 = [0.1 \ 0.1 \ 0.75 \ 0.05]$.

Allora:

$$\mathcal{L} = -[(0 \cdot \log 0.1 + 0 \cdot \log 0.1 + 1 \cdot \log 0.75 + 0 \cdot \log 0.05) + (0 \cdot \log 0.1 + 1 \cdot \log 0.1 + 0 \cdot \log 0.75 + 0 \cdot \log 0.05)]$$

- Mean Squared Error (MSE):

$$\frac{1}{N} \sum_{i=1}^{|N|} \|y_i - \hat{y}_i\|^2$$

dove y_i rappresenta un vettore in un certo spazio che indica l'output della regressione ed \hat{y}_i rappresenta la previsione dell'esempio i fatta da un modello \mathcal{M} . La scelta di questa funzione loss viene effettuata nel caso di problemi di **regressione**.

Architettura (Basic Building Blocks)

Una Neural Network è un meccanismo di Machine Learning molto potente che imita il modo in cui il cervello umano apprende le informazioni. In particolare il cervello riceve gli stimoli dal mondo esterno, elabora gli input e genera gli output. Quando questa attività di apprendimento si complica, è necessario l'attivazione di più neuroni che collaborano per formare una rete complessa, passandosi informazioni tra di loro. Una rete neurale artificiale cerca di imitare un comportamento simile. Ogni neurone della rete è caratterizzato da un peso \vec{w} , un bias b ed funzione di attivazione σ .

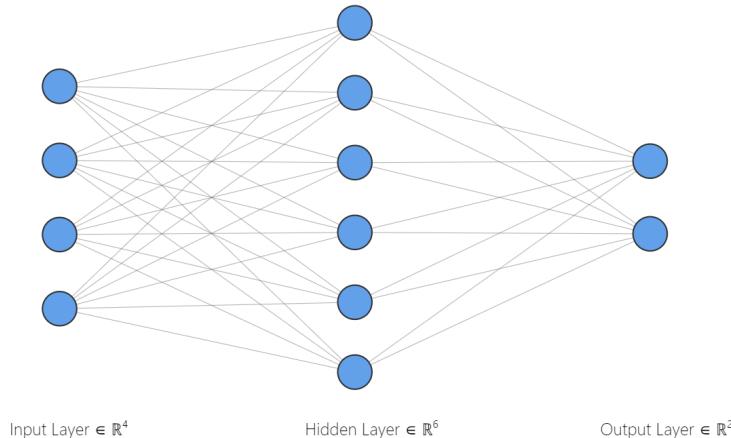
Un **Fully Connected Layer** (detto anche **Dense Layer** oppure **Linear Layer**) in una rete neurale rappresenta un layer in cui tutti gli input di quel layer vengono collegati a tutte le unità di attivazione del layer successivo. In particolare un layer Fully Connected è caratterizzato da una funzione lineare dell'input \vec{x} pesata da un vettore dei pesi \vec{w} più un bias b , ovvero:

$$f(\vec{x}) = \vec{w}^T \vec{x} + b$$

N.B: tale funzione viene spesso definita in termini di *fan-in* e *fan-out* o di dimensionalità di input/output. Ad esempio la seguente rete può essere espressa in termini di fan-in = 4 e fan-out = 2 (oppure alternativamente $D_{\text{in}} = 4$ e $D_{\text{out}} = 2$).

N.B: in PyTorch possiamo definire un Fully Connected Layer come

`torch.nn.Linear(Din, Dout)`.



In questo modo le informazioni in input vengono inviate ad un layer, nel quale i neuroni eseguono una trasformazione lineare su questo input utilizzando i pesi ed i bias.

A tale risultato viene successivamente applicata una **funzione di attivazione**.

$$s = \sigma(f(\vec{x})) = \sigma(\vec{w}^T \vec{x} + b)$$

Nonostante tale funzioni comporti una complessità maggiore al modello, risulta essere fondamentale perché permette di apprendere pattern più complessi dai dati.

N.B.: se la rete neurale non impiegasse le funzioni di attivazione si eseguirebbero solamente le trasformazioni lineari sugli input applicando a questi i pesi ed i bias. Nonostante questo semplifichi la rete, risulterebbe essere una rete meno potente.

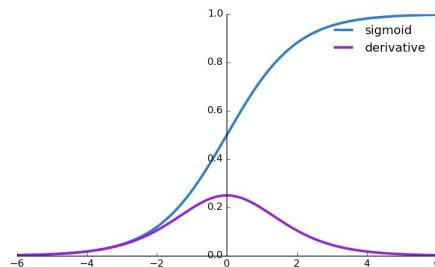
Esistono diversi tipi di funzioni di attivazione:

- **Sigmoide:**

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

La funzione sigmoide per un lungo periodo è stata considerata la funzione standard per le reti neurali. Tale funzione mappa i valori reali in valori compresi nell'intervallo $[0, 1]$.

Tale funzione di attivazione però comporta il problema delle unità saturate (che comporta il problema dei gradienti che svaniscono). Infatti come possiamo osservare dal grafico i valori del gradiente sono significativi nell'intervallo $[-3, 3]$, ma il grafico si appiattisce nella regione esterna a tale intervallo. Questo implica che per valori esterni da questo intervallo, i gradienti risultano essere molto piccoli e quindi la rete non sta più imparando.



Osserviamo inoltre che la funzione sigmoide non è simmetrica rispetto all'origine. Quindi l'output di tutti i neuroni avrà sempre lo stesso segno

⚠ Dim:

Dimostriamo che $\sigma'(x) = \sigma(x)(1 - \sigma(x))$:

$$\begin{aligned}
 \sigma'(x) &= \frac{d}{dx} \left[\frac{1}{1 + e^{-x}} \right] \\
 &= \frac{d}{dx} (1 + e^{-x})^{-1} \\
 &= -(1 + e^{-x})^{-2} (-e^{-x}) \\
 &= \frac{e^{-x}}{(1 + e^{-x})^2} \\
 &= \frac{1}{1 + e^{-x}} \cdot \frac{e^{-x}}{1 + e^{-x}} \\
 &= \frac{1}{1 + e^{-x}} \cdot \frac{(1 + e^{-x}) - 1}{1 + e^{-x}} \\
 &= \frac{1}{1 + e^{-x}} \cdot \left(\frac{1 + e^{-x}}{1 + e^{-x}} - \frac{1}{1 + e^{-x}} \right) \\
 &= \frac{1}{1 + e^{-x}} \cdot \left(1 - \frac{1}{1 + e^{-x}} \right) \\
 &= \sigma(x) \cdot (1 - \sigma(x))
 \end{aligned}$$

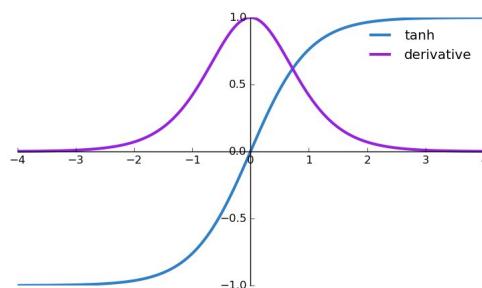
C.V.D.

- Iperbolica (\tanh):

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

Tale funzione mappa i valori reali in valori compresi nell'intervallo $[-1, 1]$.

La funzione \tanh è simile alla funzione sigmoide, con l'unica differenza che risulta simmetrica rispetto all'origine. Questa proprietà comporta risultati migliori della funzione \tanh rispetto alla sigmoide, poiché in questo caso l'output dei neuroni non avrà sempre lo stesso segno (perciò i gradienti non sono limitati a muoversi in una determinata direzione). Inoltre, come possiamo osservare dal grafico, il gradiente della funzione \tanh risulta essere più ripido rispetto alla funzione sigmoide.



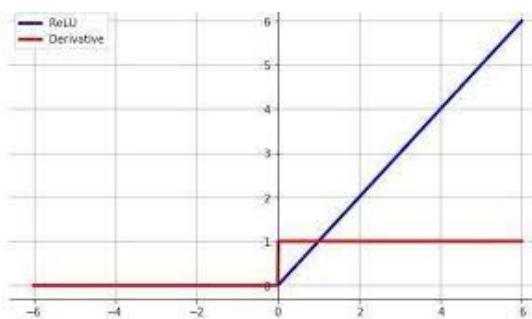
- Rectified Linear Unit (ReLU):

$$\sigma(\vec{x}) = \max\{0, \vec{x}\}$$

Il vantaggio principale dell'utilizzo della funzione ReLU rispetto alle altre funzioni di attivazione è che questa non attiva i neuroni nel caso in cui l'output della trasformazione lineare risulti essere inferiore a 0.

Poiché viene attivato solo un certo numero di neuroni, la funzione ReLU risulta essere molto più efficiente dal punto di vista computazionale rispetto alle funzioni sigmoide e tanh. Se si osserva il lato negativo del grafico, si noterà che il valore del gradiente è pari a zero. Per questo motivo, durante il processo di retropogazione, i pesi e i bias di alcuni neuroni non vengono aggiornati.

N.B: questo può creare il problema dei neuroni morti che non vengono mai attivati.



N.B: tale funzione di attivazione risulta essere buona per problemi di classificazione, ma meno buona per problemi di regressione.



Dim:

Calcoliamo la derivata della funzione ReLU:

$$\sigma'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

Inoltre è indefinita in $x = 0$ poiché la derivata sinistra e la derivata destra in questo punto non coincidono.

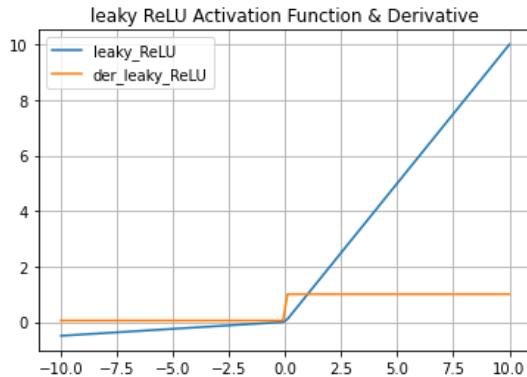
C.V.D.

- LeakyReLU:

$$\sigma(\vec{x}) = \max\{a\vec{x}, \vec{x}\}$$

dove $a < 1$ (ad esempio $a = 0.1$). Tale funzione rappresenta una versione migliorata della funzione ReLU. Quindi, mentre la funzione ReLU viene definita come 0 per i valori negativi di \vec{x} , la funzione LeakyReLU viene definita come una componente lineare estremamente piccola di \vec{x} . Con questo accorgimento il gradiente del lato negativo del grafico diventa adesso un valore non nullo. Di conseguenza, non ci saranno più neuroni morti in quella regione.

Quindi il vantaggio è che non vengono scartati i valori negativi in input mantenendo le proprietà proprietà non lineari e non saturanti della ReLU.



- **Softmax:**

$$\sigma(\vec{x})_i = \frac{e^{x_i}}{\sum_{i=0}^{|C|} e^{x_i}}$$

La funzione Softmax può essere rappresentata come una combinazione di più sigmoidi. In particolare la funzione sigmoide restituisce valori compresi nell'intervallo $[0, 1]$, che possono essere trattati come la probabilità per un punto di appartenere ad una particolare classe (perciò viene spesso utilizzata per problemi di classificazione binaria). Quindi la funzione Softmax restituisce le probabilità per un punto di appartenere a ogni singola classe (perciò viene spesso utilizzata per problemi di classificazione multiclasse).

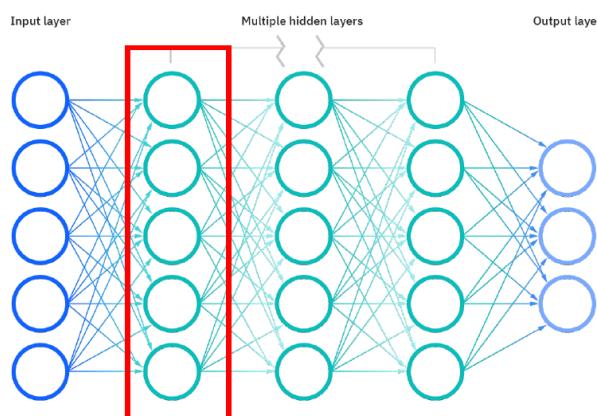
N.B: la somma di tutte le probabilità calcolate dalla Softmax per un punto è 1.

N.B: durante la creazione di una rete per un problema di classificazione multiclasse, l'output layer deve possedere tanti neuroni quante sono le classi (ad esempio se ha tre classi, ci sono tre neuroni nel livello di output).

N.B: in PyTorch possiamo definire la funzione Softmax come

`torch.nn.Softmax(dim = 1)` (dove dim = 1 rappresenta la dimensione lungo la quale verrà calcolata la Softmax).

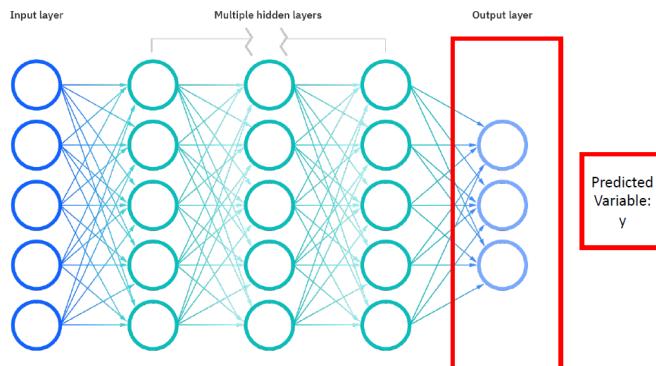
Una volta applicata la funzione di attivazione alla trasformazione lineare dell'input, l'output della funzione di attivazione passa al layer successivo che ripete lo stesso procedimento. Questo movimento in avanti delle informazioni è noto come **forward propagation**.



Quindi a partire dalle variabili di input \vec{x} che rappresentano i nostri dati (come un'immagine, un segnale, ecc...) si propagano le informazioni applicando una combinazione di trasformazioni lineari e funzioni di attivazione non lineari attraverso una serie di layer intermedi (*hidden layer*) fino ad arrivare ad un *output layer* nel quale si produce la previsione finale desiderata.

N.B: la funzione di attivazione per l'output layer può essere diversa da quella degli hidden layer, a seconda del problema.

In particolare un output layer è costituito da una serie di unità scelte in base al problema che stiamo cercando di risolvere. Ad esempio per un problema di classificazione lineare possiamo scegliere un output layer costituito da un'unica unità con funzione di attivazione sigmoide, mentre per un problema di classificazione multiclasse (con K classi) possiamo scegliere un output layer costituito da K unità con funzione di attivazione softmax. Invece per un problema di regressione possiamo scegliere un output layer costituito da una unità senza funzioni di attivazione.



Utilizzando l'output della propagazione in avanti (*forward propagation*), viene calcolato un valore di errore. Sulla base di questo valore di errore, vengono aggiornati i pesi ed i bias dei neuroni di tutti i layer del modello. Questo processo è noto come *backward propagation*.

Ottimizzazione

La **backpropagation** (o *propagazione all'indietro degli errori*) rappresenta un metodo di addestramento delle reti neurali artificiali utilizzato insieme ad un metodo di ottimizzazione come il metodo della discesa del gradiente. Tale metodo calcola il gradiente di una funzione loss rispetto a tutti i pesi della rete. Quindi tale gradiente viene utilizzato dal metodo di ottimizzazione per aggiornare i pesi della rete con l'obiettivo di minimizzare la funzione loss.



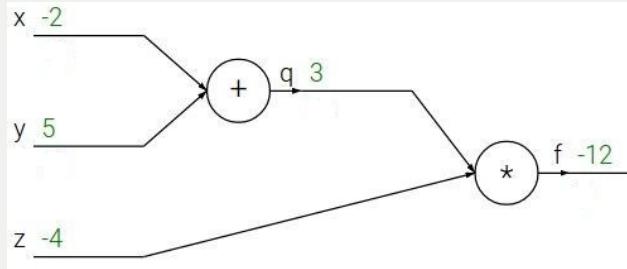
ES:

Vediamo un esempio di backward propagation. Consideriamo la seguente funzione:

$$f(x, y, z) = (x + y)z$$

Rappresentiamo tale funzione come un grafo di elaborazione, detto **computation graph** (tale grafo è un Directed Acyclic Graph). Nel grafo in questione viene mostrato uno step di forward propagation applicato all'input

$(x, y, z) = (-2, 5, 4)$:



In particolare possiamo rappresentare l'operazione di somma $x+y$ attraverso una variabile q (perciò $q = 3$). Quindi le derivate parziali rispetto ad x ed y saranno:

$$\frac{\partial q}{\partial x} = 1 \quad \frac{\partial q}{\partial y} = 1$$

Di conseguenza avremo che $f = qz$ e quindi le derivate parziali rispetto a q e z saranno:

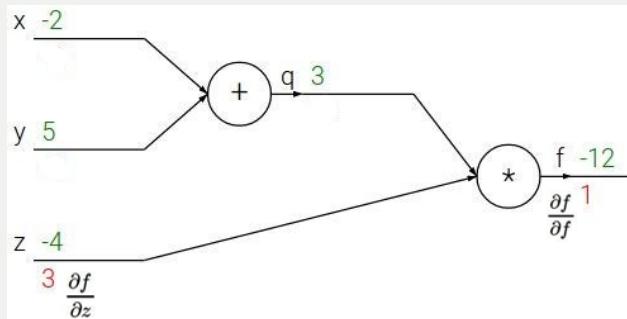
$$\frac{\partial f}{\partial q} = z \quad \frac{\partial f}{\partial z} = q$$

L'obiettivo è quello di calcolare le derivate parziali $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$ ed $\frac{\partial f}{\partial z}$.

Una volta calcolato il valore di errore f all'output layer dobbiamo aggiornare i pesi ed i bias delle unità appartenenti ai layer precedenti. In particolare vediamo che:

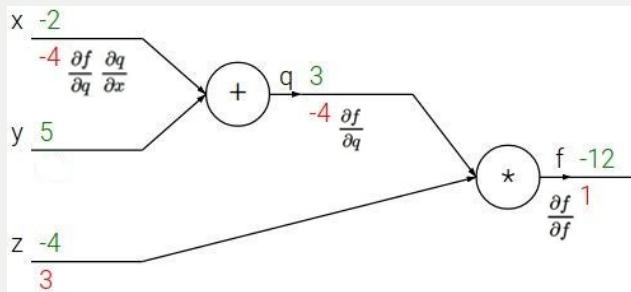
$$\frac{\partial f}{\partial z} = q = 3$$

Quindi possiamo aggiornare z :



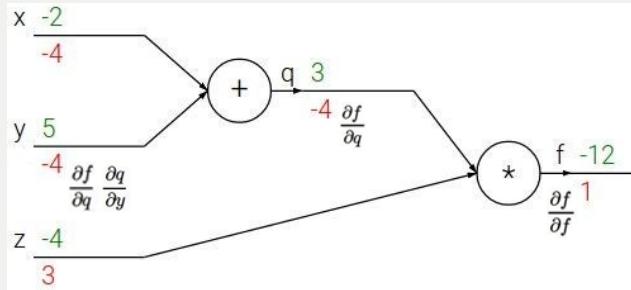
Inoltre, sapendo che $\frac{\partial f}{\partial q} = z$ possiamo ricavare:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} = z \cdot 1 = -4 \cdot 1 = -4$$



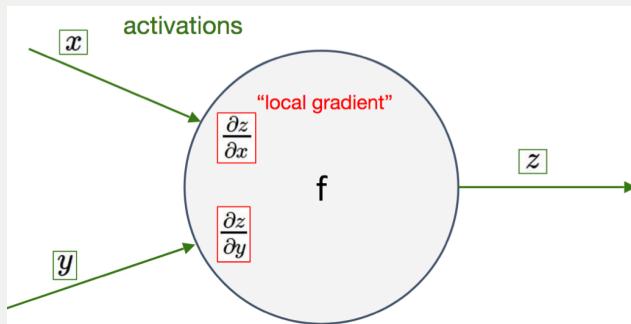
Analogamente possiamo ricavare:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} = z \cdot 1 = -4 \cdot 1 = -4$$

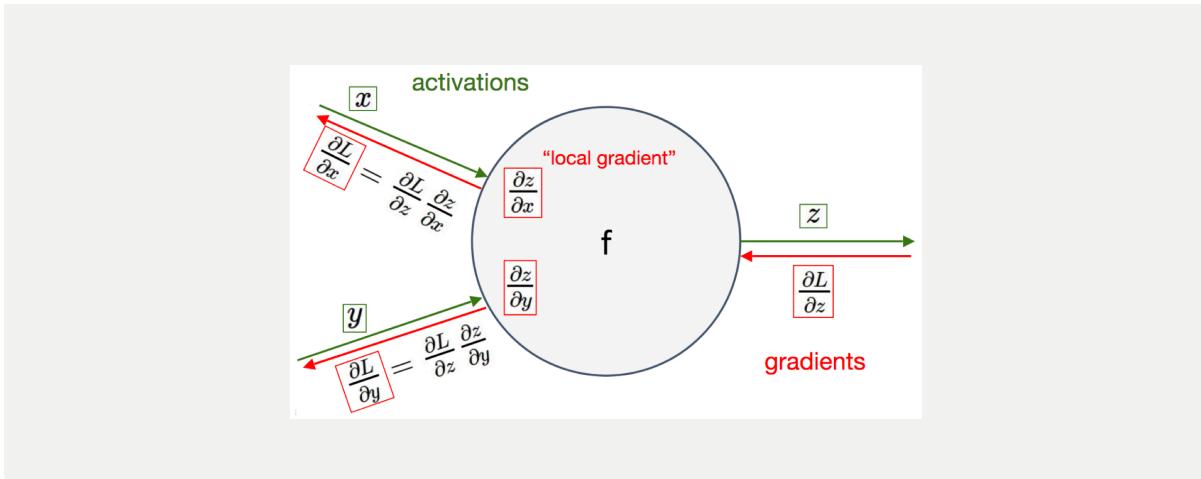


ES:

Generalizziamo l'algoritmo per un neurone della rete. Consideriamo un neurone (ovvero un nodo computazionale) f di cui inizialmente non abbiamo informazioni. Dopo la fase forward propagation, nella quale viene calcolata la funzione loss \mathcal{L} , possiamo conoscere x , y ed f . A questo punto possiamo calcolare i gradienti locali $\frac{\partial z}{\partial x}$ ed $\frac{\partial z}{\partial y}$.



Perciò possiamo calcolare $\frac{\partial \mathcal{L}}{\partial z}$ e, tramite la regola della catena, possiamo propagare all'indietro le informazioni moltiplicando i gradienti.



Quindi possiamo facilmente rappresentare una rete neurale attraverso un computation graph. In questo modo è possibile valutare la funzione loss e calcolare quindi calcolare i gradienti rispetto a tutti i parametri del modello.

N.B.: tutti i nodi computazionali (ovvero i neuroni) devono essere funzioni differenziabili e devono definire un operatore forward ed un operatore backward (derivata).

N.B.: le reti neurali sono grafi aciclici diretti (o DAG). Questo garantisce alla backward propagation di funzionare sempre.

Gli algoritmi di addestramento per modelli di Deep Learning sono solitamente iterativi e richiedono quindi all'utente di specificare un punto iniziale da cui iniziare l'iterazione. Perciò addestriamo le Deep Networks utilizzando il metodo SGD (Stochastic Gradient Descent) o qualche sua variante iterativa e quindi definiamo un metodo di inizializzazione per tutti i parametri della rete.

N.B.: l'addestramento di modelli di Deep Learning è un'attività fortemente influenzata dalla scelta dell'inizializzazione. Il valore di inizializzazione può infatti determinare se l'algoritmo converge o meno.

Progettare buone strategie di inizializzazione è un'attività piuttosto complessa perché l'ottimizzazione delle reti neurali non è ancora un processo del tutto compreso. Una difficoltà è che alcuni punti iniziali possono essere vantaggiosi dal punto di vista dell'ottimizzazione, ma dannosi dal punto di vista della generalizzazione.

Le moderne procedure di inizializzazione sono tecniche euristiche e suggeriscono di inizializzare i pesi in modo casuale, a seconda dell'architettura.

In particolare *PyTorch* segue il seguente metodo standard di inizializzazione dei pesi per un modello lineare (*Linear*):

$$W_{i,j} \sim U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$$

dove m rappresenta la dimensione di input del layer. L'idea è quella di campionare ogni peso da una distribuzione uniforme in modo da ottenere una varianza uniforme del gradiente tra i vari layer.

Quindi consideriamo un modello lineare e supponiamo di avere un input \vec{x} con n componenti ed un neurone lineare con matrice dei pesi W inizializzata in modo casuale, campionando i valori da una distribuzione uniforme, che produce un valore di output y . Allora la varianza di y è data da:

$$\text{Var}(y) = \text{Var}(W_1 x_1 + \dots + W_n x_n) = \text{Var}(x) \text{Var}(W_i) \cdot n$$

Perciò la varianza dell'output è data dalla varianza dell'input, ma scalata da $Var(W_i) \cdot n$.

Quindi se vogliamo la stessa varianza in input ed in output allora $Var(W_i) \cdot n$ deve valere 1. Questo significa che $Var(W_i) = \frac{1}{n}$ (detto **inizializzazione di Xavier**).

Perciò tale euristica consiste nell'inizializzare i pesi di un Fully Connected Layer con m input ed n ouput campionando ogni peso dalla distribuzione uniforme $U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$.

Glorot e Bengio suggeriscono di utilizzare una inizializzazione normalizzata, ovvero:

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

dove m è la dimensione di input del layer ed n è la dimensione di output.

Questa euristica è stata progettata per trovare un compromesso tra l'obiettivo di inizializzare tutti i layer di attivazione e l'obiettivo di inizializzare tutti i layer con la stessa varianza del gradiente.

Una volta definita la tecnica di inizializzazione dei parametri del modello possiamo stabilire l'algoritmo di ottimizzazione. In particolare consideriamo le seguenti possibili scelte:

- **MiniBatch SGD:** la Stochastic Gradient Descent (SGD) e le sue varianti sono probabilmente gli algoritmi di ottimizzazione più utilizzati per il Deep Learning. In particolare è possibile ottenere una stima non distorta del gradiente se valutiamo il gradiente medio su un minibatch di m campioni estratti (in modo indipendente ed identicamente distribuito) dalla distribuzione dei dati. Tale algoritmo mostra come seguire questa stima del gradiente in discesa.

Algorithm 8.1 Stochastic gradient descent (SGD) update

```

Require: Learning rate schedule  $\epsilon_1, \epsilon_2, \dots$ 
Require: Initial parameter  $\theta$ 
   $k \leftarrow 1$ 
  while stopping criterion not met do
    Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with
    corresponding targets  $\mathbf{y}^{(i)}$ .
    Compute gradient estimate:  $\hat{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$ 
    Apply update:  $\theta \leftarrow \theta - \epsilon_k \hat{g}$ 
     $k \leftarrow k + 1$ 
  end while

```

In questo algoritmo la dimensione del passo dipende unicamente dalla norma del gradiente moltiplicata per il tasso di apprendimento (ovvero $\vec{\theta} \leftarrow \vec{\theta} - \varepsilon_k \hat{g}$).

- **Momentum:** nonostante la Stochastic Gradient Descent risulti essere una strategia di ottimizzazione molto diffusa, l'apprendimento con essa può essere talvolta lento. Perciò è stato definito il metodo del Momentum (o metodo della quantità di moto) per accelerare il processo di apprendimento.
Poiché SGD aggiorna i parametri solo dopo aver valutato un sottoinsieme di esempi allora il percorso della discesa del gradiente oscillatorà verso la convergenza. L'obiettivo del Momentum è quello di ridurre queste oscillazioni aggiungendo un parametro \vec{v} che indica la direzione e la velocità con cui i parametri si muovono.

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ , momentum parameter α
Require: Initial parameter θ , initial velocity v

```

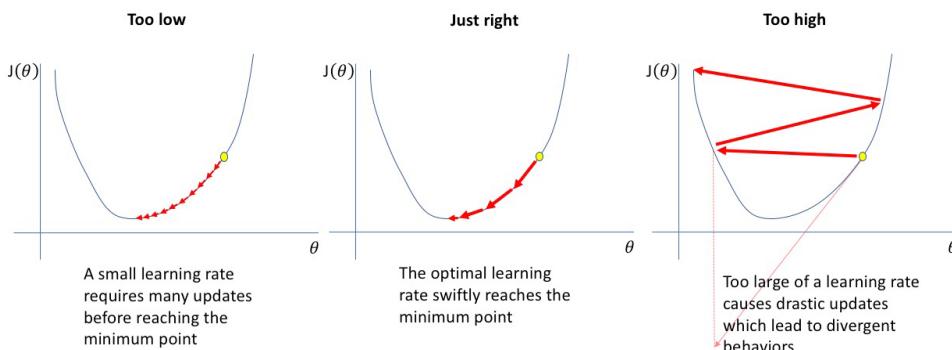
while stopping criterion not met do
    Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with
    corresponding targets  $\mathbf{y}^{(i)}$ .
    Compute gradient estimate:  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$ .
    Compute velocity update:  $v \leftarrow \alpha v - \epsilon \mathbf{g}$ .
    Apply update:  $\theta \leftarrow \theta + v$ .
end while

```

In questo algoritmo la dimensione del passo dipende dalla grandezza e dall'allineamento della sequenza di gradienti (ovvero $\vec{v} \leftarrow \alpha \vec{v} - \epsilon \vec{g}$ ed $\vec{\theta} \leftarrow \vec{\theta} + \vec{v}$).

N.B.: il parametro $\alpha \in [0, 1)$ determina la velocità con cui i contributi dei gradienti precedenti decadono esponenzialmente. Quindi più grande è α rispetto a ϵ , più i gradienti precedenti influenzano la direzione attuale.

Un importante parametro per questi algoritmi è il **learning rate** (o *tasso di apprendimento*) ϵ (fondamentale per la dimensione del passo da fare nella fase di aggiornamento dei pesi). L'obiettivo è quello di diminuire gradualmente questo parametro nel tempo (indichiamo con ϵ_k il learning rate alla k -esima iterazione). La scelta del valore iniziale del learning rate può essere fatta per tentativi ed errori, ma di solito è più opportuno farla monitorando le curve di apprendimento che tracciano la funzione obiettivo in funzione del tempo.



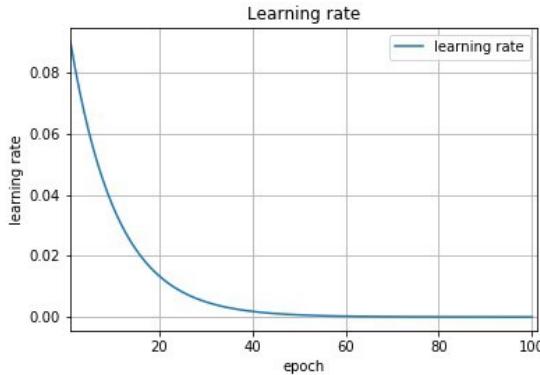
In particolare se tale valore è troppo grande, la curva di apprendimento mostrerà oscillazioni violente, con la funzione loss che aumenta in modo significativo (la soluzione potrebbe anche divergere). Al contrario se il tasso di apprendimento è troppo basso, allora l'apprendimento procede lentamente prima di raggiungere un punto di minimo. Un buon tasso di apprendimento è un valore non troppo piccolo, in modo che l'algoritmo possa convergere rapidamente, e non troppo grande, in modo che l'algoritmo non salti avanti e indietro senza raggiungere i minimi.

Idealmente vogliamo inizializzare il learning rate ad un valore più alto per poi ridurlo gradualmente facendolo decadere.

Una possibile implementazione è il *decadimento esponenziale* (**exponential decay**) che ha la seguente forma:

$$\epsilon = \epsilon_0 \cdot e^{-kt}$$

dove t indica il numero di iterazioni (epoch) e k è un'iperparametro. Tale tecnica è euristica.

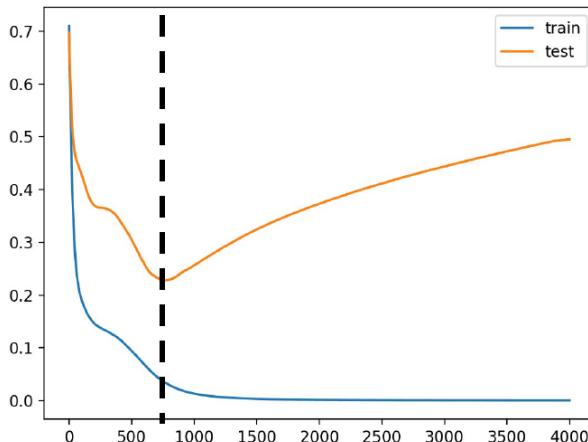


Training

Il processo di addestramento di un modello di Deep Learning segue alcune buone pratiche come lo split del dataset in un dataset di addestramento (**train set**) ed un dataset di validazione (**validation set**). In particolare risulta utile monitorare la funzione loss ed alcune metriche relative al modello su questi dataset.

Uno dei problemi più comuni nell’addestramento delle reti neurali è l’overfitting. In particolare si verifica un overfitting quando la rete ottiene risultati ottimali sui dati di addestramento, ma scarsi sui dati di test. Questo accade perché l’algoritmo di apprendimento cerca di adattarsi a tutti i dati in ingresso, anche se rappresentano un rumore, e quindi non riesce a valutare dati che non ha mai visto.

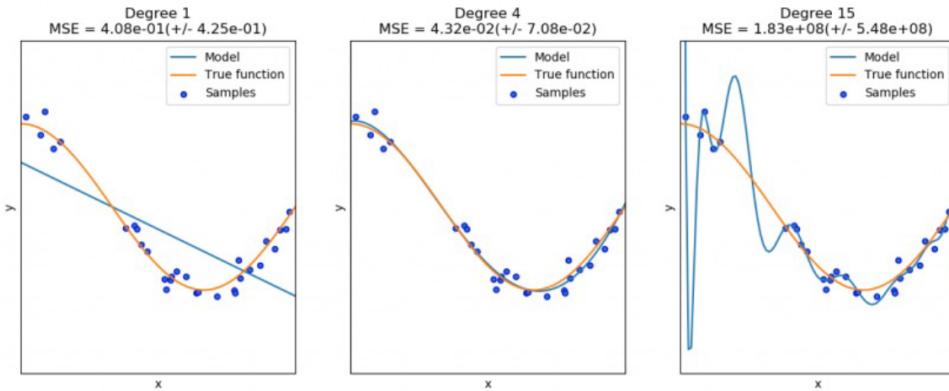
Una buona tecnica per limitare questo problema è l’utilizzo dell’**early stopping**, ovvero una forma di regolarizzazione in cui si monitorano le performance del modello in modo da interrompere l’addestramento nel caso in cui si verifica un overfitting.



Possiamo controllare i problemi di overfitting e underfitting valutando semplicemente le performance ottenute durante le fasi di training e di testing. In particolare:

	Training	Validation
Underfit	Low	Low
Overfit	High	Low

Un problema centrale del Machine Learning è quello di creare un algoritmo che abbia buone prestazioni sia sui dati di training, ma anche su nuovi input, in modo da ottenere un modello più generale (**generalizzazione**).



Molte strategie utilizzate sono progettate per ridurre l'errore di test, eventualmente a spese di un aumento dell'errore di training. Queste strategie sono note collettivamente come **regolarizzazione**. Esistono numerose forme di regolarizzazione.

Consideriamo una funzione loss $J(\vec{\theta}; X, \vec{y})$. Molti approcci di regolarizzazione aggiungono una penalità $\Omega(\vec{\theta})$ alla funzione obiettivo J (dove $\vec{\theta}$ rappresenta il vettore dei parametri del modello). In generale denotiamo la funzione loss regolarizzata con \tilde{J} :

$$\tilde{J}(\vec{\theta}; X, \vec{y}) = \alpha \Omega(\vec{\theta}) + J(\vec{\theta}; X, \vec{y})$$

dove $\alpha \in [0, \infty)$ è un iperparametro che pesa il contributo del termine di penalizzazione della norma (ovvero Ω) rispetto alla funzione loss J .

N.B.: impostando $\alpha = 0$ non si ottiene alcuna regolarizzazione, mentre valori maggiori di α corrispondono ad una maggiore regolarizzazione.

Un semplice e comune tipo di regolarizzazione è detto **Weight Decay** (oppure L^2). Questa strategia di regolarizzazione avvicina i pesi all'origine aggiungendo un termine di regolarizzazione $\Omega(\vec{\theta}) = \frac{1}{2} \|\vec{w}\|^2$ alla funzione loss. Quindi la funzione loss regolarizzata è definita come:

$$\tilde{J}(\vec{w}; X, \vec{y}) = \frac{\alpha}{2} \vec{w}^T \vec{w} + J(\vec{w}; X, \vec{y})$$

Analizziamo il gradiente della funzione obiettivo regolarizzata per studiare il comportamento di questa forma di regolarizzazione weight decay (*decadimento dei pesi*).

N.B.: per semplificazione, assumiamo che non ci sia un parametro di bias. Quindi $\vec{\theta}$ sarà semplicemente \vec{w} .

$$\nabla_{\vec{w}} \tilde{J}(\vec{w}; X, \vec{y}) = \alpha \vec{w} + \nabla_{\vec{w}} J(\vec{w}; X, \vec{y})$$

Possiamo osservare che la regola di aggiornamento per un vettore di pesi \vec{w} diventerebbe:

$$\vec{w} \leftarrow (1 - \varepsilon \alpha) \vec{w} - \varepsilon \nabla_{\vec{w}} J(\vec{w}; X, \vec{y})$$

Quindi possiamo notare che l'aggiunta del termine di weight decay modifica la regola di apprendimento in modo da ridurre iterativamente il vettore dei pesi \vec{w} di un fattore costante a ogni passo (prima di eseguire il solito aggiornamento del gradiente).

Possiamo semplificare ulteriormente l'analisi effettuando un'approssimazione quadratica alla funzione loss intorno al valore dei pesi che ottiene il minimo costo di addestramento non regolarizzato (ovvero $\vec{w}^* = \arg \min_{\vec{w}} J(\vec{w})$):

$$\tilde{J}(\vec{\theta}) = J(\vec{w}^*) + \frac{1}{2}(\vec{w} - \vec{w}^*)^T H(\vec{w} - \vec{w}^*)$$

dove H rappresenta la matrice Hessiana di J rispetto a \vec{w}^* .

In particolare il minimo di \tilde{J} si verifica dove il suo gradiente $\nabla_{\vec{w}} \tilde{J}(\vec{w}) = H(\vec{w} - \vec{w}^*)$ è nullo. Quindi aggiungiamo il gradiente del weight decay e risolviamo il minimo della versione regolarizzata di \tilde{J} (utilizziamo \tilde{w} per rappresentare il minimo):

$$\begin{aligned}\alpha \tilde{w} + H(\tilde{w} - \vec{w}^*) &= 0 \\ (H + \alpha I)\tilde{w} &= H\vec{w}^* \\ \tilde{w} &= (H + \alpha I)^{-1}H\vec{w}^*\end{aligned}$$

Osserviamo quindi che per $\alpha \rightarrow 0$, la soluzione regolarizzata \tilde{w} si avvicina a \vec{w}^* (quindi la soluzione è uguale a quella non regolarizzata).

Invece al crescere di α otteniamo:

$$\tilde{w} = Q(\Lambda + \alpha I)^{-1}\Lambda Q^T \vec{w}^*$$

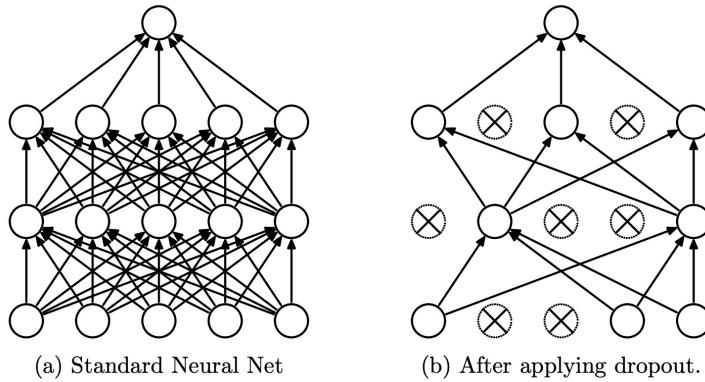
dove abbiamo decomposto H in una matrice diagonale Λ ed una base ortonormale di autovettori Q tale che $H = Q\Lambda Q^T$.

Vediamo quindi che l'effetto del weight decay consiste nel ridimensionare \vec{w}^* lungo gli assi definiti dagli autovalori di H . In particolare, la componente di \vec{w}^* allineata con l' i -esimo autovalore di H viene riscalata di un fattore di $\frac{\lambda_i}{\lambda_i + \alpha}$.

Questo implica che solo le direzioni lungo le quali i parametri contribuiscono in modo significativo alla riduzione della funzione loss vengono conservate intatte. Al contrario, un piccolo autovalore di H nelle direzioni lungo le quali i parametri non contribuiscono a ridurre la funzione loss, implica che il movimento in questa direzione non aumenterà in modo significativo il gradiente. Quindi le componenti del vettore dei pesi corrispondenti a queste direzioni decadono grazie all'uso della regolarizzazione durante l'addestramento.

Un'altra tecnica di regolarizzazione, poco costosa ma potente, è detta **Dropout**. Questa tecnica consiste nell'inibire casualmente alcune unità appartenenti a differenti layers, con una certa probabilità (ad esempio $p = 0.5$). Di conseguenza da questa unità inibita vengono temporaneamente rimossi tutti i collegamenti con il layer precedente e quello successivo della rete. In questo modo viene creata una nuova architettura di rete, differente da quella principale.

Questa tecnica viene solitamente applicata in modo stratificato.



Il dropout può quindi essere utilizzato durante l'addestramento di un modello per evitare il problema dell'overfitting. L'obiettivo dell'addestramento di una rete è quello diminuire una funzione loss attraverso le unità della rete. In particolare, un'unità può modificarsi in modo da correggere gli errori commessi da altre unità. Questo processo comporta degli adattamenti che causano il problema dell'overfitting (poiché non si riesce a generalizzare a causa di questi adattamenti). Il dropout impedisce a queste unità di correggere l'errore di altre unità, evitando quindi di adattarsi. Quindi, inibendo casualmente alcune unità (nodi), si costringe i layers ad assumersi più o meno responsabilità per l'input adottando un approccio probabilistico.

In questo modo possiamo realizzare architetture di rete più profonde e più grandi che possono fare buone previsioni su dati che la rete non ha visto prima (quindi senza overfitting).

N.B.: il dropout può essere applicato a tutti i layer, tranne che all'output layer.

N.B.: in PyTorch possiamo definire il dropout come `torch.nn.Dropout(p = 0.5)` (dove $p = 0.5$ rappresenta la probabilità che un elemento venga azzerato).

Quindi consideriamo le unità di un hidden layer i lineare:

$$h_i(\vec{x}) = W_i^T + b_i = \sum_{k=1}^m w_i^k \vec{x}^k + b_i$$

Per semplicità ipotizziamo che \vec{x} sia un vettore in \mathbb{R}^m (cioè $\vec{x} \in \mathbb{R}^m$). Analogamente le unità del layer successivo saranno definite come:

$$h_{i+1}(\vec{x}) = W_{i+1}^T + b_{i+1} = \sum_{k=1}^n w_{i+1}^k \vec{x}^k + b_{i+1}$$

In particolare, combinando due layer consecutivi abbiamo:

$$h_i(\vec{x}) = h_i(h_{i-1}(\vec{x})) = \sum_{k=1}^m w_i^k \vec{x}^k + b_i$$

Quindi in generale possiamo rappresentare una rete come:

$$f(\vec{x}) = o \circ h_1 \circ h_2 \circ \dots \circ h_N$$

dove o rappresenta le unità dell'output layer ed h_i rappresenta le unità dell' i -esimo hidden layer.

In particolare, la j -esima unità dell' i -esimo hidden layer è definita da:

$$h_i^j(\vec{x}) = w_i^1 \vec{x}^1 + \dots + w_i^m \vec{x}^m$$

Mentre la j -esima unità del successivo hidden layer è definita da:

$$h_{i+1}^j(\vec{x}) = w_{i+1}^1 z^1 + \dots + w_{i+1}^n z^n$$

dove $z^j = h_i^j(\vec{x})$ rappresenta l'unità j dell'hidden layer h_i (layer precedente). Perciò:

$$h_{i+1}^j(\vec{x}) = w_{i+1}^1 h_i^1(\vec{x}) + \dots + w_{i+1}^j h_i^j(\vec{x}) + \dots + w_{i+1}^n h_i^n(\vec{x})$$

Adesso introduciamo il dropout:

$$\mathcal{D}(z^j) = \begin{cases} h_i^j(\vec{x}) & \text{se } r > 0.5 \\ 0 & \text{altrimenti} \end{cases}$$

dove $r \sim U(0, 1)$ (probabilità campionata da una distribuzione uniforme).

Quindi otteniamo:

$$h_{i+1}^j(\vec{x}) = w_{i+1}^1 \mathcal{D}(z^1) + \dots + w_{i+1}^j \mathcal{D}(z^j) + \dots + w_{i+1}^n \mathcal{D}(z^n)$$

Durante la fase di backpropagation dobbiamo calcolare per ogni unità j il gradiente del layer h_{i+1} rispetto ai pesi w_{i+1} , ovvero:

$$\frac{\partial h_{i+1}^j}{\partial w_{i+1}} = \mathcal{D}(z^j)$$

In particolare, quando questo valore tende a 0 in modo casuale, si interrompe l'aggiornamento del gradiente per il parametro w_{i+1} . In questo modo possiamo impedire che il layer h_{i+1} dipenda dall'unità z^j .

Questo risultato può essere visto come una sorta di campionamento da $2^{|H|}$ differenti architetture di rete, dove $|H|$ rappresenta la quantità di hidden layer su cui è possibile applicare il dropout.

N.B.: poiché ogni unità può essere attivata o disattivata con una certa probabilità, allora stiamo addestrando un insieme di reti neurali diverse.

N.B.: poiché i pesi sono condivisi c'è un effetto di regolarizzazione dato dal fatto che ogni modello è vincolato dagli altri nel valore che i pesi possono assumere. Per questo motivo è meglio usare dropout rispetto alla tecnica di regolarizzazione L^2 , la quale in generale porta il peso a zero, mentre in questo caso si porta il peso verso un valore corretto (anche se con una lenta convergenza).

Dropout può essere considerato come un metodo pratico per il bagging applicato ad ensemble di reti neurali molto numerosi e di grandi dimensioni. Il bagging prevede l'addestramento di più modelli e la valutazione di più modelli su ogni campione di test. Questo sembra essere poco pratico quando ogni modello è costituito da una rete neurale di grandi dimensioni, poiché l'addestramento e la valutazione di tali reti sono costosi in termini di tempo di esecuzione e di memoria.

In particolare, il dropout addestra l'ensemble costituito da tutte le sottoreti che possono essere formate rimuovendo le unità (non appartenenti all'output layer) da una rete di base.

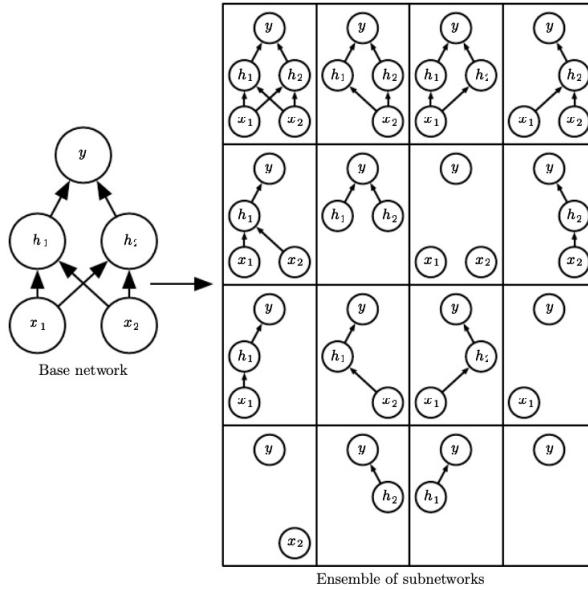


Figura 9.3: Dropout addestra un insieme costituito da tutte le sottoreti che possono essere costruite rimuovendo le unità (non appartenenti all'output layer) da una rete di base. Prendiamo come esempio una rete di base costituita da due unità in input e due unità contenute in un unico hidden layer. Allora possiamo realizzare fino a sedici possibili sottoinsiemi di queste quattro unità. Mostriamo tutte le sedici sottoreti che possono essere formate eliminando sottoinsiemi diversi di unità dalla rete originale.

Formalmente, una ensemble effettua una predizione attraverso un meccanismo a votazione. Tale processo è detto **inferenza**. In particolare l'output di un modello bagging produce una distribuzione di probabilità $p^{(i)}(y | \vec{x})$. Allora la predizione dell'ensemble è data dalla media aritmetica di tutte queste distribuzioni:

$$\frac{1}{k} \sum_{i=1}^k p^{(i)}(y | \vec{x})$$

Il problema è che k solitamente assume valori elevati e quindi questo problema risulta intrattabile da valutare (si avrebbe un numero esponenziali di termini da sommare), a meno che la struttura del modello non consenta qualche forma di semplificazione.

In particolare possiamo approssimare l'inferenza con il campionamento, ovvero si campiona in modo casuale un certo numero di istanze del modello e si fa la media dei risultati ottenuti su questi campioni (spesso 10-20 campioni sono sufficienti per ottenere buone prestazioni).

Nella pratica possiamo scalare le attivazioni in base alla probabilità di inclusione ed eseguire un singolo passo forward. Non ci sono ancora argomenti teorici a favore dell'accuratezza di questa regola di inferenza approssimata, ma empiricamente si comporta molto bene. Poiché di solito usiamo una probabilità di inclusione di $\frac{1}{2}$ (ovvero $p = 0.5$), allora la regola di scalatura dei pesi di solito equivale a dividere i pesi per 2 alla fine dell'addestramento e poi usare il modello come al solito. Un altro modo per ottenere lo stesso risultato è moltiplicare gli stati delle unità per 2 durante l'addestramento. In entrambi i casi, l'obiettivo è fare in modo che l'input totale previsto per un'unità al momento del test sia all'incirca uguale all'input totale previsto per quell'unità al momento dell'addestramento, anche se metà delle unità al momento dell'addestramento sono in media mancanti.

La **Label Smoothing** rappresenta un'ulteriore tecnica di regolarizzazione che prevede la sostituzione dei vettori targets (che hanno una rappresentazione one-hot) con degli

output più *smooth* (ad esempio si sostituisce $\vec{y} = [0 \ 0 \ 1 \ 0]$, riferito ad un campione appartenente alla terza classe, con l'output $\vec{y} = [0 \ 0 \ 0.9 \ 0]$ in cui abbiamo utilizzato uno smooth di 0.1). In questo modo rendiamo il modello più incerto rispetto ad una determinata previsione.

PyTorch

PyTorch rappresenta uno dei framework di Deep Learning, grazie alla sua facilità di utilizzo, alla possibilità di creare grafi in modo dinamico (a runtime) ed al fatto che risulta essere più *pythonic* rispetto ad altri frameworks (come TensorFlow). Inoltre permette di agire in modo più diretto con i nostri modelli e permette di effettuare il debug sulle singole operazioni.

Il componente centrale di PyTorch è la struttura dei dati detta **tensore**. Un tensore rappresenta un vettore multidimensionale, come *numpy.ndarrays*, con la differenza fondamentale che in PyTorch i vettori sono compatibili con CUDA e quindi possono essere memorizzati sulle GPU (i dati devono essere esplicitamente spostati dalla RAM della CPU alla RAM della GPU).

Un'altra caratteristica importante che i tensori possiedono è che sono ottimizzati per la differenziazione, ovvero la computazione che sta alla base della backpropagation (ovvero dell'algoritmo di addestramento della rete neurale).

Possiamo definire una sottoclasse modello semplicemente ereditando la classe di PyTorch `torch.nn.Module`, che rappresenta la classe base per tutti i moduli di rete neurale in PyTorch. Quindi dobbiamo definire la struttura del modello nel costruttore (ovvero dobbiamo definire i vari layer del modello, le dimensioni dei layer, le funzioni di attivazione, le funzioni di regolarizzazione, ecc...).

Una volta definita la struttura del modello, dobbiamo definire il metodo `forward` che specifica lo step di forward del modello.

N.B: il metodo `backward` non deve essere implementato.

Per addestrare un modello è necessario istanziare un oggetto della classe del modello definita. Quindi lo step di forward si ottiene semplicemente richiamando tale oggetto.



ES:

Consideriamo il seguente modello *Net* che eredita la classe `torch.nn.Module` e prende in ingresso la dimensione dell'input layer (D_{in} e D_{out}):

```
class Net(torch.nn.Module):
    def __init__(self, D_in, D_out):
        super(Net, self).__init__()
        self.fc1 = torch.nn.Linear(D_in, 32)
        self.fc2 = torch.nn.Linear(32, D_out)
        self.tanh = torch.nn.Tanh()
        self.dropout = torch.nn.Dropout(0.5)
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        x = self.fc1(x)
        x = self.dropout(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.dropout(x)
        return self.tanh(x)
```

Quindi nel costruttore viene definito un singolo hidden layer `self.fc1` (di dimensioni D_{in} e 32) ed un layer di output `self.fc2` (di dimensioni 32 e D_{out}). Quindi viene definita una funzione di attivazione `self.relu`, posta tra l'hidden layer ed il layer di output, ed una funzione di attivazione `self.tanh`, che determina l'output della rete. Infine viene definita una funzione di regolarizzazione `self.dropout` (con probabilità $p = 0.5$).

Il metodo `forward(x)` definisce lo step forward sui dati in ingresso x .

Per addestrare il modello, dobbiamo innanzitutto istanziare il modello.

Quindi possiamo effettuare il passo di forward semplicemente richiamando l'oggetto sui dati di ingresso:

```
D_in, D_out = 2, 1
x = torch.tensor([1, 2])

model = Net(D_in,D_out)
y = model(x)
```

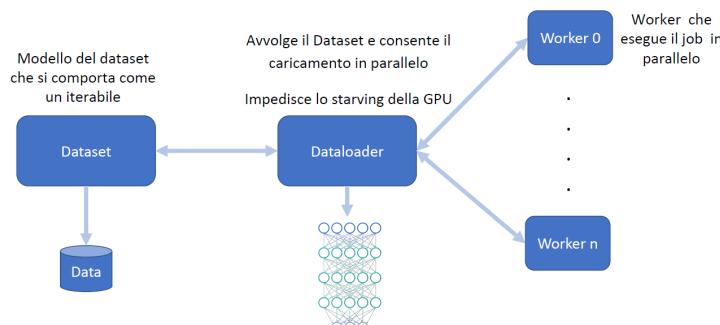
N.B.: il modello istanziato ha un input layer con 2 unità ed una singola unità di output.

Esistono due modi per integrare i dati nel modello:

1. Utilizzare dei dataset standard forniti da PyTorch e integrati nella classe `torch.utils.data.Dataset`.
2. Creare un dataset personalizzato.

Il codice per l'elaborazione dei dati può diventare disordinato e difficile da mantenere. Idealmente vogliamo che il codice del dataset risulti essere disaccoppiato dal codice di addestramento del modello, così da avere una migliore leggibilità e modularità.

PyTorch fornisce due primitive per elaborare i dati, ovvero `torch.utils.data.Dataset` e `torch.utils.data.DataLoader`. In particolare `Dataset` memorizza i campioni e le etichette corrispondenti, mentre `DataLoader` avvolge un iterabile attorno a `Dataset` per consentire un facile accesso ai campioni.





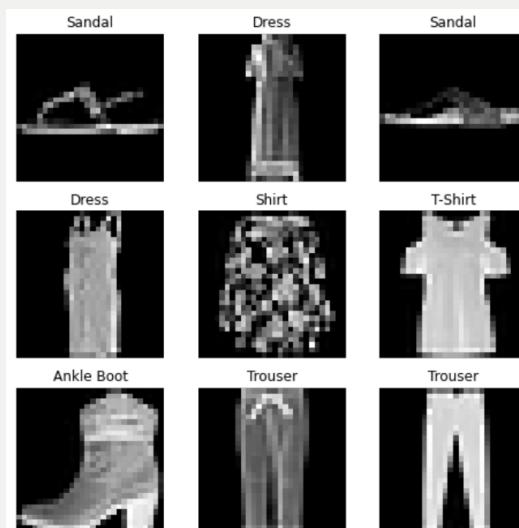
ES:

Preleviamo il dataset *FashionMNIST* fornito in `torchvision.datasets`:

```
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor

training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)
```

Tale dataset fornisce un insieme di immagini che fanno riferimenti a diverse tipologie di abbigliamento (rappresenta un dataset più complesso rispetto a MNIST che comprende un insieme di immagini relative a cifre).



N.B: possiamo iterare sui dataset come se fossero delle liste o un qualsiasi altro iterabile.

Avvolgiamo il dataset in un DataLoader:

```
from torch.utils.data import DataLoader
train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)

for x,y in train_dataloader:
    print(x.shape, y.shape)
    break
```

Il risultato della stampa è `torch.Size([64, 1, 28, 28]), torch.Size([64])` dove il primo risultato rappresenta la dimensione relativa al campione mentre il secondo risultato rappresenta la dimensione relativa al target. In particolare il primo elemento indica la dimensione del batch (in questo caso stiamo considerando insiemi batch di 64 elementi), il secondo elemento indica la dimensione del canale delle immagini (essendo in scala di

grigi avremo dimensione 1, mentre se ad esempio fosse stato RGB allora avremmo avuto 3) e gli ultimi due elementi indicano la dimensione dell'immagine (28×28 pixels). Il secondo risultato restituisce la dimensione del batch (ovvero contiene i target di tutti i 64 campioni appartenenti al batch).

Durante l'addestramento necessitiamo di un criterio di ottimizzazione. Inoltre vorremmo evitare di aggiornare manualmente i parametri del modello, quindi utilizziamo un optimizer già definito.

I criteri (ovvero le funzioni loss) possono essere direttamente derivabili da `torch.nn` (ad esempio per la loss Cross Entropy avremo `torch.nn.CrossEntropyLoss()`).

Gli ottimizzatori, invece, sono definiti nel namespace `torch.optim` (ad esempio l'ottimizzatore SGD è definito come `torch.optim.SGD()`).

Una volta valutata la loss al termine del passo forward, possiamo applicare la backpropagation invocando il metodo `loss.backward()` sulla funzione loss in modo da calcolare i gradienti.

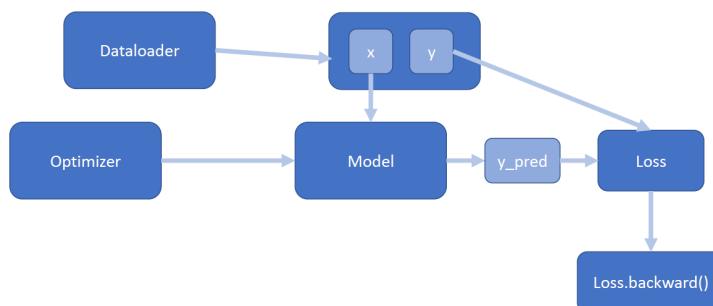


Figura 9.4: Rappresentazione del processo di training di una neural network.



ES:

Vediamo una semplice implementazione di un processo di addestramento:

```

learning_rate, batch_size, epochs = 1e-3, 64, 5
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

size = len(train_dataloader.dataset)
for epoch in range(epochs):
    #training
    for batch, (X, y) in enumerate(train_dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
  
```

Ricordiamo che l'utilizzo della GPU può risultare molto comodo e importante per velocizzare il processo di addestramento. In particolare se è disponibile un dispositivo

GPU, dobbiamo semplicemente spostare il modello su tale dispositivo e fare la stessa cosa con i tensori di input.

```
model = model.to('cuda')

#in the training loop

x = x.to('cuda')
y = y.to('cuda')
```

9.2.1 Convolutional Neural Networks

Nell'ambito del Deep Learning, una **Convolutional Neural Network (CNN)** rappresenta una classe di reti neurali artificiali, comunemente applicata all'analisi delle immagini visive.

Le CNN sono versioni regolarizzate dei Multilayer Perceptron, dove gli MLP rappresentano reti completamente connesse in cui ogni neurone di un layer viene connesso a tutti i neuroni del layer successivo. Però la totale connettività di queste reti le rende inclini all'overfitting dei dati. Quindi si prevedono metodi tipici di regolarizzazione, o di prevenzione dell'overfitting, che includono la penalizzazione dei parametri durante l'addestramento (come il weight decay) oppure la riduzione della connettività (come il dropout).

Le CNN adottano un approccio diverso alla regolarizzazione. Queste sfruttano il modello gerarchico dei dati e assemblano modelli di complessità crescente utilizzando modelli più piccoli e più semplici.

Quindi le CNN sono semplicemente reti neurali che utilizzano la convoluzione al posto della generale moltiplicazione matriciale in almeno uno dei loro layer.

Le CNN sono state ispirate dallo studio degli scienziati Hubel e Wiesel, vincitori del premio Nobel, che hanno dimostrato il funzionamento della corteccia visiva nel cervello animale. In particolare hanno inserito dei microelettrodi nella corteccia visiva di un gatto e hanno proiettato una linea luminosa attraverso la sua retina. Quindi hanno notato che i neuroni si attivavano quando la linea si trovava in un punto particolare della retina, che l'attività dei neuroni cambiava a seconda dell'orientamento della linea e che i neuroni a volte si attivavano solo quando la linea si muoveva in una particolare direzione.

Questo esperimento ha mostrato come la corteccia visiva elabori le informazioni in modo gerarchico, estraendo informazioni sempre più complesse.

In particolare hanno dimostrato che nella corteccia visiva esiste una mappa topografica che rappresenta il campo visivo, dove le cellule vicine elaborano le informazioni provenienti dai campi visivi vicini. Questo ha dato origine al concetto di interazioni sparse nelle CNN, in cui la rete si concentra sulle informazioni locali piuttosto che sull'intera informazione globale. Questa proprietà fa sì che le CNN forniscano prestazioni migliori nei problemi legati alle immagini, poiché nelle immagini i pixel vicini sono più fortemente correlati di quelli lontani.

Inoltre, il loro lavoro ha determinato che i neuroni della corteccia visiva sono disposti secondo una precisa architettura. Le cellule con funzioni simili sono organizzate in colonne. Questo è simile al modo in cui è progettata l'architettura di una CNN, in cui i

layer inferiori estraggono i bordi e altre caratteristiche comuni e gli strati superiori estraggono informazioni più specifiche della classe.

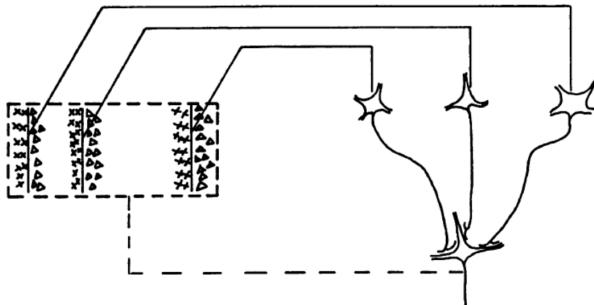


Figura 9.5: Schema che spiega l'organizzazione dei campi recettivi complessi. Si immagina che un certo numero di cellule con campi semplici, di cui tre sono mostrate schematicamente, proiettino a una singola cellula corticale di ordine superiore. Ogni neurone proiettante ha un campo recettivo disposto come mostrato a sinistra, costituito da una regione eccitatoria a sinistra e da una regione inibitoria a destra di un confine verticale rettilineo. I confini dei campi sono sfalsati all'interno di un'area delimitata dalle linee interpunktate. Qualsiasi stimolo che attraversi questo rettangolo, indipendentemente dalla sua posizione, ecciterà alcune cellule a campo semplice, portando all'eccitazione della cellula di ordine superiore.

In passato, per il riconoscimento delle immagini venivano utilizzati i tradizionali modelli di MLP. Tuttavia, la piena connettività tra i nodi causava il problema noto come *curse of dimensionality* che, nel caso di immagini ad alta risoluzione, risultava essere computazionalmente intrattabile. Ad esempio, consideriamo un'immagine in input di 128×128 pixel e ipotizziamo di realizzare un modello con un singolo hidden layer di 32 unità. Allora otteniamo circa 500 mila pesi. Quindi possiamo facilmente intuire che immagini leggermente più grandi supereranno rapidamente un numero di pesi di 1M (per un modello che utilizza un singolo hidden layer), ottenendo un modello poco efficiente.

Inoltre, tale architettura di rete non tiene conto della struttura spaziale dei dati, trattando allo stesso modo i pixel di input che risultano essere distanti e quelli che risultano essere vicini tra loro. Quindi nei dati che possiedono una topologia a griglia (come le immagini) si ignora la proprietà di Locality. Infatti, le immagini vengono rappresentate come una griglia bidimensionale di pixel (attraverso una matrice) che possono essere sia monocromatici che a colori. Utilizzando reti completamente connesse (come MLP) si ignora questa ricca struttura, appiattendola e ignorando la relazione spaziale tra i pixel.

Inoltre la completa connettività della rete non soddisfa la proprietà di Translation Invariance, ovvero non riesce a riconoscere uno stesso oggetto quando subisce dei cambiamenti nell'aspetto (come rotazioni, traslazioni, ecc...).

In generale, nei primi layer la rete dovrebbe rispondere in modo simile alla stessa *patch* (macchia), indipendentemente dalla sua posizione nell'immagine. Questo principio è chiamato **Translation Invariance** (*invarianza di traslazione*). Inoltre i primi layer della rete devono concentrarsi sulle regioni locali, senza tenere conto del contenuto dell'immagine nelle regioni lontane. Questo principio è chiamato **Locality** (*località*). Alla fine, queste rappresentazioni locali possono essere aggregate per fare previsioni a livello dell'intera immagine. Quindi i layer più profondi dovrebbero essere in grado di catturare caratteristiche dell'immagine a più lungo raggio. Vediamo come possiamo tradurre in forma matematica queste osservazioni.

Innanzitutto consideriamo una MLP che prende in input immagini bidimensionali X e che possiede una *hidden representation* H (dove X ed H hanno la stessa struttura). Siano $X_{i,j}$ ed $H_{i,j}$ i pixel in posizione (i, j) rispettivamente nell'immagine di input e nella rappresentazione nascosta (ovvero $H_{i,j}$ è la rappresentazione nascosta per il pixel $X_{i,j}$ dell'immagine di input X).

Se consideriamo un Fully Connected Layer, allora, rimuovendo i bias per semplicità, possiamo calcolare $H_{i,j}$ come:

$$H_{i,j} = \sum_k \sum_l W_{i,j,k,l} X_{k,l}$$

dove W rappresenta la matrice dei pesi.

Quindi abbiamo ottenuto un Fully Connected Layer in cui per ogni pixel della rappresentazione nascosta vengono valutati tutti i pixel in input.

Cambiando gli indici con $k = i + a$ ed $l = j + b$ otteniamo:

$$H_{i,j} = \sum_a \sum_b W_{i,j,i+a,j+b} X_{i+a,j+b}$$

Introducendo $V_{i,j,a,b} = W_{i,j,i+a,j+b}$ ricaviamo:

$$H_{i,j} = \sum_a \sum_b V_{i,j,a,b} X_{i+a,j+b}$$

Ovvero per ogni posizione (i, j) nella rappresentazione nascosta $H_{i,j}$ si calcola il suo valore sommando i pixel in X centrati intorno a (i, j) e ponderati per $V_{i,j,a,b}$.

Translation Invariance. Tale principio indica che uno spostamento dell'input X dovrebbe semplicemente portare ad uno spostamento della rappresentazione nascosta H . Questo è possibile solo se V non dipende effettivamente dalla posizione dei pixel (i, j) . Quindi, l'unico modo per ottenere l'invarianza dalla posizione dei pixel è quello di rimuovere (i, j) dal tensore dei pesi V (ovvero $V_{i,j,a,b} = V_{a,b}$). In questo modo possiamo semplificare la definizione di H :

$$H_{i,j} = \sum_a \sum_b V_{a,b} X_{i+a,j+b}$$

Questo risultato ottenuto è una *convoluzione*. Infatti stiamo ponderando i pixel in posizione $(i+1, j+b)$ in prossimità della posizione (i, j) con i coefficienti $V_{a,b}$ per ottenere il valore $H_{i,j}$.

N.B: osserviamo che che $V_{a,b}$ necessita di molti meno coefficienti rispetto a $V_{i,j,a,b}$, poiché non dipende più dalla posizione all'interno dell'immagine.

Convoluzione. In matematica, la convoluzione tra due funzioni $f, g : \mathbb{R}^d \rightarrow \mathbb{R}$ è definita come:

$$f(x) * g(x) = \int_{z=-\infty}^{\infty} f(z) \cdot g(x-z) dz$$

Questa rappresenta la sovrapposizione tra f e g quando una delle due funzioni viene ribaltata e spostata di x .

In presenza di funzioni discrete, allora l'integrale si trasforma in una somma. Quindi la convoluzione discreta è definita come:

$$f(x) * g(x) = \sum_{z=-N}^N f(z) \cdot g(x-z, y)$$

Locality. Questo principio indica che nei primi layer non si necessita di osservare l'intera immagine per ottenere una caratteristica importante. Infatti si ritiene che per ottenere informazioni rilevanti per valutare ciò che sta accadendo in $H_{i,j}$ non si debba guardare molto lontano dalla posizione (i, j) . Quindi possiamo imporre la località all'operatore di convoluzione azzerando la risposta di V al di fuori di una certa regione centrata nei pixel (i, j) (ovvero al di fuori di un certo intervallo $|a| > \Delta$ o $|b| > \Delta$, dovremmo impostare $V_{a,b} = 0$). Quindi possiamo riscrivere $H_{i,j}$ come:

$$H_{i,j} = \sum_{b=-\infty}^{\infty} \sum_{a=-\infty}^{\infty} V_{a,b} X_{i+a,j+b} \rightarrow \sum_{b=-\Delta}^{\Delta} \sum_{a=-\Delta}^{\Delta} V_{a,b} X_{i+a,j+b}$$

N.B: questo semplifica ulteriormente il numero di parametri. Mentre in precedenza potevano essere necessari miliardi di parametri per rappresentare un solo layer di una rete, adesso ne bastano poche centinaia, senza alterare la dimensionalità degli ingressi o delle rappresentazioni nascoste. Il prezzo da pagare per questa drastica riduzione dei parametri sarà che le caratteristiche sono adesso invarianti alla traduzione (Translation Invariance) e che il layer può incorporare soltanto informazioni locali (Locality).

N.B: questo risultato è detto **Convolutional Layer** e V è detto **convolutional kernel** (o **filtro**) e rappresenta un tensore di pesi. In particolare le Convolutional Neural Networks sono una famiglia speciale di reti neurali che contengono Convolutional Layers.

N.B: lavorando in due dimensioni (ad esempio con le immagini) questo risultato rappresenta il prodotto interno tra i pesi nel convolutional kernel ed i pixel contenuti nella finestra di dimensioni $-\Delta \times \Delta$.

Canali. Abbiamo visto che il Convolutional Layer seleziona delle finestre di una determinata dimensione e pesa le intensità in base al kernel V . Questo approccio però presenta un problema. Infatti fino ad adesso abbiamo ignorato il fatto che le immagini possono essere costituite da 3 canali (RGB) e quindi le immagini non sono oggetti bidimensionali, ma piuttosto tensori del terzo ordine (caratterizzati da altezza, larghezza e canale). Mentre i primi due di questi assi riguardano le relazioni spaziali, il terzo può essere considerato come l'assegnazione di una rappresentazione multidimensionale a ciascuna posizione dei pixel.

Quindi invece di avere una singola rappresentazione nascosta corrispondente a ogni posizione spaziale, vogliamo un intero vettore di rappresentazioni nascoste corrispondenti a ogni posizione spaziale.

Possiamo pensare alle rappresentazioni nascoste come a una serie di griglie bidimensionali impilate l'una sull'altra, dette **canali** oppure **feature map**, poiché ognuna di esse fornisce al layer successivo un insieme spazializzato di feature apprese. Consideriamo quindi un'immagine I definita come un tensore del terzo ordine, ad esempio $I \in \mathbb{R}^{W \times H \times 3}$. Ipotizziamo di effettuare una convoluzione con un singolo kernel $k \in \mathbb{R}^{n \times n \times 3}$. Il risultato di questa convoluzione sarà una *feature map* (ovvero una mappa delle feature) di dimensioni $[W - (n - 1)] \times [H - (n - 1)] \times 1$.

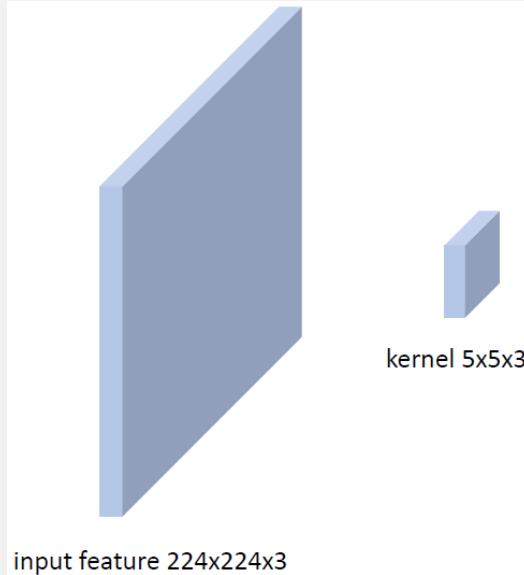
In generale possiamo applicare **convoluzioni multiple** con differenti kernel $k_c \in \mathbb{R}^{n \times n \times 3}$, ottenendo c *feature map* che della stessa forma. Quindi possiamo assemblare queste mappe in un unico tensore con c canali.

Perciò possiamo applicare differenti convoluzioni alle *feature map* successive, creando dei kernel con la stessa quantità di canali della feature map in input ai kernel.



ES:

Consideriamo un input di dimensione $224 \times 224 \times 3$ (ad esempio un'immagine RGB con risoluzione 224×224 pixel). Immaginiamo di effettuare una convoluzione di questa immagine con un kernel di dimensione $5 \times 5 \times 3$.



Idealmente, il kernel “scorre” sulla feature di input calcolando la seguente convoluzione:

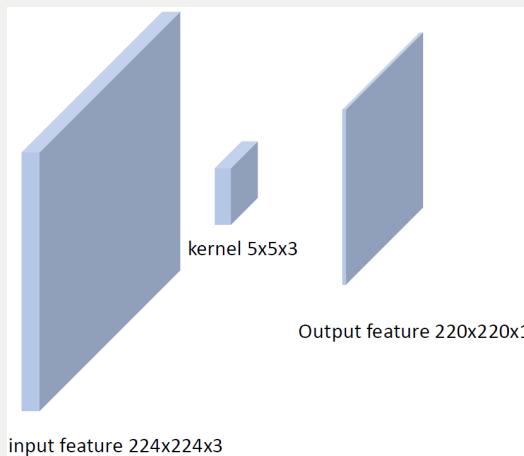
$$k(x, y) * I(x, y) = \int_{s_1=-\infty}^{\infty} \int_{s_2=-\infty}^{\infty} k(s_1, s_2) \cdot I(x - s_1, y - s_2) ds_1 ds_2$$

dove k indica la funzione kernel ed I indica l'immagine.

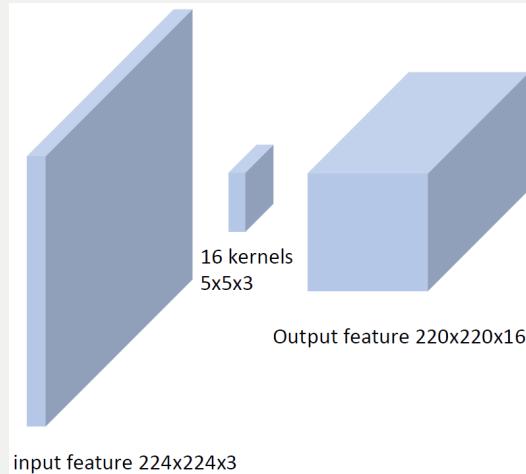
In particolare, per un'immagine definiamo una convoluzione discreta il cui output è dato da:

$$k(x, y) * I(x, y) = \sum_{s_1=-N}^{N} \sum_{s_2=-N}^{N} k(s_1, s_2) \cdot I(x - s_1, y - s_2)$$

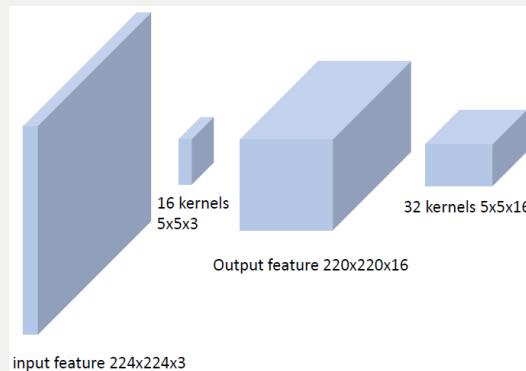
Il risultato di questa convoluzione sarà un'immagine con un singolo canale e di risoluzione ridotta (otteniamo un output di dimensione $220 \times 220 \times 1$):



Ipotizziamo adesso di applicare 16 kernel di dimensioni $5 \times 5 \times 3$ in parallelo. Allora il risultato ottenuto dalla convoluzione sarà un'immagine di dimensioni $220 \times 220 \times 16$ (abbiamo assemblato 16 *feature map* della stessa forma).



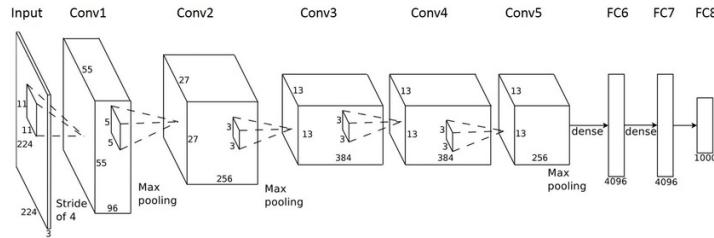
Quindi applichiamo 32 kernel di dimensioni $5 \times 5 \times 16$ (utilizziamo lo stesso numero di canali definiti dalla feature map in input ai kernel).



Il risultato di questa convoluzione sarà un'immagine di dimensione $216 \times 216 \times 32$.

Convolutional Architecture

Combiniamo un'architettura di una Neural Network costituita da sequenza di 5 Convolutional Layer, alcuni dei quali applicano un *max pooling*, combinata con 3 Fully Connected Layer per l'apprendimento delle feature. Tale architettura è detta **AlexNet**. **N.B:** le convoluzioni sono soltanto delle operazioni lineari applicate localmente e quindi differenziabili.



Vediamo come funziona l'operazione di convoluzione. Come abbiamo osservato, l'oggetto della convoluzione è rappresentato da un tensore di terzo grado (ovvero una matrice a tre dimensioni) che rappresenta i dati in input ed il corrispondente convolutional kernel.

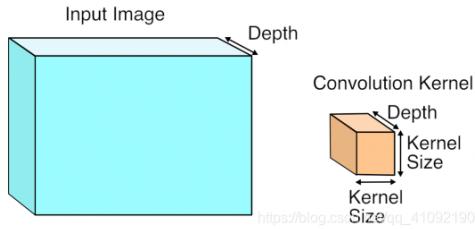


Figura 9.6: Illustrazione che mostra il funzionamento dell'operazione di convoluzione. Il blocco blu indica la matrice dei dati di input di dimensione $W \times H \times c$, mentre il blocco arancione indica il convolutional kernel di dimensione $n \times n \times c$.

Tale convolutional kernel scorre sui dati di input, da sinistra a destra e dall'alto in basso, in determinati passi, calcolando il valore di output corrispondente in ogni posizione.

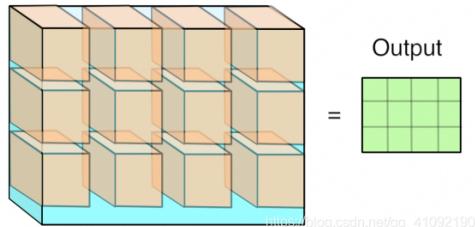


Figura 9.7: Il blocco blu indica i dati di ingresso, mentre i blocchi arancioni mostrano le regioni in cui si muove il kernel. La griglia verde mostra i dati di output dopo l'operazione di convoluzione.

Le convoluzioni sono convenientemente considerate come moltiplicazioni tra matrici. Quindi la funzione *img2col* trasforma il tensore della feature map, trasformandola in una matrice.

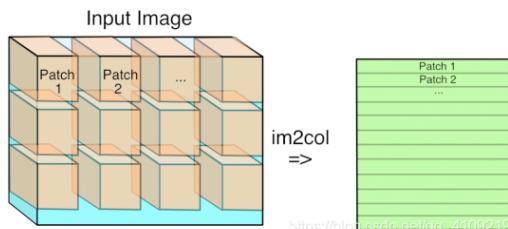
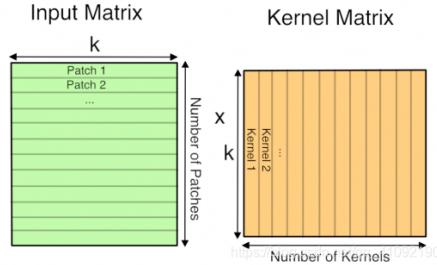


Figura 9.8: La funzione *img2col* prende i dati di input di una matrice a 3 dimensioni (cioè $W \times H \times c$) e li trasforma in una matrice a 2 dimensioni (cioè $H * W \times n * n * c$).

Qui si presume che l'altezza e la larghezza dell'input e dell'output siano uguali.

Come possiamo osservare il numero di righe della matrice di output è dato il numero di volte in cui il kernel di convoluzione si sposta sui dati di input, mentre il numero di colonne è dato dal numero di elementi del kernel.

A questo punto possiamo applicare una moltiplicazione tra la matrice ottenuta in precedenza e la matrice del kernel, ottenendo un prodotto scalare tra ogni kernel ed ogni patch.



Padding

Un problema che si verifica quando si applicano dei Convolutional Layer è che si tende a perdere pixel sui bordi dell'immagine (infatti abbiamo osservato che otteniamo come risultato della convoluzione un'immagine con minore risoluzione). Generalmente una singola convoluzione porta un piccolo numero di pixel persi, ma questo valore può diventare non indifferente quando si applicano una serie di convoluzioni successive (Che comportano un alto numero di pixel persi).

Una soluzione semplice a questo problema consiste nell'aggiungere ulteriori pixel di riempimento attorno al perimetro dell'immagine di input, aumentandone così la dimensione effettiva. Tipicamente impostiamo i valori di questi pixel aggiuntivi a 0. Tale tecnica è detta **padding**.

Questa tecnica permette di preservare le features e risulta utile per architetture molto profonde.

Input	Kernel	Output
$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 \\ 0 & 3 & 4 & 5 & 0 \\ 0 & 6 & 7 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$	$\begin{bmatrix} 0 & 3 & 8 & 4 \\ 9 & 19 & 25 & 10 \\ 21 & 37 & 43 & 16 \\ 6 & 7 & 8 & 0 \end{bmatrix}$

Figura 9.9: Consideriamo un input di dimensione 3×3 ed effettuiamo il padding, ottenendo così una matrice di dimensione aumentata 5×5 . L'output della convoluzione genera una matrice di dimensioni 4×4 .

L'output evidenziato in celeste è stato ottenuto dalla convoluzione tra gli elementi di input evidenziati ed il kernel, cioè $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$.

Stride

Quando calcoliamo la convoluzione, iniziamo il calcolo con l'angolo in alto a sinistra del tensore di input. Quindi la facciamo scorrere il kernel su tutte le posizioni sia in basso che a destra. Negli esempi precedenti scorrevamo un singolo elemento alla volta. Tuttavia, a volte, per efficienza computazionale o perché desideriamo eseguire il downsampling, spostiamo la finestra più di un elemento alla volta, saltando delle posizioni intermedie.

Il numero di righe e colonne attraversate viene chiamato **stride**.

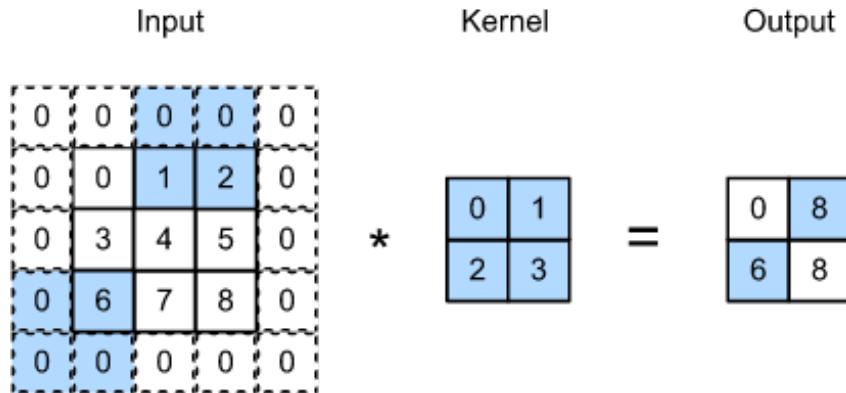


Figura 9.10: Applichiamo uno stride di $(3, 2)$, cioè ogni 2 pixel sulla larghezza e ogni 3 pixel sull'altezza.

Gli elementi dell'output evidenziati in celeste sono ottenuti dalla convoluzione tra gli elementi di input evidenziati in celeste ed il kernel, cioè $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$.

Pooling Layers

Spesso, mentre elaboriamo le immagini, vogliamo ridurre gradualmente la risoluzione spaziale delle nostre rappresentazioni nascoste, aggregando le informazioni in modo che più in alto andiamo nella rete, più grande sarà il campo ricettivo a cui ogni hidden node è sensibile. Spesso l'obiettivo finale è quello di porre qualche domanda globale sull'immagine (per esempio vogliamo sapere se l'immagine contiene un gatto). Quindi in genere le unità del nostro livello finale dovrebbero essere sensibili all'intero input. Aggregando gradualmente le informazioni, producendo mappe sempre più grossolane, raggiungiamo questo obiettivo di apprendere una rappresentazione globale, mantenendo tutti i vantaggi degli strati convoluzionali negli strati intermedi dell'elaborazione.

I **Pooling Layers** (o layer di raggruppamento) sono operatori deterministici che computano un'operazione su una finestra di forma fissa che viene fatta scorrere su tutte le regioni dell'input in base al suo passo, calcolando un singolo output per ogni posizione attraversata dalla finestra (detta **pooling window**).

In genere gli operatori pooling calcolano il valore massimo o il valore medio degli elementi contenuti nella finestra di raggruppamento (pooling window). Queste operazioni sono chiamate rispettivamente **max pooling** e **average pooling**. In particolare, un Max Pooling Layer estrae il massimo tra i valori raggruppati.

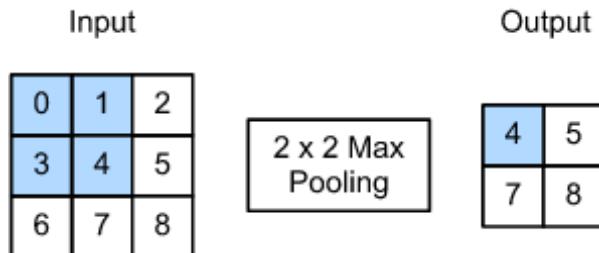


Figura 9.11: Raggruppamento massimo con una pooling window di dimensione 2×2 . L'output evidenziato in celeste viene ottenuto dal valore massimo tra quelli presenti nel raggruppamento dei dati in input (evidenziato in celeste), cioè $\max(0, 1, 3, 4) = 4$.

N.B.: a differenza della convoluzione degli input e dei kernel nel Convolutional Layer, il Pooling Layer non contiene parametri (cioè non esiste il kernel).

N.B.: il Pooling Layer aggiunge non linearità e robustezza alle localizzazioni delle risposte. Inoltre, se combinato con stride, permette di campionare le feature, riducendo la dimensionalità.

Durante la fase di backpropagation, il max pooling propagherà il gradiente solo per il neurone attivo. La funzione max agisce quindi come un commutatore di gradiente.

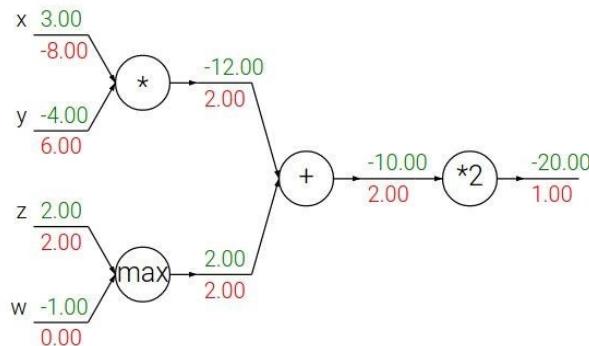


Figura 9.12: Durante la fase di backpropagation, la funzione max propaga il gradiente solamente all'unità che corrisponde al massimo.

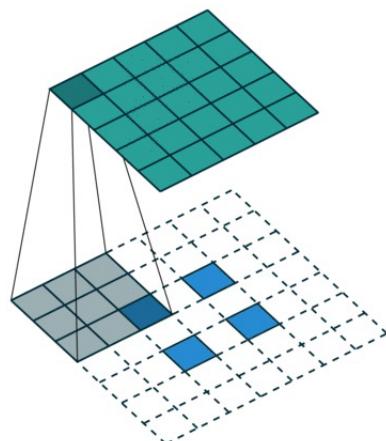
Implementazione

Possiamo definire facilmente i Convolutional Layer di PyTorch utilizzando il modulo `Conv2d`.

```
conv = torch.nn.Conv2d(in_channels=1,out_channels=1,kernel_size=5)
out = conv(im_t) #compute the output
```

In questo caso stiamo calcolando la convoluzione di un singolo kernel casuale 5×5 su un'immagine im_t .

La convoluzione può essere utilizzata sia per il downsampling che per l'upsampling. In particolare per effettuare l'upsampling vengono utilizzate le convoluzioni trasposte. Rispetto a una convoluzione standard, queste sono calcolate in modo che la feature map in output sia più grande della feature map in input.



Le Convolutional Neural Networks sono architetture che incorporano strati convoluzionali.

I Convolutional Layer sono adatti a tutti i dati per i quali è possibile apprendere caratteristiche locali e invarianti alla traslazione (ad esempio le immagini).

Le architetture di base sono create impilando una serie di Convolutional Layer.

Di solito vengono utilizzati alcuni Linear Layer per eseguire la classificazione finale.