

# Analisi Comparativa di KAN, MLP, Random forest e XGBoost con Tecniche di Ottimizzazione

Martin Tomassi  
Università di Bologna  
Corso di Laurea in Ingegneria e Scienze Informatiche  
Settembre 2025

# Indice

<b>1</b>	<b>Introduzione</b>	<b>6</b>
<b>2</b>	<b>Multi-Layer Perceptron (MLP)</b>	<b>7</b>
2.1	Fondamenti matematici . . . . .	8
2.1.1	Enunciato formale . . . . .	8
2.1.2	Teorema di approssimazione universale . . . . .	8
2.1.3	Significato di $\sup_{x \in K}$ . . . . .	9
2.1.4	Ipotesi e Precisazioni . . . . .	9
2.2	Struttura delle MLP . . . . .	10
2.2.1	Strati: input, nascosti, output . . . . .	10
2.2.2	Funzioni di attivazione comuni . . . . .	11
2.3	Procedura di forward pass . . . . .	19
2.3.1	Calcolo delle attivazioni . . . . .	19
2.3.2	Propagazione ed output . . . . .	20
2.4	Algoritmo di backpropagation . . . . .	20
2.4.1	Derivazione del gradiente . . . . .	20
2.4.2	Aggiornamento dei pesi . . . . .	21
2.4.3	Tecniche di regolarizzazione . . . . .	22
2.5	Vantaggi e limiti . . . . .	23
2.5.1	Flessibilità e capacità di generalizzazione . . . . .	23
2.5.2	Problemi di vanishing/exploding gradient . . . . .	23
2.5.3	Efficienza computazionale . . . . .	24
<b>3</b>	<b>Kolmogorov–Arnold Networks (KAN)</b>	<b>25</b>
3.1	Fondamenti matematici . . . . .	26

3.1.1	Enunciato del teorema di Kolmogorov–Arnold . . . .	26
3.1.2	Sulla natura "non costruttiva" delle dimostrazioni . .	26
3.2	Architettura delle KAN . . . . .	27
3.2.1	Funzioni univariate parametriche . . . . .	28
3.2.2	B-spline . . . . .	30
3.2.3	Scaling laws e Curse of dimensionality . . . . .	32
3.3	Funzionamento operativo . . . . .	32
3.3.1	Calcolo del mapping input–hidden–output . . . . .	32
3.3.2	Processo di training e Calcolo dei pesi . . . . .	33
3.4	Confronto con MLP tradizionali . . . . .	34
3.4.1	Architettura a confronto . . . . .	34
3.4.2	Complessità computazionale . . . . .	35
3.4.3	Interpretabilità e flessibilità locale . . . . .	36
3.4.4	Precisione controllabile tramite grid extension . . . .	37
3.4.5	Dipendenza dalla struttura compositiva . . . . .	37
3.4.6	Irregolarità nella rappresentazione di Kolmogorov .	38
3.4.7	Overhead computazionale e scelta della struttura . .	38
3.4.8	Sensibilità al rumore e necessità di regolarizzazione .	38
<b>4</b>	<b>Random forest</b>	<b>39</b>
4.1	Concetti fondamentali . . . . .	40
4.1.1	Alberi di decisione: Criteri di splitting (Gini, Entropia, Gain ratio) . . . . .	40
4.1.2	Ensemble learning e Bagging . . . . .	42
4.2	Architettura e Costruzione . . . . .	44
4.2.1	Bootstrapping e Feature bagging . . . . .	44
4.2.2	Aggregazione delle predizioni . . . . .	45
4.3	Vantaggi . . . . .	46
4.4	Svantaggi . . . . .	47
<b>5</b>	<b>eXtreme Gradient Boosting (XGBoost)</b>	<b>49</b>
5.1	Introduzione al Gradient boosting . . . . .	50
5.1.1	Principi iterativi e Weak learners . . . . .	50

5.1.2	Funzione di perdita e Discesa del gradiente . . . . .	51
5.2	Miglioramenti di XGBoost rispetto al Gradient boosting tradizionale . . . . .	52
5.2.1	Regolarizzazione e Tree pruning . . . . .	53
5.2.2	Gestione dei valori mancanti . . . . .	54
5.2.3	Ottimizzazioni (Parallelismo, Cache-awareness) . . .	54
5.3	Parametri chiave e Tuning . . . . .	55
5.4	Vantaggi . . . . .	57
5.5	Svantaggi . . . . .	58
<b>6</b>	<b>Convolutional Neural Networks (CNN)</b>	<b>60</b>
6.1	Principi fondamentali delle CNN . . . . .	60
6.1.1	Convoluzione: kernel, stride, padding . . . . .	60
6.1.2	Feature maps e profondità dei canali . . . . .	61
6.1.3	Convoluzioni 1D, 2D e 3D . . . . .	62
6.2	Pooling e normalizzazione . . . . .	63
6.2.1	Max pooling vs average pooling . . . . .	63
6.2.2	Global pooling . . . . .	63
6.2.3	Batch, Layer e Group normalization . . . . .	63
6.3	Data augmentation: rotazioni, zoom, colour jitter . . . . .	64
6.4	Funzionamento delle CNN . . . . .	65
6.4.1	Forward pass . . . . .	65
6.4.2	Strati di convoluzione . . . . .	65
6.4.3	Funzioni di attivazione . . . . .	66
6.4.4	Strati di pooling . . . . .	66
6.4.5	Gerarchia delle caratteristiche . . . . .	66
6.4.6	Strati fully-connected . . . . .	67
6.4.7	Flusso informativo e Trasformazioni progressive . . .	67
6.5	Vantaggi . . . . .	67
6.6	Svantaggi . . . . .	68
<b>7</b>	<b>Ottimizzazione degli iperparametri</b>	<b>70</b>
7.1	Cross-Validation . . . . .	70

7.1.1	Time Series Cross-Validation (TSCV)	71
7.2	Nested Cross-Validation (NCV)	73
7.3	Grid search (GS)	75
7.3.1	Spiegazione dell'algoritmo	75
7.3.2	Vantaggi	76
7.3.3	Limiti	76
7.4	Random Search (RS)	77
7.4.1	Spiegazione dell'algoritmo	77
7.4.2	Vantaggi	78
7.4.3	Limiti	78
7.5	Bayesian optimization (BO)	79
7.5.1	Spiegazione dell'algoritmo	79
7.5.2	Vantaggi	80
7.5.3	Limiti	81
7.6	Genetic algorithm (GA)	82
7.6.1	Spiegazione dell'algoritmo	82
7.6.2	Vantaggi	84
7.6.3	Limiti	85
7.7	Confronto pratico	85
7.7.1	Criteri per la scelta	85
7.7.2	Tabella riassuntiva comparativa	86
7.7.3	Matrice decisionale per la scelta del metodo	86
7.7.4	Scelta per i casi studio: Random search	86
7.7.5	Ottimizzazione del numero di iterazioni nel Random search	87
<b>8</b>	<b>Studio di ablazione e Pruning post-training</b>	<b>89</b>
8.1	Fondamenti teorici degli studi di ablazione	89
8.1.1	Definizione	89
8.1.2	Benefici	90
8.2	Pruning post-training	90
8.2.1	Definizione	90
8.2.2	Benefici	91

8.2.3	Trade-off bias-varianza . . . . .	91
8.3	Pruning L1 post-training per MLP e KAN . . . . .	92
8.3.1	Definizione . . . . .	92
8.3.2	Considerazioni specifiche per le KAN . . . . .	92
8.4	Pruning per Ensemble: Rank-based pruning per Random forest . . . . .	93
8.4.1	Principio fondamentale . . . . .	93
8.4.2	Criterio di ranking basato sulla feature importance .	93
8.4.3	Procedura di selezione . . . . .	94
8.5	Pruning per Ensemble: Cumulative pruning per XGBoost . .	94
8.5.1	Criterio di pruning cumulativo . . . . .	94
8.5.2	Procedura di selezione . . . . .	94
<b>9</b>	<b>Primo Caso Studio: Regressione su emissioni di automobili</b>	<b>96</b>
<b>10</b>	<b>Secondo Caso Studio: Classificazione di PM2.5</b>	<b>97</b>
<b>11</b>	<b>Terzo Caso Studio: Classificazione di età tramite immagini</b>	<b>98</b>
<b>12</b>	<b>Discussione comparativa sui Risultati dei casi studio</b>	<b>99</b>
<b>13</b>	<b>Conclusioni</b>	<b>100</b>

# Capitolo 1

## Introduzione

La presente tesi si propone di analizzare, confrontare e valutare diverse architetture di Machine e Deep learning: le Kolmogorov–Arnold Networks (KAN), i Multi-Layer Perceptron (MLP), le Random forest e XGBoost.

L'obiettivo principale è quello di fornire, da un lato, un'analisi teorica esaustiva di ciascun modello, includendo i fondamenti matematici e le architetture; dall'altro, valutare sperimentalmente le prestazioni dei modelli su tre casi di studio: regressione sulle emissioni di automobili, classificazione dell'inquinamento atmosferico (PM2.5) e riconoscimento di immagini mediante CNN abbinate a MLP e KAN.

Oltre allo studio comparativo dei modelli, la tesi include un'ampia indagine sui metodi di Hyperparameter Tuning: Grid search, Random search, Bayesian optimization e Genetic algorithms, selezionati in base ad uno studio preliminare di confronto.

A completamento, sono stati condotti due studi di ablazione post-training per ogni caso di studio:

- **L1 Pruning** per KAN e MLP.
- **Ensemble Pruning** dove:
  - **Rank-Based Pruning** per Random forest.
  - **Cumulative Pruning** per XGBoost.

## Capitolo 2

# Multi-Layer Perceptron (MLP)

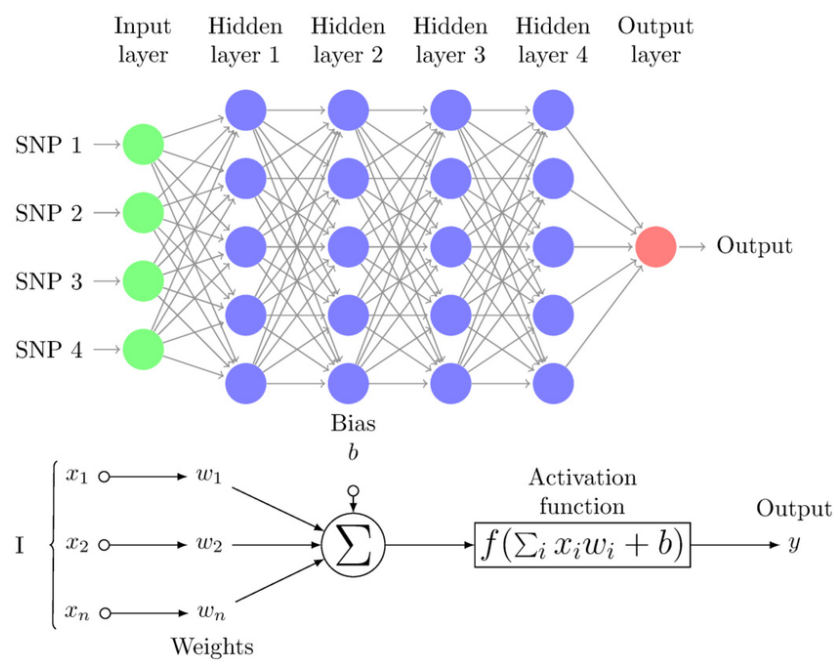


Figura 2.1: Multi-Layer Perceptron.



## 2.1 Fondamenti matematici

### 2.1.1 Enunciato formale

Sia  $K \subset \mathbb{R}^n$  uno spazio compatto e sia  $C(K)$  lo spazio delle funzioni continue su  $K$  munito della norma uniforme  $\|\cdot\|_\infty$ . Sia  $\sigma : \mathbb{R} \rightarrow \mathbb{R}$  una funzione di attivazione che soddisfa una delle seguenti ipotesi:

(A<sub>1</sub>)  $\sigma$  è continua e sigmoide, cioè  $\lim_{t \rightarrow -\infty} \sigma(t) = a$  e  $\lim_{t \rightarrow +\infty} \sigma(t) = b$  con  $a \neq b$  (Cybenko);

(A<sub>2</sub>)  $\sigma$  è continua e non polinomiale (Leshno et al.).

Allora vale il seguente risultato di approssimazione universale.

### 2.1.2 Teorema di approssimazione universale

Per ogni  $f \in C(K)$  e per ogni  $\varepsilon > 0$ , esistono un intero  $N \in \mathbb{N}$  (indica il numero di neuroni nello strato nascosto), coefficienti scalari  $c_i \in \mathbb{R}$ , vettori  $w_i \in \mathbb{R}^n$  e bias  $b_i \in \mathbb{R}$  tali che la funzione a singolo strato

$$\hat{f}_N(x) = \sum_{i=1}^N c_i \sigma(w_i \cdot x + b_i)$$

soddisfa

$$\|f - \hat{f}_N\|_\infty = \sup_{x \in K} |f(x) - \hat{f}_N(x)| < \varepsilon.$$

In altre parole, lo span delle funzioni elementari (cioè l'insieme di tutte le possibili combinazioni lineari di queste funzioni)  $x \mapsto \sigma(w \cdot x + b)$  è denso in  $C(K)$  rispetto alla norma uniforme. Ciò significa che per ogni funzione continua  $f \in C(K)$  e per ogni  $\varepsilon > 0$  esiste una combinazione lineare finita di blocchi attivati da  $\sigma$  (cioè una rete a singolo strato nascosto) che approssima  $f$  uniformemente su  $K$  con errore massimo minore di  $\varepsilon$ . Formalmente, la chiusura (nell' $\|\cdot\|_\infty$ ) dello spazio generato dalle funzioni elementari coincide con l'intero  $C(K)$ .

### 2.1.3 Significato di $\sup_{x \in K}$ .

La notazione  $\sup_{x \in K}$  denota l'estremo superiore di un insieme di reali. Nella norma uniforme

$$\|f - \hat{f}_N\|_\infty = \sup_{x \in K} |f(x) - \hat{f}_N(x)|$$

il valore indicato è l'errore massimo di approssimazione su tutto il dominio  $K$ . Poiché nel teorema  $K$  è assunto compatto e la funzione  $x \mapsto |f(x) - \hat{f}_N(x)|$  è continua, l'estremo superiore coincide con il massimo:

$$\sup_{x \in K} |f(x) - \hat{f}_N(x)| = \max_{x \in K} |f(x) - \hat{f}_N(x)|.$$

**Esempio** Se  $K = [0, 1]$  e  $f(x) = \sin(2\pi x)$ , affermare che esiste  $N$  tale che

$$\sup_{x \in [0,1]} |f(x) - \hat{f}_N(x)| < 0.01$$

significa che con quel numero di neuroni si può costruire  $\hat{f}_N$  che differisce dalla sinusoide al più di 0.01 in ogni punto dell'intervallo  $[0, 1]$ .

### 2.1.4 Ipotesi e Precisazioni

- **Compattezza del dominio  $K$ .** Il teorema è enunciato per funzioni continue definite su un insieme compatto  $K \subset \mathbb{R}^n$  (ad es. l'intervallo chiuso  $[0, 1]^n$ ). La compattezza garantisce che la norma uniforme  $\|g\|_\infty = \sup_{x \in K} |g(x)|$  sia ben definita e che l'estremo superiore sia effettivamente un massimo raggiunto su  $K$ . Su domini non limitati (per es.  $\mathbb{R}^n$ ) la formulazione uniforme non è direttamente applicabile.
- **Ipotesi sulla funzione di attivazione  $\sigma$ :** la validità del risultato dipende dalle proprietà di  $\sigma$ . Due formulazioni tipiche sono:
  - **Sigmoide limitata e continua (Cybenko):** dove  $\sigma$  ha limiti finiti agli estremi e cambia valore tra  $-\infty$  e  $+\infty$ .
  - **Funzione continua non polinomiale (Leshno et al.):** condizione più generale che garantisce densità dello span.

Queste ipotesi escludono funzioni che non introducono la non linearità richiesta per generare uno spazio denso in  $C(K)$ . Per attivazioni moderne (ad esempio, ReLU) il teorema rimane valido ma con enunciati e ipotesi tecniche differenti.

- **Natura esistenziale del risultato:** il teorema è di tipo qualitativo: afferma che esiste un numero finito di neuroni  $N$  e parametri  $(c_i, w_i, b_i)$  tali che l'approssimazione uniforme è ottenuta entro qualsiasi tolleranza prefissata  $\varepsilon$ . Non fornisce però:
  - una procedura esplicita per la ricerca dei parametri;
  - un bound quantitativo generale che esprima  $N$  in funzione di  $\varepsilon$  per una data  $f$ .
- **Non implica una fase di training facile:** anche se esiste una rete che approssima  $f$ , nella pratica:
  - gli algoritmi numerici di ottimizzazione (SGD, Adam, ecc.) non sono garantiti a trovare quei parametri ottimali: la funzione di perdita è non convessa e può avere molteplici ottimi locali o regioni piatte;
  - la buona approssimazione teorica non assicura buona generalizzazione se i dati a disposizione sono scarsi: quindi è necessario usare tecniche di regolarizzazione, validazione e controllo dell'overfitting.

## 2.2 Struttura delle MLP

### 2.2.1 Strati: input, nascosti, output

Le Multi-layer Perceptron sono una tipologia di reti neurali feed-forward, costituite da più strati (layer) di neuroni: esiste uno strato di input, che riceve i dati iniziali (ognuno dei suoi neuroni corrisponde ad una caratteristica del dato in ingresso), uno o più strati nascosti, e uno strato di output che

genera le previsioni finali. Ogni neurone, appartenente ad uno strato, è connesso a tutti i neuroni di quello successivo (architettura fully connected). Questo significa che ogni input viene trasformato dallo strato di input ai layer nascosti intermedi e infine allo strato di output, con ogni collegamento caratterizzato da un peso  $w$ . Il numero di neuroni, in ciascun layer, è un iperparametro da scegliere: tipicamente lo strato di input ha tante unità quanti sono i parametri (o dimensioni) in ingresso, gli strati nascosti possono variare da pochi a molti nodi, a seconda del problema, e lo strato di output ha un neurone per ogni valore target. In ogni neurone (esclusi quelli di input) si effettua una somma pesata degli input più un termine di bias, e quindi si applica una funzione di attivazione per produrre l'output del neurone stesso.

## 2.2.2 Funzioni di attivazione comuni

Le funzioni di attivazione sono cruciali nelle reti neurali, poiché introducono la non linearità necessaria per modellare relazioni complesse tra input e output. In assenza di tali funzioni, una rete multi-layer si ridurrebbe ad una trasformazione lineare. Di seguito vengono descritte alcune delle funzioni di attivazione più diffuse con le loro proprietà matematiche, i pro e contro.

**Proprietà rilevanti** Quando si valuta una funzione di attivazione conviene considerare:

- **differenziabilità**, che è importante per la backpropagation;
- **boundedness dell'output e zero-centering**, se l'output è centrato attorno a 0;
- **saturazione** che causa vanishing gradient;
- **sparsità**;
- **costo computazionale**.

## 1. Sigmoid (logistica)

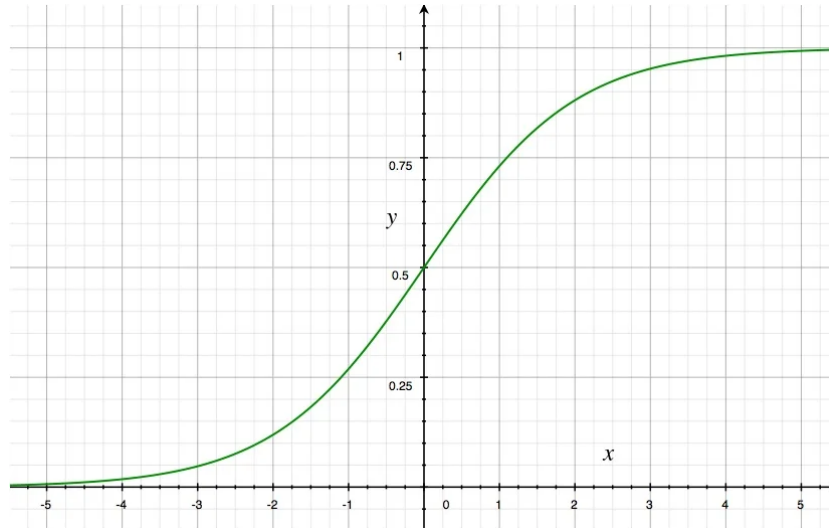


Figura 2.2: Grafico Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \quad \sigma'(x) = \sigma(x)(1 - \sigma(x)).$$

- **Pro:** output in  $(0, 1)$ , forma a "S", utile per probabilità (quindi output normalizzato); derivata semplice.
- **Contro:** saturazione per  $x \rightarrow \pm\infty$  (gradiente  $\rightarrow 0$ ), quindi soffre di vanishing gradient nei layer profondi.
- **Uso tipico:** viene usata nello strato di output per classificazione binaria (in combinazione con binary cross-entropy).

## 2. Tangente iperbolica (tanh)

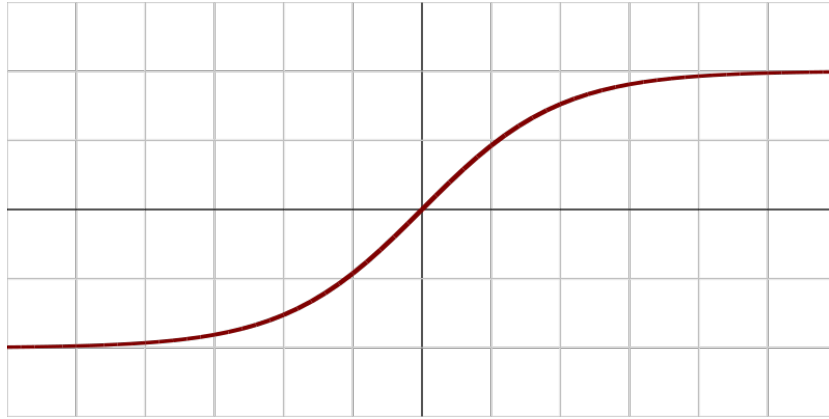


Figura 2.3: Grafico Tangente iperbolica

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \quad \tanh'(x) = 1 - \tanh^2(x).$$

- **Pro:** output in  $(-1, 1)$ , centrata in 0, che porta a convergere meglio di sigmoid quando i dati sono normalizzati.
- **Contro:** resta una funzione saturante per valori estremi, quindi può soffrire ancora di vanishing gradient.
- **Uso tipico:** viene usata nei layer nascosti, in reti poco profonde, o quando si desidera un output centrato.

### 3. Rectified Linear Unit ReLU (ReLU)

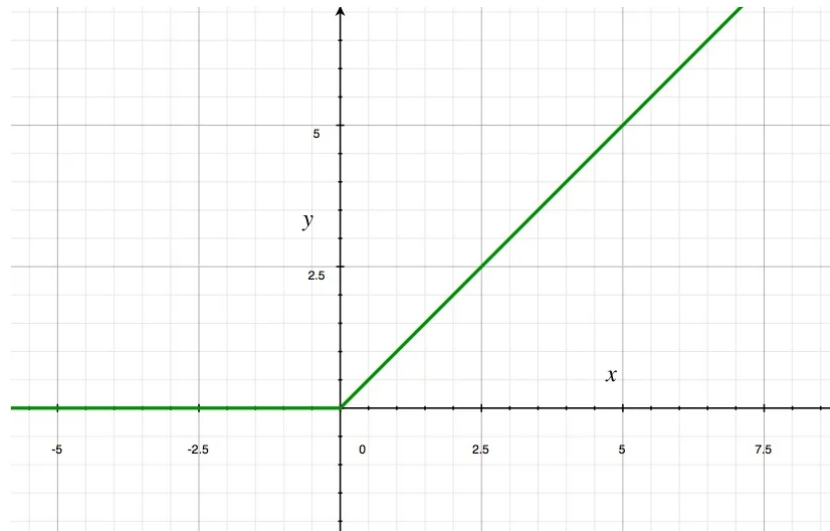


Figura 2.4: Grafico ReLU

$$\text{ReLU}(x) = \max(0, x), \quad \text{ReLU}'(x) = \begin{cases} 0 & x < 0, \\ 1 & x > 0, \end{cases}$$

- **Pro:** semplice, computazionalmente efficiente; evita, la maggior parte delle volte, il vanishing gradient sulle porzioni attive; favorisce sparsità delle attivazioni.
- **Contro:** *dying ReLU*: neuroni che restano permanentemente inattivi, se ricevono input negativi grandi; non centrata. Il neurone è morto se, per tutte (o quasi) le istanze del dataset, risulta  $x \leq 0$ , allora l'output è sempre uguale a zero. Poiché la derivata di ReLU è zero quando  $x < 0$ , il gradiente, rispetto ai pesi, è zero e quindi quel neurone non riceve più aggiornamenti: resta inattivo per tutto l'allenamento (i pesi non si aggiornano).
- **Uso tipico:** viene usata nei layer nascosti, in quasi tutte le architetture delle reti neurali.

#### 4. Leaky ReLU (LReLU) / Parametric ReLU (PReLU)

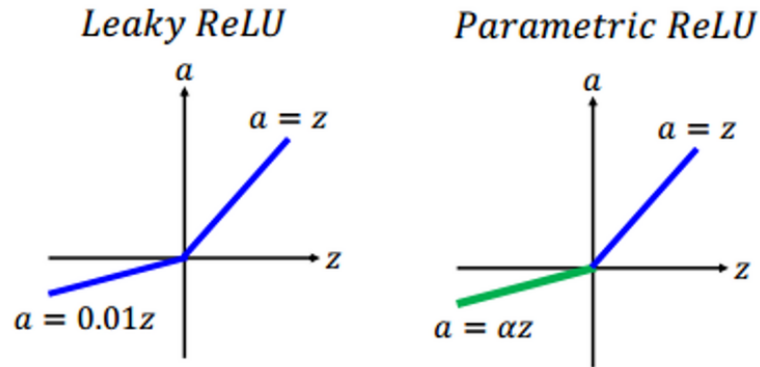


Figura 2.5: Grafico LReLU vs PReLU

$$\text{LReLU}(x) = \begin{cases} 0.01x & x \leq 0, \\ x & x > 0, \end{cases} \quad \text{LReLU}'(x) = \begin{cases} 0.01 & x < 0, \\ 1 & x > 0, \end{cases}$$
$$\text{PReLU}(x) = \begin{cases} \alpha x & x < 0, \\ x & x \geq 0, \end{cases} \quad \text{PReLU}'(x) = \begin{cases} \alpha & x < 0, \\ 1 & x > 0, \end{cases} \quad \alpha \in (0, 1)$$

PReLU apprende  $\alpha$  durante il training.

- **Pro:** mantiene un piccolo gradiente per  $x < 0$ , riducendo i dead neurons.
- **Contro:** introduce (o apprende) un iperparametro; comportamento non sempre superiore a ReLU.
- **Uso tipico:** dove si vuole evitare il problema del *dying ReLU*, mantenendo semplicità.



## 5. Exponential Linear Unit (ELU)

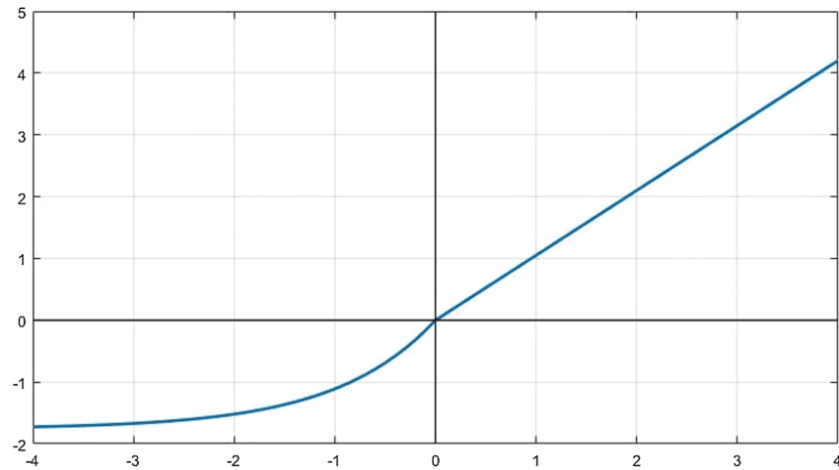


Figura 2.6: Grafico ELU

$$\text{ELU}(x) = \begin{cases} x & x \geq 0, \\ \alpha(e^x - 1) & x < 0, \end{cases} \quad \text{ELU}'(x) = \begin{cases} \alpha(e^x) & x < 0, \\ 1 & x \geq 0, \end{cases} \quad \alpha > 0$$

- **Pro:** output più centrato, gradiente non nullo per  $x < 0$ , migliore convergenza in alcuni casi.
- **Contro:** leggermente più costosa (esponenziale) e introduce parametro  $\alpha$ .

## 6. Softplus

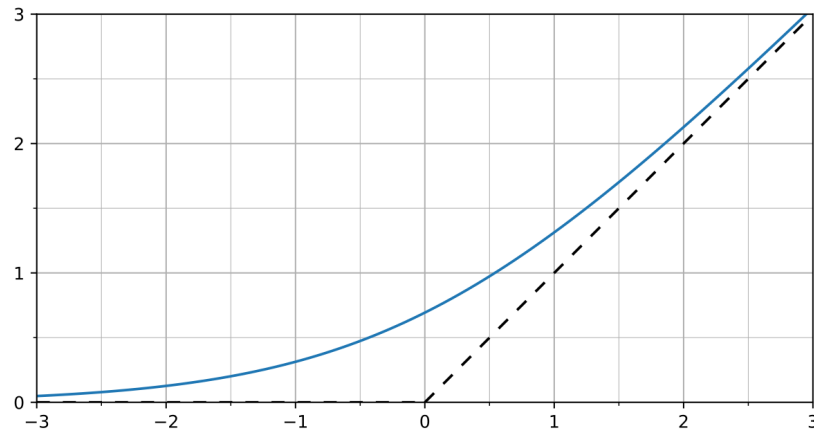


Figura 2.7: Grafico Softplus

$$\text{softplus}(x) = \log(1 + e^x), \quad \text{softplus}'(x) = \frac{1}{1 + e^{-x}}$$

- **Pro:** versione smooth e differenziabile di ReLU.
- **Contro:** più costosa e meno sparsa di ReLU.

## 7. Gaussian Error Linear Unit (GELU)

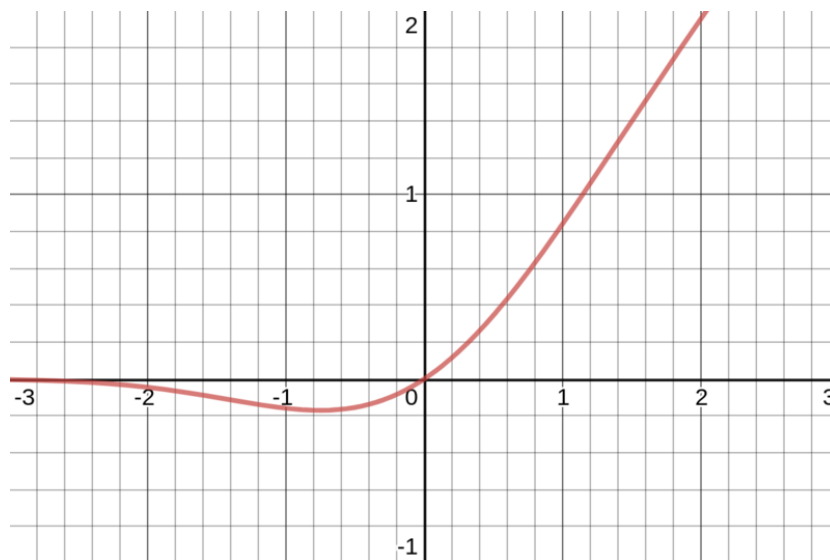


Figura 2.8: Grafico GELU

$$\text{GELU}(x) = x \cdot \Phi(x)$$

dove  $\Phi(x)$  è la funzione di distribuzione cumulativa (CDF) della distribuzione normale standard ed 'erf' è la funzione degli errori di Gauss:

$$\Phi(x) = \frac{1}{2} \left[ 1 + \text{erf} \left( \frac{x}{\sqrt{2}} \right) \right]$$

$$\text{GELU}'(x) = \Phi(x) + \frac{1}{2} x \phi(x)$$

dove

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

- **Pro:** buon comportamento empirico in modelli di linguaggio; più sofisticata della ReLU.
- **Contro:** più costosa da calcolare.
- **Uso tipico:** viene ampiamente utilizzato nelle architetture Transformer; è una funzione di attivazione soft che combina linearità e gating stocastico.

## 8. Softmax

Per vettore  $x \in \mathbb{R}^K$ :

$$\text{softmax}(x)_j = \frac{e^{x_j}}{\sum_{k=1}^K e^{x_k}}.$$

- **Uso tipico:** viene utilizzata per trasformare i *logits* in una distribuzione di probabilità, normalmente combinata con la loss di *categorical cross-entropy*.

## Riepilogo

- **Hidden layers profondi:** ReLU o varianti (LReLU, PReLU, ELU) sono in genere preferite per ridurre il problema del vanishing gradient e per semplicità computazionale.

- **Reti poco profonde o con output centrato:** tanh può essere utile quando si desidera output centrati.
- **Output:** sigmoid per output binario (con binary cross-entropy); softmax per multi-classe (con categorical cross-entropy); identità (lineare) per regressione.
- **Architetture moderne:** GELU è comune nei Transformers.
- **Sostituti lisci:** softplus può rimpiazzare ReLU quando serve differenziabilità completa.

Nella maggior parte delle applicazioni moderne la combinazione ReLU (o varianti) nei layer nascosti e softmax/sigmoid in output è una scelta robusta, mentre alternative come GELU o ELU possono migliorare le prestazioni in contesti specifici.

## 2.3 Procedura di forward pass

### 2.3.1 Calcolo delle attivazioni

Durante la fase di propagazione in avanti (forward pass), i dati attraversano la rete dallo strato di input a quello di output. Ogni neurone calcola prima un ingresso pesato sommandolo con il bias. Dato un neurone  $j$  dello strato nascosto o di output, l'attivazione lineare è

$$z_j = \sum_i w_{ji} x_i + b_j,$$

dove  $x_i$  sono gli output (o input iniziali),  $w_{ji}$  i pesi di connessione, e  $b_j$  il bias. In seconda battuta, si applica la funzione di attivazione  $\phi$  per ottenere l'uscita del neurone:

$$a_j = \phi(z_j).$$

Ad esempio, con  $\phi = \sigma$  (sigmoid), avremmo  $a_j = 1/(1 + e^{-z_j})$ . Questo processo viene eseguito strato per strato. Ogni layer trasforma in modo non

lineare i dati in ingresso, permettendo alla rete di apprendere composizioni funzionali complesse.

### 2.3.2 Propagazione ed output

Dopo aver calcolato le attivazioni in tutti gli strati intermedi, l'uscita dello strato finale ( $\mathbf{a}_{\text{out}}$ ) costituisce la previsione della rete. Se il problema è di regressione, l'ultima funzione di attivazione può essere identità (o lineare); se è di classificazione binaria, si può usare la sigmoid; se è multi-classe, si usa tipicamente la softmax. Ad esempio, in una classificazione a  $K$  classi lo strato di output contiene  $K$  neuroni con softmax, e ciascuna uscita  $a_k \in (0, 1)$  rappresenta la probabilità assegnata alla classe  $k$ . L'output finale è dunque un vettore di previsioni che dipende dalle scelte di pesi, bias e funzioni di attivazione attraverso la rete. Infine, confrontando  $\mathbf{a}_{\text{out}}$  con il valore target (ground truth) del training si calcola una funzione di perdita che misura l'errore di previsione (ad esempio, MSE per regressione o cross-entropy per classificazione). Questa funzione di perdita viene poi utilizzata nell'allenamento per aggiornare i pesi tramite backpropagation.

## 2.4 Algoritmo di backpropagation

### 2.4.1 Derivazione del gradiente

L'algoritmo di backpropagation serve a calcolare come variano i pesi della rete per ridurre l'errore. In termini operativi, si tratta di applicare la *chain rule* per calcolare il gradiente della funzione di perdita  $L$  rispetto a ciascun peso  $w$ . Poiché la rete è una composizione di funzioni (ogni neurone è una funzione di attivazione della sua somma pesata), si propaga l'errore a ritroso layer per layer. Formalmente, a partire dallo strato di output, si calcola la derivata parziale dell'errore rispetto all'attivazione netta di ciascun neurone:

$$\delta_j = \frac{\partial L}{\partial z_j}.$$

Quindi, per il layer precedente si applica la regola della catena: ciascun neurone riceve contributi di errore da tutti i neuroni a cui è connesso nel layer successivo. In pratica, questo permette di ottenere il gradiente di  $L$  rispetto a ogni peso:

$$\frac{\partial L}{\partial w_{ji}} = a_i \delta_j,$$

dove  $a_i$  è l'uscita del neurone  $i$  nel layer precedente e  $\delta_j$  è l'"errore locale" nel neurone  $j$ . In sostanza, la retropropagazione calcola il gradiente della funzione di perdita rispetto a ogni peso usando la regola della catena per determinare quanto ciascun peso contribuisce all'errore finale. In parole povere, si va all'indietro partendo dall'errore di output e si distribuisce l'errore ai layer precedenti moltiplicando per le derivate delle funzioni di attivazione. Di conseguenza, si ottiene un vettore di gradienti che indica la direzione in cui modificare ciascun peso per diminuire l'errore complessivo.

### 2.4.2 Aggiornamento dei pesi

Una volta noti i gradienti parziali  $\frac{\partial L}{\partial w}$ , i pesi vengono aggiornati solitamente tramite discesa del gradiente (gradient descent). Con un learning rate  $\eta$ , l'aggiornamento base è dato da:

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}.$$

Questo modifica ogni peso nella direzione negativa del gradiente per ridurre la perdita. In termini di formula, per l'esempio di un singolo campione e peso  $w_{ji}$ :

$$\Delta w_{ji} = -\eta \frac{\partial L}{\partial w_{ji}} = -\eta \delta_j a_i.$$

Questa è la regola standard di backpropagation con discesa del gradiente. In pratica, si iterano più epoche di allenamento aggiornando i pesi in base a molti esempi, eventualmente con varianti come la discesa del gradiente stocastico (SGD) oppure con algoritmi avanzati (momentum, Adam, ecc.). Tra un passo di forward e il successivo di backward, possono essere applicate

tecniche di batching: l'errore può essere aggregato su minibatch di esempi per stabilizzare l'aggiornamento. In ogni caso, il principio fondamentale è che i pesi vengono "aggiustati" proporzionalmente al proprio contributo all'errore complessivo, come descritto nelle sezioni precedenti.

### 2.4.3 Tecniche di regolarizzazione

Affinché la rete abbia un'ottima capacità di generalizzazione (non si limiti a riprodurre il rumore dei dati di training), si usano tecniche di regolarizzazione:

- **Dropout:** durante la fase di training, in ciascuna iterazione si disattiva casualmente una parte di neuroni in alcuni layer, impostando le loro attivazioni a zero. In pratica, si costringe la rete a non dipendere eccessivamente da un singolo neurone. Questo riduce l'overfitting in quanto sviluppa più rappresentazioni ridondanti. Ad esempio, con una dropout probability  $p$ , un neurone viene disattivato con probabilità  $p$  e gli altri sono scalati di  $1/(1 - p)$  per compensazione.
- **Regolarizzazione  $\ell_1$  (LASSO):** si aggiunge alla loss un termine proporzionale alla somma dei valori assoluti dei pesi:

$$L'(w) = L(w) + \lambda \|w\|_1 = L(w) + \lambda \sum_i |w_i|,$$

con  $\lambda > 0$ . La funzione di penalità  $\ell_1$  induce sparsità: molte componenti dei pesi vengono impostate a zero, facilitando la selezione di feature e l'interpretabilità del modello.

- **Regolarizzazione  $\ell_2$  (Ridge):** si aggiunge il quadrato della norma dei pesi:

$$L'(w) = L(w) + \frac{\lambda}{2} \|w\|_2^2 = L(w) + \frac{\lambda}{2} \sum_i w_i^2.$$

Il termine  $\ell_2$  non produce soluzioni esattamente sparse ma riduce la magnitudine di tutti i pesi verso lo zero.

- **Combinazione L1+L2 (Elastic Net):** combina entrambe le precedenti penalità:

$$L'(w) = L(w) + \alpha \left( \lambda_1 \|w\|_1 + \frac{\lambda_2}{2} \|w\|_2^2 \right),$$

e viene scelta per ottenere sia sparsità (L1) sia stabilità (L2) quando le features sono correlate. Elastic Net é spesso preferibile quando il numero di variabili supera il numero di osservazioni o quando ci sono gruppi di variabili fortemente correlate.

## 2.5 Vantaggi e limiti

### 2.5.1 Flessibilità e capacità di generalizzazione

Le reti MLP offrono grande flessibilità: grazie alla combinazione di pesi e attivazioni non lineari, possono modellare relazioni complesse e non lineari tra input e output. Possono apprendere sia compiti di regressione che di classificazione (binarie o multi-classe) e sono in grado di approssimare praticamente qualsiasi funzione continua. Tale capacità di rappresentazione rende le MLP potenti modelli predittivi in molti ambiti. Inoltre, in presenza di dati adeguati e con l'uso di tecniche di regolarizzazione, le MLP tendono ad avere una buona capacità di generalizzazione, ossia sono in grado di fare previsioni corrette su dati non visti.

### 2.5.2 Problemi di vanishing/exploding gradient

Uno dei limiti più importanti delle MLP (soprattutto se possiedono molti hidden layer) riguarda il problema del vanishing gradient. Poiché, durante la backpropagation, i gradienti vengono moltiplicati per le derivate delle funzioni di attivazione in ogni layer, se queste derivate sono piccole (come in sigmoid o tanh, che assumono valori entro  $(0, 1)$ ), il prodotto dei gradienti tende a diminuire esponenzialmente con la profondità. Di conseguenza, i pesi nei primi strati (vicini all'input) ricevono gradienti quasi nulli e la rete impara molto lentamente le rappresentazioni nei layer bassi. Al contrario,



se derivate o pesi sono grandi ( $> 1$ ), può manifestarsi un exploding gradient, dove i gradienti crescono esponenzialmente e portano ad instabilità numeriche (pesanti oscillazioni o overflow). Per questo si usano funzioni, come le ReLU, che hanno derivate più stabili, normalizzazione dei dati, inizializzazione specifiche dei pesi, o architetture speciali per alleviare il fenomeno.

### **2.5.3 Efficienza computazionale**

Dal punto di vista computazionale, le MLP possono diventare costose da addestrare se il numero di layer o di neuroni è elevato. Ogni propagazione in avanti ed indietro richiede calcoli intensivi e su set di dati di grandi dimensioni l'allenamento può richiedere molto tempo e risorse computazionali. Il costo cresce con il numero di connessioni del modello. Inoltre, le MLP richiedono un tuning accurato degli iperparametri (learning rate, struttura della rete, regolarizzazione, ecc.) per ottimizzare performance ed efficienza.

## Capitolo 3

# Kolmogorov–Arnold Networks (KAN)

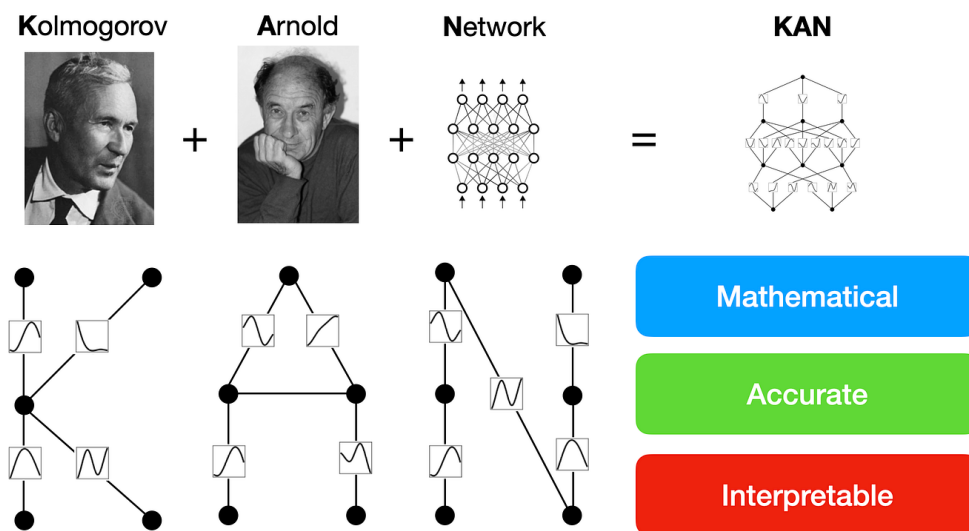


Figura 3.1: Kolmogorov–Arnold Networks

## 3.1 Fondamenti matematici

### 3.1.1 Enunciato del teorema di Kolmogorov–Arnold

Il teorema di Kolmogorov–Arnold (KART) stabilisce che ogni funzione continua multivariata (cioè su più variabili), su un intervallo compatto, può essere rappresentata come una combinazione di somme di funzioni univariate (cioè su una variabile). In forma esplicita: per una funzione continua  $f : [0, 1]^n \rightarrow \mathbb{R}$  esistono funzioni continue univariate  $\phi_{q,p}$  e  $\Phi_q$  tali che

$$f(x_1, \dots, x_n) = \sum_{q=1}^{2n+1} \Phi_q \left( \sum_{p=1}^n \phi_{q,p}(x_p) \right),$$

per  $q = 1, \dots, 2n + 1$ . In altre parole, ogni funzione multivariata può essere ricondotta a combinazioni di funzioni unidimensionali tramite opportune funzioni interne  $\phi_{q,p}$ , che agiscono come "feature extractors" individuali, cioè estraggono informazioni rilevanti da ciascuna dimensione dei dati in modo indipendente, ed esterne  $\Phi_q$ , che agiscono come un "classificatore" di queste features, cioè combinano le caratteristiche estratte per prendere una decisione. Questo risultato mostra che l'unica vera interazione tra le variabili, nella rappresentazione data, è la somma: tutte le altre dipendenze possono essere "scomposte" in funzioni di una sola variabile.

### 3.1.2 Sulla natura "non costruttiva" delle dimostrazioni

Le dimostrazioni originali del KART, eseguite da Kolmogorov nel 1957 e successivamente Arnold nel 1967, sono di natura esistenziale: garantiscono l'esistenza delle funzioni  $\phi_{q,p}$  e  $\Phi_q$ , ma non forniscono una procedura esplicita o una formula chiusa per costruirle. Pertanto il teorema è fondamentale dal punto di vista teorico ma, senza ulteriori risultati costruttivi, ha limitata utilità pratica per la costruzione diretta di architetture neurali basate su tali funzioni, spingendo i ricercatori a privilegiare le reti neurali multistrato (MLP) che, pur con i loro limiti, erano più facili da implementare.

## 3.2 Architettura delle KAN

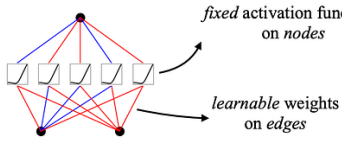
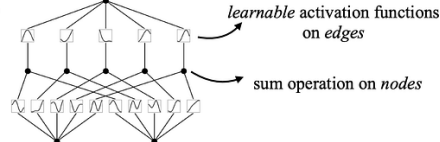
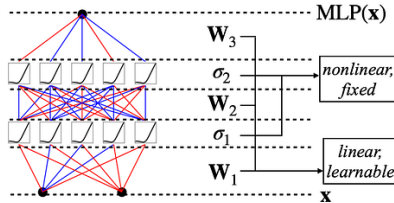
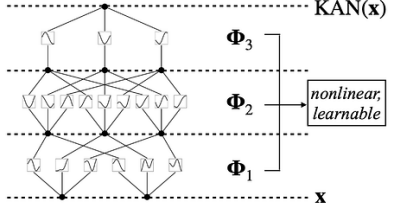
Model	Multi-Layer Perceptron (MLP)	Kolmogorov-Arnold Network (KAN)
Theorem	Universal Approximation Theorem	Kolmogorov-Arnold Representation Theorem
Formula (Shallow)	$f(\mathbf{x}) \approx \sum_{i=1}^{N(\epsilon)} a_i \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i)$	$f(\mathbf{x}) = \sum_{q=1}^{2n+1} \Phi_q \left( \sum_{p=1}^n \phi_{q,p}(x_p) \right)$
Model (Shallow)	(a)  fixed activation functions on nodes learnable weights on edges	(b)  learnable activation functions on edges sum operation on nodes
Formula (Deep)	$\text{MLP}(\mathbf{x}) = (\mathbf{W}_3 \circ \sigma_2 \circ \mathbf{W}_2 \circ \sigma_1 \circ \mathbf{W}_1)(\mathbf{x})$	$\text{KAN}(\mathbf{x}) = (\Phi_3 \circ \Phi_2 \circ \Phi_1)(\mathbf{x})$
Model (Deep)	(c)  $\mathbf{W}_3$ $\sigma_2$ $\mathbf{W}_2$ $\sigma_1$ $\mathbf{W}_1$ $\mathbf{x}$ nonlinear, fixed linear, learnable	(d)  $\Phi_3$ $\Phi_2$ $\Phi_1$ $\mathbf{x}$ nonlinear, learnable

Figura 3.2: Confronto tra un Multi-Layer Perceptron (MLP) ed una Kolmogorov-Arnold Network (KAN).

Una Kolmogorov-Arnold Network (KAN) è strutturalmente simile ad una rete feedforward completamente connessa, simile ad una MLP, ma differiscono in modo sostanziale nell'uso delle funzioni di attivazione: ogni arco (collegamento) tra i neuroni di strati consecutivi porta con sé una funzione univariata parametricamente definita (spesso una B-spline), anziché un peso scalare come nei MLP. Ciascun neurone di uno strato riceve gli output dei collegamenti in ingresso e calcola semplicemente la somma di tali output; non vi sono pesi lineari né attivazioni non lineari aggiuntive sui nodi stessi.

Il modello generale si descrive così: se lo strato  $(\ell - 1)$  ha  $d_{\ell-1}$  neuroni e lo strato  $\ell$  ne ha  $d_\ell$ , allora esiste una matrice di funzioni unidimensionali  $\{f_{ij}^{(\ell)}\}_{i=1, \dots, d_\ell}^{j=1, \dots, d_{\ell-1}}$  tale che, dati gli output degli  $d_{\ell-1}$  neuroni precedenti  $x_i^{(\ell-1)}$ ,

l'uscita  $x_j^{(\ell)}$  del  $j$ -esimo neurone del livello  $\ell$  è:

$$x_j^{(\ell)} = \sum_{i=1}^{d_{\ell-1}} f_{ij}^{(\ell)}(x_i^{(\ell-1)}) .$$

In forma matriciale si può scrivere  $x^{(\ell)} = f^{(\ell)}(x^{(\ell-1)})$ , dove  $f^{(\ell)}$  è l'insieme delle funzioni collegamento per quello strato. L'output complessivo della KAN è quindi dato dalla composizione degli strati successivi:

$$y = x^{(L)} = f^{(L)}(f^{(L-1)}(\dots f^{(1)}(x^{(0)}) \dots)),$$

dove  $x^{(0)}$  è il vettore di input della rete. Questa architettura "fonde" le trasformazioni lineari e non-lineari in un'unica funzione  $f_{ij}$  per ogni arco, permettendo alla rete di apprendere la forma esatta della funzione di attivazione necessaria per ogni connessione.

Si noti che un MLP applica funzioni di attivazione predefinite (ReLU, sigmoid, ecc.) sui singoli neuroni e moltiplica gli input per pesi scalari; al contrario, una KAN utilizza funzioni parametriche sugli archi e non applica nessuna attivazione aggiuntiva sui neuroni.

### 3.2.1 Funzioni univariate parametriche

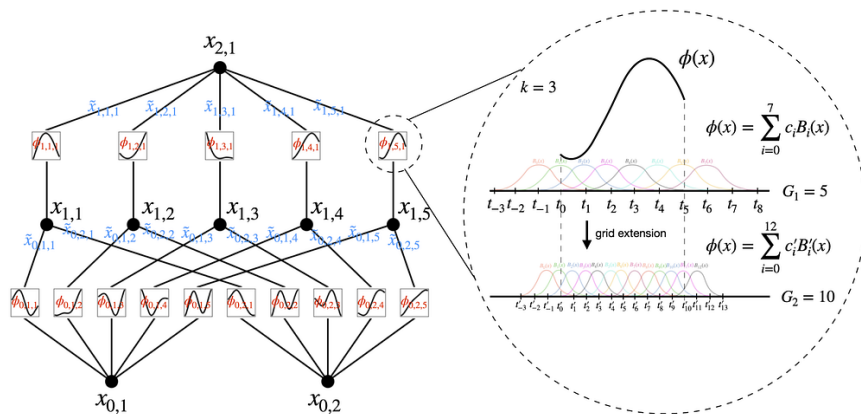


Figura 3.3: Funzioni univariate parametriche.

Le funzioni di attivazione utilizzate nelle KAN sono funzioni univariate parametriche scelte in modo da poter apprendere flessibilmente la forma durante il training. Nell'implementazione classica proposta da Liu et al. (2024), queste funzioni sono rappresentate tramite B-spline (polinomi a tratti di basso grado), che offrono un buon trade-off tra flessibilità locale e complessità di calcolo.

Ciascuna funzione su un collegamento è quindi espressa nella forma

$$f_{ij}(t) = t + g_{ij}(t),$$

dove  $g_{ij}(t)$  è una combinazione lineare di B-spline:

$$g_{ij}(t) = \sum_{k=1}^{G+p} c_k B_{k,p} \left( \frac{t - t_{\min}}{t_{\max} - t_{\min}} \right).$$

Il termine lineare  $t$  garantisce un comportamento iniziale affine, migliorando la stabilità dell'ottimizzazione, mentre il contributo spline introduce la non linearità appresa dalla rete.

Un aspetto importante è la definizione della spline grid, ovvero la suddivisione dell'asse degli input in nodi che determinano dove iniziano e finiscono gli intervalli di ogni polinomio. Il numero di nodi  $G$  e la loro posizione influenzano la capacità espressiva e la risoluzione locale delle funzioni attivazione. In generale, oltre alle B-spline, si possono utilizzare anche altre famiglie di funzioni unidimensionali parametriche, come i polinomi di Chebyshev o altre basi ortogonali, in base alle caratteristiche del problema specifico. Infatti, le reti KAN sono in grado di adattare i coefficienti di queste basi durante il training, permettendo alla rete di apprendere funzioni di attivazione ottimali per ciascun collegamento.

### 3.2.2 B-spline

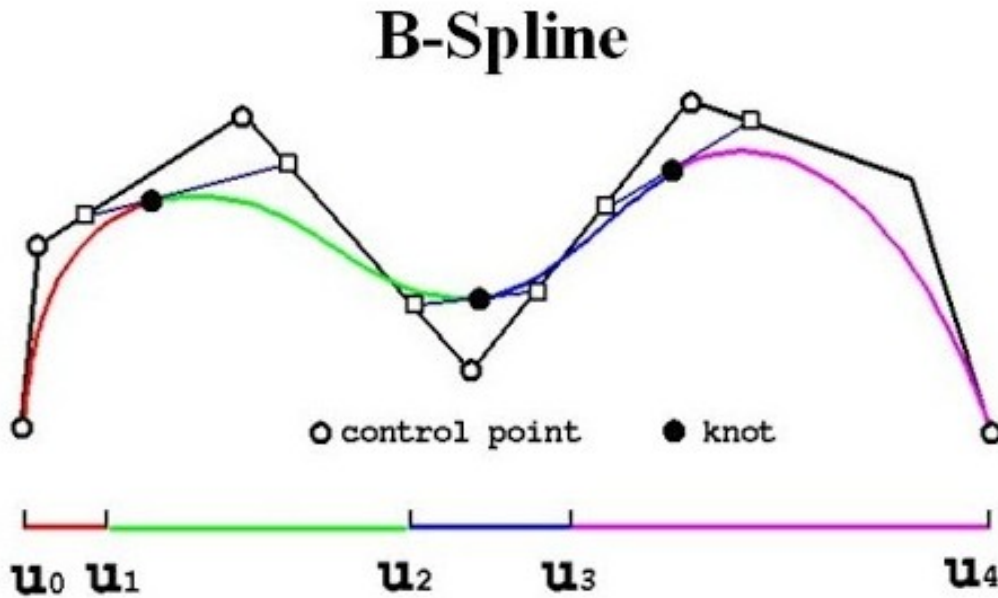


Figura 3.4: Struttura B-spline.

#### Definizione

Le B-spline sono funzioni polinomiali a tratti che costituiscono la base per la rappresentazione di funzioni spline di un dato grado. Una B-spline di ordine  $p + 1$  (ovvero di grado  $p$ ) è definita ricorsivamente come segue:

$$B_{i,0}(t) = \begin{cases} 1 & \text{se } t_i \leq t < t_{i+1} \\ 0 & \text{altrimenti} \end{cases}$$

e per  $p > 0$ :

$$B_{i,p}(t) = \frac{t - t_i}{t_{i+p} - t_i} B_{i,p-1}(t) + \frac{t_{i+p+1} - t}{t_{i+p+1} - t_{i+1}} B_{i+1,p-1}(t),$$

dove  $\{t_i\}$  è il vettore dei nodi (knot vector), che suddivide il dominio della funzione in intervalli. Ogni B-spline  $B_{i,p}(t)$  è diversa da zero solo sull'intervallo  $[t_i, t_{i+p+1})$ , conferendo proprietà di supporto locale.

## Implementazione delle B-spline nelle KAN

Nelle implementazioni standard delle KAN, le funzioni di attivazione sui collegamenti sono parametrizzate come combinazioni lineari di B-spline cubiche ( $p = 3$ ):

$$f_{ij}(x) = w_0x + \sum_{k=1}^{G+p} c_k B_{k,p} \left( \frac{x - x_{\min}}{x_{\max} - x_{\min}} \right),$$

dove:

- $w_0$  è il termine residuo lineare che garantisce un comportamento iniziale affine,
- $\{c_k\}$  sono i coefficienti addestrabili della spline,
- $G$  è il numero di intervalli della griglia di nodi,
- $B_{k,p}$  sono le funzioni base B-spline di grado  $p$ .

Il termine residuo lineare  $w_0x$  stabilizza l'ottimizzazione, permettendo alla funzione di partire come lineare e apprendere non linearità tramite la componente spline.

## Adattamento della griglia

Un aspetto importante delle KAN è l'adattamento dinamico della spline grid:

- La griglia iniziale è in genere uniforme, ma si può ampliare aumentando il numero di nodi  $G$ ,
- I nodi possono essere riposizionati per concentrare la risoluzione dove la funzione varia maggiormente,
- Tecniche di interpolazione lineare dei parametri dell'ottimizzatore sono usate per mantenere la stabilità durante le transizioni della griglia.



### 3.2.3 Scaling laws e Curse of dimensionality

Come suggerito dal teorema matematico KART, le KAN godono di proprietà di approssimazione analoghe ma più raffinate rispetto alle MLP. In particolare, il teorema di Kolmogorov-Arnold garantisce che ogni funzione continua multivariata definita su un dominio compatto possa essere rappresentata esattamente come composizione di funzioni univariate e somme, realizzabile da una KAN di profondità 2 e larghezza proporzionale all'input  $n$  (in particolare, larghezza  $2n + 1$ ). Di conseguenza, per ogni tolleranza  $\epsilon > 0$  esiste una KAN sufficientemente ampia che approssima la funzione  $f$  entro errore  $\epsilon$ , cioè

$$\|f_{\text{KAN}} - f\| < \epsilon.$$

Questa capacità permette alle KAN di superare la curse of dimensionality (COD), sotto l'ipotesi che la funzione da approssimare possieda una struttura additivo-composizionale sufficientemente regolare. In particolare, l'errore di approssimazione di una KAN dipende principalmente dalla risoluzione della griglia spline utilizzata nelle funzioni univariate, risultando approssimativamente indipendente dalla dimensione dell'input. Questa proprietà si traduce in scaling laws più favorevoli rispetto alle MLP tradizionali, per cui il numero di parametri necessari per garantire una certa accuratezza generalmente cresce esponenzialmente con la dimensione dell'input. Questo vantaggio teorico deriva dal fatto che le KAN, a differenza delle MLP, apprendono non solo la struttura composizionale, ma anche le funzioni univariate con elevata precisione, grazie all'uso di attivazioni parametrizzate da spline.

## 3.3 Funzionamento operativo

### 3.3.1 Calcolo del mapping input-hidden-output

Nel funzionamento operativo di una KAN, i dati scorrono in avanti attraverso gli strati esattamente come in una normale MLP, ma usando le funzioni di attivazione sugli archi. Dato un vettore di input  $x^{(0)}$ , il calcolo procede layer

dopo layer: per ciascun neurone nel primo strato nascosto si valuta la somma delle funzioni dei collegamenti in ingresso applicate alle componenti di  $x^{(0)}$ , ottenendo  $x^{(1)}$  e così via. Al livello successivo si ripete lo stesso procedimento prendendo  $x^{(1)}$  come input, e così via fino allo strato di output. Formalmente, ogni layer  $\ell$  esegue la trasformazione  $x^{(\ell)} = f^{(\ell)}(x^{(\ell-1)})$ , e l'output finale è  $y = x^{(L)}$ . Grazie al fatto che tutte le operazioni (applicazione delle funzioni univariate e somme) sono differenziabili, il modello complessivo è addestrabile tramite backpropagation. Questo significa che i coefficienti delle funzioni di attivazione (ad esempio, delle spline) possono essere ottimizzati tramite discesa del gradiente, minimizzando una funzione di perdita, allo stesso modo di quanto avviene in un MLP.

### 3.3.2 Processo di training e Calcolo dei pesi

Il training di una KAN segue la procedura standard di addestramento supervisionato con discesa del gradiente. Si assume un dataset di training ed una funzione loss, quindi si aggiornano iterativamente i parametri delle funzioni di attivazione. Nelle KAN, i "pesi" da addestrare sono i coefficienti che definiscono le funzioni unidimensionali sui collegamenti. Per esempio, una spline di ordine 3 su  $r$  intervalli ha  $r + 3$  coefficienti; ciascun coefficiente è un parametro della rete. È prassi includere un termine base lineare (di solito la stessa identità), come in

$$f_{ij}(t) = t + g_{ij}(t),$$

dove  $g_{ij}$  è la spline addestrabile. Questo facilita la convergenza iniziale. Durante l'ottimizzazione si può anche aggiornare adattivamente la griglia di definizione delle spline, così da coprire automaticamente i nuovi intervalli di attivazione che emergono durante il training (grid extension). In pratica, ogni volta che un valore di attivazione supera la griglia corrente, si estende dinamicamente il supporto della spline per mantenere il dominio di apprendimento adeguato. In sintesi, il flusso di calcolo è identico ad un MLP: si effettua forward pass, si calcola la loss, e poi si retropropaga

l'errore calcolando gradienti rispetto ai coefficienti delle funzioni  $g_{ij}$ .

### 3.4 Confronto con MLP tradizionali

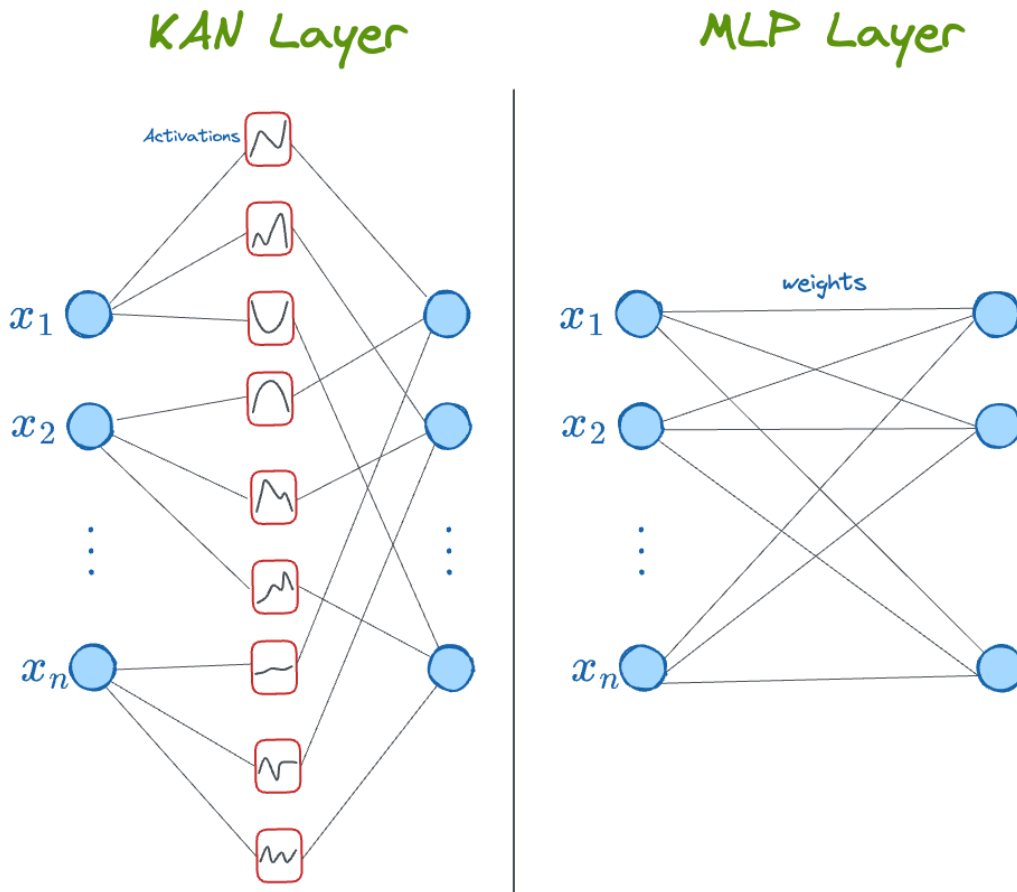


Figura 3.5: Differenze tra Layer MLP e KAN.

#### 3.4.1 Architettura a confronto

Dal punto di vista architetturale, l'architettura di base di una KAN è feedforward e totalmente connessa come quella di un MLP. La differenza fondamentale risiede nel collocamento delle non-linearità e nell'assenza di matrici di pesi lineari. Come abbiamo visto, in un MLP ogni neurone applica una funzione di attivazione fissa, dopo una combinazione lineare dei suoi

input, mentre in una KAN ogni collegamento possiede direttamente una funzione di attivazione addestrabile. Di conseguenza, una KAN "unisce" le trasformazioni lineari e non-lineari in un'unica funzione  $f_{ij}$  per ogni arco, anziché trattarle separatamente come in un MLP.

In termini pratici, un MLP a  $L$  strati alterna operazioni  $x \mapsto Wx + b$  e  $x \mapsto \sigma(x)$ , mentre una KAN sostituisce ogni prodotto  $W_{ij}x_i$  con  $f_{ij}(x_i)$ . Questo implica che una KAN può essere vista come un MLP "con pesi che variano in modo non-lineare col valore dell'input".

### 3.4.2 Complessità computazionale

Dal punto di vista parametrico, una KAN può avere un numero di parametri superiore rispetto ad un MLP di dimensioni simili. Ad esempio, supponiamo una KAN con  $L$  strati, ciascuno di larghezza  $m$ , che usa spline di ordine  $p$  su  $r$  intervalli. Allora il numero totale dei parametri della rete KAN cresce come  $O(L m^2 p r)$ , mentre un MLP con  $L$  strati e larghezza  $m$  avrebbe circa  $O(L m^2)$  pesi scalari. In teoria quindi le KAN appaiono meno efficienti in termini di numero di parametri. Tuttavia, empiricamente si osserva che spesso basta una KAN con dimensioni molto più piccole per eguagliare le prestazioni di un MLP molto più grande. Dal punto di vista computazionale, l'impiego di funzioni parametriche sugli archi comporta un overhead rispetto alle semplici moltiplicazioni peso-input di un MLP. In pratica, valutare una B-spline su ogni collegamento è più costoso del prodotto scalare in un MLP, soprattutto se la rete è profonda o le spline sono molto finemente discretizzate (cioè utilizzano un numero elevato di punti di controllo). Quindi, l'addestramento di una KAN, può essere più lento, con stime che indicano un tempo di training circa 10 volte superiore rispetto alle MLP a parità di condizioni.

### 3.4.3 Interpretabilità e flessibilità locale

Uno dei principali vantaggi delle KAN è la loro interpretabilità. Poiché ogni arco implementa una funzione univariata ben definita, è possibile visualizzare direttamente le forme delle attivazioni apprese. Questo permette di comprendere i singoli contributi delle variabili di input e, in alcuni casi, di dedurre formule simboliche sottostanti, oltre a rendere più semplice il debug e la semplificazione del modello.

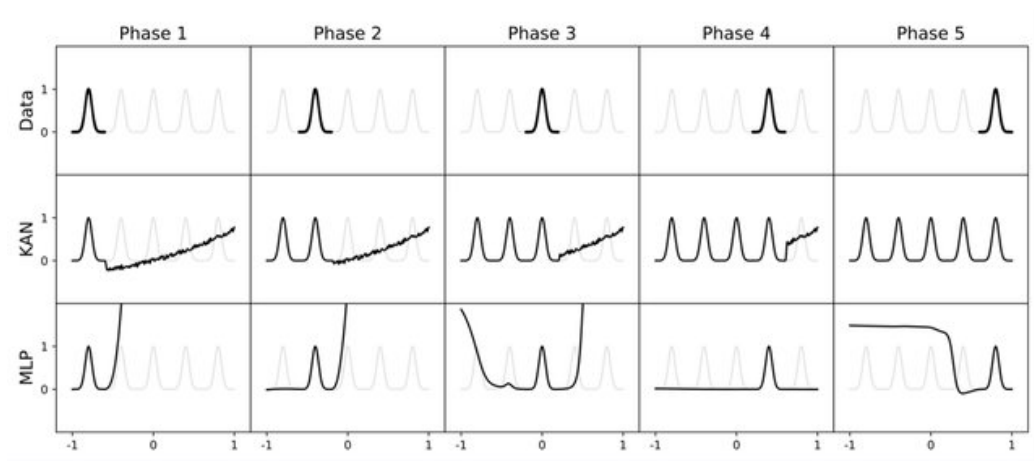


Figura 3.6: Differenza nel Catastrophic forgetting di MLP e KAN.

Un altro vantaggio importante è la flessibilità locale che deriva dalla natura delle spline. A differenza delle MLP che usano funzioni di attivazione globali (come ReLU o Tanh), una KAN modifica solo una piccola regione di input quando apprende una nuova informazione. Ciò riduce significativamente il rischio di "catastrophic forgetting", un fenomeno in cui l'addestramento su nuovi dati può distruggere le informazioni precedentemente apprese.

### 3.4.4 Precisione controllabile tramite grid extension

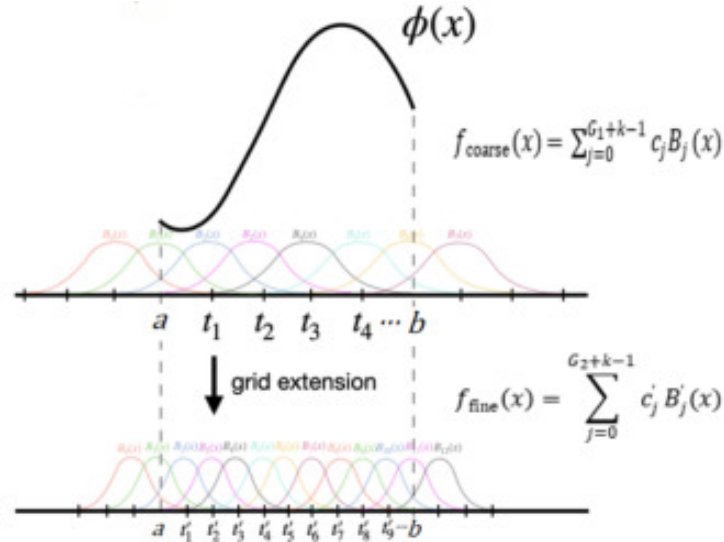


Figura 3.7: Grid extension della spline grid delle KAN.

Le funzioni univariate nelle KAN sono parametrizzate tramite B-spline definite su una griglia. È possibile aumentare la risoluzione della griglia ("grid extension") per incrementare la precisione in modo controllato: partendo da spline grossolane si possono ottenere spline più fini con una procedura di inizializzazione che conserva la continuità e permette rapide riduzioni della loss senza costi computazionali esponenziali.

### 3.4.5 Dipendenza dalla struttura compositiva

I vantaggi più netti emergono solo se la funzione target è effettivamente vicina a una decomposizione in somme di funzioni univariate. Quando la funzione è intrinsecamente non decomponibile o presenta forti interazioni multivariate, la rappresentazione KAN perde efficacia e può risultare meno efficiente rispetto ad una parametrizzazione densa tipica delle MLP.

### 3.4.6 Irregolarità nella rappresentazione di Kolmogorov

Il teorema di Kolmogorov–Arnold garantisce l’esistenza di una rappresentazione, ma non la regolarità delle funzioni intermedie  $\varphi_{q,p}$ . In casi pratici, queste funzioni possono essere non-smooth o altamente oscillanti; per approssimarle con spline possono essere necessarie griglie molto fitte, annullando i vantaggi teorici in termini di parametri e costo computazionale.

### 3.4.7 Overhead computazionale e scelta della struttura

Parametrizzare ogni arco come funzione spline introduce overhead in memoria ed in tempo di calcolo (valutazione e aggiornamento di B-spline, gestione di griglie differenziate). Inoltre, la scelta automatica della topologia (numero di rami, profondità, risoluzione delle griglie per ciascuna spline) non è banale e richiede procedure di pruning o ricerca strutturale che aumentano la complessità del workflow.

### 3.4.8 Sensibilità al rumore e necessità di regolarizzazione

In presenza di dati molto rumorosi, una parametrizzazione spline troppo fine tende al sovralfitting locale. È quindi necessario un attento tuning degli iperparametri (ordine della spline, numero di nodi, termine di regolarizzazione, smoothing), e in alcuni casi una MLP ben regolarizzata può mostrare maggiore robustezza.

## Capitolo 4

### Random forest

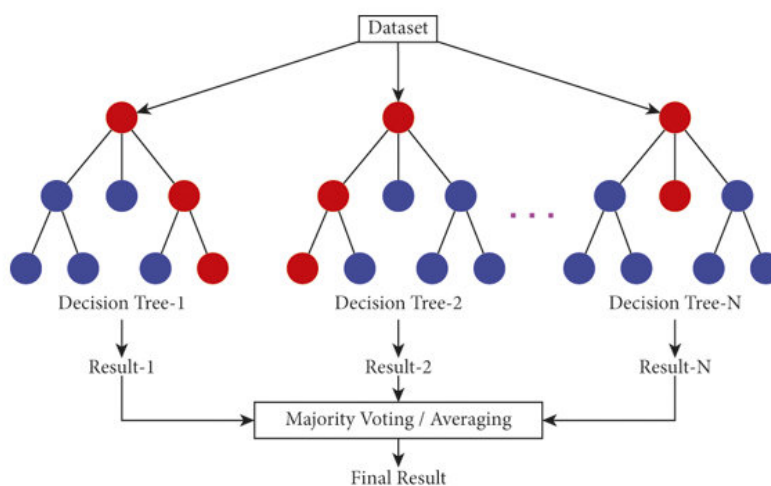


Figura 4.1: Random forest.

Il Random forest è un algoritmo di Machine learning ampiamente utilizzato, noto per la sua robustezza e versatilità sia in problemi di classificazione e regressione. La sua efficacia deriva dalla combinazione di più alberi di decisione "deboli", formando un "ensemble" che supera le prestazioni di un singolo albero, mitigando le debolezze di ciascuno.



## 4.1 Concetti fondamentali

### 4.1.1 Alberi di decisione: Criteri di splitting (Gini, Entropia, Gain ratio)

Gli alberi di decisione costituiscono i "learner deboli" fondamentali all'interno di un Random forest. La loro costruzione implica la divisione iterativa dei dati basata sulle features per creare sottoinsiemi sempre più omogenei. La qualità di queste divisioni è misurata da specifici criteri di impurità o Information gain. Il Gini impurity (o Gini index) è una misura di non-omogeneità ampiamente utilizzata negli alberi di classificazione. Essa quantifica la probabilità che un elemento scelto casualmente da un set venga erroneamente etichettato, se classificato in modo casuale, secondo la distribuzione delle classi nel sottoinsieme. Un valore di 0 indica purezza perfetta, dove tutti gli elementi appartengono alla stessa classe, mentre un valore massimo di  $1 - \frac{1}{n}$ , dove  $n$  è il numero di classi, indica la massima impurità, con le classi equamente distribuite. La formula per il Gini impurity è data da:

$$Gini = 1 - \sum_{i=1}^n (p_i)^2$$

dove  $p_i$  rappresenta la proporzione delle istanze della classe  $i$  nel set. Ad esempio, se un nodo contiene 50 campioni, di cui 25 di una classe e 25 di un'altra, l'impurità di Gini sarebbe 0.5, indicando la massima incertezza. Dopo uno split, l'algoritmo seleziona la variabile che produce la maggiore diminuzione dell'impurità di Gini, portando a nodi più puri.

L'Entropia (Entropy) misura il grado di disordine, imprevedibilità o incertezza in un dataset. Un'entropia di 0 indica un set perfettamente puro, mentre un valore di  $\log_2(n)$  indica la massima incertezza. L'Information gain misura la riduzione dell'entropia ottenuta da uno split, indicando quanta "informazione" viene acquisita sulla variabile target. La formula per l'Entropia è:

$$Entropy = - \sum_{i=1}^n p_i \log_2(p_i)$$

dove  $p_i$  è la probabilità della classe  $i$ .

Il Gain ratio è stato introdotto per mitigare un problema dell'Information gain, che ha un bias verso attributi con un gran numero di valori. Questi attributi tendono a creare molti nodi piccoli e puri, il che può condurre a un potenziale overfitting. Il Gain ratio normalizza l'Information gain, penalizzando gli split che creano molti sottoinsiemi. La sua formula è:

$$Gainratio = \frac{Informationgain}{Splitinformation}$$

dove lo *Split information* è l'entropia dello split stesso:

$$SplitInformation(S, A) = - \sum_{i=1}^v \frac{|S_i|}{|S|} \log_2\left(\frac{|S_i|}{|S|}\right)$$

dove  $S$  è il set di dati del nodo,  $A$  è la feature su cui si sta splittando,  $v$  è il numero di valori unici della feature,  $|S_i|$  è il numero di istanze nel sottoinsieme  $i$  e  $|S|$  è il numero totale di istanze.

Un confronto tra Gini impurity, Entropia e Gain ratio rivela differenze importanti. Il Gini impurity ha un intervallo di valori compreso tra  $[0, 1 - \frac{1}{n}]$ , mentre l'Entropia ha un intervallo tra  $[0, \log_2(n)]$ . Dal punto di vista computazionale, il Gini index è generalmente più efficiente da calcolare rispetto all'Entropia, poiché quest'ultima richiede l'uso di logaritmi. La scelta tra i criteri non è arbitraria, ma implica un compromesso. Il Gini, essendo computazionalmente più veloce, è meno incline a produrre alberi di decisione molto profondi, poiché privilegia split che generano nodi più bilanciati. Al contrario, l'Entropia, sebbene più onerosa, tende a generare alberi che massimizzano la riduzione dell'incertezza, ma il suo bias può portare a preferire caratteristiche con molte categorie, aumentando il rischio di overfitting. Per mitigare ciò, il Gain Ratio si dimostra più robusto.

Per dataset molto grandi o per applicazioni con stringenti requisiti di velocità, il Gini potrebbe essere la scelta preferibile. Per contro, in contesti

dove la massima purezza dei nodi è cruciale, l'Entropia (o, più precisamente, il Gain ratio) potrebbe rivelarsi più efficace, a condizione che il rischio di overfitting venga gestito adeguatamente. Questa decisione fondamentale, a livello del singolo albero, si ripercuote sulla performance e sulla struttura complessiva del Random forest. Un albero "più debole" ma più rapido, generato con il Gini, può essere efficacemente compensato dall'approccio Ensemble, mentre alberi "più forti" ma più lenti, derivanti dall'Entropia, potrebbero non scalare con la stessa efficienza.

#### 4.1.2 Ensemble learning e Bagging

Il principio alla base dell'Ensemble learning è spesso descritto come la "saggezza della folla": un gruppo di learner deboli, che individualmente potrebbero non performare in modo ottimale, a causa di alta varianza o alto bias, può, quando le loro previsioni vengono aggregate, formare un "learner forte" con prestazioni notevolmente migliorate.

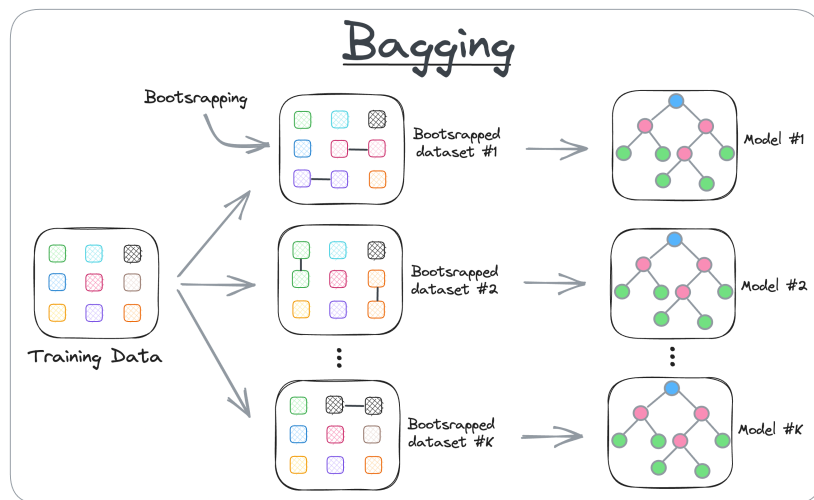


Figura 4.2: Bagging.

Il Bagging è un metodo di Ensemble learning il cui scopo principale è ridurre la varianza all'interno di un dataset "rumoroso", migliorando così

la capacità di generalizzazione del modello e mitigando l'overfitting. Il processo di Bagging si articola in tre fasi fondamentali:

1. **Bootstrapping:** questa tecnica di ricampionamento genera diversi sottoinsiemi del training set. Il campionamento avviene selezionando istanze in modo casuale con re-immissione, ciò significa che una singola istanza può essere scelta più volte all'interno dello stesso sottoinsieme. Questo processo è cruciale per creare diversità tra i campioni su cui verranno addestrati i modelli individuali.
2. **Addestramento parallelo:** i campioni bootstrap così generati vengono utilizzati per addestrare, in modo indipendente e parallelo, una serie di learner deboli, che nel contesto di Random forest sono tipicamente alberi di decisione.
3. **Aggregazione:** una volta che i modelli individuali hanno prodotto le loro previsioni, queste vengono combinate. Per i problemi di regressione, si utilizza un processo noto come Soft voting (cioè la media di tutti gli output previsti dai singoli learner), mentre, per i problemi di classificazione, si utilizza l'Hard o Majority voting (cioè viene scelta la classe più votata).

Il beneficio più significativo è la riduzione della varianza, particolarmente utile con dati ad alta dimensionalità o in presenza di valori mancanti, dove un'alta varianza può rendere il modello più incline all'overfitting. La diversità introdotta nei dati di training per ciascun modello contribuisce a ridurre la varianza nelle previsioni finali. Questo processo mitiga l'influenza del rumore nei dati e degli outlier, producendo un modello aggregato più stabile ed affidabile. Tuttavia, il Bagging può portare ad una perdita di interpretabilità, rendendo difficile estrarre intuizioni precise a causa del processo di media delle previsioni. È anche computazionalmente costoso, rallentando e diventando più intensivo all'aumentare del numero di iterazioni. Infine, è meno flessibile con algoritmi già stabili o con un alto bias, poiché i benefici, in termini di riduzione della varianza, sono meno visibili.

Il Bagging non è semplicemente un metodo per combinare modelli, ma agisce come una forma intrinseca di regolarizzazione. Addestrando modelli su sottoinsiemi diversi del dataset, attraverso il campionamento di tipo bootstrap, si garantisce che ogni modello acquisisca una prospettiva leggermente differente del problema. Quando le previsioni di questi modelli, sebbene diversi, sono aggregate, gli errori casuali e la varianza intrinseca di un singolo modello tendono a compensarsi reciprocamente. Ciò porta a una previsione finale che è più stabile e meno sensibile al rumore oppure agli outlier. Questo meccanismo è la ragione fondamentale per cui il Bagging è così efficace nel ridurre l'overfitting, specialmente per learner ad alta varianza, come gli alberi di decisione profondi. Questa capacità di ridurre la varianza senza introdurre un bias significativo è ciò che rende il Bagging una base così potente per algoritmi come Random forest, che altrimenti sarebbero molto inclini all'overfitting. È una dimostrazione del principio che la "diversità" all'interno di un ensemble conduce a una maggiore robustezza del modello complessivo.

## **4.2 Architettura e Costruzione**

Random forest estende il concetto di Bagging introducendo un ulteriore livello di casualità, rendendo l'Ensemble ancora più robusto e meno propenso all'overfitting.

### **4.2.1 Bootstrapping e Feature bagging**

La costruzione di un Random forest si basa su due fonti principali di casualità, essenziali per garantire che gli alberi individuali siano il più possibile non correlati tra loro.

La prima fonte di casualità è il campionamento di tipo bootstrap. Per ogni albero che fa parte della foresta, viene estratto un campione casuale di dati dal set di training originale con re-immissione. Questo sottoinsieme di dati è noto come Bootstrap sample. Una caratteristica importante di questo processo è che circa un terzo dei dati originali non viene selezionato

per un dato bootstrap sample; questi dati non utilizzati sono chiamati campioni "out-of-bag" e possono essere impiegati per la validazione interna del modello. Questo tipo di campionamento assicura che ogni albero sia addestrato su un sottoinsieme leggermente diverso dei dati originali, promuovendo una diversità fondamentale tra gli alberi.

La seconda fonte di casualità è legata alle features, nota come feature bagging o random subspace method. Ad ogni split dei nodi, all'interno di un albero di decisione, Random forest non considera tutte le feature disponibili, ma seleziona solo un sottoinsieme casuale di esse. Questa è una differenza importante rispetto agli alberi di decisione standard, che valuterebbero tutte le feature possibili per trovare lo split migliore. L'introduzione di questa casualità nella selezione delle feature aggiunge ulteriore diversità al dataset per ogni albero e riduce significativamente la correlazione tra gli alberi di decisione individuali.

La casualità introdotta, tramite le due fonti, non è ridondante, ma complementare e strategica. Il Bootstrapping riduce la varianza generale del modello assicurando che ogni albero acquisisca una prospettiva leggermente diversa del dataset complessivo. Il Feature bagging, d'altra parte, previene che caratteristiche particolarmente forti o dominanti influenzino la costruzione di tutti gli alberi. Se una singola caratteristica fosse sempre scelta come il miglior punto di split, tutti gli alberi all'interno della foresta risulterebbero molto simili e altamente correlati, rendendo inutili i benefici derivanti dall'approccio Ensemble. Introducendo la casualità nella selezione delle feature, si forza ogni albero ad esplorare diverse combinazioni di caratteristiche, riducendo ulteriormente la loro correlazione e, di conseguenza, la varianza dell'intero Ensemble.

#### **4.2.2 Aggregazione delle predizioni**

Una volta che tutti gli alberi di decisione sono stati costruiti e addestrati sui rispettivi sottoinsiemi di dati e feature, le loro previsioni vengono combinate per ottenere il risultato finale del Random forest. Il metodo di aggregazione dipende dalla natura del problema:

- **Regressione:** le previsioni numeriche generate da ciascun albero individuale vengono semplicemente mediate. Il valore medio di tutte le previsioni degli alberi costituisce la previsione finale del modello.
- **Classificazione:** la classe finale viene determinata attraverso un processo di voto di maggioranza. Ogni albero nella foresta produce una previsione e la classe che riceve il maggior numero di "voti" (cioè, la più scelta) tra tutti gli alberi viene accettata come previsione finale del Random forest.

L'aggregazione delle previsioni, sia essa tramite media o voto di maggioranza, rappresenta il passo finale che trasforma un insieme di learner deboli in uno forte. Sebbene gli alberi di decisione individuali possano presentare un'alta varianza, l'atto di mediare o votare le loro previsioni tende a ridurre la varianza e sensibilità del modello agli outlier. Questo processo mitiga la tendenza del singolo albero a sovrastimare o sottostimare i valori, producendo una previsione finale più stabile e robusta. Questo meccanismo di aggregazione è ciò che consente al Random forest di raggiungere un'elevata accuratezza mantenendo al contempo una solida capacità di generalizzazione. È un esempio pratico dell'applicazione del principio della "saggezza della folla" nel Machine learning, dove la combinazione di molteplici opinioni individuali, seppur imperfette, conduce ad un risultato complessivo superiore.

### 4.3 Vantaggi

- **Riduzione della varianza:** è intrinsecamente meno propenso all'overfitting rispetto a un singolo albero di decisione. Questo è dovuto alla sua natura ensemble, che aggrega le previsioni di molti alberi diversi, ed all'introduzione di casualità sia nel campionamento dei dati che nella selezione delle features.
- **Robustezza agli outlier ed ai dati rumorosi:** il modello è meno sensibile all'influenza degli outlier e del rumore nei dati, poiché

le previsioni vengono mediate su più alberi, diluendo l'impatto di singole osservazioni estreme.

- **Gestione dei valori mancanti:** Random forest può gestire efficacemente i valori mancanti nel dataset, rendendo la fase di pre-processing dei dati più semplice.
- **Gestione dei dati sbilanciati:** ha la capacità di gestire dataset in cui le classi sono sbilanciate.
- **Facilità di determinazione dell'importanza delle feature:** l'algoritmo rende relativamente semplice valutare il contributo di ciascuna variabile al modello.
- **Parallelizzabile:** i singoli alberi, all'interno del Random forest, possono essere addestrati in parallelo. Questa caratteristica contribuisce significativamente alla sua velocità di addestramento, specialmente quando si lavora con dataset di grandi dimensioni e hardware multi-core.

## 4.4 Svantaggi

- **Costo computazionale e Tempo di addestramento:** l'addestramento di un Random forest può essere lento, in particolare con un numero molto elevato di alberi o su dataset di grandi dimensioni, poiché la costruzione di ogni albero è un'operazione computazionalmente intensiva.
- **Requisiti di risorse:** poiché elabora dataset potenzialmente grandi e costruisce molti alberi, richiede più risorse di memoria per archiviare i dati rispetto ad un singolo albero.
- **Complessità e Perdita di interpretability (rispetto a un singolo albero):** la previsione di una "foresta" di centinaia o migliaia di alberi diventa molto più difficile da interpretare a livello globale.



- **Mancanza di regolarizzazione esplicita:** A differenza di altri algoritmi ensemble, Random forest non include internamente tecniche di regolarizzazione dirette. Si affida principalmente alla sua natura ensemble e all'introduzione di casualità per prevenire l'overfitting.

## Capitolo 5

# eXtreme Gradient Boosting (XGBoost)

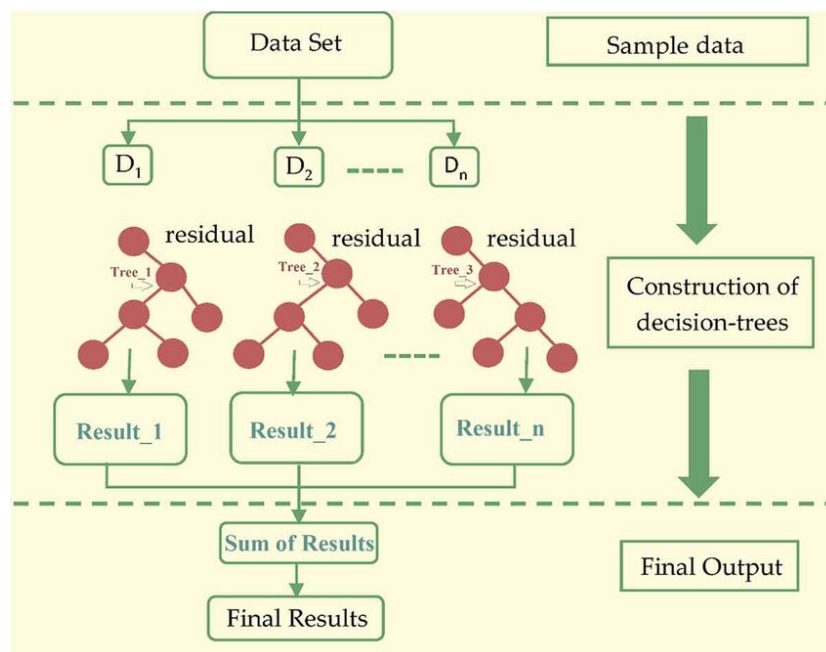


Figura 5.1: XGBoost.

## 5.1 Introduzione al Gradient boosting

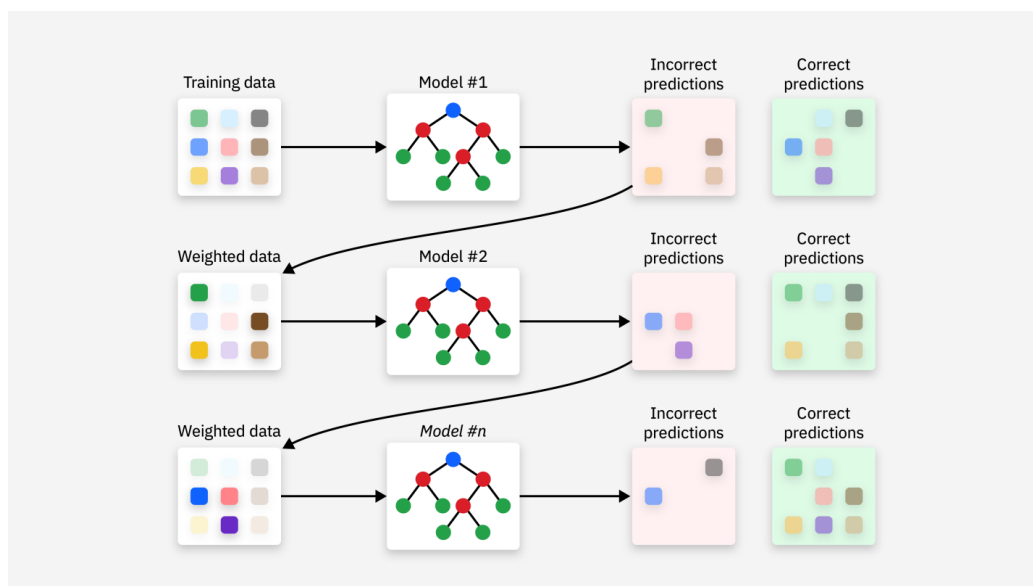


Figura 5.2: Gradient boosting.

Il Gradient boosting è una tecnica di Ensemble learning che costruisce un modello predittivo forte combinando iterativamente i risultati di numerosi "learner deboli". A differenza del Bagging, che addestra i modelli in parallelo, il Boosting adotta un approccio sequenziale, dove ogni nuovo modello cerca di correggere gli errori commessi dai modelli precedenti.

### 5.1.1 Principi iterativi e Weak learners

Il gradient boosting si fonda sull'idea di migliorare progressivamente un modello addestrando nuovi learner per correggere gli errori commessi dai precedenti. Questo processo è intrinsecamente iterativo e sequenziale. A differenza del Bagging, dove i learner sono addestrati in modo parallelo ed indipendente, il gradient boosting costruisce un modello additivo passo dopo passo. Ogni nuovo "albero" (che funge da learner debole) viene costruito specificamente per ridurre gli errori, o più precisamente gli "pseudo-residui", generati dalle previsioni degli alberi addestrati nelle

iterazioni precedenti. I "learner deboli", in questo caso, sono modelli che, se utilizzati singolarmente, classificano o predicono i dati in modo scarso e presentano un alto tasso di errore. Nel framework del gradient boosting, questi learner deboli sono tipicamente alberi di decisione semplici. Possono essere molto poco profondi, a volte ridotti ad un singolo split, in tal caso sono noti come "decision stumps". L'obiettivo generale del gradient boosting è minimizzare una funzione di perdita predefinita, aggiungendo iterativamente funzioni (i learner deboli) che puntano nella direzione del gradiente negativo della funzione di perdita stessa. Mentre il Bagging mira a ridurre la varianza addestrando modelli indipendenti e poi mediandone i risultati, il Boosting si concentra sulla riduzione del bias del modello. Addestrando sequenzialmente nuovi learner per correggere gli errori dei precedenti, il modello impara a concentrarsi sulle istanze più difficili da classificare o predire. Questo processo iterativo consente al modello di adattarsi in modo più efficace alle relazioni complesse presenti nei dati, riducendo il bias complessivo e portando spesso a una maggiore accuratezza predittiva. Questa differenza fondamentale nell'approccio, ovvero la riduzione della varianza contro la riduzione del bias, spiega perché il Bagging ed il Boosting eccellono in contesti diversi.

### 5.1.2 Funzione di perdita e Discesa del gradiente

Il processo di apprendimento nel gradient boosting è formalizzato come un algoritmo di discesa del gradiente nello spazio delle funzioni, dove l'obiettivo è trovare la funzione che minimizza la funzione di perdita, che quantifica quanto bene il modello sta eseguendo le previsioni sui dati forniti. La scelta della funzione di errore dipende dalla natura del problema: ad esempio, per problemi di regressione si potrebbe usare l'errore quadratico medio (MSE), mentre per problemi di classificazione si potrebbe usare la log-loss. L'algoritmo di gradient boosting mira a minimizzare questa funzione di perdita. In ogni iterazione, il modello calcola i cosiddetti "pseudo-residui", che non sono i residui tradizionali (differenza tra valore osservato e previsto), ma piuttosto i gradienti negativi della funzione di perdita rispetto alle

previsioni attuali del modello. Il nuovo learner debole (tipicamente un albero di decisione) viene quindi addestrato per predire questi pseudo-residui, imparando così a correggere gli errori del modello ensemble cumulativo. Il processo di aggiunta di nuovi alberi può essere interpretato come un passo nella direzione del gradiente negativo della funzione di perdita nello spazio delle funzioni. Un'innovazione significativa in XGBoost rispetto al gradient boosting tradizionale è l'utilizzo di un'approssimazione di Taylor del secondo ordine nella funzione di perdita. Questo approccio collega il processo di ottimizzazione di XGBoost al metodo Newton-Raphson, che è più robusto e può portare a una convergenza più rapida rispetto alla discesa del gradiente del primo ordine. L'utilizzo di un'approssimazione di Taylor del secondo ordine nella funzione di perdita distingue XGBoost dal gradient boosting tradizionale, che si basa sul gradiente del primo ordine. Ciò implica che XGBoost considera, non solo la direzione di discesa più ripida (data dal gradiente), ma anche la curvatura della funzione di perdita (data dall'hessiana). Ciò consente al modello di compiere passi più informati e potenzialmente più ampi verso il minimo della funzione di perdita, portando a una convergenza più rapida e stabile. Di conseguenza, XGBoost risulta meno sensibile a problemi come i minimi locali rispetto agli algoritmi che utilizzano esclusivamente il gradiente del primo ordine. Questa modifica è uno dei motivi principali della superiorità prestazionale di XGBoost in numerose competizioni di machine learning ed in svariate applicazioni reali. Dimostra come un'implementazione più avanzata dei principi fondamentali possa tradursi in miglioramenti significativi in termini di prestazioni ed efficienza del modello.

## **5.2 Miglioramenti di XGBoost rispetto al Gradient boosting tradizionale**

XGBoost è riconosciuto come un'evoluzione del gradient boosting, grazie all'integrazione di una serie di ottimizzazioni e tecniche di regolarizzazione che ne migliorano significativamente prestazioni, velocità e robustezza.

### 5.2.1 Regularizzazione e Tree pruning

XGBoost si distingue per l'integrazione di meccanismi di regolarizzazione configurabili, che sono cruciali per prevenire l'overfitting e migliorare la sua capacità di generalizzazione. Il modello incorpora termini di regolarizzazione L1 (Lasso) e L2 (Ridge), direttamente nella sua funzione obiettivo. Questi termini penalizzano la complessità del modello ed i pesi di grandi dimensioni. In particolare, la regolarizzazione L1 incoraggia la sparsità, spingendo i pesi meno importanti verso lo zero, mentre la regolarizzazione L2 incoraggia pesi più piccoli e distribuiti.

Una componente fondamentale della regolarizzazione nel gradient boosting è il "learning rate" (o shrinkage). Questo parametro riduce il contributo di ogni nuovo albero aggiunto al modello. Inoltre, è stato dimostrato che l'uso di "learning rate" piccoli (ad esempio, 0.01 o 0.1) migliori di molto la capacità di generalizzazione del modello, sebbene ciò comporti un aumento del numero di iterazioni e, di conseguenza, del tempo di calcolo.

XGBoost implementa anche tecniche avanzate di tree pruning. A differenza di alcuni algoritmi che costruiscono alberi fino alla massima profondità, XGBoost non lo fa. Il pruning avviene in base ad un "guadagno" del nodo. L'iperparametro "gamma" specifica la riduzione minima della funzione di errore richiesta per effettuare un'ulteriore partizione su un nodo foglia. Un valore più grande di questo rende l'algoritmo più conservativo, limitando la crescita dell'albero ed aiutando a prevenire l'overfitting.

Un altro parametro cruciale per la regolarizzazione è la "somma minima dei pesi delle istanze". Questo definisce la somma minima del peso delle istanze (basata sull'hessiana della funzione di perdita) necessaria in un nodo figlio per consentire un ulteriore split. Se esso si traduce in un nodo foglia con un peso inferiore all'iperparametro, la partizione viene annullata, mentre un valore più grande, rende l'algoritmo più conservativo, riducendo la complessità degli alberi individuali.

## 5.2.2 Gestione dei valori mancanti

XGBoost implementa un algoritmo, consapevole della sparsità, che gestisce i valori mancanti in modo flessibile. Durante la costruzione degli alberi di decisione, quando viene incontrato un valore mancante per una determinata feature, XGBoost non si limita a ignorare o richiedere una rimozione preliminare di essa. Invece, l'algoritmo è in grado di prendere una decisione su quale ramo del nodo il dato debba prendere, basandosi sui dati disponibili. Internamente, durante la fase di training, XGBoost apprende la direzione ottimale da seguire per le istanze con valori mancanti, trattando la "mancanza" del dato come una caratteristica informativa a sé stante. Molti algoritmi richiedono una rimozione esplicita dei valori mancanti, un processo che può introdurre bias o rumore nel dataset ed aumentare la complessità della fase di pre-processing. La capacità di XGBoost di gestire i valori mancanti internamente, imparando la direzione ottimale per gli split, rappresenta un vantaggio significativo. Questo non solo semplifica la pipeline di preparazione dei dati, ma può anche portare a modelli più robusti, poiché il modello apprende direttamente dalla "mancanza" dell'informazione, piuttosto che da un'eliminazione potenzialmente errata o arbitraria. Questa funzionalità rende XGBoost particolarmente efficiente e pratico per dataset reali, che spesso contengono valori mancanti, riducendo la necessità di avere una fase di pre-processing complessa, migliorando così l'affidabilità del modello in scenari del mondo reale.

## 5.2.3 Ottimizzazioni (Parallelismo, Cache-awareness)

Una delle ottimizzazioni chiave di XGBoost è il parallelismo. Sebbene il gradient boosting sia intrinsecamente sequenziale nella costruzione degli alberi (ogni albero corregge gli errori del precedente), XGBoost introduce il parallelismo, nella costruzione dei singoli alberi, in particolare sui livelli dell'albero o di split. Questo significa che, anziché costruire gli alberi in modo strettamente sequenziale, XGBoost può scansionare i valori del gradiente ed utilizzare somme parziali per valutare la qualità degli split in parallelo. Il sistema sfrutta tutti i core della CPU disponibili su una singola macchina e

può operare in modalità distribuita, massimizzando l'utilizzo della potenza di calcolo. Questo parallelismo su larga scala accelera significativamente il processo di addestramento.

Un'altra ottimizzazione interessante è il Cache-awareness. XGBoost è progettato per utilizzare in modo intelligente la cache della CPU per accelerare l'accesso ai dati. Durante l'addestramento, memorizza nella cache i calcoli intermedi e le statistiche importanti, evitando così di ricalcolare gli stessi valori ripetutamente. Questo riduce i ritardi nel recupero dei dati tra la CPU e memoria principale, portando ad un'elaborazione e previsioni molto più veloci.

Infine, XGBoost beneficia della GPU acceleration, che velocizza significativamente l'addestramento del modello e contribuisce a migliorare l'accuratezza delle previsioni. L'algoritmo sfrutta il calcolo parallelo per eseguire operazioni veloci, per ripartizionare i dati e costruire gli alberi un livello alla volta, elaborando l'intero dataset contemporaneamente sulla GPU.

## 5.3 Parametri chiave e Tuning

XGBoost offre un'ampia e dettagliata lista di iperparametri che possono essere ottimizzati per personalizzare il comportamento del modello e massimizzare le sue prestazioni. Questa flessibilità, sebbene potente, richiede una comprensione approfondita ed un'attenta strategia di tuning. I parametri per il Tree booster controllano la costruzione dei singoli alberi all'interno dell'Ensemble:

- `learning_rate` (o `eta`): questo è il tasso di apprendimento, un parametro cruciale che controlla la dimensione del passo di shrinkage per prevenire l'overfitting. Valori più piccoli di `eta` rendono il processo di boosting più conservativo, riducendo il rischio di overfitting ma richiedendo più iterazioni. Il suo intervallo è  $(0, 1]$ .
- `max_depth`: definisce la profondità massima di un albero. Aumentare questo valore rende il modello più complesso e potenzialmente più



incline all'overfitting. Un valore di 0 indica nessuna limitazione di profondità, ma ciò può portare a un elevato consumo di memoria. L'intervallo è  $[0, \infty]$ .

- **min\_child\_weight**: specifica la somma minima del peso delle istanze (basata sull'hessiana) necessaria in un nodo figlio per consentire un ulteriore split. Un valore più grande rende l'algoritmo più conservativo, limitando la crescita dell'albero. L'intervallo è  $[0, \infty]$ .
- **subsample**: rappresenta la frazione di osservazioni (istanze) campionate casualmente per la costruzione di ogni albero. Questo campionamento avviene una volta per ogni iterazione di boosting ed aiuta a prevenire l'overfitting. L'intervallo è  $(0, 1]$ .
- **colsample\_bytree**: indica la frazione di features campionate casualmente per la costruzione di ogni albero. Questo parametro contribuisce anch'esso a prevenire l'overfitting introducendo casualità nella selezione delle caratteristiche. L'intervallo è  $(0, 1]$ .
- **lambda** (o **reg\_lambda**): è il termine di regolarizzazione L2 sui pesi. Aumentare questo valore rende il modello più conservativo, penalizzando i pesi grandi. L'intervallo è  $[0, \infty)$ .
- **alpha** (o **reg\_alpha**): è il termine di regolarizzazione L1 sui pesi. Un valore più grande rende il modello più conservativo e incoraggia la sparsità, spingendo i pesi meno importanti verso lo zero. L'intervallo è  $[0, \infty]$ .
- **objective**: definisce la funzione obiettivo che il modello mira a minimizzare. Ad esempio, **reg:squarederror** per problemi di regressione, **binary:logistic** per classificazione binaria e **multi:softprob** per classificazione multiclasse.
- **eval\_metric**: specifica la metrica di valutazione da monitorare durante l'addestramento. È possibile specificare più metriche.

Per l'ottimizzazione dei parametri, sono essenziali diverse tecniche:

- **Hyperparameter tuning:** tecniche come Grid search, Random search, Bayesian optimization o Genetic algorithm sono cruciali per esplorare lo spazio degli iperparametri e trovare la combinazione ottimale che massimizza le prestazioni del modello.
- **Early stopping:** consiste nell'interrompere l'addestramento del modello se la metrica di validazione su un set di dati separato non migliora per un certo numero di round consecutivi. Ciò impedisce al modello di continuare ad apprendere il rumore nei dati di training una volta che le sue prestazioni di generalizzazione iniziano a peggiorare, evitando l'overfitting.

A differenza di Random forest, che tende ad avere meno parametri da ottimizzare, XGBoost richiede una comprensione più approfondita e una sperimentazione più estesa per raggiungere le sue prestazioni ottimali. Questo implica che, sebbene XGBoost possa teoricamente superare Random forest in termini di accuratezza, raggiungere tale superiorità richiede un investimento maggiore in termini di tempo e risorse per il tuning.

## 5.4 Vantaggi

- **Robustezza all'overfitting:** grazie alle sue varie tecniche di regolarizzazione integrate, XGBoost è molto robusto contro l'overfitting. Questo controllo granulare sulla complessità del modello è un fattore chiave della sua capacità di generalizzazione.
- **Gestione efficiente di grandi dataset:** è stato progettato per scalabilità ed efficienza, gestendo efficacemente dati di grandi dimensioni, dati sparsi e valori mancanti. La sua architettura ottimizzata consente di elaborare grandi quantità di dati che sarebbero impossibili da gestire per altri algoritmi.
- **Velocità di addestramento:** nonostante la sua natura sequenziale, XGBoost è altamente ottimizzato per la velocità. Può essere più

veloce di Random forest nell'addestramento, specialmente quando si sfruttano le sue capacità di parallelismo e l'accelerazione GPU, oltre al supporto per sistemi distribuiti.

- **Flessibilità e Personalizzazione:** offre un'ampia gamma di iperparametri che consentono un fine-tuning profondo del modello, adattandolo alle specifiche esigenze del problema e dataset.
- **Gestione dati sbilanciati:** è particolarmente efficace nella gestione di dataset con classi sbilanciate.

## 5.5 Svantaggi

- **Processo di addestramento sequenziale:** nonostante le ottimizzazioni per il parallelismo interno, la sua natura di boosting sequenziale (dove ogni albero dipende dal precedente) può renderlo intrinsecamente più lento di Random forest nella costruzione degli alberi completi, a meno che non si sfruttino appieno le sue capacità di parallelismo e le accelerazioni hardware.
- **Complessità e Tuning:** è un algoritmo più complesso da comprendere ed implementare rispetto al Random forest. La sua vasta gamma di iperparametri richiede una maggiore conoscenza ed esperienza per un fine-tuning efficace, rendendo il processo più dispendioso in termini di tempo e risorse.
- **Meno interpretabile di Random forest (a volte):** sebbene fornisca l'importanza delle feature, la complessità dell'ensemble di alberi sequenziali può rendere le sue decisioni specifiche più difficili da interpretare rispetto ad un Random forest, dove spesso si richiedono strumenti aggiuntivi per la spiegabilità.
- **Consumo di memoria:** può consumare una notevole quantità di memoria, specialmente quando si addestrano alberi molto profondi,

il che può essere una limitazione per dataset estremamente grandi su hardware con risorse limitate.

## Capitolo 6

# Convolutional Neural Networks (CNN)

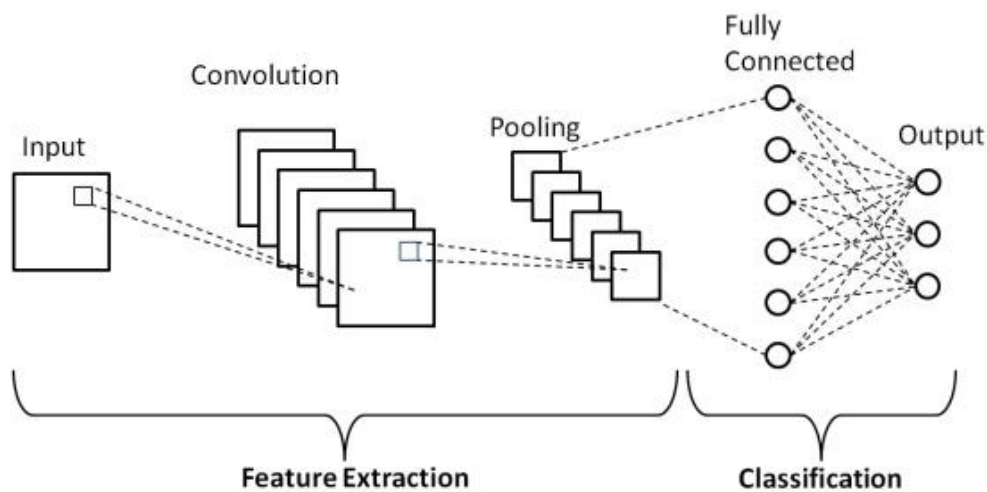


Figura 6.1: Convolutional Neural Networks.

## 6.1 Principi fondamentali delle CNN

### 6.1.1 Convoluzione: kernel, stride, padding

Un kernel (o filtro) è una matrice di piccole dimensioni, tipicamente  $3 \times 3$ ,  $5 \times 5$  o  $7 \times 7$ , contenente pesi apprendibili che vengono moltiplicati elemento

per elemento con porzioni locali dell'input. Durante l'operazione di convoluzione, il kernel viene fatto scorrere attraverso l'intera immagine di input, calcolando il prodotto scalare tra i pesi del filtro e i valori corrispondenti dell'input in ogni posizione.

Lo stride rappresenta la grandezza in pixel di cui il kernel si sposta ad ogni passo durante la convoluzione. Uno stride di 1 significa che il filtro si muove di un pixel alla volta, producendo un output con dimensioni spaziali simili all'input. Uno stride maggiore (ad esempio 2 o 3) riduce significativamente le dimensioni dell'output, fornendo un effetto di downsampling.

Il padding è una tecnica che consiste nell'aggiungere pixel (solitamente con valore zero) ai bordi dell'immagine di input. Esistono due tipi principali di padding:

- Valid padding: nessun padding viene aggiunto, l'output risulta più piccolo dell'input.
- Same padding: viene aggiunto padding sufficiente per mantenere le stesse dimensioni spaziali dell'input.

La formula per calcolare le dimensioni dell'output di una convoluzione è:

$$O = \frac{W - K + 2P}{S} + 1$$

dove  $W$  è la dimensione dell'input,  $K$  è la dimensione del kernel,  $P$  è il padding e  $S$  è lo stride.

### 6.1.2 Feature maps e profondità dei canali

Le feature maps rappresentano l'output prodotto dall'applicazione di filtri convoluzionali all'input. Ogni feature map corrisponde ad un filtro specifico e rappresenta la risposta di quel filtro all'immagine di input. Negli strati iniziali della rete, le feature maps possono catturare caratteristiche semplici come bordi, linee e angoli, mentre negli strati più profondi possono rappresentare pattern più complessi come forme, texture o addirittura oggetti interi.

La profondità dei canali si riferisce al numero di feature maps prodotte da uno strato convoluzionale. Aumentare il numero di feature maps consente alla rete di apprendere caratteristiche più complesse e astratte, ma incrementa anche il costo computazionale e può portare ad overfitting se la rete è troppo grande per i dati disponibili. Un aspetto cruciale è che la profondità di un filtro deve corrispondere alla profondità dell'input. Ad esempio, per un'immagine RGB (3 canali), ogni filtro deve avere profondità 3. L'output di ogni convoluzione è una feature map 2D, indipendentemente dalla profondità dell'input.

### 6.1.3 Convoluzioni 1D, 2D e 3D

Le CNN possono operare su dati di diversa dimensionalità, richiedendo tipi specifici di convoluzione:

- **Convoluzione 1D:** viene utilizzata per dati sequenziali come segnali temporali, dati finanziari, segnali biomedici (EEG, ECG) o testi. Il kernel si muove lungo una sola direzione (asse temporale), producendo un output 1D.
- **Convoluzione 2D:** è la più comune nelle CNN per Computer vision. Il kernel si muove in due direzioni (x e y) attraverso l'immagine, producendo feature maps 2D. Anche quando l'input è 3D (come un'immagine RGB), la convoluzione opera ancora in 2D poiché la profondità del filtro deve corrispondere al numero di canali dell'input.
- **Convoluzione 3D:** è utilizzata per dati volumetrici come video, scansioni MRI o TAC. Il kernel si muove in tre direzioni (x,y,z), producendo un output 3D. È essenziale che la profondità del filtro sia minore di quella dell'input per generare un output volumetrico.

## 6.2 Pooling e normalizzazione

### 6.2.1 Max pooling vs average pooling

Il pooling è un'operazione di downsampling che riduce le dimensioni spaziali delle feature maps mantenendo le informazioni più importanti. Questa operazione migliora l'efficienza computazionale e introduce una forma di invarianza alle traslazioni, rendendo la rete meno sensibile a piccoli spostamenti nell'input.

Il max pooling seleziona il valore massimo all'interno di ogni finestra di pooling. È particolarmente efficace nel preservare le caratteristiche più importanti e nell'introdurre invarianza alle traslazioni. Ad esempio, con una finestra  $2 \times 2$  e stride 2, il max pooling riduce le dimensioni dell'input della metà mantenendo le attivazioni più forti.

L'average pooling calcola la media dei valori all'interno di ogni finestra di pooling. Mentre preserva più informazione locale rispetto al max pooling, può essere meno efficace nel gestire variazioni sottili delle caratteristiche o features significative in certe regioni dell'immagine.

### 6.2.2 Global pooling

Il global pooling è una variante che riduce ogni feature map ad un singolo valore, eliminando completamente le dimensioni spaziali. Il global max pooling seleziona il valore massimo da ogni feature map intera, mentre il global average pooling calcola la media di tutti i valori in ogni feature map. Questa tecnica è spesso utilizzata prima degli strati fully connected finali nelle architetture CNN per la classificazione, riducendo drasticamente il numero di parametri e prevenendo l'overfitting.

### 6.2.3 Batch, Layer e Group normalization

La batch normalization è una tecnica introdotta per accelerare l'addestramento delle reti neurali profonde e ridurre la sensibilità all'inizializzazione dei parametri. Normalizza gli input di ogni strato utilizzando la media e



varianza del mini-batch corrente durante l'addestramento.

Per ogni canale, durante la fase di apprendimento, la batch normalization calcola:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

dove  $\mu_B$  e  $\sigma_B^2$  sono la media e varianza del batch, ed  $\epsilon$  è una costante piccola per evitare divisioni per zero. Successivamente applica una trasformazione affine apprendibile:

$$y_i = \gamma \hat{x}_i + \beta$$

dove  $\gamma$  e  $\beta$  sono parametri apprendibili.

La layer normalization normalizza tutti i neuroni in un particolare strato per ogni input individualmente, rendendola indipendente dalla dimensione del batch.

La group normalization divide i canali in gruppi e normalizza all'interno di ogni gruppo, offrendo un compromesso tra batch e layer normalization.

## 6.3 Data augmentation: rotazioni, zoom, colour jitter

La data augmentation è una tecnica che aumenta artificialmente le dimensioni del dataset applicando trasformazioni realistiche agli esempi di training, migliorando la generalizzazione e riducendo l'overfitting.

Le rotazioni ruotano le immagini di angoli casuali (tipicamente tra  $-50^\circ$  e  $50^\circ$ ), aiutando la rete a riconoscere oggetti indipendentemente dal loro orientamento. Studi hanno dimostrato che la rotazione può migliorare significativamente le prestazioni.

Lo zoom (o scaling) modifica le dimensioni degli oggetti nell'immagine, permettendo alla rete di riconoscere oggetti a diverse scale. Il random crop è una variante che estrae porzioni casuali dell'immagine originale.

Il colour jitter modifica luminosità, contrasto, saturazione e tonalità delle immagini. Questa tecnica varia i canali RGB con valori casuali, producendo

cambiamenti casuali nel colore che aiutano la rete a essere invariante alle variazioni di illuminazione e colore.

## 6.4 Funzionamento delle CNN

### 6.4.1 Forward pass

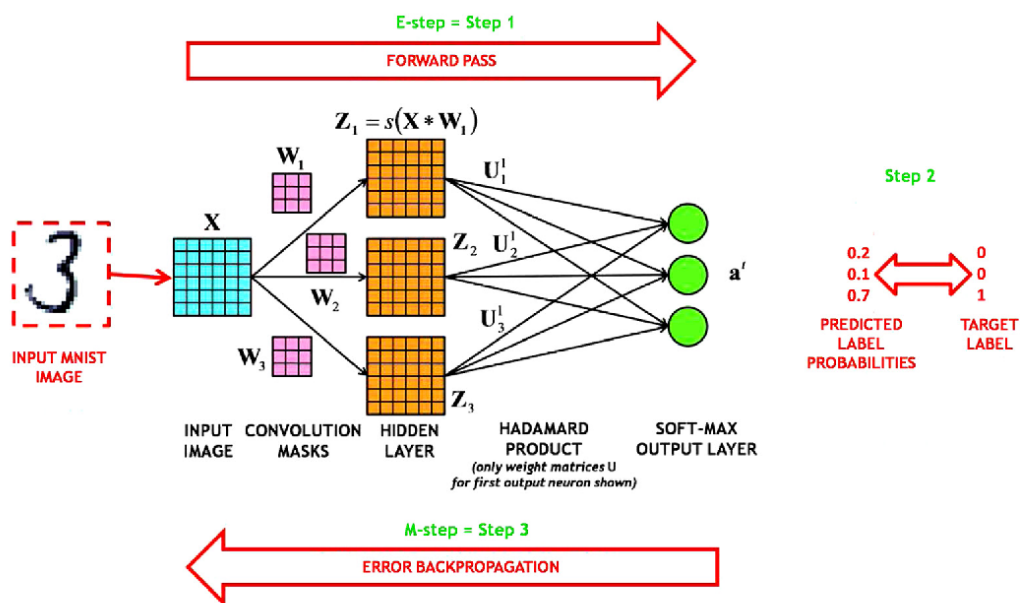


Figura 6.2: Diagramma del forward pass in una CNN.

Il forward pass rappresenta il percorso che i dati seguono dall'input verso l'output attraverso l'intera architettura della rete. Questo processo inizia con l'immagine grezza e termina con la predizione finale, passando attraverso una serie di trasformazioni matematiche che estraggono progressivamente caratteristiche sempre più complesse.

### 6.4.2 Strati di convoluzione

Il primo stadio, per l'elaborazione delle immagini, coinvolge gli strati convoluzionali, che rappresentano il cuore dell'architettura CNN. Quando

un'immagine di input entra nella rete, essa viene elaborata attraverso un insieme di filtri (kernel) che eseguono l'operazione di convoluzione. Ogni filtro è responsabile del rilevamento di una specifica caratteristica: negli strati iniziali, questi filtri apprendono a riconoscere caratteristiche di basso livello come bordi, linee e angoli. L'operazione di convoluzione produce le feature maps, che rappresentano la risposta di ciascun filtro all'immagine di input. Ogni elemento nella feature map indica l'intensità della presenza di quella specifica caratteristica in quella posizione dell'immagine.

### **6.4.3 Funzioni di attivazione**

Dopo ogni operazione di convoluzione, viene applicata una funzione di attivazione, tipicamente ReLU. Questa fase è cruciale perché introduce non-linearità nel modello, permettendo alla rete di apprendere relazioni complesse nei dati.

### **6.4.4 Strati di pooling**

Successivamente agli strati convoluzionali, i dati passano attraverso gli strati di pooling. Questi strati eseguono un'operazione di downsampling che riduce le dimensioni spaziali delle feature maps mantenendo le informazioni più importanti.

### **6.4.5 Gerarchia delle caratteristiche**

Man mano che i dati attraversano strati successivi, si verifica un fenomeno fondamentale: la costruzione gerarchica delle caratteristiche. Gli strati iniziali rilevano caratteristiche elementari (bordi, texture), gli strati intermedi combinano queste caratteristiche per formare pattern più complessi (forme geometriche, motivi), mentre gli strati più profondi assemblano questi pattern in rappresentazioni di alto livello che corrispondono a parti di oggetti o oggetti interi.

### 6.4.6 Strati fully-connected

Dopo l'estrazione gerarchica delle caratteristiche, i dati raggiungono gli strati fully-connected. Prima di entrare in questi strati, le feature maps multidimensionali vengono convertite in un vettore unidimensionale. Negli strati fully-connected, ogni neurone è collegato a tutti i neuroni dello strato precedente, permettendo alla rete di combinare tutte le caratteristiche estratte per prendere la decisione finale, in base al tipo di problema.

### 6.4.7 Flusso informativo e Trasformazioni progressive

Durante tutto questo processo, l'immagine originale, inizialmente rappresentata come una matrice di valori pixel, viene gradualmente trasformata in rappresentazioni sempre più astratte e significative dal punto di vista semantico. Ogni strato della rete contribuisce a questa trasformazione: gli strati convoluzionali estraggono e raffinano le caratteristiche, gli strati di pooling riducono la complessità computazionale e introducono invarianza, mentre gli strati fully-connected integrano tutte le informazioni per la classificazione finale.

Questo design architetturale permette alle CNN di apprendere automaticamente le rappresentazioni ottimali per il compito specifico, eliminando la necessità di progettare manualmente gli estrattori di caratteristiche. La capacità di costruire rappresentazioni gerarchiche rende le CNN particolarmente efficaci per compiti di computer vision, dove la comprensione dell'immagine richiede l'integrazione di informazioni a diversi livelli di astrazione.

## 6.5 Vantaggi

- **Rilevamento automatico delle caratteristiche:** a differenza delle reti neurali tradizionali che richiedono l'estrazione manuale delle caratteristiche, le CNN sono in grado di apprendere ed identificare automaticamente le features rilevanti direttamente dai dati grezzi.

Questo riduce significativamente lo sforzo nella fase di data pre-processing e permette al modello di adattarsi meglio ai dati.

- **Efficienza computazionale e riduzione dei parametri:** l'uso del weight sharing (una tecnica che consente ad un singolo filtro di rilevare la stessa caratteristica in qualsiasi posizione dell'immagine, utilizzando lo stesso set di pesi) e degli strati di pooling riduce notevolmente il numero di parametri da addestrare rispetto alle reti neurali fully-connected. Questo non solo accelera l'addestramento, ma aiuta anche a prevenire l'overfitting.
- **Invarianza alla traslazione:** grazie all'operazione di convoluzione, le CNN sono in grado di riconoscere un oggetto indipendentemente dalla sua posizione nell'immagine. Un filtro che impara a riconoscere un occhio, ad esempio, lo riconoscerà sia che si trovi in alto a sinistra che in basso a destra.
- **Scalabilità:** le architetture CNN possono essere adattate facilmente a dataset di grandi dimensioni ed a compiti complessi. Ad esempio, la profondità e larghezza della rete possono essere aumentate per gestire immagini ad alta risoluzione o per apprendere pattern più astratti.

## 6.6 Svantaggi

- **Dipendenza da grandi dataset:** le CNN, specialmente quelle con architetture complesse, richiedono enormi quantità di dati etichettati per essere addestrate in modo efficace. La mancanza di un dataset sufficientemente grande può portare all'overfitting o ad una scarsa generalizzazione.
- **Invarianza limitata:** le CNN standard sono intrinsecamente invarianti alla traslazione, ma non lo sono rispetto ad altre trasformazioni geometriche. Se un'immagine viene ruotata o scalata in modo significativo, il modello potrebbe non riconoscerla correttamente, a meno che non

si utilizzino tecniche di data augmentation per esporre il modello a queste variazioni durante l'addestramento.

- **Mancanza di interpretazione:** spesso è difficile comprendere perché una CNN prenda una determinata decisione. Le feature map intermedie possono essere visualizzate, ma il processo decisionale rimane una "black box", rendendo complicato il debug e l'adozione in settori critici come la medicina, dove è necessaria la trasparenza.
- **Costo computazionale:** l'addestramento di architetture CNN molto profonde può richiedere una notevole potenza di calcolo, spesso necessitando di hardware specializzato come le GPU. Anche l'inferenza su dispositivi a bassa potenza può essere problematica.

# Capitolo 7

## Ottimizzazione degli iperparametri

### 7.1 Cross-Validation

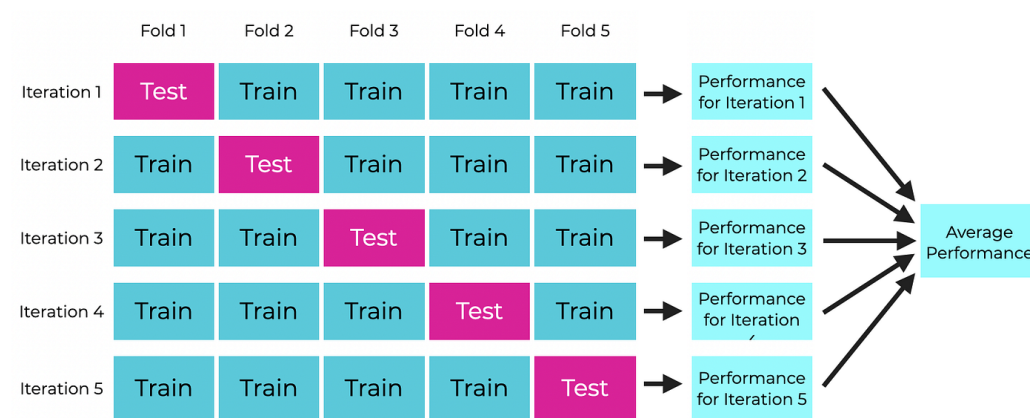


Figura 7.1: Cross-Validation.

La CV è una tecnica statistica per la valutazione delle prestazioni di un modello predittivo. Il suo scopo principale è quello di stimare quanto un modello è in grado di generalizzare su dati indipendenti non visti durante la fase di training. Dato un dataset  $D = \{z_1, \dots, z_N\}$  e una procedura di training che produce un modello  $\hat{f}_D$ , la K-fold cross-validation

divide i dati in  $K$  parti disgiunte (fold), dove ognuno ha una dimensione approssimativamente uguale. Il processo si ripete  $K$  volte. Per ogni iterazione  $k \in \{1, \dots, K\}$ :

- Si usano le restanti  $K - 1$  parti, che costituiscono il training set  $D^{(k)} = D \setminus D_k$ , per allenare il modello. Questo produce un modello parziale  $\hat{f}_{D^{(k)}}$ .
- Si valuta l'errore del modello  $\hat{f}_{D^{(k)}}$  sul fold lasciato fuori  $D_k$ , che ricopre il ruolo del validation set per questa iterazione. L'errore viene calcolato come  $E_k = \text{Errore}(\hat{f}_{D^{(k)}}, D_k)$ .

Al termine delle  $K$  iterazioni, si ottiene una lista di  $K$  errori  $\{E_1, E_2, \dots, E_K\}$ . La stima finale della performance del modello è data dalla media degli errori calcolati su ogni fold,  $\bar{E} = \frac{1}{K} \sum_{k=1}^K E_k$ .

### 7.1.1 Time Series Cross-Validation (TSCV)

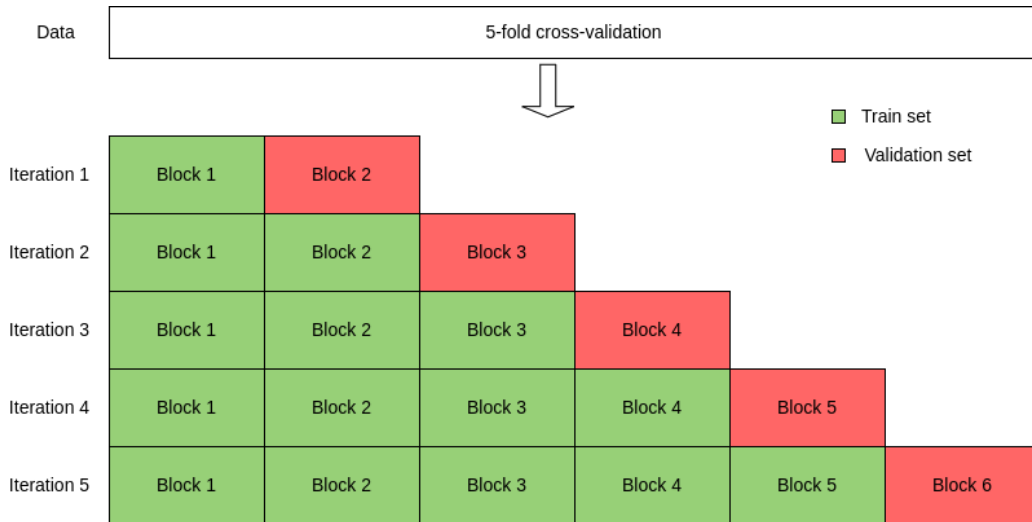


Figura 7.2: Time Series Cross-Validation.

Per i dati che hanno una dipendenza temporale, come le serie storiche, la CV standard non è adatta. Suddividere i dati in fold casuali romperebbe la struttura temporale, e l'addestramento su dati futuri per testare il modello su



dati passati (fenomeno noto come data leakage) non avrebbe senso pratico e porterebbe a risultati fuorvianti. La TSCV, sviluppata per risolvere questa problematica, crea un training set che cresce sequenzialmente nel tempo, ed il validation set è sempre un blocco di dati che segue immediatamente il dataset di allenamento. Il processo funziona così:

- **Prima iterazione:** il modello è addestrato sui primi  $m$  punti temporali e testato sui successivi  $n$  punti.
- **Seconda iterazione:** il modello è addestrato sui primi  $m + n$  punti temporali e testato sui successivi  $n$  punti.
- **Iterazioni successive:** il processo continua, con il training set che si espande ad ogni passo ed il validation set che avanza nel tempo.

Questo approccio rispecchia fedelmente lo scenario reale in cui un modello di serie storica viene addestrato su dati passati e utilizzato per fare previsioni su dati futuri non ancora visti. La media degli errori di validazione su tutte le iterazioni fornisce una stima robusta e realistica delle prestazioni del modello.

## 7.2 Nested Cross-Validation (NCV)

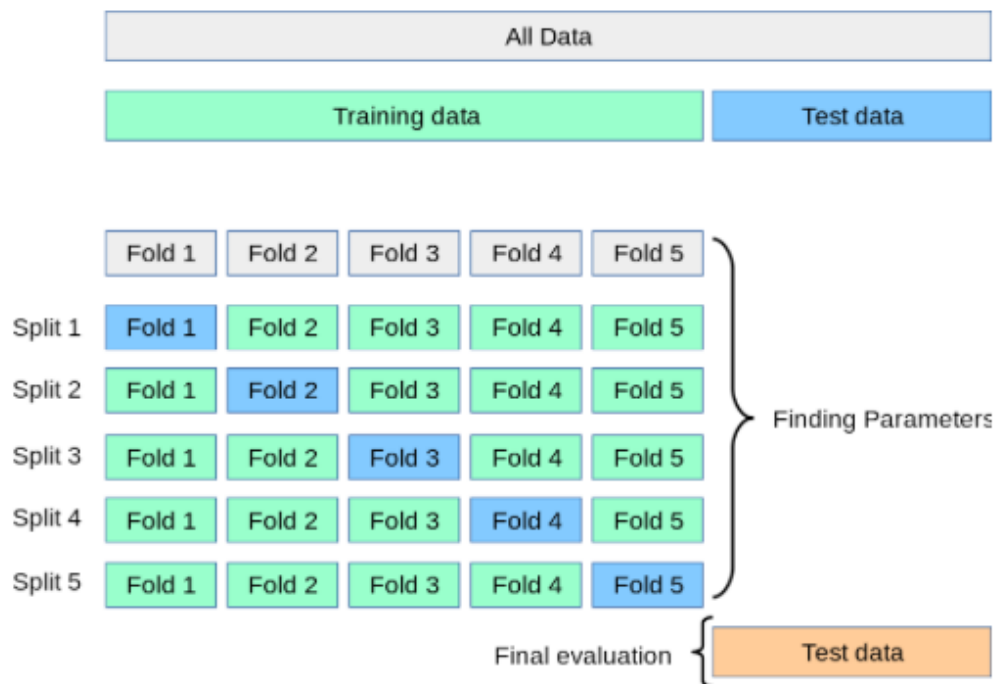


Figura 7.3: Nested Cross-Validation.

La NCV é un'estensione della classica CV. Il suo scopo principale è quello di fornire una stima imparziale ed affidabile dell'errore di generalizzazione di un modello, risolvendo il problema del bias di selezione che può verificarsi quando gli stessi dati vengono utilizzati sia per la scelta degli iperparametri che per la valutazione finale del modello. L'idea alla base della NCV è quella di creare due cicli di CV: un ciclo esterno ed uno interno.

1. **Ciclo esterno (Outer loop):** ha il compito di stimare l'errore di generalizzazione del modello. Il dataset viene diviso in  $K$  fold. Per ogni iterazione di questo ciclo, vengono utilizzati  $K - 1$  parti per costruire il training set esterno ed il restante fold agisce da test set finale, che non verrà mai utilizzato per la selezione degli iperparametri, garantendo una valutazione finale imparziale.

2. **Ciclo interno (Inner loop):** all'interno di ogni iterazione del ciclo esterno, si esegue un altro ciclo di CV (solitamente con  $L$  fold) sul training set esterno. Questo ciclo interno è dedicato esclusivamente all'ottimizzazione degli iperparametri. Per ogni combinazione di essi da testare (ad esempio, utilizzando Grid o Random Search), si addestra il modello sui  $L - 1$  fold interni e si valuta la sua performance sul fold interno rimanente. La combinazione di iperparametri che ottiene la migliore performance media su tutte le  $L$  iterazioni viene selezionata.
3. **Valutazione del modello ottimizzato:** una volta trovata la migliore combinazione di iperparametri nel ciclo interno, il modello viene addestrato nuovamente sull'intero training set esterno utilizzando proprio quella combinazione ottimale. Infine, la performance di questo modello viene valutata sul test set esterno, che è stato lasciato fuori all'inizio dell'iterazione  $K$ . L'errore ottenuto in questa fase è la stima delle prestazioni di generalizzazione del modello per quella specifica iterazione del ciclo esterno.

Questo processo viene ripetuto per tutti i  $K$  fold del ciclo esterno. La stima finale dell'errore del modello è la media dei  $K$  errori ottenuti sui test set esterni.

## 7.3 Grid search (GS)

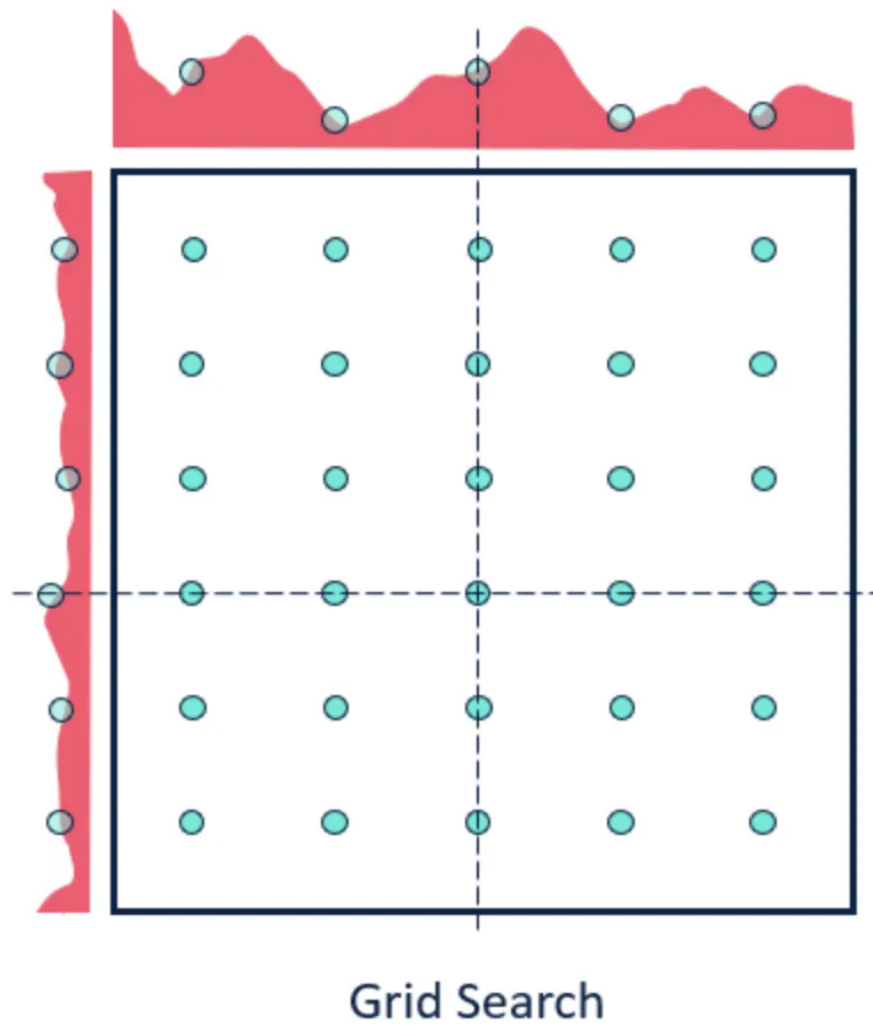


Figura 7.4: Grid search.

### 7.3.1 Spiegazione dell'algoritmo

Il GS consiste nel definire una griglia discreta di possibili valori per ciascun iperparametro e nell'eseguire una valutazione esaustiva del modello per

ogni combinazione. Alla fine, si seleziona il set di iperparametri che ottimizza la metrica di interesse. Il metodo non introduce casualità, risultando completamente ripetibile e deterministico.

### 7.3.2 Vantaggi

- **Esaustivo:** esplora tutte le combinazioni predefinite nello spazio di ricerca, consentendo di trovare l'ottimo globale, se questo è incluso nella griglia.
- **Deterministico e Riproducibile:** poiché non c'è casualità, ogni esecuzione dell'algoritmo ripete gli stessi test fornendo risultati replicabili.
- **Semplicità di implementazione:** ideale in spazi di ricerca ridotti e ben definiti, o come baseline quando si dispone di risorse elevate.

### 7.3.3 Limiti

- **Costo computazionale esponenziale:** la complessità cresce rapidamente con il numero di iperparametri (curse of dimensionality), rendendo Grid search impraticabile per spazi ampi o ad alta dimensionalità.
- **Inefficiente:** molte valutazioni potrebbero riguardare regioni poco promettenti; spesso solo alcuni parametri influenzano la prestazione ottimale.
- **Discretizzazione e loss di ottimi:** la necessità di fissare una griglia per iperparametri continui può portare a saltare valori potenzialmente migliori che non sono inclusi.
- **Non adatto a modelli e dataset complessi:** il costo aumenta drasticamente con la complessità e con la dimensione del dataset.

## 7.4 Random Search (RS)

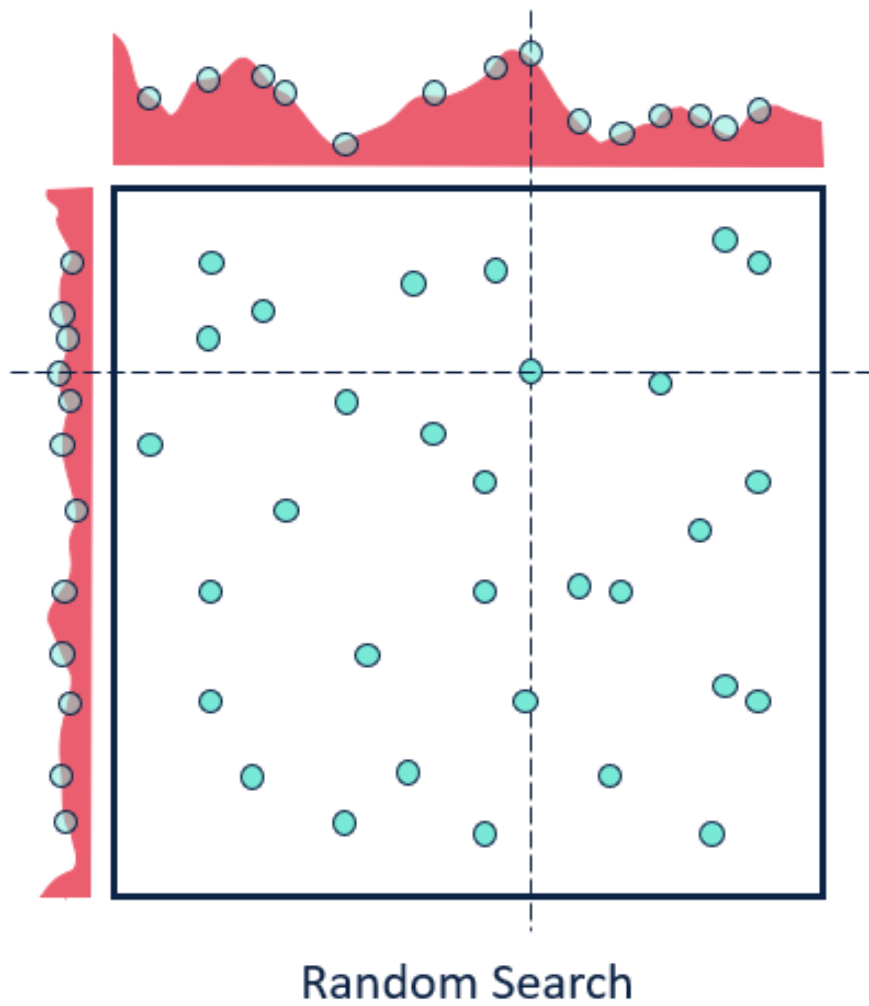


Figura 7.5: Random search.

### 7.4.1 Spiegazione dell'algoritmo

Il RS seleziona casualmente  $N$  configurazioni dagli intervalli o distribuzioni scelte per ciascun iperparametro, valutando il modello solo in quei punti. La campionatura può avvenire secondo distribuzioni uniformi, log-uniformi o

guidate da conoscenze pregresse. Questo approccio si basa, quindi, sulla randomizzazione invece che su una griglia predefinita.

### 7.4.2 Vantaggi

- **Efficienza su spazi estesi:** campionando casualmente, si ha maggior probabilità di individuare combinazioni efficaci, specialmente quando solo pochi parametri sono percepiti come determinanti.
- **Scalabilità:** il numero di valutazioni può essere fissato (es.  $N = 100$ ), spesso ottenendo performance simili a Grid Search che richiede molte più combinazioni.
- **Facile da parallelizzare:** ogni valutazione è indipendente, consentendo l'esecuzione in parallelo.
- **Adattabile:** è possibile scegliere distribuzioni di campionamento informate da conoscenze a priori.

### 7.4.3 Limiti

- **Non sistematico:** non esplora esaustivamente lo spazio delle ipotesi, quindi può saltare l'ottimo globale.
- **Dipendenza dalla distribuzione di campionamento:** campionamenti mal scelti possono ignorare regioni promettenti.
- **Risultati non ripetibili:** salvo fissare un seed random, i risultati possono variare da un'esecuzione all'altra.
- **Sotto-esplorazione:** su spazi piccoli e ben definiti, Grid Search resta più efficace.

## 7.5 Bayesian optimization (BO)

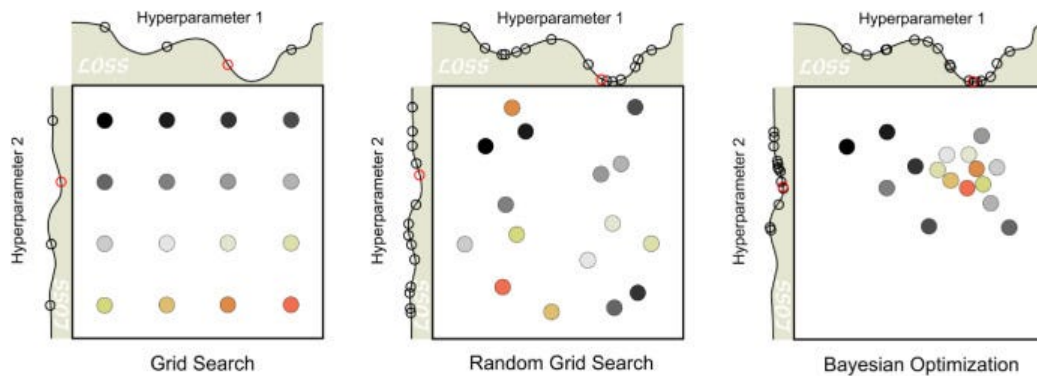


Figura 7.6: Bayesian optimization.

### 7.5.1 Spiegazione dell'algoritmo

La BO interpreta l'ottimizzazione degli iperparametri come la ricerca del massimo/minimo di una funzione obiettivo costosa ed ignota. Si costruisce un modello surrogato probabilistico (come un Gaussian Process, GP) che stima la funzione obiettivo e quantifica l'incertezza predittiva. Ad ogni iterazione, una funzione di acquisizione determina il prossimo punto da valutare.

#### Modello surrogato: Gaussian Process (GP)

Un GP definisce, per ogni punto  $\lambda$ , una distribuzione normale per il valore  $f(\lambda)$  con media  $\mu(\lambda)$  e varianza  $\sigma^2(\lambda)$ . Dopo aver osservato alcune valutazioni, si aggiorna  $\mu$  e  $\sigma$  per riflettere ciò che si è imparato.

#### Le fasi del BO

1. **Campionamento iniziale:** Si comincia con una serie di prove iniziali, selezionando casualmente diverse combinazioni di iperparametri. Per ogni combinazione, si addestra il modello e si calcola una metrica di performance, come la precisione (accuracy) o l'errore quadratico medio (MSE), che funge da risultato della funzione obiettivo.



2. **Modello surrogato:** sulla base dei risultati del campionamento iniziale, si costruisce un modello probabilistico, solitamente un Gaussian process, che approssima la funzione obiettivo. Questo modello non solo predice la performance attesa per una data combinazione di iperparametri, ma fornisce anche un'incertezza sulla predizione.
3. **Funzione di acquisizione:** viene usata per decidere la prossima combinazione di iperparametri da testare. Questa funzione bilancia l'esplorazione (provare combinazioni di cui si sa poco, per ridurre l'incertezza) e lo sfruttamento (provare combinazioni che il modello surrogato ritiene promettenti, per migliorare la performance).
4. **Valutazione della performance:** la nuova combinazione di iperparametri viene utilizzata per addestrare il modello e valutarne le prestazioni. Il risultato di questa valutazione rappresenta il nuovo punto dati che viene aggiunto al nostro set di informazioni.
5. **Aggiornamento del modello surrogato:** viene aggiornato con i nuovi risultati. Questo affina le sue predizioni e riduce l'incertezza, permettendo alla funzione di acquisizione di prendere decisioni più informate nelle iterazioni successive.
6. **Ripetizione:** i passaggi 3, 4 e 5 vengono ripetuti. Il ciclo si ferma quando si raggiunge un criterio predefinito, come un budget di tempo, un numero massimo di iterazioni, o quando la performance del modello smette di migliorare significativamente.

### 7.5.2 Vantaggi

- **Efficiente:** riduce drasticamente il numero di valutazioni necessarie, ideale per funzioni complesse dove ogni valutazione richiede molto tempo.
- **Bilanciamento exploration-exploitation:** la funzione di acquisizione permette di esplorare regioni nuove e raffinare aree note.

- **Modellazione dell'incertezza:** il modello surrogato stima l'incertezza ed indirizza la ricerca nelle zone potenzialmente migliori.

### 7.5.3 Limiti

- **Overhead computazionale:** il mantenimento ed aggiornamento del modello surrogato aggiunge complessità computazionale.
- **Difficile parallelizzazione:** il processo guidato dalle valutazioni precedenti rende più complessa la parallelizzazione rispetto alla Random o Grid Search.
- **Scalabilità limitata:** efficiente principalmente su spazi continui di media dimensione; su spazi molto ampi, l'ottimizzazione può diventare difficoltosa.

## 7.6 Genetic algorithm (GA)

### Genetic Algorithms

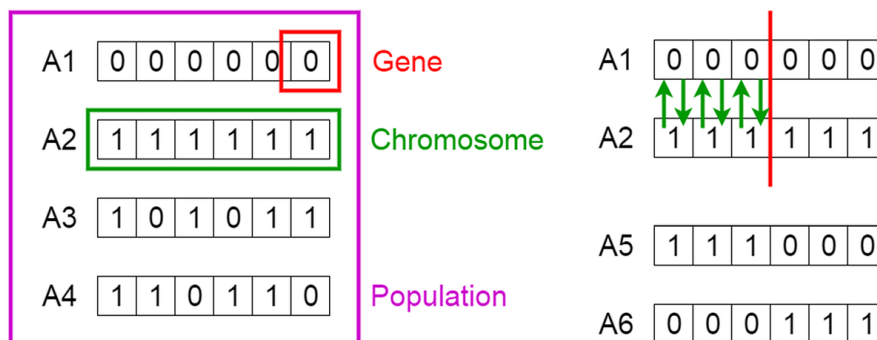


Figura 7.7: Genetic algorithm.

### 7.6.1 Spiegazione dell'algoritmo

Gli algoritmi genetici (GA) sono metodi di ottimizzazione di iperparametri, ispirati al processo di selezione naturale. Funzionano mantenendo una popolazione di soluzioni e migliorandole nel tempo tramite un processo iterativo. Ad ogni iterazione, detta anche generazione, vengono applicati tre operatori evolutivi principali:

- **Selezione:** gli individui con un punteggio di fitness più alto (le soluzioni "migliori" o più "adatte") vengono scelti per la riproduzione. Questo processo assicura che le caratteristiche positive si diffondano nella popolazione.

- **Crossover:** le soluzioni selezionate vengono combinate per creare nuovi individui "figli". Questo operatore permette di esplorare nuove combinazioni e di mescolare le caratteristiche delle soluzioni migliori.
- **Mutazione:** vengono introdotte piccole, casuali variazioni nelle nuove soluzioni. La mutazione è fondamentale per mantenere la diversità genetica e per evitare che l'algoritmo rimanga bloccato in un'unica soluzione.

Questi operatori permettono, agli algoritmi genetici, di esplorare un vasto spazio di soluzioni (esplorazione globale) ed, allo stesso tempo, di migliorare le soluzioni promettenti (esplorazione locale), migliorandole sempre di più nel corso delle generazioni.

### Le fasi del GA

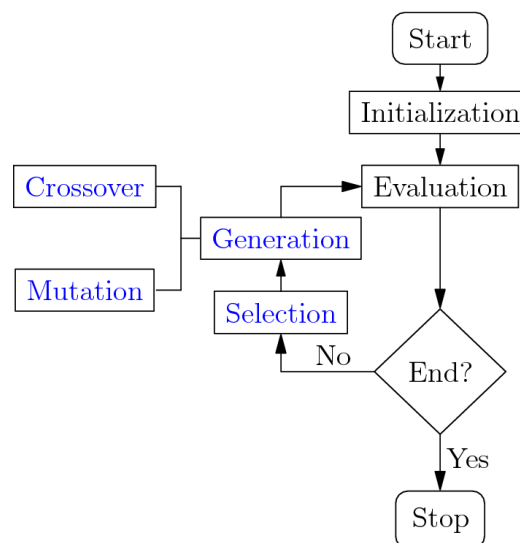


Figura 7.8: Le fasi del processo del Genetic Algorithm.

- **Inizializzazione:**
  - Definire lo spazio di ricerca degli iperparametri.
  - Generare una popolazione iniziale di  $N$  individui casuali.

- Impostare i parametri: dimensione popolazione ( $N$ ), generazioni massime ( $G_{max}$ ), tassi di crossover ( $p_c$ ) e mutazione ( $p_m$ ).
- **Ciclo evolutivo** (per ogni generazione  $g = 1, 2, \dots, G_{max}$ ):
  - **Valutazione:** allenare il modello per ogni individuo e calcolare il fitness  $f(x_i)$ .
  - **Selezione:** scegliere i genitori migliori.
  - **Crossover:** combinare coppie di genitori con probabilità  $p_c$  per generare figli.
  - **Mutazione:** introdurre variazioni casuali nei figli con probabilità  $p_m$ .
  - **Sostituzione:** formare la nuova popolazione (strategia elitista o generazionale).
- **Terminazione:**
  - Fermarsi quando: raggiunto  $G_{max}$ , nessun miglioramento per  $k$  generazioni, o fitness target raggiunto.
  - Restituire l'individuo con il miglior fitness come soluzione ottimale.

### 7.6.2 Vantaggi

- **Esplorazione robusta:** grazie a crossover e mutazioni, i GA sono molto adatti a spazi complessi e non lineari, e trovano ottimi globali ove altri metodi fallirebbero.
- **Gestione di spazi misti:** funzionano bene con iperparametri discreti, continui e categorici.
- **Parallelizzazione:** le valutazioni della fitness possono essere distribuite in parallelo.

### 7.6.3 Limiti

- **Costo computazionale elevato:** richiedono molte generazioni e valutazioni, quindi la convergenza può essere lenta e costosa.
- **Dipendenza da iperparametri evolutivi:** i risultati sono sensibili alla dimensione della popolazione, ai tassi di mutazione/crossover ed altri iperparametri di GA, che vanno ottimizzati. Ciò porta maggior complessità.
- **Nessuna garanzia di convergenza:** potrebbero bloccarsi in minimi sub-ottimali se la diversità non viene mantenuta.

## 7.7 Confronto pratico

### 7.7.1 Criteri per la scelta

- **Efficienza:** quante valutazioni servono per trovare una buona soluzione;
- **Scalabilità:** comportamento al crescere della dimensionalità degli iperparametri;
- **Supporto per vari tipi di variabili:** continue, discrete, categoriche;
- **Parallelizzazione:** facilità di esecuzione su cluster/GPU;
- **Robustezza:** capacità di gestire rumore e funzioni complesse;
- **Risorse computazionali richieste:** overhead e costi aggiuntivi dell'algoritmo.

### 7.7.2 Tabella riassuntiva comparativa

Metodo	Efficienza	Scalabilità	Parallelizz.	Complessità	Spazi misti
Grid search	Bassa	Pessima	Eccellente	Bassa	Bassa
Random search	Media	Buona	Eccellente	Bassa	Bassa
Bayesian opt.	Alta	Limitata	Moderata	Alta	Media
Genetic alg.	Variabile*	Discreta	Eccellente	Media-Alta	Alta

\* può essere molto efficace su spazi complessi/discreti, ma richiede molte valutazioni rispetto a BO in spazi piccoli; meno efficiente se valutazioni sono molto care.

### 7.7.3 Matrice decisionale per la scelta del metodo

Condizioni	Metodo consigliato	Alternativa	Da evitare
Spazi piccoli (2-3 param.)	Grid Search	Random Search	Bayesian Opt.
Spazi medi/ampi (> 4 param.)	Random Search	Bayesian Opt.	Grid Search
Valutazioni costose, budget limitato	Bayesian Opt.	Random Search	Grid Search
Spazi molto complessi	Genetic Algorithms	Bayesian Opt.	Grid Search
Multi-obiettivo	Genetic Algorithms	-	Grid/Random Search
Risorse limitate	Random Search	Bayesian Opt.	Grid Search

### 7.7.4 Scelta per i casi studio: Random search

#### Motivazioni della scelta

- **Efficienza su spazi ampi:** questo metodo performa meglio di Grid search quando alcuni iperparametri hanno impatto marginale;
- **Scalabilità:** non soffre dell'aumento esponenziale della complessità, mantenendo buone performance anche con molti iperparametri;
- **Semplicità implementativa:** triviale da implementare e parallelizzare, riducendo la complessità del codice e facilitando l'esecuzione su cluster/GPU;

- **Flessibilità operativa:** permette interruzione anticipata (early stopping) e riutilizzo semplice dei risultati per analisi successive;
- **Costo-beneficio ottimale:** bilancia efficienza esplorativa e semplicità computazionale, ideale per il contesto di questa ricerca.

### 7.7.5 Ottimizzazione del numero di iterazioni nel Random search

Per massimizzare l'efficienza del Random search, è fondamentale stimare il numero minimo di iterazioni necessarie per garantire un'alta probabilità di trovare configurazioni quasi-ottimali. Questo approccio consente di bilanciare efficacia esplorativa e costo computazionale.

#### Derivazione teorica della formula

La stima del numero di iterazioni richieste si basa sulla teoria della probabilità discreta. Data la seguente situazione:

- $M$ : numero totale di configurazioni possibili nello spazio di ricerca
- $k$ : numero delle migliori configurazioni che consideriamo "quasi-ottimali"
- $P$ : probabilità desiderata di includere almeno una configurazione top- $k$
- $n$ : numero di iterazioni del Random search

Il processo di derivazione segue questi passaggi logici:

#### 1. Probabilità di fallimento in un singolo tentativo

Se ci sono  $M$  configurazioni totali e  $k$  di esse sono "quasi-ottimali", la probabilità di non selezionare una configurazione top- $k$  in un singolo campionamento casuale è:

$$P(\text{fallimento singolo}) = 1 - \frac{k}{M}$$



## 2. Probabilità di fallimento in $n$ tentativi indipendenti

$$P(\text{fallimento totale}) = \left(1 - \frac{k}{M}\right)^n$$

## 3. Probabilità di successo: trovare almeno una configurazione top- $k$ in $n$ tentativi

$$P(\text{successo}) = 1 - \left(1 - \frac{k}{M}\right)^n$$

## 4. Ricavare $n$ dalla formula

$$1 - \left(1 - \frac{k}{M}\right)^n = P \implies n = \frac{\ln(1 - P)}{\ln\left(1 - \frac{k}{M}\right)}$$

## 5. Approssimazione per

$$k \ll M$$

Poiché

$$\ln(1 - x) \approx -x$$

per  $x$  piccolo:

$$n \approx -\frac{\ln(1 - P)}{k/M}$$

## Vantaggi dell'approccio probabilistico

- **Efficienza computazionale:** riduzione drastica del numero di valutazioni necessarie
- **Controllo statistico:** garanzia probabilistica di trovare soluzioni quasi-ottimali
- **Flessibilità:** possibilità di modulare il trade-off tra accuratezza ( $P$ ) e costo computazionale ( $n$ )
- **Scalabilità:** il metodo si adatta automaticamente alla dimensione dello spazio di ricerca

## Capitolo 8

# Studio di ablazione e Pruning post-training

Gli studi di ablazione rappresentano una metodologia fondamentale nell'analisi e nell'ottimizzazione dei modelli di machine e deep learning, che consiste nella rimozione temporanea di una componente del modello, come un layer o gruppo di parametri, per osservare l'impatto sulle prestazioni. Si tratta di un esperimento diagnostico che aiuta a comprendere quali elementi contribuiscono maggiormente alla capacità predittiva del modello. In questo contesto, il pruning post-training viene utilizzato come una tecnica complementare che permette non solo di comprendere l'importanza delle componenti del modello, ma anche di ottenere versioni più efficienti senza compromettere significativamente le prestazioni.

### 8.1 Fondamenti teorici degli studi di ablazione

#### 8.1.1 Definizione

Per formalizzare matematicamente il concetto di ablazione, consideriamo un modello  $f_\theta$  dove  $\theta$  rappresenta l'insieme completo dei suoi parametri, e  $L(\theta)$  indica la funzione di perdita del modello. Dato un sottoinsieme

specifico di parametri  $S$ , la variazione di performance dovuta all'ablazione può essere quantificata come:

$$\Delta_S = L(\theta_{-S}) - L(\theta)$$

In questa formulazione,  $\theta_{-S}$  rappresenta il modello con la componente  $S$  rimossa. Un valore elevato di  $\Delta_S$  indica che la componente  $S$  fornisce un contributo importante alle prestazioni del modello. Questa definizione matematica fornisce una base quantitativa per valutare l'importanza relativa delle diverse componenti del modello.

### 8.1.2 Benefici

- **Identificazione delle componenti critiche:** questo processo aiuta a comprendere la sensibilità del modello a specifiche parti della sua architettura, rivelando potenziali vulnerabilità o punti di forza.
- **Guida per la semplificazione:** i risultati degli studi di ablazione guidano lo sviluppo di strategie di semplificazione del modello, fornendo una base empirica per le decisioni di pruning.
- **Interpretazione del comportamento del modello:** l'analisi di come la rimozione di una componente specifica influenzi le previsioni, facilita l'interpretazione del modello, contribuendo alla comprensione dei meccanismi interni che portano alle predizioni finali.

## 8.2 Pruning post-training

### 8.2.1 Definizione

Il pruning post-training può essere considerato come l'applicazione di un operatore  $\mathcal{P}$  ad un modello già addestrato, che elimina parametri secondo criteri specifici (come l'ampiezza del parametro, l'importanza stimata, o il contributo marginale) per avere un modello più leggero.

La scelta del criterio di pruning influenza significativamente sia l'efficacia

della compressione sia il mantenimento delle prestazioni. I criteri più comuni includono la magnitudine dei parametri, misure di sensibilità basate sui gradienti, e metriche di importanza derivate dall'analisi della struttura del modello.

### 8.2.2 Benefici

- **Riduzione della memoria e del tempo di inferenza:** il pruning riduce la memoria richiesta e il tempo di inferenza del modello, aspetti critici per il deployment in ambienti con risorse limitate.
- **Mantenimento delle prestazioni:** l'obiettivo del pruning è bilanciare l'efficienza e l'accuratezza del modello, mantenendo le prestazioni entro una tolleranza accettabile.
- **Aumento dell'interpretabilità:** riducendo il numero di elementi attivi, il pruning facilita l'analisi e la comprensione del contributo delle parti rimanenti.

### 8.2.3 Trade-off bias-varianza

La riduzione della complessità del modello, attraverso l'eliminazione di parametri, tende a diminuire la varianza, riducendo il rischio di overfitting sui dati di training. Tuttavia, questa semplificazione può introdurre bias, limitando la capacità del modello di catturare pattern complessi nei dati. L'obiettivo pratico del pruning consiste nel trovare il punto ottimale dove la riduzione della varianza compensa l'aumento del bias, mantenendo prestazioni globalmente superiori. Questo equilibrio è altamente dipendente dalla natura dei dati, dalla complessità del task, e dalle caratteristiche specifiche dell'architettura del modello.

## 8.3 Pruning L1 post-training per MLP e KAN

### 8.3.1 Definizione

Dato un insieme di parametri del modello  $\Theta = \{\theta_1, \theta_2, \dots, \theta_N\}$ , la procedura di L1 pruning opera applicando una trasformazione a ciascun parametro  $\theta_i$ , che è definita da una soglia di potatura  $\lambda$ , che viene tipicamente determinata a livello globale per il modello o a livello locale per ciascun strato.

La formula è:

$$\theta_i^{\text{pruned}} = \begin{cases} 0 & \text{se } |\theta_i| \leq \lambda \\ \theta_i & \text{se } |\theta_i| > \lambda \end{cases}$$

In questa formulazione, se la magnitudine assoluta ( $|\theta_i|$ ) di un parametro è inferiore o uguale alla soglia  $\lambda$ , esso viene permanentemente impostato a zero. Al contrario, se la sua magnitudine è superiore a  $\lambda$ , il parametro viene mantenuto invariato.

### 8.3.2 Considerazioni specifiche per le KAN

L'applicazione del L1 pruning alle KAN é abbastanza diverso rispetto a MLP. La motivazione principale risiede nella diversa natura dei parametri che compongono questi modelli. Nelle MLP, i parametri sono pesi e bias che operano su connessioni lineari, mentre nelle KAN sono coefficienti che definiscono funzioni univariate (tipicamente B-spline) su ogni arco della rete.

Il pruning su KAN può essere implementato in due modi. Il primo approccio prevede la rimozione di coefficienti locali delle spline, riducendo la risoluzione locale della rappresentazione funzionale mantenendo la struttura generale. Il secondo approccio elimina intere funzioni su specifici archi della rete, semplificando direttamente l'architettura della KAN.

La scelta tra questi approcci deve considerare l'impatto sulla continuità delle funzioni e sulla loro interpretabilità. La rimozione di coefficienti locali mantiene la struttura generale ma può creare delle discontinuità

o irregolarità indesiderate nella curva che la spline rappresenta, mentre l'eliminazione di intere funzioni preserva la continuità locale ma può alterare significativamente la capacità espressiva del modello.

## **8.4 Pruning per Ensemble: Rank-based pruning per Random forest**

### **8.4.1 Principio fondamentale**

Il rank-based pruning per Random forest si basa sul principio che non tutti gli alberi nell'ensemble contribuiscono equamente alle prestazioni finali. Alcuni alberi possono essere ridondanti o addirittura dannosi per la capacità di generalizzazione dell'ensemble, rendendo la loro rimozione vantaggiosa sia in termini di efficienza e prestazioni. L'approccio implementato definisce per ogni albero  $m$  un contributo stimato alle prestazioni, quantificato attraverso la variazione di errore  $\Delta L_m$  che si osserverebbe rimuovendo l'albero dall'ensemble. Gli alberi con contributo minore sono candidati prioritari per la rimozione durante il processo di pruning.

### **8.4.2 Criterio di ranking basato sulla feature importance**

Invece di valutare l'impatto della rimozione di ogni singolo albero sull'errore complessivo, il criterio di ranking si basa sulle feature importance di ogni singolo albero. Specificamente, l'importanza di un albero  $m$  viene calcolata come la somma delle feature importance che l'albero utilizza. La logica sottostante è che un albero che si basa su feature più discriminative, che riducono significativamente l'entropia o l'indice di Gini, è probabile che fornisca un contributo maggiore all'ensemble. Questo metodo offre un vantaggio computazionale significativo rispetto a un'analisi diretta dell'errore.

### 8.4.3 Procedura di selezione

La procedura di selezione implementa una strategia greedy che ordina gli alberi per importanza decrescente e seleziona i primi  $k$  alberi, dove  $k$  è determinato dal pruning ratio desiderato. Questa scelta greedy è giustificata dall'assunzione che l'utilità marginale degli alberi decresce monotonamente con il loro ranking. La validazione empirica di questa assunzione rappresenta un aspetto critico della metodologia, poiché la sub-modularità della funzione di utilità non è sempre garantita negli ensemble reali. L'implementazione, infatti, include procedure di validazione che verificano che la selezione greedy produca risultati coerenti.

## 8.5 Pruning per Ensemble: Cumulative pruning per XGBoost

### 8.5.1 Criterio di pruning cumulativo

Il pruning cumulativo implementato si basa su un principio di selezione basato sull'ordine: vengono mantenuti solo i primi  $n$  round di boosting, scartando i successivi. Il presupposto è che le prime iterazioni, che hanno l'obiettivo di ridurre al massimo l'errore iniziale, contribuiscano in modo più significativo e non siano ridondanti come gli alberi meno performanti che si trovano alla fine del processo di training. La percentuale di iterazioni da mantenere è controllata direttamente dal pruning ratio, che determina la frazione di modello da eliminare. In pratica, l'algoritmo mantiene le iterazioni da 1 a  $n$ , dove  $n$  è calcolato come  $(1 - \text{pruning ratio})$  moltiplicato per il numero totale di round di boosting.

### 8.5.2 Procedura di selezione

A differenza di Random forest, che può semplicemente rimuovere alberi dalla lista degli "estimators", XGBoost richiede un'operazione più delicata a causa della natura additiva delle sue predizioni.

Il processo tecnico consiste in:

1. **Calcolo del numero di iterazioni da mantenere:** si determina il numero di alberi da utilizzare per la predizione. Questo valore viene calcolato in base ad un pruning ratio definito. Nei problemi di classificazione multiclasse, ogni round di boosting aggiunge un albero per ogni classe.
2. **Predizione con iterazioni limitate:** invece di modificare la struttura del modello, si sfrutta una funzionalità di XGBoost che permette di specificare il numero di alberi da usare per la predizione. Il modello addestrato mantiene al suo interno tutti gli alberi, ma per calcolare il risultato finale si usa solo l'insieme limitato di alberi specificato.

Se l'obiettivo è creare un modello più leggero da salvare o esportare, è possibile potare il modello in modo permanente. Questo si ottiene limitando l'ensemble agli alberi selezionati e salvando il nuovo modello ridotto. Questo processo è utile per ridurre l'occupazione di memoria e rendere il modello più efficiente per la distribuzione in produzione, una volta che la migliore configurazione è stata identificata tramite lo studio di ablazione.



## **Capitolo 9**

### **Primo Caso Studio: Regressione su emissioni di automobili**

## **Capitolo 10**

### **Secondo Caso Studio: Classificazione di PM2.5**

## **Capitolo 11**

### **Terzo Caso Studio: Classificazione di età tramite immagini**

## **Capitolo 12**

### **Discussione comparativa sui Risultati dei casi studio**

## **Capitolo 13**

## **Conclusioni**

# Bibliografia

- [1] Popescu et al. , *Multilayer perceptron and neural networks*, 2009. [https://www.researchgate.net/publication/228340819\\_Multilayer\\_perceptron\\_and\\_neural\\_networks](https://www.researchgate.net/publication/228340819_Multilayer_perceptron_and_neural_networks)
- [2] Hornick et al. , *Multilayer feedforward networks are universal approximators*, 1988. [https://www.cs.cmu.edu/~epxing/Class/10715/reading/Kornick\\_et\\_al.pdf](https://www.cs.cmu.edu/~epxing/Class/10715/reading/Kornick_et_al.pdf)
- [3] J. Schmidhuber, *Deep learning in neural networks: An overview*, vol. 61, 2015.
- [4] G. Cybenko, *Approximation by superpositions of a sigmoidal function*, vol. 2, 1989. <https://doi.org/10.1007/BF02551274>
- [5] Leshno et al. , *Multilayer feedforward networks with a nonpolynomial activation function can approximate any functionn*, vol. 6, 1993. <https://www.sciencedirect.com/science/article/pii/S0893608005801315>
- [6] I. Goodfellow, Y. Bengio, A.Courville, *Deep Learning*, 2016.
- [7] Liu et al. , *Kan: Kolmogorov-arnold networks*, 2024. <https://arxiv.org/pdf/2404.19756>
- [8] A.N. Kolmogorov, *On the representation of continuous functions of several variables by superpositions of continuous functions of one variable and addition*, *Doklady Akademii Nauk SSSR*, vol. 114, no. 5, 1957.
- [9] V.I. Arnold, *On functions of three variables*, *Doklady Akademii Nauk SSSR*, vol. 141, no. 4, 1963.

- [10] A. Pinkus, *Approximation theory of the mlp model in neural networks*, *Acta Numerica*, vol. 8, 1999.
- [11] T. Poggio, F. Girosi, *Networks for approximation and learning*, *Proceedings of the IEEE*, vol. 78, no. 9, 1990.
- [12] A. Chaudhuri, *B-Splines*, 2021. <https://arxiv.org/pdf/2108.06617>
- [13] J. Henseler, *Back Propagation*, 1995. <https://link.springer.com/chapter/10.1007/BFb0027022>
- [14] D. Donoho, *High-Dimensional Data Analysis: The Curses and Blessings of Dimensionality*, 2000. [https://www.researchgate.net/publication/220049061\\_High-Dimensional\\_Data\\_Analysis\\_The\\_Curses\\_and\\_Blessings\\_of\\_Dimensionality](https://www.researchgate.net/publication/220049061_High-Dimensional_Data_Analysis_The_Curses_and_Blessings_of_Dimensionality).
- [15] R. French, *Catastrophic forgetting in connectionist networks*, 1999. [https://www.researchgate.net/publication/12977135\\_Catastrophic\\_forgetting\\_in\\_connectionist\\_networks](https://www.researchgate.net/publication/12977135_Catastrophic_forgetting_in_connectionist_networks)
- [16] L. Breiman, *Bagging Predictors*, vol. 24, 1996. <https://link.springer.com/article/10.1007/BF00058655>
- [17] L. Breiman, *Random Forests*, vol. 45, 2001. <https://link.springer.com/article/10.1023/A:1010933404324>
- [18] J.R. Quinlan, *C4.5: Programs for Machine Learning*, 1993. [https://scholar.google.com/scholar\\_lookup?&title=C4.5%20Programs%20for%20Machine%20Learning&publication\\_year=1992&author=Quinlan%2CJ.R.](https://scholar.google.com/scholar_lookup?&title=C4.5%20Programs%20for%20Machine%20Learning&publication_year=1992&author=Quinlan%2CJ.R.)
- [19] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, 2009. <https://link.springer.com/book/10.1007/978-0-387-84858-7>
- [20] J. H. Friedman, *Greedy function approximation: A gradient boosting machine*, 2001. <https://jerryfriedman.su.domains/ftp/trebst.pdf>

- [21] T. Chen, C. Guestrin, *XGBoost: A Scalable Tree Boosting System*, 2016.  
<https://arxiv.org/pdf/1603.02754>
- [22] A. E. Hoerl, R. W. Kennard, *Ridge regression: Biased estimation for nonorthogonal problems*, 1970. <https://homepages.math.uic.edu/~lreyzin/papers/ridge.pdf>
- [23] R. Tibshirani, *Regression shrinkage and selection via the lasso*, 1996.  
[https://webdoc.agsci.colostate.edu/koontz/arec-econ535/papers/Tibshirani%20\(JRSS-B%201996\).pdf](https://webdoc.agsci.colostate.edu/koontz/arec-econ535/papers/Tibshirani%20(JRSS-B%201996).pdf)
- [24] H. Zou, T. Hastie, *Regularization and variable selection via the elastic net*, 2005. <https://academic.oup.com/jrsssb/article-abstract/67/2/301/7109482>
- [25] N. Srivastava et al. , *Dropout: A simple way to prevent neural networks from overfitting*, 2014. <https://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf>