



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

**Valutazione Metodologica ed Applicativa di KAN,
MLP, Random Forest e XGBoost con Tecniche di
Ottimizzazione su differenti casi di studio**

Relatore:
Prof. Damiana Lazzaro

Presentata da:
Martin Tomassi

Sessione Unica
Anno Accademico 2024/2025

Indice

1	Multi-Layer Perceptron (MLP)	12
1.1	Introduzione	12
1.2	Fondamenti matematici	13
1.2.1	Enunciato formale	13
1.2.2	Teorema di approssimazione universale	13
1.2.3	Significato di $\sup_{x \in K}$	14
1.2.4	Ipotesi e Precisazioni	15
1.3	Struttura delle MLP	16
1.3.1	Strati: input, nascosti, output	16
1.3.2	Funzioni di attivazione comuni	17
1.4	Procedura di forward pass	24
1.4.1	Calcolo delle attivazioni	24
1.4.2	Propagazione ed output	25
1.5	Algoritmo di backpropagation	25
1.5.1	Derivazione del gradiente	25
1.5.2	Aggiornamento dei pesi	27
1.5.3	Tecniche di regolarizzazione	27
1.6	Vantaggi e limiti	28
1.6.1	Flessibilità e capacità di generalizzazione	28
1.6.2	Problemi di vanishing/exploding gradient	29
1.6.3	Efficienza computazionale	29
2	Kolmogorov–Arnold Networks (KAN)	31
2.1	Introduzione	31

2.2	Fondamenti matematici	32
2.2.1	Enunciato del teorema di Kolmogorov–Arnold	32
2.2.2	Sulla natura "non costruttiva" delle dimostrazioni . .	33
2.3	Architettura delle KAN	34
2.3.1	Funzioni univariate parametriche	36
2.3.2	B-spline	37
2.3.3	Scaling laws e Curse of dimensionality	39
2.4	Funzionamento operativo	40
2.4.1	Calcolo del mapping input–hidden–output	40
2.4.2	Processo di training e Calcolo dei pesi	41
2.5	Confronto con MLP tradizionali	42
2.5.1	Architettura a confronto	42
2.5.2	Complessità computazionale	43
2.5.3	Interpretabilità e flessibilità locale	43
2.5.4	Precisione controllabile tramite grid extension	45
2.5.5	Dipendenza dalla struttura compositiva	45
2.5.6	Irregolarità nella rappresentazione di Kolmogorov .	46
2.5.7	Overhead computazionale e scelta della struttura . .	46
2.5.8	Sensibilità al rumore e necessità di regolarizzazione .	46
3	Random forest (RF)	47
3.1	Introduzione	47
3.2	Concetti fondamentali	48
3.2.1	Alberi di decisione: Criteri di splitting (Gini, Entropia, Gain ratio)	48
3.2.2	Ensemble learning e Bagging	50
3.3	Architettura e Costruzione	53
3.3.1	Bootstrapping e Feature bagging	53
3.3.2	Aggregazione delle predizioni	54
3.4	Vantaggi	55
3.5	Svantaggi	56

4	eXtreme Gradient Boosting (XGBoost)	57
4.1	Introduzione	57
4.2	Introduzione al Gradient boosting	58
4.2.1	Principi iterativi e Weak learners	59
4.2.2	Funzione di perdita e Discesa del gradiente	60
4.3	Miglioramenti di XGBoost rispetto al Gradient boosting tradizionale	61
4.3.1	Regolarizzazione e Tree pruning	61
4.3.2	Gestione dei valori mancanti	63
4.3.3	Ottimizzazioni (Parallelismo, Cache-awareness)	64
4.4	Parametri chiave e Tuning	65
4.5	Vantaggi	66
4.6	Svantaggi	67
5	Convolutional Neural Networks (CNN)	69
5.1	Introduzione	69
5.2	Principi fondamentali delle CNN	70
5.2.1	Convoluzione: kernel, stride, padding	70
5.2.2	Feature maps e profondità dei canali	71
5.3	Pooling e normalizzazione	72
5.3.1	Max pooling vs average pooling	72
5.3.2	Global pooling	72
5.3.3	Batch, Layer e Group normalization	72
5.4	Data augmentation: rotazioni, zoom, colour jitter	73
5.5	Funzionamento delle CNN	74
5.5.1	Forward pass	74
5.5.2	Strati di convoluzione	74
5.5.3	Funzioni di attivazione	75
5.5.4	Strati di pooling	75
5.5.5	Gerarchia delle caratteristiche	75
5.5.6	Strati fully-connected	76
5.5.7	Flusso informativo e Trasformazioni progressive	76
5.6	Vantaggi	77

5.7	Svantaggi	77
6	Ottimizzazione degli iperparametri	79
6.1	Introduzione	79
6.2	Cross-Validation (CV)	80
6.2.1	Time Series Cross-Validation (TSCV)	81
6.3	Nested Cross-Validation (NCV)	82
6.4	Grid search (GS)	84
6.4.1	Spiegazione dell'algoritmo	84
6.4.2	Vantaggi	85
6.4.3	Limiti	85
6.5	Random Search (RS)	86
6.5.1	Spiegazione dell'algoritmo	86
6.5.2	Vantaggi	87
6.5.3	Limiti	87
6.6	Bayesian optimization (BO)	88
6.6.1	Spiegazione dell'algoritmo	88
6.6.2	Vantaggi	89
6.6.3	Limiti	90
6.7	Genetic algorithm (GA)	91
6.7.1	Spiegazione dell'algoritmo	91
6.7.2	Vantaggi	93
6.7.3	Limiti	93
6.8	Confronto pratico	94
6.8.1	Criteri per la scelta	94
6.8.2	Tabella riassuntiva comparativa	94
6.8.3	Matrice decisionale per la scelta del metodo	95
6.8.4	Scelta per i casi studio: Random search	95
6.8.5	Ottimizzazione del numero di iterazioni nel Random search	95
7	Studio di ablazione e Pruning post-training	98
7.1	Introduzione	98

7.2	Studi di ablazione	99
7.2.1	Definizione	99
7.2.2	Benefici	99
7.3	Pruning post-training	100
7.3.1	Definizione	100
7.3.2	Benefici	100
7.3.3	Trade-off bias-varianza	101
7.4	Pruning L1 post-training per MLP e KAN	101
7.4.1	Definizione	101
7.4.2	Considerazioni specifiche per le KAN	101
7.5	Pruning per Ensemble: Rank-based pruning per Random forest	102
7.5.1	Principio fondamentale	102
7.5.2	Criterio di ranking basato sulla feature importance .	103
7.5.3	Procedura di selezione	103
7.6	Pruning per Ensemble: Cumulative pruning per XGBoost .	103
7.6.1	Criterio di pruning cumulativo	103
7.6.2	Procedura di selezione	104

8 Metodologie e Procedure comuni per la Verifica sperimentale dei

Casi di studio	105
8.1	Introduzione
8.2	Progettazione dei casi di studio
8.2.1	Tecnologie e librerie
8.2.2	Ambienti di sviluppo e infrastruttura
8.2.3	Linguaggi, scripting e automazione del workflow . .
8.2.4	Script di sottomissione (Cluster GPU)
8.2.5	Scelte architetturali ed iperparametri
8.3	Addestramento dei modelli
8.3.1	Pipeline di preprocessing
8.3.2	Suddivisione dei dati e validazione
8.3.3	Metriche e intervalli di confidenza
8.4	Valutazione dei modelli

8.4.1	Fase 1: Analisi qualitativa e quantitativa integrata . .	113
8.4.2	Fase 2: Selezione del miglior modello	114
8.4.3	Fase 3: Conclusione finale	120
8.5	Studio di ablazione	120
8.5.1	L1 pruning su MLP e KAN	120
8.5.2	Ensemble pruning su Random Forest e XGBoost . . .	126
8.5.3	Confronto complessivo	131
9	Primo Caso Studio: Regressione su emissioni di automobili	135
9.1	Introduzione	135
9.2	Data preparation	136
9.2.1	Riassunto delle principali trasformazioni	136
9.3	Addestramento dei modelli	136
9.3.1	Strategia di training comune e griglie di iperparametri	137
9.3.2	Scelte architetturali finali	144
9.4	Valutazione dei modelli	146
9.4.1	Analisi dei risultati sperimentali	146
9.4.2	Selezione del miglior modello	148
9.4.3	Conclusioni	148
9.5	Studio di ablazione	149
9.5.1	Ablation study: L1 pruning su MLP e KAN	149
9.5.2	Ablation study: ensemble pruning su Random Forest e XGBoost	151
9.5.3	Ablation study — Confronto complessivo (Neural Networks vs Ensemble)	152
10	Secondo Caso Studio: Classificazione di PM2.5	155
10.1	Introduzione	155
10.2	Data preparation	156
10.2.1	Fonti e descrizione generale del dataset	156
10.2.2	Caricamento dati e organizzazione iniziale	156
10.2.3	Indicizzazione temporale	157
10.2.4	Riduzione e unificazione di feature ridondanti	157

10.2.5	Verifica e gestione dei valori mancanti	162
10.2.6	Analisi esplorativa e selezione delle feature rilevanti .	164
10.2.7	Ricampionamento ed aggregazione a livello statale .	169
10.2.8	Rilevamento e rimozione degli outlier	170
10.2.9	Feature engineering ed arricchimento temporale . . .	172
10.2.10	Creazione di lag-features e categorizzazione di PM2.5	176
10.3	Addestramento dei modelli	177
10.3.1	Strategia di training comune e griglie di iperparametri	178
10.3.2	Scelte architetturali finali	186
10.4	Valutazione dei modelli	188
10.4.1	Analisi dei risultati sperimentali	192
10.4.2	Selezione del miglior modello	193
10.4.3	Conclusioni	194
10.5	Studio di ablazione	195
10.5.1	Ablation study: L1 pruning su MLP e KAN	195
10.5.2	Ablation study: ensemble pruning su Random Forest e XGBoost	197
10.5.3	Ablation study — Confronto complessivo (Neural Networks vs Ensemble)	199
11	Terzo Caso Studio: Classificazione di fasce d'età tramite immagini	201
11.1	Introduzione	201
11.2	Data preparation	202
11.2.1	Pre-processing e costruzione delle etichette	202
11.2.2	Data exploration	205
11.3	Addestramento dei modelli	219
11.3.1	Strategia di training comune e criteri di arresto	219
11.3.2	Scelte architetturali finali	228
11.4	Valutazione dei modelli	229
11.4.1	Analisi dei risultati sperimentali	231
11.4.2	Selezione del miglior modello	231
11.4.3	Conclusioni	232

Introduzione

Contesto della Tesi

La presente tesi si propone di analizzare, confrontare e valutare diverse architetture di Machine e Deep Learning: Random Forest, XGBoost, Multi-Layer Perceptron (MLP) e Kolmogorov-Arnold Networks (KAN). Questi modelli rappresentano paradigmi differenti nel panorama dell'apprendimento automatico, coprendo sia le reti neurali tradizionali, che quelle di recente introduzione, che i metodi ensemble. L'obiettivo 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 classificazione di immagini mediante CNN abbinate a MLP e KAN. Oltre allo studio comparativo dei modelli, la tesi include un'ampia indagine sui metodi di Ottimizzazione degli iperparametri ed uno studio di ablazione post-training, che valutano il compromesso tra la complessità del modello e le prestazioni.

Obiettivo della Tesi

L'obiettivo primario è quello di condurre una valutazione metodologica ed applicativa completa dei modelli selezionati, con un focus specifico sulla loro efficacia in scenari del mondo reale. Per la valutazione empirica sono stati scelti tre casi di studio diversificati per tipologia di problema

(regressione e classificazione) e natura dei dati (tabellari, serie storiche e immagini). Le metodologie adottate, come la *Nested Cross-Validation* e la *Time Series Cross-Validation*, sono state impiegate per garantire stime robuste e non distorte della capacità di generalizzazione dei modelli.

Un ulteriore obiettivo è stato quello di investigare l'efficacia di diverse strategie di ottimizzazione, tra cui *Grid Search*, *Random Search*, *Bayesian Optimization* e *Genetic Algorithms*, selezionando il *Random Search* per la sua efficienza e scalabilità nei casi di studio. A completamento, sono stati condotti studi di ablazione post-addestramento per analizzare l'impatto del *pruning* sulla performance dei modelli. In particolare, è stato applicato il *pruning* L1 per le reti neurali (KAN e MLP) ed il *pruning ensemble* (basato sul ranking per Random Forest e cumulativo per XGBoost), al fine di misurare il compromesso tra la riduzione dei parametri ed il mantenimento della qualità predittiva.

Struttura della Tesi

La tesi è organizzata in capitoli che si susseguono in modo logico, partendo dalle basi teoriche dei modelli fino all'analisi dei risultati sperimentali.

- **Multi-Layer Perceptron (MLP):** Questo capitolo introduce il Multi-Layer Perceptron, descrivendone l'architettura, i fondamenti matematici e l'algoritmo di backpropagation. Vengono analizzate le funzioni di attivazione e le tecniche di regolarizzazione, con una discussione finale sui vantaggi e sui limiti di questa architettura.
- **Kolmogorov-Arnold Networks (KAN):** Il capitolo introduce le KAN, un'architettura neurale che si ispira al teorema di approssimazione di Kolmogorov-Arnold. Vengono esaminati i fondamenti matematici, l'architettura, il funzionamento operativo ed un confronto dettagliato con le MLP tradizionali.

- **Random Forest (RF):** Questo capitolo descrive il Random Forest, un modello ensemble basato su alberi di decisione. Vengono trattati i concetti fondamentali come il bagging ed il feature bagging, che ne garantiscono la robustezza e la capacità di generalizzazione, e vengono discussi vantaggi e svantaggi.
- **eXtreme Gradient Boosting (XGBoost):** Il capitolo si concentra su XGBoost, un'implementazione ottimizzata del gradient boosting. Vengono analizzati i principi iterativi, i miglioramenti rispetto al boosting tradizionale e l'importanza dei suoi iperparametri per il tuning del modello.
- **Convolutional Neural Networks (CNN):** Questo capitolo introduce le CNN, un'architettura fondamentale per l'analisi di immagini. Vengono descritti i principi di convoluzione, pooling ed il suo funzionamento, con una panoramica dei vantaggi e dei limiti.
- **Ottimizzazione degli Iperparametri:** Il capitolo offre una panoramica delle metodologie di ottimizzazione degli iperparametri, tra cui *Grid Search*, *Random Search*, *Bayesian Optimization* e *Genetic Algorithms*, con un focus sulla scelta del *Random Search* per gli studi di caso.
- **Studio di Ablazione e Pruning post-training:** Vengono introdotte le metodologie di ablazione e di pruning post-training, analizzando in dettaglio le tecniche di pruning L1 per le reti neurali e quelle specifiche per gli ensemble.
- **Metodologie e Procedure Comuni:** Questo capitolo descrive le scelte condivise che costituiscono la base per la verifica sperimentale dei casi di studio, inclusi gli ambienti di sviluppo, le librerie software, le metriche di valutazione, le procedure di validazione e di ablazione.
- **Primo, Secondo e Terzo Caso studio:** I capitoli finali presentano i tre casi di studio applicativi (regressione su emissioni di automobili, classificazione di PM2.5 e classificazione di fasce d'età), con una trattazione

dettagliata delle fasi di preparazione dei dati, training, valutazione e pruning per ciascuno.

Tecnologie e Strumenti

Il workflow sperimentale e di analisi è stato interamente sviluppato in Python, facendo uso di un insieme di librerie consolidate nel settore del machine learning e dell'analisi dati. Per la manipolazione e l'analisi dei dati sono state utilizzate le librerie Pandas e NumPy. La visualizzazione è stata realizzata con Matplotlib e Seaborn. Le reti neurali sono state implementate in PyTorch, sfruttando anche la libreria pykan per l'implementazione delle Kolmogorov-Arnold Networks. Gli approcci ensemble sono stati gestiti tramite le librerie scikit-learn e XGBoost. Il framework di lavoro ha previsto l'utilizzo di Jupyter notebooks per la prototipazione e il debug, mentre gli esperimenti su larga scala sono stati eseguiti su un cluster HPC dell'Università di Bologna, con allocazione di GPU tramite SLURM. La gestione dei metadati, il tracciamento degli esperimenti e l'automazione del workflow sono stati gestiti con moduli standard come json, os e logging.

Capitolo 1

Multi-Layer Perceptron (MLP)

1.1 Introduzione

Questo capitolo presenta una descrizione del Multi-Layer Perceptron (MLP), un'architettura fondamentale delle reti neurali. L'obiettivo è esplorare la sua struttura e le sue capacità di modellare relazioni complesse, a partire dai principi matematici. Verranno analizzate le sue componenti principali, tra cui la distinzione tra gli strati e le funzioni di attivazione, che sono cruciali per l'apprendimento di relazioni non lineari. Il testo spiegherà poi il meccanismo attraverso cui la rete impara, descrivendo i passaggi di calcolo che avvengono durante il forward pass e l'algoritmo di backpropagation che aggiorna i parametri del modello. Verranno inoltre affrontate le tecniche di regolarizzazione, indispensabili per evitare che la rete memorizzi i dati di addestramento invece di imparare a generalizzare. Infine, il capitolo riassume i vantaggi ed i limiti delle MLP, ponendo l'accento sulla loro flessibilità e sui problemi legati alla profondità della rete, come il vanishing/exploding gradient. [1, 2, 4, 5, 10, 22, 3]

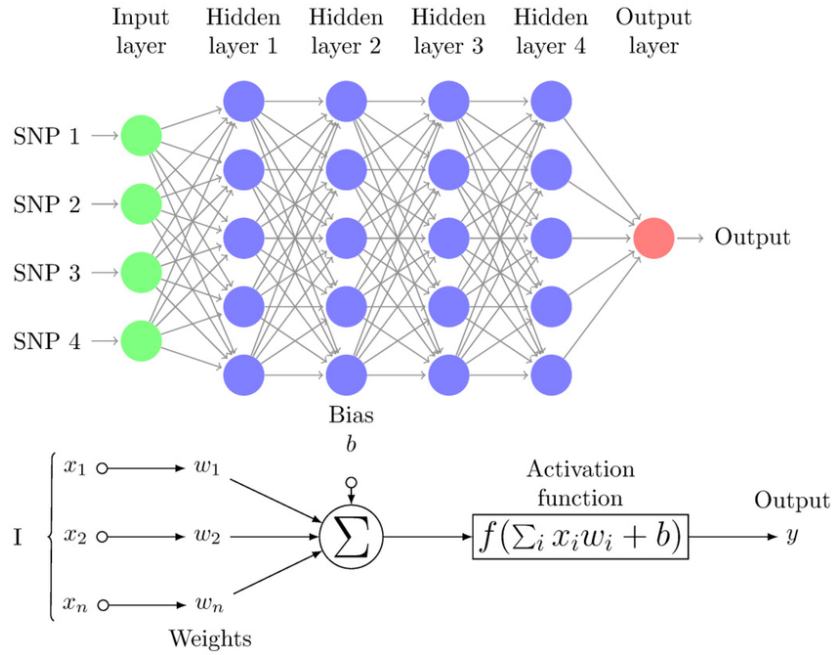


Figura 1.1: Architettura della rete neurale Multi-Layer Perceptron.

1.2 Fondamenti matematici

1.2.1 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:

- (A₁) σ 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);
- (A₂) σ è continua e non polinomiale (Leshno et al.).

Allora vale il seguente risultato di approssimazione universale.

1.2.2 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 σ (cioè una rete a singolo strato nascosto) che approssima f uniformemente su K con errore massimo minore di ε . Formalmente, la chiusura (nell' $\|\cdot\|_\infty$) dello spazio generato dalle funzioni elementari coincide con l'intero $C(K)$.

1.2.3 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 $[0, 1]$.

1.2.4 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 σ :** la validità del risultato dipende dalle proprietà di σ . Due formulazioni tipiche sono:
 - **Sigmoide limitata e continua (Cybenko):** dove σ 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 ε . Non fornisce però una procedura esplicita per la ricerca dei parametri ed un bound quantitativo generale che esprima N in funzione di ε 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.

1.3 Struttura delle MLP

1.3.1 Strati: input, nascosti, output

Le Multi-layer Perceptron (MLP) sono una tipologia di reti neurali feed-forward, costituite da più strati (layer) di neuroni: uno strato di input, che riceve i dati iniziali; uno o più strati nascosti; ed 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 ed 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 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 calcola una somma pesata degli input e di un termine di bias, per poi applicare una funzione di attivazione. L'output di un neurone i nel layer l è dato da:

$$z_i^{(l)} = \sum_{j=1}^{N_{l-1}} w_{ij}^{(l)} a_j^{(l-1)} + b_i^{(l)}$$

$$a_i^{(l)} = \sigma(z_i^{(l)})$$

Dove $z_i^{(l)}$ è la somma pesata, $w_{ij}^{(l)}$ è il peso della connessione, $a_j^{(l-1)}$ è l'output del neurone precedente, $b_i^{(l)}$ è il bias, e σ è la funzione di attivazione non lineare.

1.3.2 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 la sua **differenziabilità**, che è fondamentale per la backpropagation; la **boundedness dell'output e zero-centering**, cioè se l'output è centrato attorno a 0; la **saturazione**, che può causare il *vanishing gradient*; la **sparsità** ed il **costo computazionale**.

1. Sigmoide

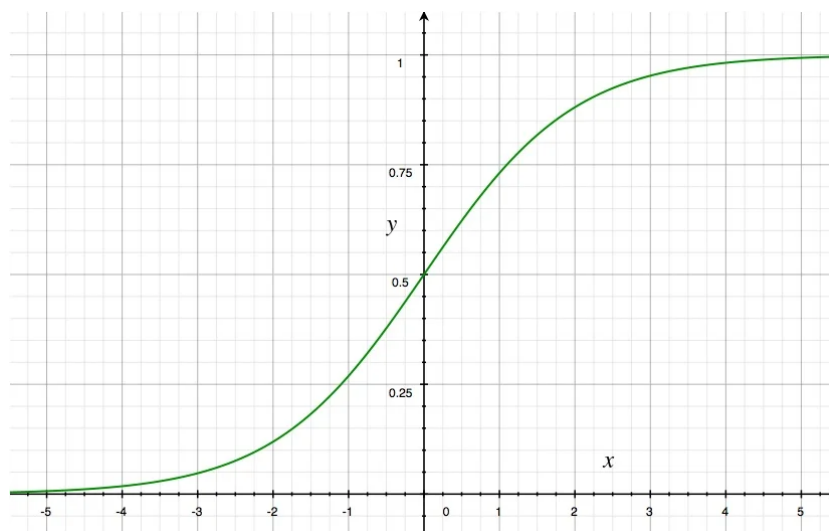


Figura 1.2: Grafico della funzione di attivazione Sigmoide.

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

Questa funzione ha un output che si colloca nell'intervallo $(0, 1)$ ed una caratteristica forma a "S", rendendola utile per rappresentare la probabilità (quindi un output normalizzato); la sua derivata è semplice da calcolare. Tuttavia, la sua principale debolezza è la saturazione per $x \rightarrow \pm\infty$, dove il gradiente tende a zero. Questo fenomeno porta al problema del vanishing gradient nei layer profondi. È tipicamente utilizzata nello strato di output per la classificazione binaria, in combinazione con la binary cross-entropy.

2. Tangente iperbolica (tanh)

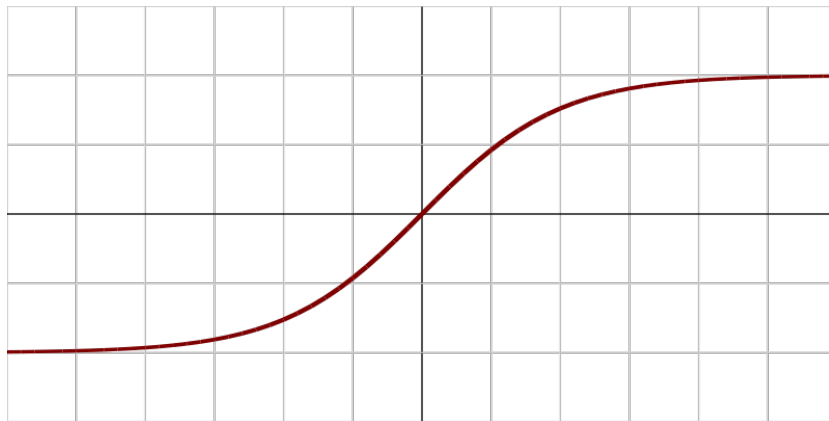


Figura 1.3: Grafico della funzione di attivazione Tangente iperbolica.

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

La funzione \tanh ha un output compreso tra $(-1, 1)$ ed è centrata in 0, il che, quando i dati sono normalizzati, porta ad una convergenza migliore rispetto alla sigmoide. Nonostante questo vantaggio, rimane una funzione saturante per valori estremi, e di conseguenza può soffrire ancora del problema del vanishing gradient. Il suo uso tipico è nei layer nascosti di reti poco profonde o quando è desiderabile un output centrato.

3. Rectified Linear Unit ReLU (ReLU)

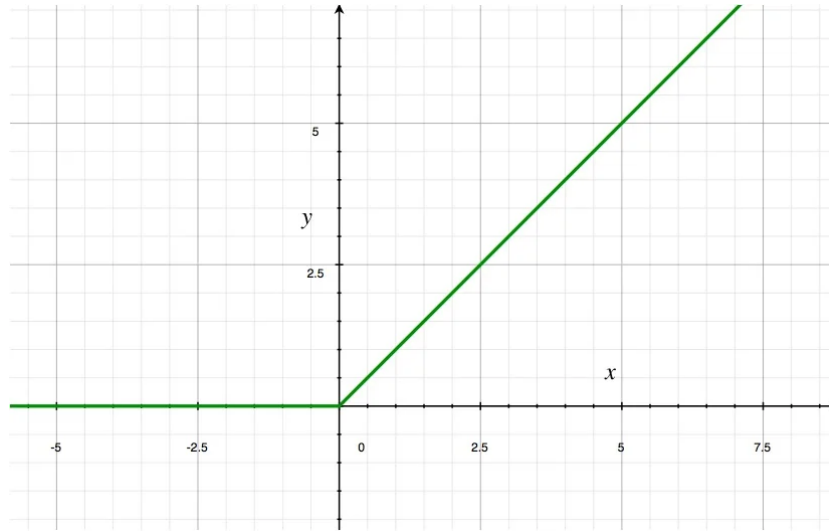


Figura 1.4: Grafico della funzione di attivazione ReLU.

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

La ReLU è una funzione semplice e computazionalmente efficiente, che, nella maggior parte dei casi, evita il problema del vanishing gradient sulle porzioni attive e favorisce la sparsità delle attivazioni. Il principale svantaggio è il problema della "dying ReLU": i neuroni possono diventare permanentemente inattivi se ricevono input negativi. Un neurone si considera "morto" se, per tutte (o quasi) le istanze del dataset, l'input $x \leq 0$, dato che in tal caso l'output è sempre nullo. Poiché la derivata di ReLU è zero per $x < 0$, il gradiente non si propaga a ritroso, e il neurone non riceve più aggiornamenti dei pesi, rimanendo inattivo per tutta la durata dell'allenamento. Viene ampiamente utilizzata nei layer nascosti in quasi tutte le architetture di reti neurali.

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

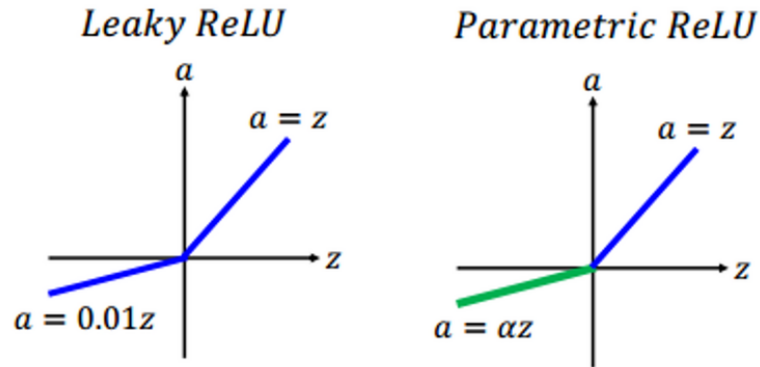


Figura 1.5: Grafico delle funzioni di attivazione LReLU e PReLU.

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

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

PReLU apprende il parametro α durante il training. Entrambe le varianti mantengono un piccolo gradiente per $x < 0$, riducendo significativamente il problema dei "dead neurons". L'introduzione (o l'apprendimento) di un iperparametro è un potenziale svantaggio, ed il loro comportamento non è sempre superiore a quello della semplice ReLU. Sono impiegate dove si vuole evitare il problema della "dying ReLU", mantenendo la semplicità.

5. Exponential Linear Unit (ELU)

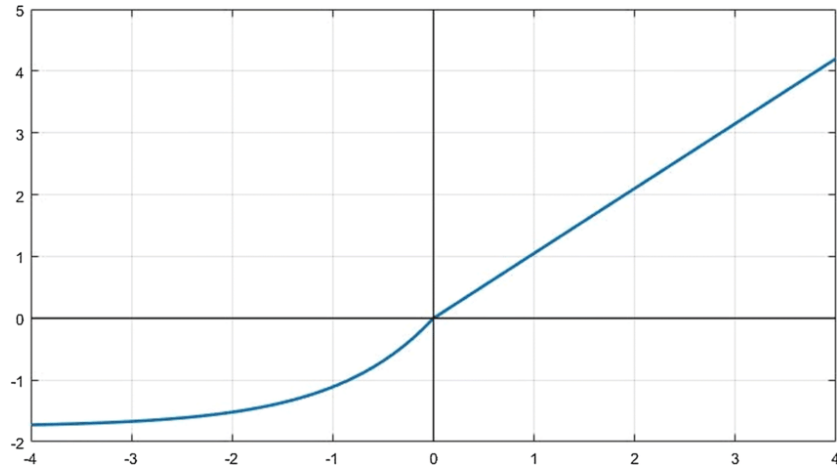


Figura 1.6: Grafico della funzione di attivazione ELU.

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

L'ELU produce un output più centrato rispetto alla ReLU e ha un gradiente non nullo per $x < 0$, che in alcuni casi può portare a una migliore convergenza. Tuttavia, è leggermente più costosa a livello computazionale a causa della funzione esponenziale ed introduce un parametro α da ottimizzare.

6. Softplus

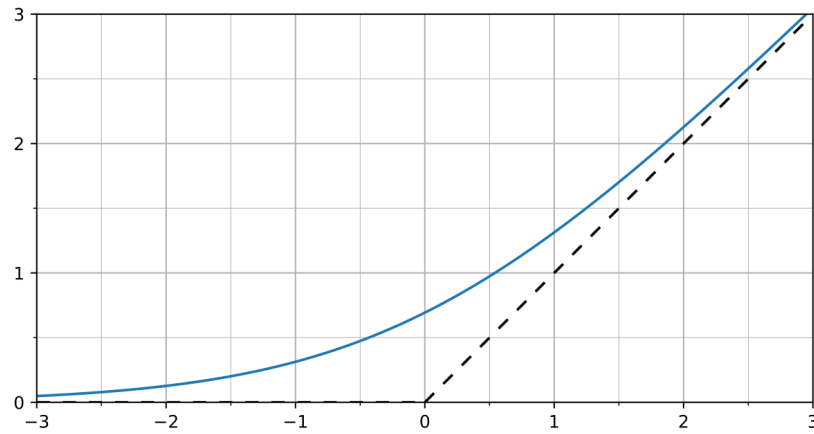


Figura 1.7: Grafico della funzione di attivazione Softplus.

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

La Softplus è una versione continua e completamente differenziabile della ReLU. Nonostante questa proprietà, è più costosa dal punto di vista computazionale e meno sparsa rispetto alla ReLU.

7. Gaussian Error Linear Unit (GELU)

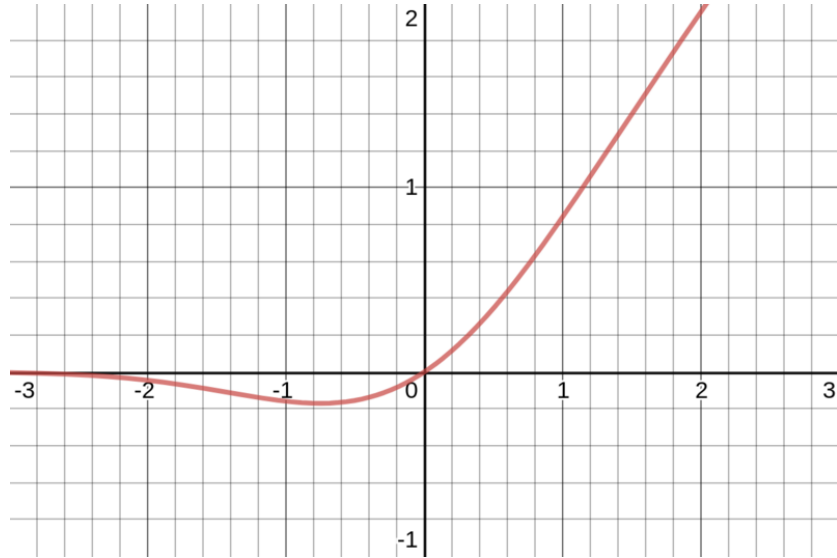


Figura 1.8: Grafico della funzione di attivazione GELU.

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

dove $\Phi(x)$ è la funzione di distribuzione cumulativa (CDF) della distribuzione normale standard e '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

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

La GELU ha dimostrato un eccellente comportamento empirico, specialmente nei modelli di linguaggio, ed è considerata una funzione di attivazione "soft" che combina linearità e gating stocastico. Il suo principale svantaggio è il costo computazionale più elevato. Viene ampiamente utilizzata nelle architetture Transformer.

8. Softmax

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

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

Questa funzione viene utilizzata per trasformare i logits (gli output grezzi di un layer) in una distribuzione di probabilità. È normalmente combinata con la funzione di perdita di categorical cross-entropy per problemi di classificazione multi-classe.

1.4 Procedura di forward pass

1.4.1 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'eccitazione, o 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 ϕ per ottenere l'uscita del neurone:

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

Ad esempio, con $\phi = \sigma$ (sigmoide), 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.

1.4.2 Propagazione ed output

Dopo aver calcolato le attivazioni in tutti gli strati intermedi, l'uscita dello strato finale (\mathbf{a}_{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 sigmoide; 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 predizioni che dipende dalle scelte di pesi, bias e funzioni di attivazione attraverso la rete. Infine, confrontando \mathbf{a}_{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.

1.5 Algoritmo di backpropagation

L'algoritmo di backpropagation è fondamentale per l'addestramento delle reti neurali, poiché calcola come i pesi della rete devono essere modificati per minimizzare l'errore. Questo processo si basa sull'applicazione della regola della catena (chain rule) per derivare il gradiente della funzione di perdita L rispetto a ogni peso w .

1.5.1 Derivazione del gradiente

Il processo inizia con la definizione dell'errore, calcolato attraverso una funzione di perdita L , che misura la differenza tra il valore previsto dalla rete e il valore target effettivo. Ad esempio, per la regressione si può usare l'errore quadratico medio (MSE), mentre per la classificazione si ricorre spesso alla cross-entropy.

Una volta definito l'errore, l'algoritmo procede calcolando la derivata parziale della funzione di perdita L rispetto all'attivazione netta di ciascun

neurone. Questo valore è noto come "errore locale" o δ_j per il neurone j , ed è una misura di quanto l'errore di output sia influenzato dall'input di quel neurone prima dell'applicazione della funzione di attivazione.

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

Per un neurone nello strato di output, il calcolo di δ_j è diretto (dipende dalla perdita e dalla derivata dell'attivazione). Per i neuroni nei layer nascosti, invece, δ_j si determina propagando a ritroso gli errori dei neuroni del layer successivo: ogni neurone nascosto riceve contributi di errore da tutti i neuroni a cui è collegato nel layer seguente. In forma esplicita, se indichiamo con k gli indici dei neuroni del layer successivo e con w_{kj} il peso che connette il neurone j (nascosto) al neurone k (del layer successivo), allora

$$\delta_j = \sigma'(z_j) \sum_k w_{kj} \delta_k^{(\text{next})},$$

dove z_j è la net-input del neurone j e σ' è la derivata della funzione di attivazione valutata in z_j . Questa formula mostra chiaramente due passaggi: si sommano i contributi di errore $\delta_k^{(\text{next})}$ dei neuroni del layer successivo pesati dai corrispondenti pesi w_{kj} ed il risultato viene moltiplicato per $\sigma'(z_j)$ per tener conto della non linearità locale del neurone j .

L'applicazione della chain rule permette quindi di ottenere il gradiente della perdita rispetto a ciascun peso w_{ji} che connette il neurone i al neurone j :

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

dove a_i è l'output (attivazione) del neurone i nel layer precedente e δ_j è l'errore locale del neurone j calcolato come sopra. Quindi, l'algoritmo calcola l'errore allo strato di output e lo distribuisce a ritroso lungo la rete, scalando i contributi con le derivate delle attivazioni; il prodotto $a_i \delta_j$ fornisce per ogni peso la misura del suo contributo all'errore complessivo e, quindi, la direzione in cui occorre modificarlo per ridurre la perdita.

1.5.2 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 η , l'aggiornamento 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.

1.5.3 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. Questo costringe la rete a non fare affidamento eccessivo su singoli neuroni, favorendo lo sviluppo di rappresentazioni ridondanti e riducendo l'overfitting. 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 L1 (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 ℓ_1 induce sparsità: molte componenti dei pesi vengono impostate a zero, facilitando la selezione di feature e l'interpretabilità del modello.

- **Regolarizzazione L2 (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 ℓ_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 è spesso preferibile quando il numero di variabili supera il numero di osservazioni o quando ci sono gruppi di variabili fortemente correlate.

1.6 Vantaggi e limiti

1.6.1 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.

1.6.2 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 sigmoide 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.

1.6.3 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 2

Kolmogorov–Arnold Networks (KAN)

2.1 Introduzione

Il presente capitolo introduce le Kolmogorov–Arnold Networks (KAN), una nuova classe di reti neurali che si ispira direttamente al teorema di approssimazione di Kolmogorov–Arnold. L'obiettivo è esplorare la loro architettura innovativa, che differisce dalle tradizionali reti MLP nel modo in cui gestiscono le funzioni di attivazione. Verranno esaminati i fondamenti matematici che giustificano la loro efficacia, in particolare come le KAN si propongono di superare la "maledizione della dimensionalità" (curse of dimensionality) grazie alla loro capacità di approssimare funzioni complesse. Il capitolo si concentra sul funzionamento operativo, descrivendo come le funzioni parametriche basate su B-spline sostituiscono i pesi scalari e le attivazioni fisse degli MLP. Infine, verranno discussi in dettaglio i vantaggi e gli svantaggi delle KAN, come la loro interpretabilità e maggiore flessibilità locale rispetto agli MLP, ma anche la loro maggiore complessità computazionale e la dipendenza dalla struttura del problema. [6, 7, 8, 11, 9, 12]

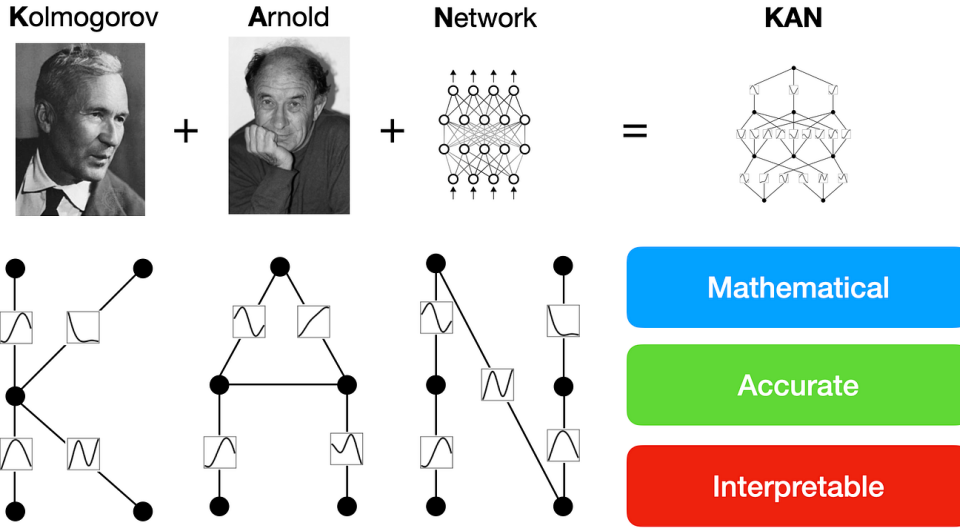


Figura 2.1: Kolmogorov–Arnold Networks

2.2 Fondamenti matematici

2.2.1 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 espressa 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 Φ_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 $q = 1, \dots, 2n + 1$.

Ciò significa che ogni funzione continua multivariata può essere scomposta in una somma di funzioni univariate più semplici Φ_q , (dette funzioni esterne), ciascuna delle quali dipende da una combinazione lineare delle variabili originali trasformate tramite funzioni univariate, dette funzioni

interne $\phi_{q,p}$. In questo contesto le funzioni interne agiscono da "features extractor", cioè estraggono informazioni rilevanti da ciascuna variabile, mentre le funzioni esterne combinano queste caratteristiche per produrre il valore finale della funzione, agendo in modo simile ad un classificatore delle features estratte. Questo risultato afferma che qualsiasi funzione continua di più variabili può essere completamente rappresentata tramite combinazioni di funzioni continue di una sola variabile.

2.2.2 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 Φ_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 un'utilità pratica limitata 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.

2.3 Architettura delle KAN

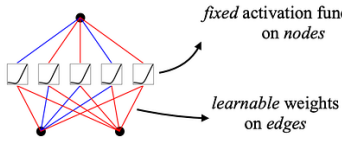
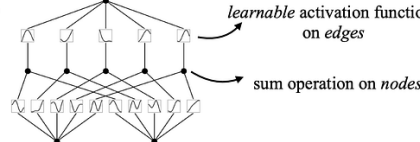
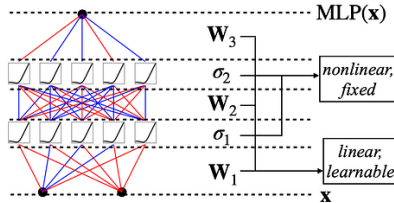
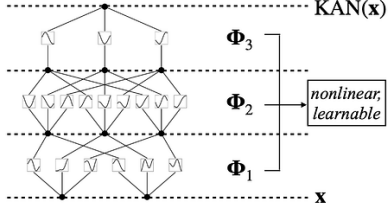
Model	Multi-Layer Perceptron (MLP)	Kolmogorov-Arnold Network (KAN)
Theorem	Universal Approximation Theorem	Kolmogorov-Arnold Representation Theorem
Formula (Shallow)	$f(\mathbf{x}) \approx \sum_{i=1}^{N(\epsilon)} a_i \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i)$	$f(\mathbf{x}) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^n \phi_{q,p}(x_p) \right)$
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 σ_2 \mathbf{W}_2 σ_1 \mathbf{W}_1 \mathbf{x} nonlinear, fixed linear, learnable	(d)  Φ_3 Φ_2 Φ_1 \mathbf{x} nonlinear, learnable

Figura 2.2: Confronto tra Multi-Layer Perceptron (MLP) e Kolmogorov-Arnold Network (KAN).

Una KAN è strutturalmente simile ad una rete feedforward completamente connessa, simile ad una MLP, ma differisce 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 nelle MLP. Ciascun neurone di uno strato riceve gli output dei collegamenti in ingresso e calcola semplicemente la somma di tali output, senza l'uso di pesi lineari o di funzioni di attivazione non lineari applicate ai singoli nodi stessi.

Il modello generale si descrive così: se lo strato $(\ell - 1)$ ha $d_{\ell-1}$ neuroni e lo strato ℓ ne ha d_ℓ , 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 ℓ 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 collegate a 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 combina, in un'unica funzione f_{ij} per ogni arco, le trasformazioni lineari e non lineari, permettendo alla rete di apprendere la forma esatta della funzione di attivazione necessaria per ogni arco di connessione.

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

2.3.1 Funzioni univariate parametriche

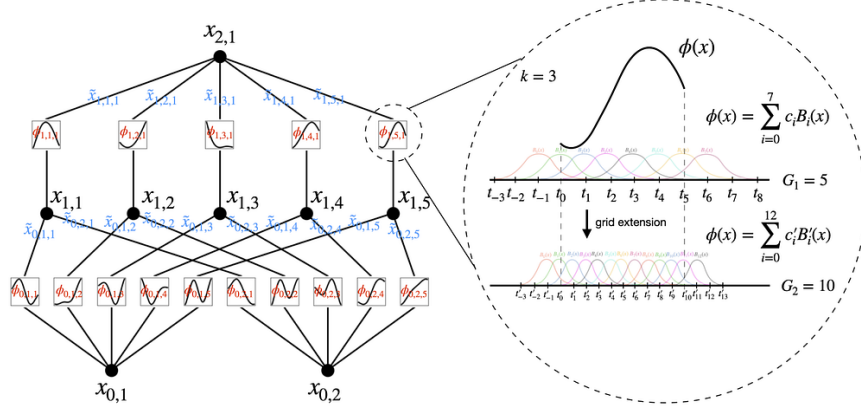


Figura 2.3: Struttura delle funzioni univariate parametriche.

Le funzioni di attivazione utilizzate nelle KAN sono funzioni univariate parametriche che possono 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 di attivazione su un collegamento in una KAN 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 affine iniziale, migliorando la stabilità dell'ottimizzazione durante il training, mentre il contributo spline introduce la non linearità appresa dalla rete, permettendo di model-

lare funzioni di attivazione flessibili e adattabili.

Un aspetto importante è la definizione della spline grid, ovvero la suddivisione dell'asse degli input in nodi che delimitano gli intervalli su cui sono definiti i singoli segmenti delle B-spline. 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 esigenze specifiche del problema. 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.

2.3.2 B-spline

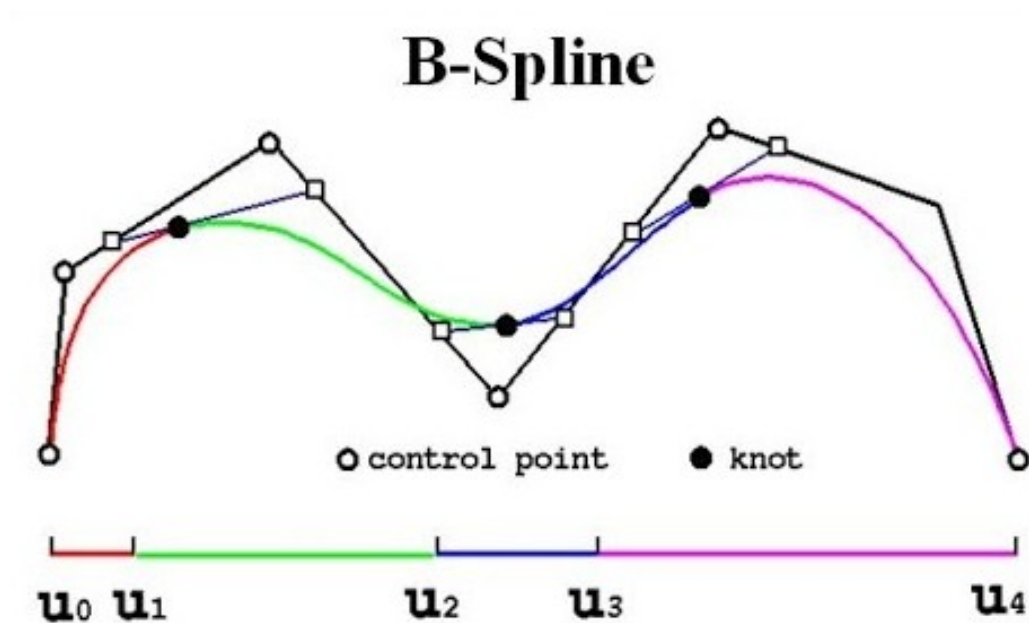


Figura 2.4: Struttura delle B-spline.

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 ($p = 3$):

$$f_{ij}(x) = w_0 x + \sum_{k=1}^{G+p} c_k B_{k,p} \left(\frac{x - x_{\min}}{x_{\max} - x_{\min}} \right),$$

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. Inizialmente, la griglia è generalmente uniforme, ma può essere ampliata aumentando il numero di nodi. Durante il training, i nodi possono essere riposizionati strategicamente per concentrare la risoluzione nelle aree in cui la funzione varia maggiormente, migliorando così la capacità di modellazione locale della rete. Per mantenere la stabilità durante queste modifiche alla griglia, si utilizzano tecniche di interpolazione lineare sui parametri dell'ottimizzatore, garantendo transizioni fluide e controllate nella definizione della griglia.

2.3.3 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 con 2 strati 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 ϵ , cioè

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

Questa capacità permette alle KAN di superare il problema noto come curse of dimensionality, a condizione 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 le quali il numero di parametri necessari per garantire una certa accuratezza generalmente cresce esponenzialmente con la dimensione dell'input. Il vantaggio teorico delle KAN deriva dal fatto che esse, a differenza delle MLP, apprendono non solo la struttura compositiva della funzione, ma sono anche in grado di modellare con elevata precisione le funzioni univariate interne, grazie all'uso di attivazioni parametriche basate su spline.

2.4 Funzionamento operativo

2.4.1 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 ℓ esegue la trasformazione $x^{(\ell)} = f^{(\ell)}(x^{(\ell-1)})$, e l'output finale è $y = x^{(L)}$.

Poiché tutte le operazioni, cioè l'applicazione delle funzioni univariate e le somme, sono differenziabili, l'intero modello 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.

2.4.2 Processo di training e Calcolo dei pesi

Il training di una KAN segue la procedura standard di addestramento supervisionato con discesa del gradiente. Si parte da 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 comune 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} .

2.5 Confronto con MLP tradizionali

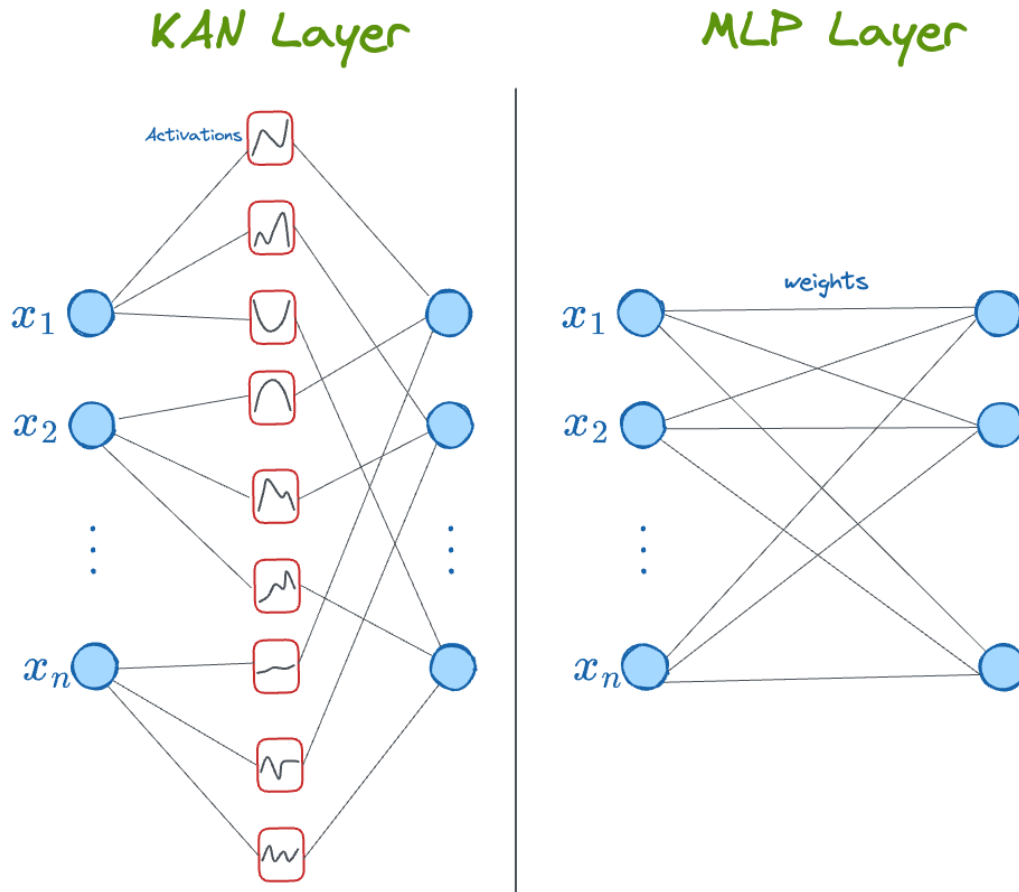


Figura 2.5: Differenze tra i Layer delle reti MLP e KAN.

2.5.1 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, una 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".

2.5.2 Complessità computazionale

Dal punto di vista parametrico, una KAN può avere un numero di parametri superiore rispetto ad una 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 una 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 una 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 una MLP. In pratica, valutare una B-spline su ogni collegamento è più costoso del prodotto scalare in una 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.

2.5.3 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.

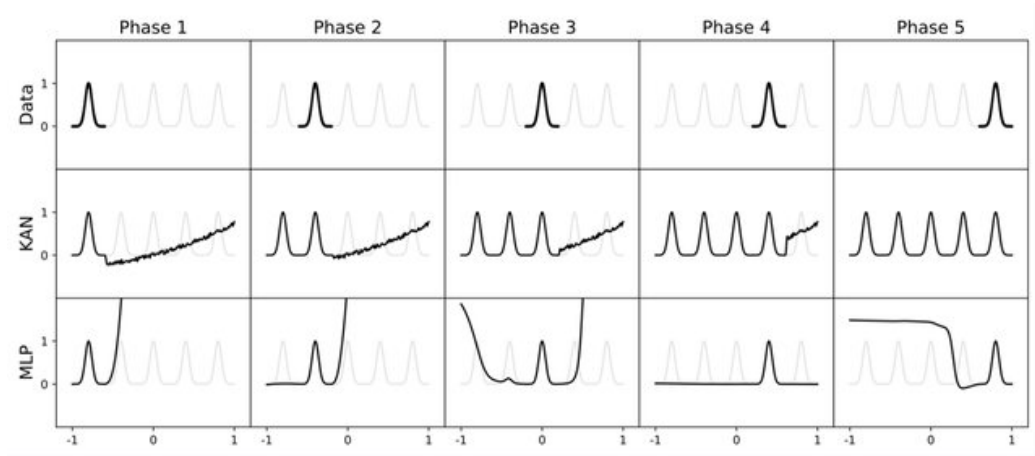


Figura 2.6: Differenza nel Catastrophic forgetting di MLP e KAN.

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.

2.5.4 Precisione controllabile tramite grid extension

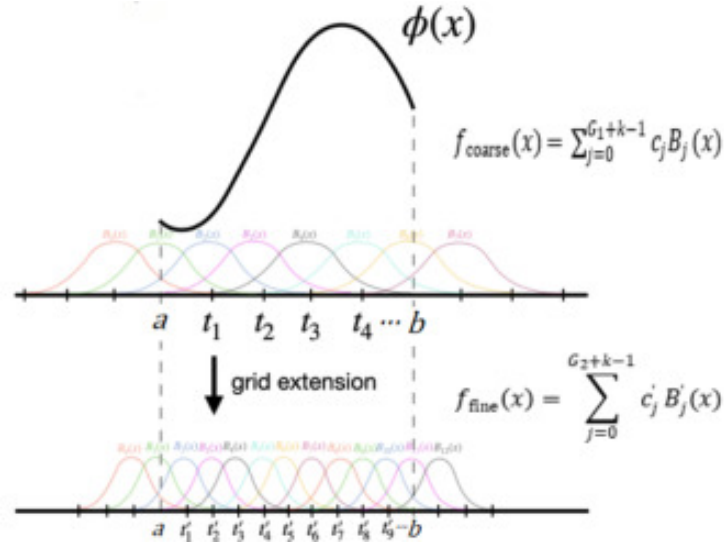


Figura 2.7: Grid extension delle B-spline nelle KAN.

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.

2.5.5 Dipendenza dalla struttura compositiva

I vantaggi più evidenti delle KAN si manifestano quando la funzione target da approssimare ha una struttura che si avvicina ad una decomposizione in somme di funzioni univariate, cioè può essere sufficientemente rappresentata come somma di trasformazioni su singole variabili. 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.

2.5.6 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.

2.5.7 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.

2.5.8 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 3

Random forest (RF)

3.1 Introduzione

In questo capitolo viene descritto il Random forest, un algoritmo di Machine learning molto versatile, in grado di gestire sia compiti di classificazione che di regressione. Il suo funzionamento si basa sull'idea dell'ensemble learning, combinando la forza di più alberi di decisione per ottenere un modello finale più robusto e preciso. L'introduzione del capitolo si concentra sui concetti fondamentali che guidano la costruzione del modello, come i criteri di divisione dei dati e la tecnica di bagging, che sfrutta il campionamento casuale per ridurre la varianza. Si approfondisce poi l'architettura specifica del Random forest, che aggiunge un ulteriore livello di casualità nella selezione delle variabili per ogni albero, rendendo la "foresta" più diversificata e meno soggetta ad overfitting. Il capitolo si conclude con una valutazione dei vantaggi e degli svantaggi dell'algoritmo, evidenziando la sua robustezza e la sua capacità di generalizzazione, ma anche i suoi requisiti in termini di risorse computazionali e la minore interpretabilità rispetto ad un singolo albero. [14, 15, 13, 18]

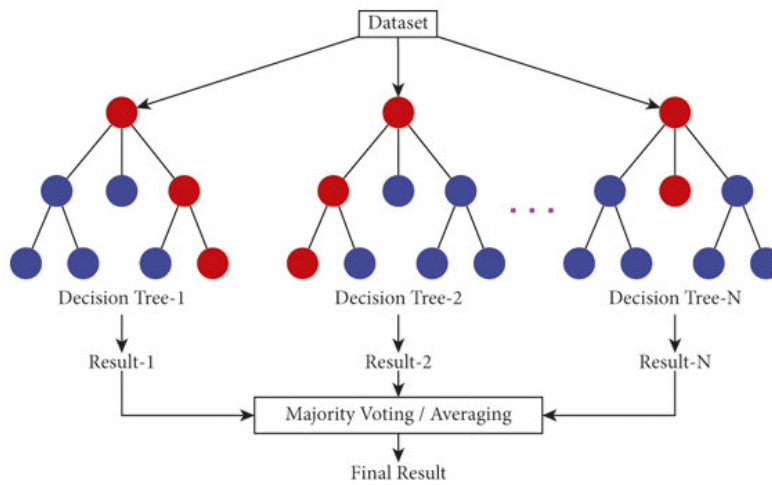


Figura 3.1: Architettura del Random forest.

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.

3.2 Concetti fondamentali

3.2.1 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.

3.2.2 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.

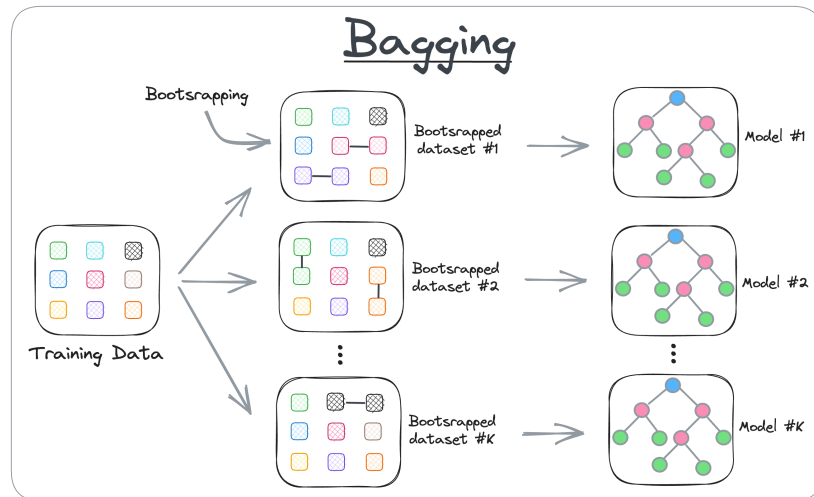


Figura 3.2: Funzionamento del Bagging.

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, ottenuti attraverso il campionamento di tipo bootstrap, ciascun modello apprende 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 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 "di-

versità" all'interno di un ensemble conduce a una maggiore robustezza del modello complessivo.

3.3 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.

3.3.1 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 nel Random Forest, tramite bootstrapping e feature bagging, 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.

3.3.2 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. Nei problemi di 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. Nei problemi di 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.

Questo processo di aggregazione trasforma un insieme di learner deboli in un modello forte. Sebbene gli alberi di decisione individuali possano presentare un'alta varianza, la media o il voto di maggioranza sulle loro previsioni riduce significativamente la varianza complessiva e rende il modello meno sensibile ad errori ed 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 finale superiore.

3.4 Vantaggi

Il Random Forest si distingue per la sua robustezza e flessibilità, offrendo diversi vantaggi che lo rendono una scelta popolare nel machine learning. Un punto di forza principale è la **riduzione della varianza**, che lo rende intrinsecamente meno propenso all'overfitting rispetto a un singolo albero di decisione. Questo risultato è ottenuto grazie alla sua natura ensemble, che aggrega le previsioni di molti alberi indipendenti, ed all'introduzione di casualità sia nel campionamento dei dati (bootstrap) sia nella selezione delle feature per ogni singolo albero. La sua robustezza non si limita alla varianza. L'algoritmo mostra anche una notevole **robustezza agli outlier**, poiché l'impatto di singole osservazioni estreme viene diluito e mediato tra i vari alberi. Il Random Forest eccelle anche nella **gestione dei valori mancanti**, semplificando notevolmente la fase di pre-elaborazione dei dati. Allo stesso modo, è in grado di gestire efficacemente **dataset sbilanciati**. Dal punto di vista della usabilità, il Random Forest facilita la **determinazione dell'importanza delle feature**, offrendo un modo diretto per valutare il contributo di ogni variabile al modello. La sua architettura è inoltre **parallelizzabile**, il che significa che i singoli alberi possono essere addestrati in parallelo. Questa caratteristica contribuisce in modo significativo alla sua velocità di addestramento, in particolare con dataset di grandi dimensioni e hardware multi-core.

3.5 Svantaggi

Nonostante i numerosi vantaggi, il Random Forest presenta alcune limitazioni, principalmente legate ai requisiti computazionali e alla complessità. Uno svantaggio notevole è il **costo computazionale ed il tempo di addestramento**. L'addestramento può essere lento, soprattutto con un numero elevato di alberi o su dataset molto grandi, poiché la costruzione di ogni singolo albero è un'operazione computazionalmente intensiva. Di conseguenza, il modello ha anche **requisiti di memoria** più elevati rispetto a un singolo albero, dato che deve memorizzare la struttura di tutti gli alberi che compongono la foresta. Un'altra debolezza significativa è **la complessità e la perdita di interpretabilità**. A differenza di un singolo albero di decisione, la cui logica è facile da visualizzare e comprendere, una "foresta" composta da centinaia o migliaia di alberi rende la previsione molto più difficile da interpretare a livello globale. Infine, il Random Forest **non include una regolarizzazione esplicita** nel suo nucleo. A differenza di altri algoritmi ensemble che possono incorporare tecniche di regolarizzazione dirette, il Random Forest si affida principalmente alla sua natura ensemble ed all'introduzione di casualità per prevenire l'overfitting.

Capitolo 4

eXtreme Gradient Boosting (XGBoost)

4.1 Introduzione

Questo capitolo presenta eXtreme Gradient Boosting (XGBoost), un algoritmo che ha rivoluzionato il Machine Learning per la sua efficacia e velocità. L'obiettivo è analizzare come XGBoost, pur basandosi sul Gradient Boosting, lo superi grazie ad una serie di importanti ottimizzazioni. Verranno esaminati i miglioramenti che lo rendono così performante, come l'uso di una funzione di perdita avanzata ed una gestione più efficiente della regolarizzazione e dei dati mancanti. Il testo spiegherà come la sua architettura sfrutti il parallelismo e l'uso intelligente della cache per velocizzare i calcoli, rendendo l'addestramento più rapido. Si affronterà anche il tema degli iperparametri, che offrono una grande flessibilità per personalizzare il modello, sebbene richiedano un'attenta calibrazione. Il capitolo si conclude con un riassunto dei suoi vantaggi, come la robustezza e l'efficienza, e dei suoi svantaggi, tra cui la complessità e la maggiore richiesta di risorse per il tuning. [17, 16, 19, 20, 21, 18]

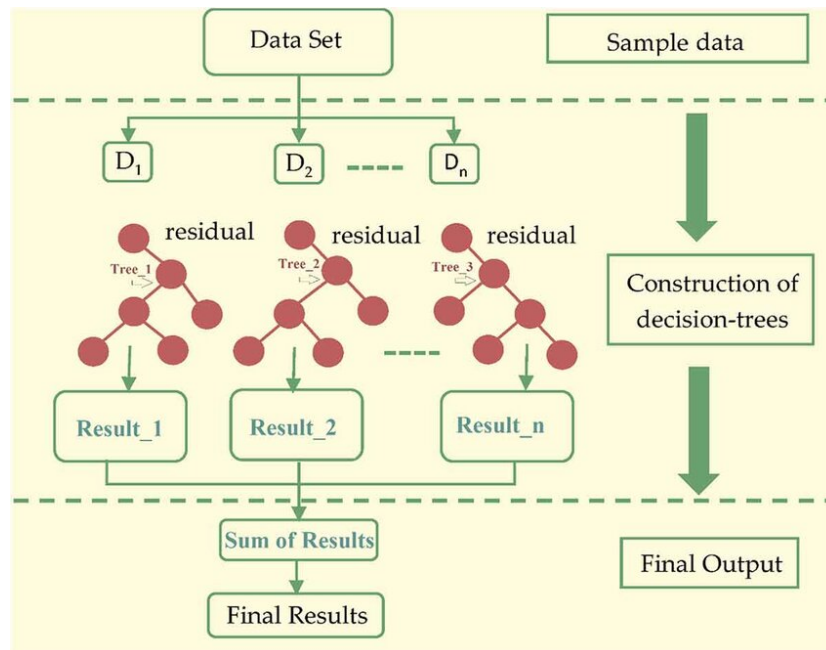


Figura 4.1: Architettura del XGBoost.

4.2 Introduzione al Gradient boosting

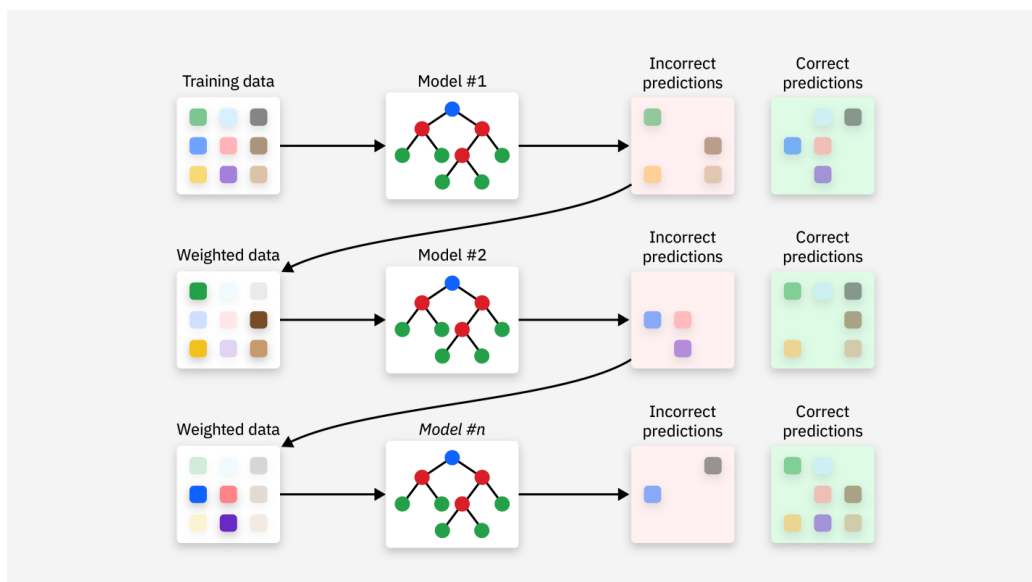


Figura 4.2: Funzionamento del Gradient boosting.

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.

4.2.1 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, i learner deboli sono tipicamente alberi di decisione semplici. Possono essere molto semplici, 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 di tale funzione. 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.

4.2.2 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.

4.3 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.

4.3.1 Regolarizzazione 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 e i pesi di grandi dimensioni. In particolare, la regolarizzazione L1, che si basa sulla somma del valore assoluto dei pesi:

$$\Omega(w) = \lambda \|w\|_1 = \lambda \sum_j |w_j|$$

incoraggia la sparsità, spingendo i pesi meno importanti verso lo zero. Al contrario, la regolarizzazione L2, che si basa sulla somma dei quadrati dei

pesi:

$$\Omega(w) = \lambda \|w\|_2^2 = \lambda \sum_j w_j^2$$

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. L'uso di learning rate piccoli (ad esempio, 0.01 o 0.1) migliora notevolmente 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 pota gli alberi in base a un "guadagno" del nodo. La decisione di effettuare una ulteriore divisione su un nodo foglia dipende dal raggiungimento di un guadagno minimo, specificato dall'iperparametro γ (gamma). Un valore più grande di γ rende l'algoritmo più conservativo, limitando la crescita dell'albero e aiutando a prevenire l'overfitting. Il guadagno di un nodo in XGBoost quantifica il miglioramento nella funzione obiettivo che si ottiene dividendo un nodo. Questo guadagno deve superare una soglia minima, definita dall'iperparametro γ , per autorizzare la divisione. La formula per il guadagno è:

$$\text{Gain} = \frac{1}{2} \left[\frac{(\sum_{i \in I_L} g_i)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{(\sum_{i \in I} g_i)^2}{\sum_{i \in I} h_i + \lambda} \right] - \gamma$$

Dove I_L e I_R sono gli insiemi di istanze (dati di addestramento) nei nodi figli sinistro e destro, mentre I rappresenta l'insieme di istanze nel nodo padre. In questa formula, g_i è la derivata prima della funzione di perdita rispetto all'output dell'istanza i , mentre h_i è la derivata seconda della funzione di perdita rispetto allo stesso output. Il parametro λ è un termine di regolarizzazione L2 che penalizza i pesi elevati, e γ è la soglia di guadagno minima richiesta per la divisione. Se il guadagno calcolato è inferiore a γ , la divisione viene annullata.

Un altro parametro cruciale per la regolarizzazione è la somma minima dei pesi delle istanze, noto come "min_child_weight". Questo iperparametro agisce come un criterio di regolarizzazione per controllare la crescita degli alberi. Un nodo figlio può essere creato solo se la somma dei pesi delle istanze al suo interno supera un valore soglia predefinito. Questa somma è calcolata usando l'***hessiana***, e la condizione per uno split può essere espressa come:

$$\sum_{i \in I_L} h_i \geq \text{min_child_weight} \quad \text{e} \quad \sum_{i \in I_R} h_i \geq \text{min_child_weight}$$

Se una delle somme è inferiore al valore di "min_child_weight", la partizione viene annullata. Un valore più grande di questo iperparametro rende l'algoritmo più conservativo, riducendo la complessità degli alberi individuali e aiutando a prevenire l'overfitting.

4.3.2 Gestione dei valori mancanti

XGBoost è in grado di lavorare efficacemente anche quando nel dataset ci sono dati assenti o non registrati per alcune feature. XGBoost incorpora internamente un meccanismo che permette di decidere automaticamente, durante la costruzione degli alberi, come trattare i valori mancanti. 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 impara quale direzione (ramo dell'albero) seguire per le istanze con valori mancanti per ottimizzare le performance. Internamente, durante la fase di training, XGBoost tratta la "mancanza" del dato come una caratteristica informativa a sé stante, apprendendo la direzione ottimale per i dati mancanti in modo da ottimizzare le suddivisioni. Questa capacità semplifica la pipeline di preparazione dei dati, evitando bias o rumori introdotti da imputazioni errate o rimozioni arbitrarie, e rende il modello più robusto e affidabile in scenari del mondo reale, dove i dati spesso presentano valori mancanti. 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.

4.3.3 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.

4.4 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.

A differenza di Random forest, che tende ad avere meno parametri da ottimizzare, XGBoost richiede una comprensione più approfondita ed 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.

4.5 Vantaggi

L'algoritmo XGBoost è rinomato per le sue prestazioni e si distingue per una serie di vantaggi chiave che lo rendono uno standard nel machine learning competitivo. Uno dei suoi punti di forza principali è **la robustezza all'overfitting**, garantita da varie tecniche di regolarizzazione integrate. Questo controllo granulare sulla complessità del modello è un fattore determinante per la sua notevole capacità di generalizzazione. XGBoost è stato progettato per la scalabilità e l'efficienza, permettendogli **una gestione efficiente di grandi dataset**, dati sparsi e valori mancanti.

La sua architettura ottimizzata consente di elaborare volumi di dati che per altri algoritmi sarebbero difficili da gestire. Nonostante la sua natura sequenziale, **la velocità di addestramento** di XGBoost è eccezionale; può superare quella del Random Forest, specialmente quando si sfruttano le sue capacità di parallelismo e l'accelerazione GPU, oltre al supporto per sistemi distribuiti.

Un altro vantaggio significativo è **la sua flessibilità e personalizzazione**. L'algoritmo offre un'ampia gamma di iperparametri che consentono un fine-tuning profondo, adattando il modello alle specifiche esigenze di ogni problema e dataset. Inoltre, XGBoost è particolarmente efficace nella **gestione di dati sbilanciati**, un problema comune in molti contesti di classificazione.

4.6 Svantaggi

Nonostante i suoi punti di forza, XGBoost presenta alcune limitazioni, principalmente legate alla sua complessità. **Il processo di addestramento sequenziale** può renderlo intrinsecamente più lento del Random Forest nella costruzione completa degli alberi, dato che ogni albero dipende dal precedente. Sebbene siano state introdotte ottimizzazioni per il parallelismo interno, la sua natura sequenziale rimane una potenziale barriera, a meno che non si sfruttino appieno le sue capacità di parallelismo e le accelerazioni hardware.

La complessità e la necessità di tuning rappresentano un'altra sfida. XGBoost è un algoritmo più complesso da comprendere e 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.

Inoltre, XGBoost può risultare **meno interpretabile del Random Forest**. Sebbene fornisca l'importanza delle feature, la complessità dell'ensemble di alberi sequenziali può rendere le sue decisioni specifiche più difficili da interpretare. Spesso sono necessari strumenti aggiuntivi per la spiegabilità,

a differenza del Random Forest che, in alcuni casi, può essere più trasparente.

Infine, **il consumo di memoria** può essere una limitazione significativa. XGBoost può consumare una notevole quantità di memoria, in particolare quando si addestrano alberi molto profondi. Questo può rappresentare un problema per dataset estremamente grandi su hardware con risorse limitate.

Capitolo 5

Convolutional Neural Networks (CNN)

5.1 Introduzione

Questo capitolo si occupa di introdurre le Convolutional Neural Networks (CNN), un'architettura di rete neurale che eccelle nell'analisi di immagini e di dati con una struttura a griglia. L'obiettivo è esplorare come queste reti riescano ad estrarre automaticamente le caratteristiche rilevanti, superando i limiti delle reti tradizionali. Il testo descriverà il funzionamento dei principali blocchi costruttivi, come la convoluzione e il pooling, che permettono alla rete di identificare forme e pattern in modo gerarchico e di ridurre la complessità. Verrà poi spiegato come, attraverso una sequenza di operazioni, un'immagine viene trasformata in rappresentazioni via via più astratte fino a giungere ad una decisione finale. Infine, il capitolo riassume i vantaggi delle CNN, come la loro efficienza e la capacità di riconoscere oggetti a prescindere dalla loro posizione, e i loro svantaggi, tra cui la necessità di enormi quantità di dati e il loro costo computazionale. [23, 24, 25, 26, 27, 28, 29]

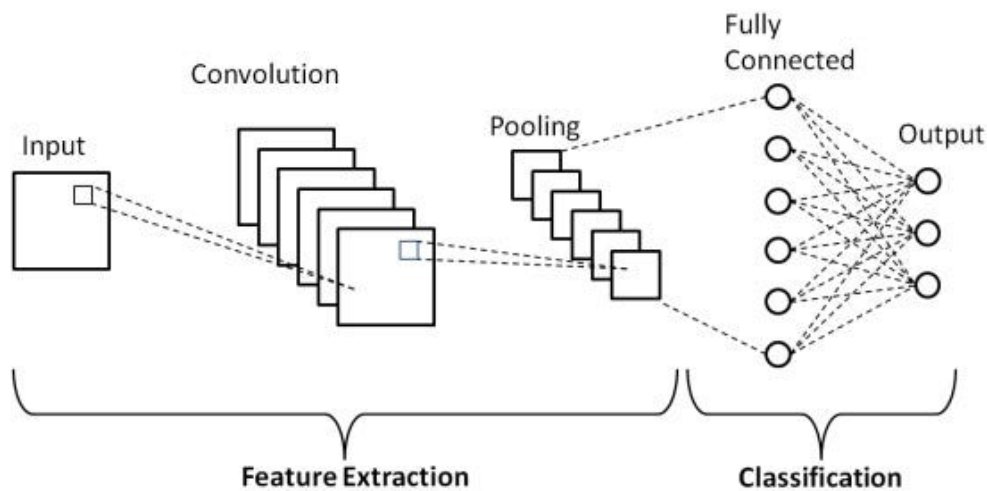


Figura 5.1: Architettura delle Convolutional Neural Networks.

5.2 Principi fondamentali delle CNN

5.2.1 Convoluzione: kernel, stride, padding

Un kernel (o filtro) è una matrice di piccole dimensioni, tipicamente 3×3 , 5×5 o 7×7 , 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.

5.2.2 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.

5.3 Pooling e normalizzazione

5.3.1 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 2×2 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.

5.3.2 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.

5.3.3 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 μ_B e σ_B^2 sono la media e varianza del batch, ed ϵ è una costante piccola per evitare divisioni per zero. Successivamente applica una trasformazione affine apprendibile:

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

dove γ e β 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.

5.4 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° e 50°), 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.

5.5 Funzionamento delle CNN

5.5.1 Forward pass

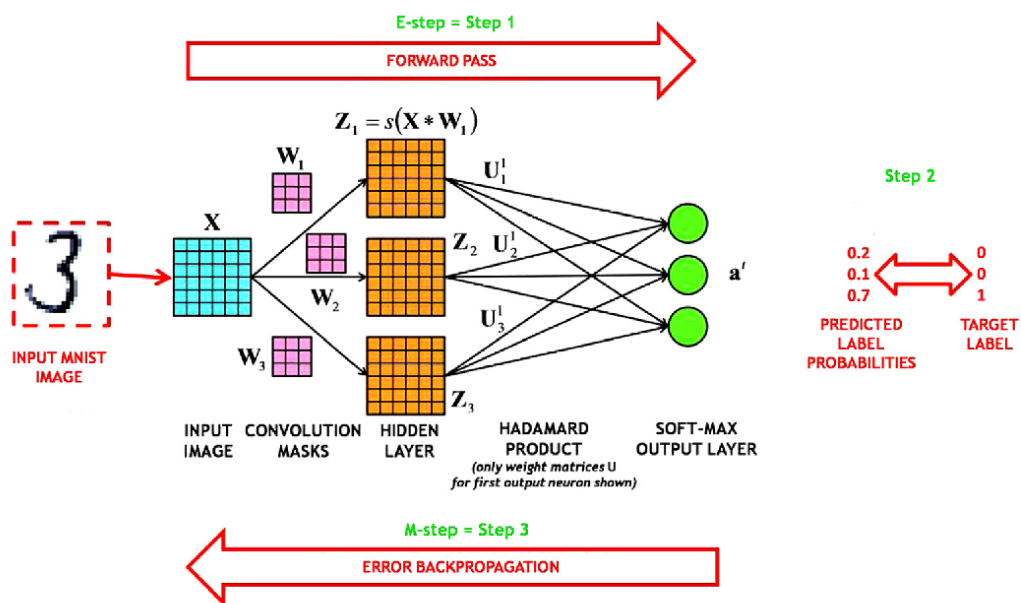


Figura 5.2: Diagramma del forward pass in una CNN.

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.

5.5.2 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.

5.5.3 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.

5.5.4 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.

5.5.5 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.

5.5.6 Strati fully-connected

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

Durante il terzo caso di studio, verranno esplorate le modifiche agli strati fully-connected delle CNN, analizzando due diverse architetture: MLP e KAN.

5.5.7 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.

5.6 Vantaggi

Le CNN si distinguono per una serie di vantaggi che le rendono lo standard per l'elaborazione delle immagini. Il loro punto di forza principale è il **rilevamento automatico delle caratteristiche**. A differenza delle reti neurali tradizionali che richiedono un'estrazione manuale delle feature, le CNN apprendono e identificano autonomamente le caratteristiche rilevanti direttamente dai dati grezzi. Questo approccio riduce drasticamente lo sforzo di pre-processing e permette al modello di adattarsi in modo più efficace ai dati.

Un'altra caratteristica distintiva è la loro **efficienza computazionale e riduzione dei parametri**. L'uso del weight sharing (una tecnica che permette 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 fully-connected. Questo non solo accelera l'addestramento, ma contribuisce anche a prevenire l'overfitting.

Grazie all'operazione di convoluzione, le CNN offrono **invarianza alla traslazione**. Sono in grado di riconoscere un oggetto indipendentemente dalla sua posizione nell'immagine. Un filtro che ha imparato a riconoscere un occhio, ad esempio, lo riconoscerà sia che si trovi in alto a sinistra che in basso a destra.

Le architetture CNN sono anche estremamente **scalabili**: possono essere adattate facilmente a dataset di grandi dimensioni ed a compiti complessi, aumentando la profondità e la larghezza della rete per gestire immagini ad alta risoluzione o per apprendere pattern più astratti.

5.7 Svantaggi

Nonostante i loro numerosi vantaggi, le CNN presentano alcune limitazioni. La principale è la loro **dipendenza da grandi dataset**. Specialmente le architetture più complesse richiedono enormi quantità di dati etichettati per un addestramento efficace. La mancanza di un dataset sufficientemente

grande può portare all'overfitting o ad una scarsa capacità di generalizzazione.

Un altro limite è l'**invarianza limitata**. Le CNN standard sono 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 usino tecniche di data augmentation per esporre il modello a tali variazioni durante l'addestramento.

Le CNN sono spesso criticate per la loro **mancanza di interpretazione**. È difficile capire il motivo per cui un modello prenda una determinata decisione. Le feature map intermedie possono essere visualizzate, ma il processo decisionale complessivo rimane una "black box", rendendo complicato il debugging e l'adozione in settori critici come la medicina, dove è essenziale la trasparenza.

Infine, l'addestramento di architetture CNN molto profonde ha un elevato **costo computazionale**. Spesso richiede una notevole potenza di calcolo e hardware specializzato come le GPU. Anche l'inferenza su dispositivi a bassa potenza può risultare problematica.

Capitolo 6

Ottimizzazione degli iperparametri

6.1 Introduzione

Questo capitolo presenta una panoramica delle metodologie di ottimizzazione degli iperparametri, essenziali per migliorare le performance dei modelli di Machine e Deep Learning. L'obiettivo è esplorare come queste tecniche permettano di navigare lo spazio delle configurazioni di un modello per trovare la combinazione ideale. Il testo inizia descrivendo diverse forme di Cross-Validation, tra cui la K-fold CV, la Nested Cross-Validation e la Time Series Cross-Validation, pensata specificamente per i dati temporali. Successivamente, il capitolo si concentra sulle strategie di ricerca, a partire dai metodi più semplici come il Grid Search e il Random Search, per poi introdurre approcci più sofisticati come l'Ottimizzazione Bayesiana e gli Algoritmi Genetici. Infine, viene fornito un confronto pratico tra questi metodi per aiutare a comprendere quando applicare ciascuno di essi. Per la sua efficienza e scalabilità, in questa ricerca è stato scelto il Random Search, che è stato ottimizzato utilizzando una formula che utilizza le probabilità per determinare il numero ideale di iterazioni. [31, 34, 35, 30, 32, 33]

6.2 Cross-Validation (CV)

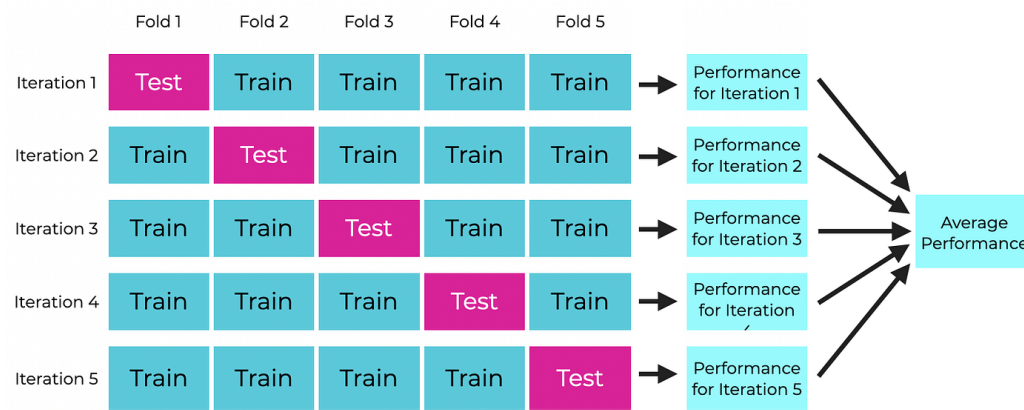


Figura 6.1: Funzionamento della tecnica di Cross-Validation.

La CV è una metodologia utilizzata 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 $\hat{f}_{D^{(k)}}$. Poi, 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, $\bar{E} = \frac{1}{K} \sum_{k=1}^K E_k$.

6.2.1 Time Series Cross-Validation (TSCV)

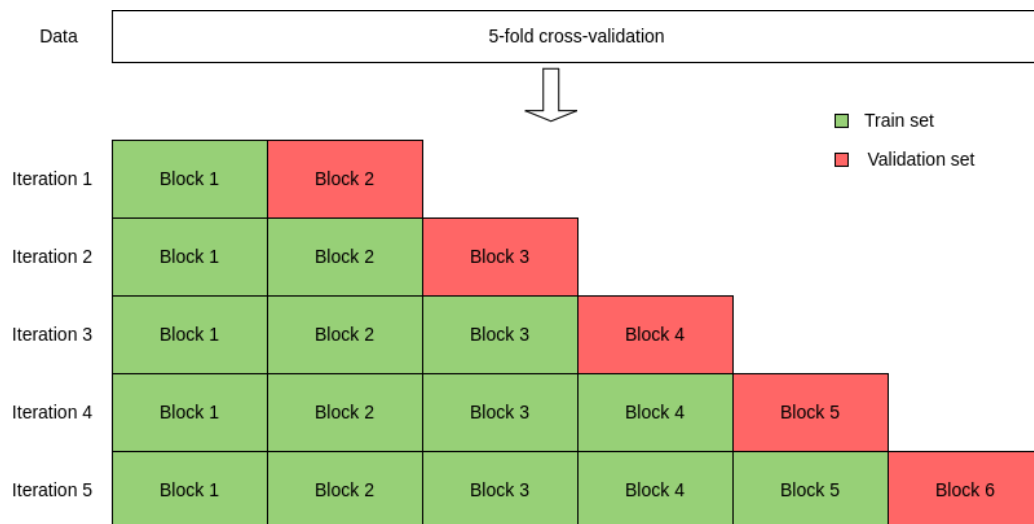


Figura 6.2: Svolgimento della tecnica di Time Series Cross-Validation.

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.

6.3 Nested Cross-Validation (NCV)

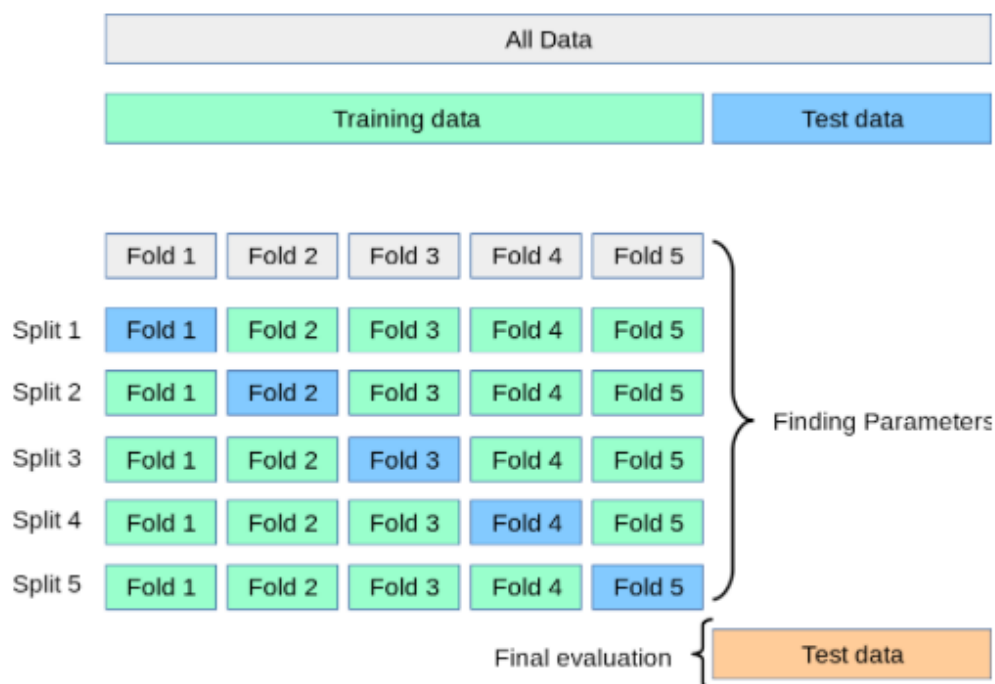


Figura 6.3: Funzionamento della tecnica di Nested Cross-Validation.

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.

6.4 Grid search (GS)

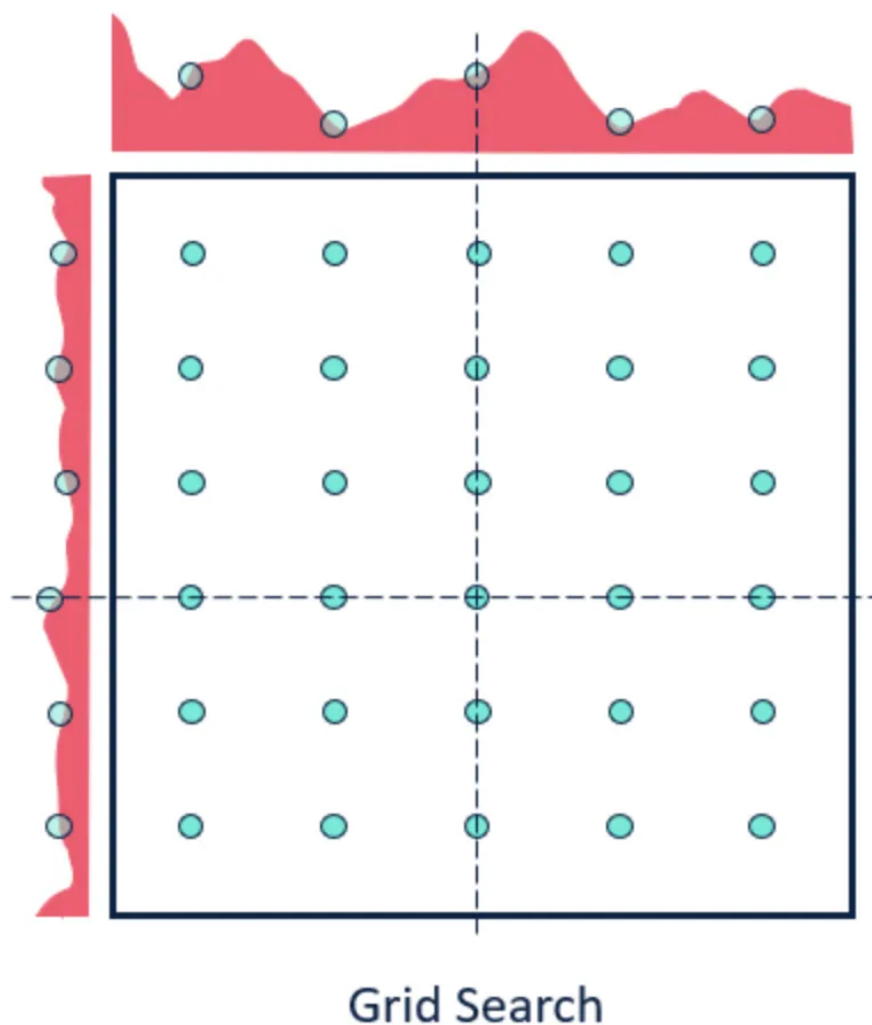


Figura 6.4: Grafico che mostra come Grid search ispeziona lo spazio degli iperparametri.

6.4.1 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.

6.4.2 Vantaggi

L'approccio del Grid Search offre diversi vantaggi. Primo fra tutti, la sua natura **esaustiva**: esplora tutte le combinazioni predefinite nello spazio di ricerca, permettendo di trovare l'ottimo globale se questo è incluso nella griglia. Inoltre, è un metodo **deterministico e riproducibile**, dato che l'assenza di casualità assicura che ogni esecuzione dell'algoritmo fornisca risultati replicabili. Infine, la sua **semplicità di implementazione** lo rende ideale in spazi di ricerca ridotti e ben definiti, o come baseline quando si dispone di elevate risorse computazionali.

6.4.3 Limiti

Nonostante i suoi vantaggi, il Grid Search presenta anche dei limiti significativi. Il suo **costo computazionale è esponenziale**, a causa della curse of dimensionality, rendendolo impraticabile per spazi di ricerca ampi o ad alta dimensionalità. Il metodo è spesso **inefficiente**, poiché molte valutazioni potrebbero riguardare regioni poco promettenti, specialmente quando solo alcuni parametri influenzano la performance ottimale. **La discretizzazione e la perdita di ottimi** sono altri problemi rilevanti: la necessità di fissare una griglia per iperparametri continui può portare a saltare valori potenzialmente migliori che non sono inclusi. Infine, non è **adatto a modelli e dataset complessi**, poiché il costo aumenta drasticamente con la complessità e la dimensione del dataset.

6.5 Random Search (RS)

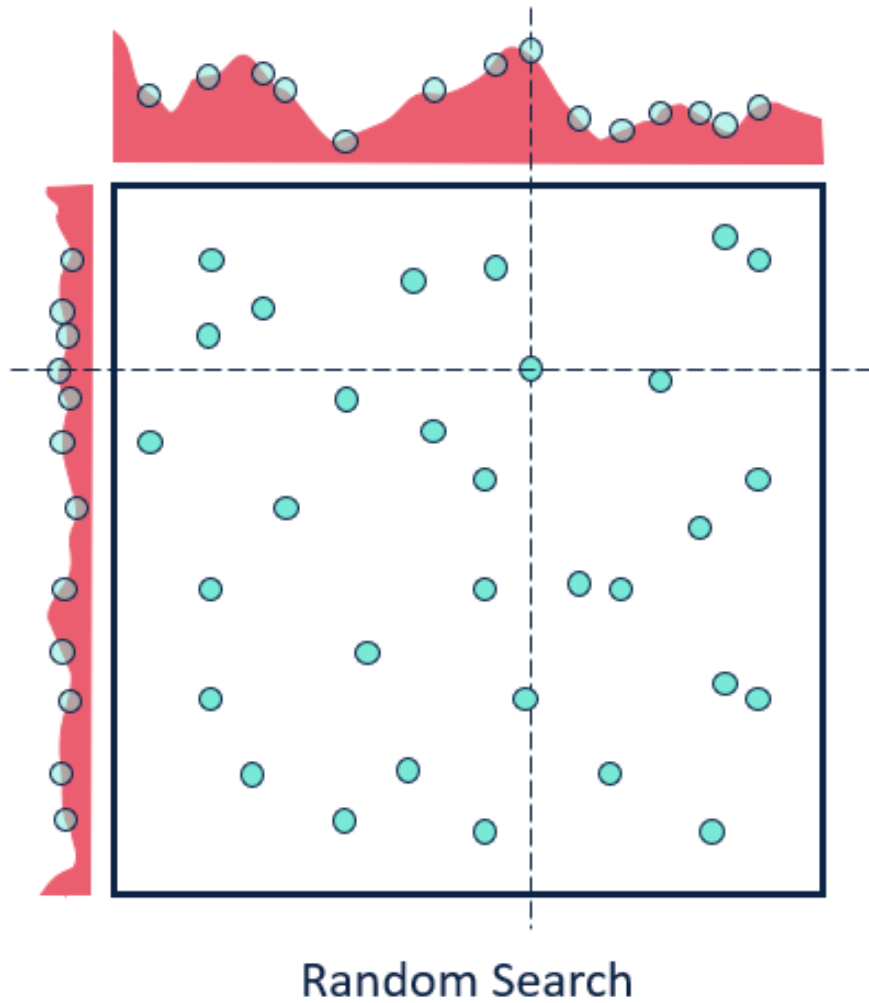


Figura 6.5: Grafico che mostra come Random search esegue l'ottimizzazione degli iperparametri sullo spazio definito.

6.5.1 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.

6.5.2 Vantaggi

Il Random Search presenta numerosi vantaggi, tra cui l'**efficienza su spazi estesi**. Campionando casualmente, si ha una maggiore probabilità di individuare combinazioni efficaci, specialmente quando solo pochi parametri sono percepiti come determinanti per le prestazioni. Un altro punto di forza è la sua **scalabilità**: il numero di valutazioni può essere fissato (es. $N=100$), spesso ottenendo performance simili a quelle del Grid Search con molte meno combinazioni. L'algoritmo è anche **facile da parallelizzare**, poiché ogni valutazione è indipendente, consentendo l'esecuzione in parallelo. Infine, è molto **adattabile**, in quanto è possibile scegliere distribuzioni di campionamento informate da conoscenze a priori.

6.5.3 Limiti

Tra i limiti del Random Search, il più evidente è la sua natura **non sistematica**, che non esplora esaustivamente lo spazio delle ipotesi e può quindi saltare l'ottimo globale. Le prestazioni dipendono in gran parte dalla **distribuzione di campionamento** scelta; campionamenti mal scelti possono ignorare regioni promettenti. Inoltre, i risultati **non sono ripetibili** a meno che non si fissi un random seed. Infine, su spazi di ricerca piccoli e ben definiti, il Grid Search può risultare più efficace.

6.6 Bayesian optimization (BO)

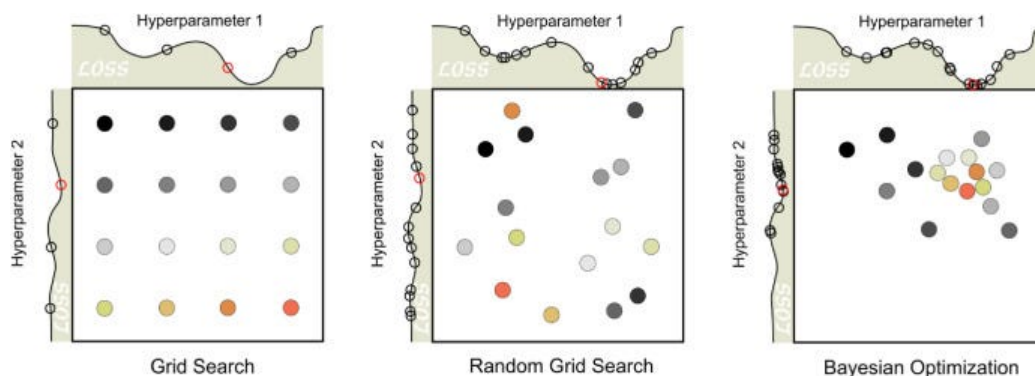


Figura 6.6: Confronto tra la ricerca di Grid e Random search con l'ottimizzazione bayesiana.

6.6.1 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 λ , una distribuzione normale per il valore $f(\lambda)$ con media $\mu(\lambda)$ e varianza $\sigma^2(\lambda)$. Dopo aver osservato alcune valutazioni, si aggiorna μ e σ 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.

6.6.2 Vantaggi

La Bayesian optimization è un metodo molto **efficiente**, riducendo drasticamente il numero di valutazioni necessarie, il che la rende ideale per funzioni costose. La sua funzione di acquisizione permette un eccellente

bilanciamento tra exploration ed exploitation, permettendo di esplorare nuove regioni e raffinare quelle già note. Inoltre, il modello surrogato offre una **modellazione dell'incertezza**, che indirizza la ricerca nelle zone potenzialmente più promettenti.

6.6.3 Limiti

Tra i limiti della BO, si include l'**overhead computazionale** dovuto al mantenimento e all'aggiornamento del modello surrogato. La sua natura sequenziale rende più **complessa la parallelizzazione** rispetto a Random o Grid Search. Infine, la sua **scalabilità è limitata**: pur essendo efficiente su spazi continui di media dimensione, l'ottimizzazione può diventare difficoltosa su spazi molto ampi.

6.7 Genetic algorithm (GA)

Genetic Algorithms

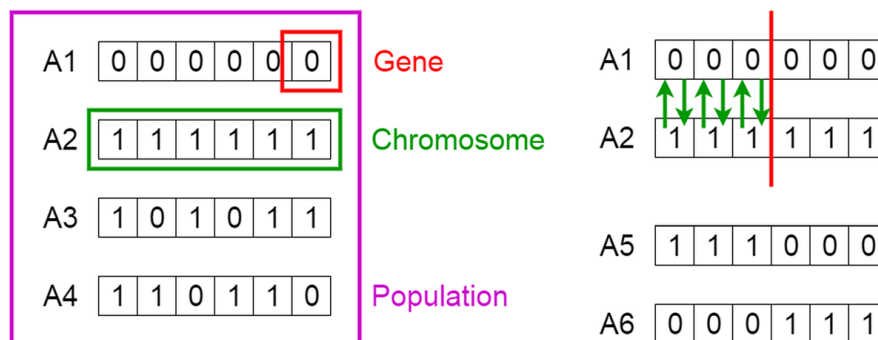


Figura 6.7: Definizione dei concetti principali alla base del funzionamento degli Algoritmi genetici.

6.7.1 Spiegazione dell'algoritmo

I Genetic algorithm(GA) sono metodi di ottimizzazione degli 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

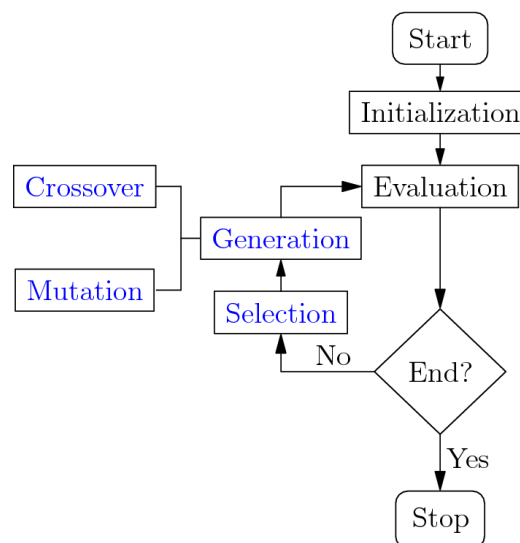


Figura 6.8: Le varie fasi del processo del Genetic algorithm.

- **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.

6.7.2 Vantaggi

Gli algoritmi genetici offrono una **esplorazione robusta** grazie a crossover e mutazioni, il che li rende adatti a spazi complessi e non lineari per trovare ottimi globali. Sono inoltre efficaci nella **gestione di spazi misti**, lavorando bene con iperparametri discreti, continui e categorici. Un altro vantaggio è la loro **parallelizzazione**, in quanto le valutazioni del fitness possono essere distribuite in parallelo.

6.7.3 Limiti

Tra gli svantaggi, si evidenzia il **costo computazionale elevato**, poiché richiedono molte generazioni e valutazioni. I risultati dipendono anche

da **iperparametri evolutivi**, come la dimensione della popolazione e i tassi di mutazione/crossover, che devono essere a loro volta ottimizzati. Infine, non c'è una **garanzia di convergenza** all'ottimo globale, dato che l'algoritmo potrebbe bloccarsi in minimi sub-ottimali se la diversità non viene mantenuta.

6.8 Confronto pratico

6.8.1 Criteri per la scelta

Nella scelta del metodo di ottimizzazione degli iperparametri, è fondamentale valutare diversi criteri per identificare la soluzione più adatta ad un problema specifico. I criteri principali includono **l'efficienza**, che si riferisce al numero di valutazioni necessarie per trovare una buona soluzione; **la scalabilità**, ovvero il comportamento dell'algoritmo all'aumentare della dimensionalità degli iperparametri; **il supporto per vari tipi di variabili**, che possono essere continue, discrete o categoriche; **la parallelizzazione**, ovvero la facilità con cui l'algoritmo può essere eseguito su cluster o GPU; **la robustezza**, ovvero la capacità di gestire rumore e funzioni complesse; e infine **le risorse computazionali richieste**, che considerano l'overhead e i costi aggiuntivi dell'algoritmo.

6.8.2 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.

6.8.3 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

6.8.4 Scelta per i casi studio: Random search

Motivazioni della scelta

Per i casi studio, si è scelto di adottare il Random Search per diverse ragioni. Questo approccio garantisce una notevole **efficienza su spazi ampi**, offrendo prestazioni superiori al Grid Search quando l'impatto di alcuni iperparametri è marginale. Inoltre, la sua **scalabilità** lo rende immune all'aumento esponenziale della complessità, mantenendo performance elevate anche con un gran numero di iperparametri. A livello implementativo, la sua **semplicità** lo rende un metodo ideale per l'implementazione e la parallelizzazione, riducendo la complessità del codice e facilitando l'esecuzione su cluster e GPU. Un'altra motivazione importante è la sua **flessibilità operativa**, che permette interruzioni anticipate tramite Early stopping ed un facile riutilizzo dei risultati per analisi successive. Infine, il Random Search offre **il miglior rapporto costo-beneficio**, bilanciando efficienza esplorativa e semplicità computazionale, rendendolo la scelta ideale per il contesto di questa tesi.

6.8.5 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 l'efficacia esplorativa con il costo computazionale, evitando valutazioni superflue.

Derivazione teorica della formula

La stima del numero di iterazioni richieste si basa sulla teoria della probabilità discreta. Il processo di derivazione segue una logica chiara, partendo dalla probabilità di fallimento in un singolo tentativo. Se in uno spazio di ricerca con M configurazioni totali ne esistono k che consideriamo "quasi-ottimali", la probabilità di non selezionarne una in un singolo campionamento casuale è:

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

La probabilità di fallire in tutti gli n tentativi indipendenti è quindi:

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

Da qui, si ricava la probabilità di successo, ovvero di trovare almeno una configurazione top- k in n tentativi:

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

Infine, per determinare il numero di iterazioni n necessarie per raggiungere una probabilità di successo P desiderata, si risolve la formula per n :

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

Quando il numero delle migliori configurazioni (k) è molto più piccolo del numero totale di configurazioni (M), si può usare l'approssimazione $\ln(1 - x) \approx -x$ per x piccolo. In questo caso, la formula si semplifica in:

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

Vantaggi dell'approccio probabilistico

L'approccio probabilistico al Random search offre notevoli vantaggi. Garantisce un'elevata **efficienza computazionale**, riducendo drasticamente il numero di valutazioni necessarie. Fornisce inoltre un forte **controllo statistico**, offrendo una garanzia probabilistica di trovare soluzioni quasi-ottimali. Grazie alla sua **flessibilità**, permette di modulare il trade-off tra accuratezza desiderata (P) e il costo computazionale (n). Infine, il metodo si distingue per la sua **scalabilità**, adattandosi automaticamente alla dimensione dello spazio di ricerca, un aspetto cruciale in contesti con molti iperparametri.

Capitolo 7

Studio di ablazione e Pruning post-training

7.1 Introduzione

Questo capitolo introduce e analizza due metodologie fondamentali per l'ottimizzazione e la comprensione dei modelli di Machine e Deep Learning: lo studio di ablazione ed il post-training pruning. L'obiettivo è esplorare come queste tecniche permettano non solo di diagnosticare il funzionamento interno di un modello, ma anche di migliorarne l'efficienza senza comprometterne significativamente le prestazioni. Il testo inizia descrivendo la natura degli studi di ablazione e la loro utilità nell'identificare le componenti critiche di un modello. Successivamente, il capitolo si concentra sul pruning post-training, partendo da una definizione generale per poi focalizzarsi sulla tecnica di L1 pruning applicata a diverse architetture, come i classici MLP e le KAN. Infine, viene fornito un confronto tra il pruning rank-based per le Random Forest e il cumulative pruning per XGBoost, illustrando come queste tecniche possano essere adattate per ottimizzare diversi tipi di ensemble. Per la sua efficienza e la sua flessibilità, il pruning L1 è stato scelto come tecnica di riferimento per l'ottimizzazione dei modelli di deep learning presentati in questa ricerca. [38, 37, 36]

7.2 Studi di ablazione

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.

7.2.1 Definizione

Per formalizzare matematicamente il concetto di ablazione, consideriamo un modello f_θ dove θ 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, θ_{-S} rappresenta il modello con la componente S rimossa. Un valore elevato di Δ_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.

7.2.2 Benefici

Gli studi di ablazione offrono diversi benefici significativi. Innanzitutto, permettono l'**identificazione delle componenti critiche**, aiutando a comprendere la sensibilità del modello a specifiche parti della sua architettura

e rivelando potenziali vulnerabilità o punti di forza. Inoltre, i risultati di questi studi fungono da **guida per la semplificazione** del modello, fornendo una base empirica per le decisioni di pruning. Infine, l'analisi di come la rimozione di una componente specifica influenzi le previsioni facilita l'**interpretazione del comportamento del modello**, contribuendo a una maggiore comprensione dei meccanismi interni che portano alle predizioni finali.

7.3 Pruning post-training

7.3.1 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.

7.3.2 Benefici

I benefici del pruning sono molteplici. Il primo e più evidente è la **riduzione della memoria e del tempo di inferenza**: il pruning riduce lo spazio di memoria richiesto e il tempo necessario per le predizioni, aspetti critici per il deployment in ambienti con risorse limitate. L'obiettivo primario è il **mantenimento delle prestazioni**, bilanciando l'efficienza e l'accuratezza del modello in modo che le performance rimangano entro una tolleranza accettabile. Infine, riducendo il numero di elementi attivi, il pruning contribuisce a un aumento dell'**interpretabilità**, facilitando l'analisi e la comprensione del contributo delle parti rimanenti.

7.3.3 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.

7.4 Pruning L1 post-training per MLP e KAN

7.4.1 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 θ_i , che è definita da una soglia di potatura λ , che viene tipicamente determinata a livello globale per il modello o a livello locale per ciascuno 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 λ , esso viene permanentemente impostato a zero. Al contrario, se la sua magnitudine è superiore a λ , il parametro viene mantenuto invariato.

7.4.2 Considerazioni specifiche per le KAN

L'applicazione del L1 pruning alle KAN è 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.

7.5 Pruning per Ensemble: Rank-based pruning per Random forest

7.5.1 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 ΔL_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.

7.5.2 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.

7.5.3 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.

7.6 Pruning per Ensemble: Cumulative pruning per XGBoost

7.6.1 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.

7.6.2 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 8

Metodologie e Procedure comuni per la Verifica sperimentale dei Casi di studio

8.1 Introduzione

In questo capitolo vengono illustrate le scelte comuni che costituiscono la base della verifica sperimentale delle metodologie proposte. L'obiettivo è fornire un quadro chiaro e coerente delle strategie condivise e specifiche adottate per ciascun caso di studio. Verranno descritte la pipeline sperimentale condivisa: progettazione, pipeline di preprocessing, suddivisione dei dati e validazione, metriche utilizzate, valutazione e studio di ablazione dei modelli precedentemente definiti teoricamente.

La parte dedicata alla preparazione dei dati non è riportata qui in forma aggregata: per ciascun caso di studio è presente una sottosezione specifica che documenta le scelte di Data Preparation e le motivazioni sperimentali. Per ogni caso saranno inoltre presentate le architetture testate, la procedura di tuning e validazione, i risultati numerici e grafici con intervalli di confidenza e le conclusioni interpretative.

Il capitolo comprende infine uno studio di ablazione trasversale, volto ad esplorare il compromesso tra compressione dei modelli e mantenimento

delle prestazioni; i risultati di questo studio vengono discussi e messi a confronto per facilitare considerazioni pratiche sul deployment. Le sezioni che seguono sviluppano in dettaglio quanto qui sintetizzato, distinguendo esplicitamente le componenti comuni da quelle specifiche di ciascun caso.

8.2 Progettazione dei casi di studio

8.2.1 Tecnologie e librerie

Il workflow sperimentale e di analisi è stato interamente sviluppato in Python, con un insieme di librerie comuni ai tre casi studio e componenti specifiche in base alla natura del problema affrontato.

Per la manipolazione dei dati si è fatto uso di **NumPy** e **pandas**, mentre la visualizzazione dei grafici è stata realizzata con **matplotlib** e **seaborn**. I moduli **json**, **os**, **inspect** e **copy** sono stati utilizzati per la gestione dei metadati, il salvataggio dei risultati e la serializzazione delle configurazioni. Le reti neurali e loro estensioni CNN sono state implementate in **PyTorch**. Nel terzo caso studio è stato inoltre utilizzato **torchvision** per le trasformazioni e la gestione delle immagini. La libreria **pykan** è stata integrata in tutti i casi studio per l'implementazione della Kolmogorov-Arnold Network, che si basa anch'essa su **PyTorch**. Sono stati adottati **scikit-learn** e **XGBoost** per gli approcci ensemble, le metriche, le pipeline di preprocessing e per l'ottimizzazione degli iperparametri. Nel secondo caso studio si è inoltre fatto uso di **imblearn** (SMOTE, ImbPipeline) per il bilanciamento delle classi. Inoltre, sono stati utilizzati **tqdm** per le progress bar, **logging** per il tracciamento degli esperimenti e librerie standard come **random**, **time**, **glob**, **zipfile**, **pathlib** e **PIL**. **Image** per la gestione dei dataset e delle immagini.

8.2.2 Ambienti di sviluppo e infrastruttura

La prototipazione ed il debug sono stati effettuati su **Google Colab**; le sperimentazioni su larga scala e l'addestramento intensivo dei modelli sono stati condotti sul **Cluster HPC** dell'Università di Bologna, con allocazione

di GPU tramite SLURM.

Per la riproducibilità delle dipendenze, sono stati creati ambienti virtuali dedicati (venv) ed un file `requirements.txt`. I risultati di ogni run, insieme a configurazioni ed iperparametri, sono stati serializzati in JSON, mentre i notebook eseguiti sul cluster sono stati automaticamente convertiti in HTML per la documentazione.

8.2.3 Linguaggi, scripting e automazione del workflow

Il linguaggio principale è Python. Le analisi esplorative e la reportistica sono state realizzate in Jupyter notebooks, mentre l'automazione degli esperimenti e la sottomissione su cluster sono state gestite tramite script bash integrati con SLURM.

La conversione automatica ed esecuzione dei notebook è stata gestita con `nbconvert`, così da produrre sia versioni aggiornate in formato notebook che report in HTML. Ogni esperimento ha registrato seed, configurazioni ed eventi salienti del training con logging strutturato, assicurando piena riproducibilità.

8.2.4 Script di sottomissione (Cluster GPU)

La struttura degli script SLURM è risultata uniforme tra i casi studio, con differenze solo nelle risorse richieste, nel nome del job e nel notebook eseguito. Un esempio generico è riportato di seguito:

```
#!/bin/bash
#SBATCH --job-name=Generic_Case
#SBATCH --mail-type=ALL
#SBATCH --mail-user=martin.tomassi@studio.unibo.it
#SBATCH --time=120:00:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=8
#SBATCH --mem=60G
```



```

#SBATCH --partition=l40
#SBATCH --output=output_jupyter_exec_job_%j.txt
#SBATCH --chdir=/scratch.hpc/martin.tomassi
#SBATCH --gres=gpu:1

export JUPYTER_CONFIG_DIR=
    "/scratch.hpc/martin.tomassi/jupyter_config_${SLURM_JOB_ID}"
export MPLCONFIGDIR=
    "/scratch.hpc/martin.tomassi/matplotlib_cache_${SLURM_JOB_ID}"
mkdir -p "$JUPYTER_CONFIG_DIR"
mkdir -p "$MPLCONFIGDIR"

source venv_case/bin/activate
jupyter nbconvert --to notebook
    --execute case_notebook.ipynb
    --output case_notebook_trained.ipynb
jupyter nbconvert --to html case_notebook_trained.ipynb
deactivate

rm -rf "$JUPYTER_CONFIG_DIR"
rm -rf "$MPLCONFIGDIR"

```

8.2.5 Scelte architetturali ed iperparametri

Le scelte architetturali e gli iperparametri finali sono stati specifici per ciascun caso studio. Le configurazioni dettagliate, incluse nelle tabelle dei singoli capitoli, documentano gli iperparametri selezionati dopo la fase di ottimizzazione. In tutti i casi, si è fatto ricorso a *Early stopping* per migliorare la generalizzazione.

8.3 Addestramento dei modelli

In tutti i casi studio considerati è stata adottata una pipeline di addestramento fondamentalmente omogenea che mira a garantire comparabilità sperimentale tra le architetture.

8.3.1 Pipeline di preprocessing

Le trasformazioni iniziali applicate alle variabili indipendenti sono state incapsulate in pipeline riutilizzabili (ad es. `ColumnTransformer`) per evitare data leakage e garantire riproducibilità. La selezione delle feature ha rimosso colonne non informative o ridondanti; le variabili categoriche sono state codificate con `OneHotEncoder` eseguito all'interno della pipeline; le feature numeriche sono state standardizzate tramite `StandardScaler`. I missing values sono stati trattati in modo coerente tra gli esperimenti: dove possibile si è preferita la rimozione controllata delle righe; si sottolinea che, sebbene `XGBoost` gestisca NaN nativamente, per confronto omogeneo i NaN sono stati generalmente eliminati. Per i dataset di immagini il preprocessing include resize e crop centrato a 224×224 , normalizzazione canale-per-canale e data augmentation (flip orizzontale, rotazioni lievi, jitter di luminosità/contrasto) applicata esclusivamente in fase di training. Per problemi di classificazione sbilanciata si è fatto ricorso a tecniche di bilanciamento approcciate esclusivamente sul training fold (`SMOTE` per dati tabellari, pesi di classe o `WeightedRandomSampler` per immagini).

8.3.2 Suddivisione dei dati e validazione

Il dataset è stato suddiviso in test set indipendente (circa 20%) e training pool (circa 80%). Per i dataset non temporali si è impiegata una Nested Cross-Validation con outer `KFold` = 5 e inner `KFold` = 3 in combinazione con Random Search per la selezione degli iperparametri e la stima della performance generalizzata. Per i dataset temporali si è usata una Time Series Cross Validation che rispetta l'ordine cronologico delle osservazioni.

8.3.3 Metriche e intervalli di confidenza

Le metriche sono state scelte in funzione del tipo di problema. Per i compiti di regressione, sono state utilizzate:

- **MSE** (Mean Squared Error): $MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$.
- **MAE** (Mean Absolute Error): $MAE = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$.
- **MAPE** (Mean Absolute Percentage Error): $MAPE = \frac{100}{n} \sum_{i=1}^n \left| \frac{y_i - \hat{y}_i}{y_i} \right|$, calcolata escludendo i casi con $y_i = 0$ o valori non finiti.
- **R² e R²-adjusted**:

$$R^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}, \quad R^2_{adj} = 1 - (1 - R^2) \frac{n - 1}{n - k - 1},$$

dove n è il numero di osservazioni, k il numero di predittori, \bar{y} è la media dei valori reali, y_i rappresenta il valore reale (ground truth) e \hat{y}_i rappresenta il valore predetto da un modello.

- **Max Error**: $\max_i |y_i - \hat{y}_i|$.

Per i compiti di classificazione, sono state considerate:

- **Accuracy**:

$$Accuracy = \frac{\text{Numero di predizioni corrette}}{\text{Numero totale di predizioni}} = \frac{\sum_{i=1}^N \mathbb{I}(y_i = \hat{y}_i)}{N}$$

dove y_i è il valore vero, \hat{y}_i è il valore predetto e $\mathbb{I}(\cdot)$ è la funzione indicatrice.

- **Precisione (P) e Recall (R)**:

$$P = \frac{TP}{TP + FP}, \quad R = \frac{TP}{TP + FN}$$

dove TP = True Positives, FP = False Positives, FN = False Negatives.

- **F1-score:** basato su Precisione e Recall della classe specifica.

$$F1 = 2 \cdot \frac{P \cdot R}{P + R}$$

- **F1-score macro e weighted:**

$$F1_{\text{macro}} = \frac{1}{C} \sum_{c=1}^C F1_c$$

$$F1_{\text{weighted}} = \sum_{c=1}^C F1_c \cdot \frac{N_c}{N}$$

dove C è il numero di classi, $F1_c$ è l'F1-score della classe c , N_c è il numero di istanze della classe c , e N è il numero totale di istanze.

- **Confusion Matrix:** la matrice di confusione è una tabella che riassume la performance di un modello di classificazione. Per un problema di classificazione binaria, la sua struttura è:

	Predetto	
	Positivo	Negativo
Reale		
Positivo	TP	FN
Negativo	FP	TN

dove TP = True Positives, FP = False Positives, FN = False Negatives, TN = True Negatives.

- **AUC-ROC weighted:** l'area sotto la curva ROC (Receiver Operating Characteristic), che traccia il True Positive Rate (TPR) contro il False Positive Rate (FPR) per diverse soglie.

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad \text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

L'AUC-ROC weighted è la media pesata dell'AUC di ogni classe.

- **AUC-PR weighted:** l'area sotto la curva Precision-Recall, che traccia la Precisione in funzione del Recall.

$$\text{AUC-PR} = \int_0^1 \text{Precision}(\text{Recall}) d(\text{Recall})$$

L'AUC-PR weighted è la media pesata dell'AUC-PR di ogni classe.

Intervalli di confidenza Per tutte le metriche principali è stata stimata l'incertezza statistica tramite bootstrap, un metodo di ricampionamento che approssima la distribuzione empirica delle metriche. La procedura consiste nel generare B campioni bootstrap S_1, S_2, \dots, S_B estraendo con ripetizione N elementi dal test set S ; su ciascun campione viene calcolata la metrica $M(S_i)$. L'intervallo di confidenza al livello $(1 - \alpha)$ si ottiene quindi dai percentili della distribuzione delle metriche:

$$\text{CI}_{(1-\alpha)} = (\text{Percentile}_{\alpha/2}(M), \text{Percentile}_{1-\alpha/2}(M)) .$$

Nel presente lavoro sono riportati intervalli al 95%, calcolati in modo uniforme in tutti i casi studio per le seguenti metriche:

- Regressione: R^2 , MSE, MAE, MAPE;
- Classificazione: Accuracy, F1 macro, F1 weighted, AUC-ROC weighted, AUC-PR weighted.

Complessità dei modelli Oltre alle metriche predittive, per tutti i casi studio è stata calcolata la complessità dei modelli:

- Per le KAN, la formula per calcolare la sua complessità è:

$$P_{\text{KAN}} = \sum_{i=0}^{L-1} (N_i \cdot N_{i+1}) \cdot (G + k + 3) + N_{i+1}$$

dove L è il numero di strati, N_i indica le dimensioni dello strato i , G è la dimensione della griglia e k è l'ordine della spline.

- Per le MLP, la formula è:

$$P_{\text{MLP}} = \sum_{i=1}^{L-1} N_i \cdot N_{i+1} + N_{i+1}$$

- Per Random Forest e XGBoost, la complessità è il numero totale di nodi in tutti gli alberi dell'ensemble.

$$\text{Complessità}_{\text{Albero}} = \sum_{i=1}^T \text{Nodi}_i$$

dove T è il numero totale di alberi nell'ensemble e Nodi_i è il numero di nodi nell'albero i .

8.4 Valutazione dei modelli

La procedura di valutazione adottata in tutti e tre i casi di studio è organizzata in tre fasi principali: (1) analisi qualitativa e quantitativa, (2) selezione del miglior modello e (3) conclusione finale. Le fasi sono pensate per essere applicabili, in modo coerente, a problemi di regressione e classificazione.

8.4.1 Fase 1: Analisi qualitativa e quantitativa integrata

La prima fase integra l'analisi visiva dei barplot e l'analisi numerica delle metriche, con l'obiettivo di ottenere in un unico passaggio una panoramica iniziale delle prestazioni comparative dei modelli, evidenziando quelli che si sono dimostrati più performanti e quelli che presentano limitazioni. Per ciascun modello si selezionano le configurazioni iperparametriche ottimali risultanti dalla fase di tuning e si generano i grafici per le metriche rilevanti del task. I barplot consentono un confronto visivo immediato tra modelli, evidenziando quale soluzione appare dominante per metriche da massimizzare o da minimizzare. Parallelamente, per le stesse configurazioni ottimali si riportano i valori numerici delle metriche calcolate sul test set,

corredati da intervalli di confidenza (tipicamente CI95%). Questo approccio integrato combina la rapidità interpretativa dei grafici con la solidità delle stime numeriche, permettendo di identificare rapidamente i modelli più promettenti, ed al contempo di validare le impressioni visive con stime oggettive e replicabili. Qualsiasi segnale visivo di instabilità o possibile overfitting individuato nei barplot viene verificato sui valori calcolati.

8.4.2 Fase 2: Selezione del miglior modello

Sulla base delle evidenze raccolte nella fase di analisi qualitativa e quantitativa, la selezione finale del modello preferibile viene condotta mediante una procedura di ranking multi-criterio che incorpora in modo esplicito sia le prestazioni sia la complessità.

Per prima cosa si definisce l'insieme delle metriche rilevanti per il task e la direzione di ottimizzazione per ciascuna.

```
regression_metrics = {
    'MSE_Test'           : 'min',
    'MAE_Test'           : 'min',
    'MAPE_Test'          : 'min',
    'R2_Test'            : 'max',
    'R2_Adjusted_Test'   : 'max'
}

classification_metrics = {
    'Accuracy_Test': 'max',
    'F1_Weighted_Test': 'max',
    'F1_Macro_Test': 'max',
    'AUC_ROC_OVR_Weighted': 'max',
    'AUC_PR_OVR_Weighted': 'max'
}
```

Per ogni metrica e per ciascun modello si calcola un rank (rank = 1 corrisponde allo score migliore). Il `performance_score` di un modello è la media aritmetica dei rank sulle metriche di interesse:

$$\text{performance_score}_m = \frac{1}{M} \sum_{j=1}^M \text{rank}_{m,j},$$

dove M è il numero di metriche considerate e $\text{rank}_{m,j}$ è il rank del modello m sulla metrica j .

In parallelo si ottiene un indicatore di complessità, il `complexity_rank`, calcolato ordinando i modelli in base alla formula di complessità scelta (definita precedentemente); anche in questo caso rank 1 indica il modello meno complesso. Per rendere confrontabili i due contributi, è utile normalizzare gli score su una scala comune. Una normalizzazione semplice e robusta consiste nel trasformare ogni score s_m in un valore normalizzato nell'intervallo $[0, 1]$ con la formula:

$$\tilde{s}_m = \frac{s_m - \min_m s_m}{\max_m s_m - \min_m s_m},$$

applicata separatamente a `performance_score` ed a `complexity_rank`.

L'aggregazione finale esplora più strategie operative per bilanciare performance e complessità. Nella strategia Equal Weight il punteggio aggregato è la somma diretta delle componenti normalizzate:

$$\text{agg}_m^{(1:1)} = \tilde{p}_m + \tilde{c}_m,$$

dove \tilde{p}_m rappresenta il punteggio di performance (`performance_score`) normalizzato del modello m , mentre \tilde{c}_m rappresenta il punteggio di complessità (`complexity_rank`) normalizzato del modello m .

Nella strategia Complexity Weighted la complessità è maggiormente penalizzata tramite un fattore di scala w_c (ad esempio $w_c = 2$ nella variante 1:2 e $w_c = 3$ nella variante extreme 1:3):

$$\text{agg}_m^{(1:w_c)} = \tilde{p}_m + w_c \cdot \tilde{c}_m.$$

Nel Pareto Approach le due componenti vengono normalizzate e combinate con pesi specifici (in questo contesto, 0.4 per la performance e 0.6 per la complessità).

Nel caso in cui ci siano dei punteggi pari, viene scelto il modello con minore complessità normalizzata.

I risultati della procedura vengono documentati con una tabella riassuntiva che riporta per ciascun modello: numero totale di parametri/nodi, `performance_score`, `complexity_rank` e punteggi aggregati per ciascuna strategia. Inoltre, viene mostrata la classifica dei migliori modelli basata sulla strategia Complexity Weighted.

Di seguito, il codice responsabile del ranking multi-criterio:

```
df_ranks = results_df.set_index('Model')
ranks = pd.DataFrame(index=df_ranks.index)

for metric, direction in metrics.items():
    if direction == 'max':
        ranks[f'{metric}_rank'] = df_ranks[metric]
        .rank(ascending=False, method='average')
    elif direction == 'min':
        ranks[f'{metric}_rank'] = df_ranks[metric]
        .rank(ascending=True, method='average')

ranks['Complexity_rank'] = df_ranks['Parameters']
    .rank(ascending=True, method='average')

performance_cols = [col for col in ranks.columns
    if col.endswith('_rank') and col != 'Complexity_rank']
ranks['performance_score'] = ranks[performance_cols]
    .mean(axis=1)

ranks['equal_weight_score'] = ranks['performance_score'] +
    ranks['Complexity_rank']
```

```

ranks['complexity_weighted_score'] = ranks['performance_score'] +
    (2 * ranks['Complexity_rank'])

ranks['extreme_complexity_score'] = ranks['performance_score'] +
    (3 * ranks['Complexity_rank'])

performance_norm = (
    ranks['performance_score'] - ranks['performance_score'].min()) /
    (ranks['performance_score'].max() - ranks['performance_score'].min())
complexity_norm = (
    ranks['Complexity_rank'] - ranks['Complexity_rank'].min()) /
    (ranks['Complexity_rank'].max() - ranks['Complexity_rank'].min())
ranks['pareto_score'] = 0.4 * performance_norm +
    0.6 * complexity_norm

methods = {
    'Equal Weight (1:1)': 'equal_weight_score',
    'Complexity Weighted (1:2)': 'complexity_weighted_score',
    'Extreme Complexity (1:3)': 'extreme_complexity_score',
    'Pareto Approach (40:60)': 'pareto_score'
}

results_summary = pd.DataFrame(index=df_ranks.index)
results_summary['Performance_Score'] = ranks['performance_score']
results_summary['Complexity_Rank'] = ranks['Complexity_rank']
results_summary['Parameters'] = df_ranks['Parameters']

print("---")
print("Best Models by Weighting Scheme")
print("---")
best_models_summary_data = {
    'Weighting Scheme': [],

```

```

    'Best Model(s)': []
}

for method_name, score_col in methods.items():
    best_model = ranks[score_col].idxmin()
    best_models_summary_data['Weighting Scheme'].append(method_name)
    best_models_summary_data['Best Model(s)'].append(best_model)

summary_df = pd.DataFrame(best_models_summary_data)
print(summary_df.to_markdown(index=False))

print("\n---")
print("Detailed Ranking Table")
print("---")

ranking_display = pd.DataFrame(index=df_ranks.index)
ranking_display['Parameters'] = df_ranks['Parameters']
    .astype(int)
ranking_display['Avg_Performance_Rank'] = ranks['performance_score']
    .round(2)
ranking_display['Complexity_Rank'] = ranks['Complexity_rank']
    .astype(int)

for method_name, score_col in methods.items():
    ranking_display[f'{method_name.split()[0]}_Rank'] = ranks[score_col]
        .rank().astype(int)

ranking_display_sorted = ranking_display.sort_values('Complexity_Rank')
print(ranking_display_sorted.to_markdown())

print("\n---")
print("Recommendation")
print("---")

```

```

recommended_model = ranks['complexity_weighted_score']
    .idxmin()
recommended_score = ranks.loc[
    recommended_model,
    'complexity_weighted_score'
]
recommended_params = df_ranks.loc[
    recommended_model,
    'Parameters'
]
recommended_mse = df_ranks.loc[
    recommended_model,
    'MSE_Test'
]

print(f"***RECOMMENDED MODEL:** {recommended_model}")
print(f"***Parameters:** {int(recommended_params):,}")
print(f"***MSE Test Score:** {recommended_mse:.4f}")
print(f"***Complexity-Weighted Rank Score:** {recommended_score:.3f}")

print(f"\n***TOP 3 MODELS** (Based on Complexity-Weighted Ranking):")
top_3 = ranks.sort_values('complexity_weighted_score').head(3)
for i, (model, row) in enumerate(top_3.iterrows(), 1):
    params = int(df_ranks.loc[model, 'Parameters'])
    mse_score = df_ranks.loc[model, 'MSE_Test']
    print(f" {i}. **{model}** |
          Parameters: {params:>8,} |
          MSE: {mse_score:.4f} |
          Score: {row['complexity_weighted_score']:.3f}")

```

8.4.3 Fase 3: Conclusione finale

La conclusione finale sintetizza le evidenze emerse dalla fase di analisi e dalla procedura di selezione. La sintesi riporta i punti di forza ed i limiti del modello selezionato, confronta i margini numerici rispetto ai concorrenti e valuta la sostenibilità operativa in funzione della complessità stimata. Si motivano le scelte effettuate mediante riferimenti chiari ai barplot ed alle tabelle con le metriche.

8.5 Studio di ablazione

L'obiettivo dello studio di ablazione è valutare il compromesso tra compressione del modello, intesa come numero di parametri attivi e rapporto di compressione, e mantenimento delle prestazioni predittive. Lo scopo finale è confrontare le prestazioni dei modelli alle quattro soglie di pruning prefissate (30%, 50%, 70%, 90%), analizzando per ciascuna soglia sia la retention delle metriche target sia il rapporto di compressione, al fine di identificare quale modello offra il miglior trade-off tra efficienza computazionale e qualità predittiva.

8.5.1 L1 pruning su MLP e KAN

La tecnica di pruning impiegata è il L1 post-training pruning, applicato ai pesi lineari della MLP ed ai coefficienti spline della KAN. Sono stati testati diversi pruning ratios, specificamente l'insieme {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95}. Ciascuna versione del modello potata è stata valutata, sia sul test set che sul train set, utilizzando le metriche definite precedentemente. Per ogni configurazione, sono stati calcolati il numero totale di parametri, il numero di parametri attivi post-pruning ed il rapporto di compressione. Per interpretare i risultati, sono state definite alcune definizioni operative. La baseline è il modello non potato (pruning ratio = 0.0). Il punto di degrado significativo è il primo pruning ratio che comporta una perdita relativa in R^2 /F1-Weighted superiore al 5% rispetto

alla baseline. Il best trade-off è il punto di massima compressione che causa una perdita relativa in R^2 /F1-Weighted inferiore o uguale al 2%. Il codice mostrato di seguito racchiude le principali funzioni che costituiscono il nucleo dello studio sull'ablazione.

```
class PruningAblationStudy:
    def __init__(self, device='cpu'):
        self.device = device
        self.pruning_results = []

    def get_model_sparsity(self, model):
        if hasattr(model, 'width') and hasattr(model, 'act_fun'):
            try:
                total_params = count_params(model)

                zero_params = 0
                for i in range(len(model.width) - 1):
                    if i < len(model.act_fun):
                        layer = model.act_fun[i]
                        if hasattr(layer, 'coef') and layer.coef is not None:
                            zero_params += float(torch.sum(layer.coef == 0))

                return zero_params / total_params
                if total_params > 0 else 0.0

            except Exception as e:
                print(f" Error calculating KAN sparsity: {e}")
                return 0.0

        else:
            zero_params = 0
            total_params = count_params(model)

            for module in model.modules():
```

```

    if isinstance(module, torch.nn.Linear):
        if hasattr(module, 'weight'):
            zero_params += float(torch.sum(module.weight == 0))
        if hasattr(module, 'bias') and module.bias is not None:
            zero_params += float(torch.sum(module.bias == 0))

    return zero_params / total_params
    if total_params > 0 else 0.0

def count_active_parameters(self, model):
    if hasattr(model, 'width') and hasattr(model, 'act_fun'):
        try:
            active_params = 0
            for i in range(len(model.width) - 1):
                if i < len(model.act_fun):
                    layer = model.act_fun[i]
                    if hasattr(layer, 'coef') and layer.coef is not None:
                        active_params += float(torch.sum(layer.coef != 0))

            return int(active_params)
        except:
            return count_params(model)

    else:
        active_params = 0
        for module in model.modules():
            if isinstance(module, torch.nn.Linear):
                if hasattr(module, 'weight'):
                    active_params += float(torch.sum(module.weight != 0))
                if hasattr(module, 'bias') and module.bias is not None:
                    active_params += float(torch.sum(module.bias != 0))

        return int(active_params)

```

```

def apply_l1_pruning(self, model, pruning_ratio):
    pruned_model = copy.deepcopy(model)

    if hasattr(model, 'width') and hasattr(model, 'act_fun'):
        try:
            if pruning_ratio == 0.0:
                return pruned_model

            kan_modules_to_prune = []
            for i in range(len(model.width) - 1):
                if i < len(model.act_fun):
                    layer = pruned_model.act_fun[i]
                    if hasattr(layer, 'coef') and layer.coef is not None:
                        temp_module = torch.nn.Linear(1, 1, bias=False)
                        temp_module.weight = torch.nn
                            .Parameter(layer.coef.view(-1, 1))
                        kan_modules_to_prune
                            .append((temp_module, 'weight'))

            if kan_modules_to_prune:
                prune.global_unstructured(
                    kan_modules_to_prune,
                    pruning_method=prune.L1Unstructured,
                    amount=pruning_ratio,
                )

            idx = 0
            for i in range(len(model.width) - 1):
                if i < len(model.act_fun):
                    layer = pruned_model.act_fun[i]
                    if hasattr(layer, 'coef') and layer.coef is not None:

```



```

        original_shape = layer.coef.shape
        mask = kan_modules_to_prune[idx][0]
            .weight_mask.view(original_shape)
        layer.coef.data = layer.coef.data * mask
        idx += 1

    for module, param_name in kan_modules_to_prune:
        prune.remove(module, param_name)

    print(f"  Applied L1 pruning to KAN
          with ratio: {pruning_ratio:.3f}")
    return pruned_model

except Exception as e:
    print(f"  Error during KAN L1 pruning: {e}")
    return model

else:
    modules_to_prune = []

for module in pruned_model.modules():
    if isinstance(module, torch.nn.Linear):
        modules_to_prune.append((module, 'weight'))
        if hasattr(module, 'bias') and module.bias is not None:
            modules_to_prune.append((module, 'bias'))

if modules_to_prune:
    prune.global_unstructured(
        modules_to_prune,
        pruning_method=prune.L1Unstructured,
        amount=pruning_ratio,
    )

```

```

    for module, param_name in modules_to_prune:
        prune.remove(module, param_name)

    print(f"    Applied L1 pruning to MLP
          with ratio: {pruning_ratio:.3f}")
    return pruned_model

def run_pruning_study(self, model, model_name,
    X_test, y_test, X_train, y_train, pruning_ratios):

    if hasattr(model, 'width') and hasattr(model, 'act_fun'):
        model_type = "KAN"
    else:
        model_type = "MLP"

    total_params = count_params(model)
    print(f"Total Parameters: {total_params:,}")

    for pruning_ratio in pruning_ratios:
        print(f"\nTesting {model_type} pruning ratio:
              {pruning_ratio:.4f}")

        if pruning_ratio == 0.0:
            pruned_model = model
            sparsity = 0.0
            active_params = total_params
        else:
            pruned_model = self
            .apply_l1_pruning(model, pruning_ratio)
            pruned_model.to(self.device)
            sparsity = self
            .get_model_sparsity(pruned_model)
            active_params = self

```

```

        .count_active_parameters(pruned_model)

metrics = self.evaluate_pruned_model(
    pruned_model, model_name, X_test, y_test, X_train, y_train
)

compression_ratio = total_params / active_params
if active_params > 0 else float('inf')

result = {
    'model_name': model_name,
    'model_type': model_type,
    'pruning_ratio': pruning_ratio,
    'sparsity': sparsity,
    'total_params': total_params,
    'active_params': active_params,
    'compression_ratio': compression_ratio,
    'metrics': metrics
}

self.pruning_results.append(result)

```

8.5.2 Ensemble pruning su Random Forest e XGBoost

La metodologia di pruning varia per i due modelli. Per il Random Forest, è stato utilizzato un Rank-Based Pruning: gli alberi sono stati ordinati per importanza e quelli meno rilevanti sono stati rimossi fino a raggiungere il `pruning_ratio` desiderato. Per XGBoost, si è optato per un Cumulative Pruning, mantenendo solo le prime iterazioni di boosting fino al numero corrispondente a $(1 - \text{pruning_ratio})$ della lunghezza originale, troncando di fatto la sequenza di boosting. Anche in questo caso, i pruning ratios testati sono stati 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95. Per ogni versione

potata, sono stati calcolati il numero totale e residuo di alberi, il rapporto di compressione e le metriche di performance su train e test set. Le definizioni operative di baseline, punto di degrado significativo e best trade-off sono rimaste le medesime.

Il codice mostrato di seguito racchiude le principali funzioni che costituiscono il nucleo dello studio sull'ablazione:

```
class EnsemblePruningAblationStudy:
    def __init__(self):
        self.pruning_results = []

    def rank_based_pruning_rf(self, rf_model, pruning_ratio):
        if pruning_ratio == 0.0:
            return rf_model, list(range(len(rf_model.estimators_)))

        tree_importances = []
        for i, tree in enumerate(rf_model.estimators_):
            tree_importance = np.sum(tree.feature_importances_)
            tree_importances.append((i, tree_importance))

        tree_importances.sort(key=lambda x: x[1], reverse=True)

        n_trees_to_keep = max(1,
                               int(len(rf_model.estimators_) * (1 - pruning_ratio)))

        selected_indices = [idx
                             for idx, _ in tree_importances[:n_trees_to_keep]]
        selected_indices.sort()

        pruned_rf = copy.deepcopy(rf_model)
        pruned_rf.estimators_ = [rf_model.estimators_[i]
                                 for i in selected_indices]
        pruned_rf.n_estimators = len(selected_indices)
```

```

return pruned_rf, selected_indices

def cumulative_pruning_xgb(self, xgb_model, pruning_ratio):
    if pruning_ratio == 0.0:
        return xgb_model, list(range(xgb_model.n_estimators))

    total_trees = xgb_model.n_estimators
    num_classes = xgb_model.n_classes_
    total_rounds = total_trees // num_classes
    keep_rounds = max(1, int(total_rounds * (1 - pruning_ratio)))
    n_keep = keep_rounds * num_classes

    pruned_model = copy.deepcopy(xgb_model)
    pruned_model.n_estimators = n_keep

    def predict_pruned(self, X):
        dmat = xgb.DMatrix(X)
        raw_predictions = self.get_booster()
            .predict(dmat, iteration_range=(0, n_keep))
        predicted_labels = np.argmax(raw_predictions, axis=1)
        return predicted_labels

    def predict_proba(self, X):
        dmat = xgb.DMatrix(X)
        raw_predictions = self.get_booster()
            .predict(dmat, iteration_range=(0, n_keep))
        proba = np.exp(raw_predictions) /
            np.sum(np.exp(raw_predictions), axis=1, keepdims=True)
        return proba

    pruned_model.predict = types
        .MethodType(predict_pruned, pruned_model)
    pruned_model.predict_proba = types

```

```

        .MethodType(predict_proba, pruned_model)

    selected_indices = list(range(n_keep))
    return pruned_model, selected_indices

def calculate_ensemble_sparsity(self, original_count, pruned_count):
    return (original_count - pruned_count) / original_count
        if original_count > 0 else 0.0

def run_rf_pruning_study(self, rf_model, model_name,
    X_test, y_test, X_train, y_train, pruning_ratios):
    print(f"\n=== Rank-Based Pruning Study for {model_name} ===")

    total_trees = len(rf_model.estimators_)
    print(f"Total Trees: {total_trees:,}")

    for pruning_ratio in pruning_ratios:
        print(f"\nTesting RF pruning ratio: {pruning_ratio:.2f}")

        pruned_model, selected_indices = self
            .rank_based_pruning_rf(rf_model, pruning_ratio)

        remaining_trees = len(selected_indices)
        sparsity = self
            .calculate_ensemble_sparsity(total_trees, remaining_trees)
        compression_ratio = total_trees / remaining_trees
            if remaining_trees > 0 else float('inf')

        metrics = self.evaluate_pruned_ensemble(
            pruned_model, model_name, X_test, y_test, X_train, y_train
        )

    result = {

```

```

        'model_name': model_name,
        'model_type': 'Random Forest',
        'pruning_method': 'Rank-Based',
        'pruning_ratio': pruning_ratio,
        'sparsity': sparsity,
        'total_trees': total_trees,
        'remaining_trees': remaining_trees,
        'compression_ratio': compression_ratio,
        'metrics': metrics
    }

    self.pruning_results.append(result)

def run_xgb_pruning_study(self, xgb_model, model_name,
    X_test, y_test, X_train, y_train, pruning_ratios):
    print(f"\n=== Cumulative Pruning Study for {model_name} ===")

    total_trees = xgb_model.n_estimators
    print(f"Total Trees: {total_trees:,}")
    print(f"Number of Classes: {xgb_model.n_classes_}")

    for pruning_ratio in pruning_ratios:
        print(f"\nTesting XGB pruning ratio: {pruning_ratio:.2f}")

        pruned_model, selected_indices = self
            .cumulative_pruning_xgb(xgb_model, pruning_ratio)

        remaining_trees = len(selected_indices)
        sparsity = self
            .calculate_ensemble_sparsity(total_trees, remaining_trees)
        compression_ratio = total_trees / remaining_trees
        if remaining_trees > 0 else float('inf')

```

```

metrics = self.evaluate_pruned_ensemble(
    pruned_model, model_name, X_test, y_test, X_train, y_train
)

result = {
    'model_name': model_name,
    'model_type': 'XGBoost',
    'pruning_method': 'Cumulative',
    'sparsity': sparsity,
    'pruning_ratio': pruning_ratio,
    'total_trees': total_trees,
    'remaining_trees': remaining_trees,
    'compression_ratio': compression_ratio,
    'metrics': metrics
}

self.pruning_results.append(result)

```

8.5.3 Confronto complessivo

Questa sezione si concentra sul confronto sistematico degli studi di ablazione eseguiti su ciascun modello, con l'obiettivo di identificare quale modello offra il miglior compromesso tra qualità predittiva e compressione. Il confronto viene effettuato sulle quattro soglie di pruning prefissate (30%, 50%, 70%, 90%), valutando per ciascuna soglia la retention delle metriche e il rapporto di compressione, così da evidenziare il modello più efficiente in termini di trade-off tra prestazioni e riduzione dei parametri.

Le informazioni principali vengono presentate in forma tabellare, riportando baseline delle metriche principali, best trade-off e compressione ottenuta per ciascun modello, consentendo un confronto rapido e chiaro delle prestazioni relative. L'analisi delle soglie tipiche permette inoltre di osservare trend di stabilità, degrado o miglioramento dei modelli, fornendo

indicazioni pratiche per la scelta del modello più adatto a scenari di deployment con vincoli di memoria o risorse computazionali.

```
def compare_all_pruning_methods(metrics):

    all_models_comparison = []

    for _, result in results_df.iterrows():
        if result['pruning_ratio'] in [0.0, 0.3, 0.5, 0.7, 0.9]:
            all_models_comparison.append({
                'Model': result['model_name'],
                'Type': 'Neural Network',
                'Pruning_Method': 'L1 Norm',
                'Pruning_Ratio': result['pruning_ratio'],
                'Metrics': result['metrics'],
                'Compression': result['compression_ratio'],
                'Components': f"{result['active_params']}/{result['total_params']}"
            })

    for _, result in ensemble_results_df.iterrows():
        if result['pruning_ratio'] in [0.0, 0.3, 0.5, 0.7, 0.9]:
            pruning_method = 'Rank-Based'
            if result['model_name'] == 'Random Forest' else 'Cumulative'
            all_models_comparison.append({
                'Model': result['model_name'],
                'Type': 'Ensemble',
                'Pruning_Method': pruning_method,
                'Pruning_Ratio': result['pruning_ratio'],
                'Metrics': result['metrics'],
                'Compression': result['compression_ratio'],
                'Trees': f"{result['remaining_trees']}/{result['total_trees']}"
            })
```

```

comparison_df = pd.DataFrame(all_models_comparison)

for m in metrics:
    pivot = comparison_df.pivot_table(
        values=m,
        index=['Model', 'Type', 'Pruning_Method'],
        columns='Pruning_Ratio',
        fill_value=np.nan
    )

    print(f"\n{m} Performance Across Pruning Levels:")
    print(pivot.round(4))

pruning_levels = [0.3, 0.5, 0.7, 0.9]

for pruning_level in pruning_levels:
    print(f"\n{'-'*60}")
    print(f"PERFORMANCE RETENTION AT {int(pruning_level*100)}% PRUNING")
    print(f"{'-'*60}")

    retention_summary = []
    for model in comparison_df['Model'].unique():
        model_data = comparison_df[comparison_df['Model'] == model]
        baseline = model_data[model_data['Pruning_Ratio'] == 0.0]
        pruned = model_data[model_data['Pruning_Ratio'] == pruning_level]

        if len(baseline) > 0 and len(pruned) > 0:
            baseline_metric = baseline.iloc[0][metrics[0]]
            pruned_metric = pruned.iloc[0][metrics[0]]
            retention = pruned_metric / baseline_metric
            if baseline_metric != 0 else 0

```

```

retention_summary.append({
    'Model': model,
    'Type': baseline.iloc[0]['Type'],
    'Method': baseline.iloc[0]['Pruning_Method'],
    f'Baseline_{metrics[0]}': baseline_metric,
    f'Pruned_{metrics[0]}': pruned_metric,
    'Retention': retention,
    'Compression': pruned.iloc[0]['Compression']
})

if retention_summary:
    retention_df = pd.DataFrame(retention_summary)
    .sort_values('Retention', ascending=False)
    print(retention_df.round(4))

best_model = retention_df.iloc[0]
print(f"\nBEST PRUNING METHOD AT {int(pruning_level*100)}% LEVEL:")
print(f"Model: {best_model['Model']} ({best_model['Type']})")
print(f"Method: {best_model['Method']}")
print(f"Performance Retention: {best_model['Retention']:.1%}")
print(f"Compression Achieved: {best_model['Compression']:.1f}x")

```

Capitolo 9

Primo Caso Studio: Regressione su emissioni di automobili

9.1 Introduzione

Il presente caso studio affronta la previsione delle emissioni di CO₂ di veicoli a partire da caratteristiche tecniche e di consumo (anno di produzione, cilindrata, tipo di trasmissione e carburante, consumo L/100km, ecc.). Il dataset utilizzato deriva dalla fusione di tre sorgenti ufficiali: i dati originali provengono dalla Vehicle Certification Agency (VCA) del Department for Transport del Regno Unito, dal sito ufficiale del Governo canadese e dal Instituto para la Diversificación y Ahorro de la Energía (IDAE), istituzione spagnola; il materiale è stato fornito in forma pre-elaborata da terze parti [41]. Per questo motivo, la trattazione qui riportata si concentra sulle fasi di modellazione, validazione ed analisi dei risultati, più che sulle operazioni di preprocessing. Inoltre, sono condotti studi di ablazione che valutano l'impatto di tecniche di pruning sui diversi tipi di modello per investigare il trade-off tra compressione e mantenimento della qualità predittiva.

Nel capitolo vengono presentati, in ordine: una sintetica nota sull'origine e lo stato del dataset; la pipeline di preprocessing e le scelte di split; la descrizione delle procedure di training e ottimizzazione; i risultati principali con le valutazioni comparative e gli esperimenti di ablazione con le considerazioni

finali sul compromesso performance/complessità.

9.2 Data preparation

9.2.1 Riassunto delle principali trasformazioni

Le principali trasformazioni applicate come descritto in [41]: l'unione delle tre sorgenti in un unico dataframe coerente; l'omogeneizzazione degli schemi attraverso la mappatura e la ridenominazione delle colonne per ottenere uno schema comune; la pulizia testuale con la standardizzazione e capitalizzazione dei valori e la rimozione di suffissi o annotazioni spurie. Inoltre, è stata effettuata una rimozione conservativa, utilizzando la mediana per riempire i valori mancanti numerici significativi, con eliminazioni conservative per categoria quando necessario. Infine, sono stati applicati dei filtri per eliminare righe prive della variabile target (CO2_Emissions) o prive di attributi ritenuti fondamentali per l'allenamento dei modelli.

9.3 Addestramento dei modelli

Il dataset finale comprende le seguenti features:

- Year: anno di produzione del veicolo (numerica);
- Manufacturer: casa automobilistica (categorica);
- Model: nome del modello (categorica);
- Engine_cm3: cilindrata in cm³ (numerica);
- Transmission_type: Automatic / Manual (categorica);
- Fuel_type: Petrol / Diesel / LPG / ... (categorica);
- Fuel_consumption: L/100km (numerica).

Per poter essere utilizzate negli algoritmi di machine learning, le variabili categoriche sono state codificate in formato numerico tramite **One-Hot Encoding**.

La variabile target da prevedere è CO2_Emissions (g/km).

9.3.1 Strategia di training comune e griglie di iperparametri

La strategia di training è stata mantenuta coerente per tutti i modelli considerati e si è basata su una Nested Cross-Validation, con un KFold esterno a 5 fold e uno interno a 3 fold. La configurazione che minimizzava la MSE di validazione interna è stata selezionata come ottimale per ogni outer split. Per le reti neurali, la procedura ha distinto i parametri di modello da quelli di addestramento, con training condotto tramite ottimizzatore Adam e funzione di perdita MSELoss. Il numero massimo di epoche è stato fissato a 1000, ma controllato da un meccanismo di Early Stopping per interrompere l'addestramento in caso di mancato miglioramento. Per i modelli ensemble, la logica è rimasta analoga: ogni configurazione di iperparametri è stata valutata nei fold interni e quella con la migliore media di MSE è stata riaddestrata sull'intero sottoinsieme interno e testata sull'outer fold corrispondente. I risultati finali di ciascun outer fold includono i migliori iperparametri individuati e le metriche di test, successivamente aggregati per la costruzione delle statistiche riassuntive e degli intervalli di confidenza.

```
def nested_random_search_neural(model_builder, param_dist, dataset,
    outer_folds=5, inner_folds=3, n_iter=10,
    early_patience=10, early_min_delta=1e-4):
    train_keys = ['lr', 'batch_size', 'l2_lambda']
    outer_cv = KFold(
        n_splits=outer_folds,
        shuffle=True,
        random_state=42)
    results = []
```

```

for train_idx, test_idx in outer_cv.split(
    range(len(dataset))):
    inner_train = Subset(dataset, train_idx)
    inner_test = Subset(dataset, test_idx)
    best_val_loss = float('inf')
    best_model_params, best_train_params = None, None

    for params in ParameterSampler(
        param_dist, n_iter=n_iter, random_state=42):
        model_params = {k: v for k, v in params.items()
            if k not in train_keys}
        train_params = {k: v for k, v in params.items()
            if k in train_keys}

        inner_cv = KFold(
            n_splits=inner_folds, shuffle=True, random_state=42)
        val_losses = []
        for subtrain_idx, val_idx in inner_cv.split(range(len(inner_train))):
            subtrain = Subset(inner_train, subtrain_idx)
            valset = Subset(inner_train, val_idx)
            train_loader = DataLoader(
                subtrain,
                batch_size=train_params['batch_size'],
                shuffle=True)
            val_loader = DataLoader(
                valset,
                batch_size=train_params['batch_size'],
                shuffle=False)

            model = model_builder(**model_params)
            if hasattr(model, 'speed'):
                model.speed()
            model.to(device)

```

```

optimizer = optim.Adam(
    model.parameters(),
    lr=train_params['lr'],
    weight_decay=train_params.get(
        'l2_lambda', 0.0)
)
stopper = EarlyStopper(
    patience=early_patience,
    min_delta=early_min_delta
)

for epoch in range(1000):
    train_epoch(
        model, train_loader,
        optimizer, nn.MSELoss(),
        l2_lambda=train_params.get(
            'l2_lambda', 0.0)
    )
    val_loss = eval_loss(model, val_loader, nn.MSELoss())
    if stopper.early_stop(val_loss):
        break

val_losses.append(eval_loss(model, val_loader, nn.MSELoss()))

mean_val = np.mean(val_losses)
if mean_val < best_val_loss:
    best_val_loss = mean_val
    best_model_params = model_params
    best_train_params = train_params

full_train_loader = DataLoader(
    inner_train,
    batch_size=best_train_params['batch_size'],

```



```

        shuffle=True)
test_loader = DataLoader(
    inner_test,
    batch_size=best_train_params['batch_size'],
    shuffle=False)

final_model = model_builder(**best_model_params)
if hasattr(final_model, 'speed'):
    final_model.speed()
final_model.to(device)

optimizer = optim.Adam(
    final_model.parameters(),
    lr=best_train_params['lr'],
    weight_decay=train_params.get(
        'l2_lambda', 0.0))
stopper = EarlyStopper(
    patience=early_patience,
    min_delta=early_min_delta)

for epoch in range(1000):
    train_epoch(final_model, full_train_loader, optimizer,
        nn.MSELoss(), l2_lambda=best_train_params.get(
            'l2_lambda', 0.0))
    if stopper.early_stop(eval_loss(
        final_model, full_train_loader, nn.MSELoss())):
        break

test_loss = eval_loss(
    final_model,
    test_loader,
    nn.MSELoss()
)

```

```

        results.append((
            best_model_params,
            best_train_params,
            test_loss)
        )

    return results

def nested_random_search_ensemble(model_builder, param_dist,
                                   X_data, y_data, outer_folds=5,
                                   inner_folds=3, n_iter=10):
    outer_cv = KFold(
        n_splits=outer_folds,
        shuffle=True,
        random_state=42)
    results = []

    for train_idx, test_idx in outer_cv.split(X_data):
        X_inner_train, X_inner_test =
            X_data.iloc[train_idx],
            X_data.iloc[test_idx]
        y_inner_train, y_inner_test =
            y_data.iloc[train_idx],
            y_data.iloc[test_idx]

        best_val_mse = float('inf')
        best_params = None

        for params in ParameterSampler(
            param_dist, n_iter=n_iter, random_state=42):
            inner_cv = KFold(
                n_splits=inner_folds,
                shuffle=True,

```

```

    random_state=42)
val_mses = []

for subtrain_idx, val_idx in inner_cv.split(X_inner_train):
    X_subtrain, X_val =
        X_inner_train.iloc[subtrain_idx],
        X_inner_train.iloc[val_idx]
    y_subtrain, y_val =
        y_inner_train.iloc[subtrain_idx],
        y_inner_train.iloc[val_idx]

    model = model_builder(**params)
    model.fit(X_subtrain, y_subtrain)
    val_pred = model.predict(X_val)
    val_mse = mean_squared_error(y_val, val_pred)
    val_mses.append(val_mse)

mean_val_mse = np.mean(val_mses)
if mean_val_mse < best_val_mse:
    best_val_mse = mean_val_mse
    best_params = params

final_model = model_builder(**best_params)
final_model.fit(X_inner_train, y_inner_train)
test_pred = final_model.predict(X_inner_test)
test_mse = mean_squared_error(y_inner_test, test_pred)

results.append((best_params, {}, test_mse))

return results

```

Di seguito sono riportate le griglie di iperparametri entro cui la Random

Search esplora le combinazioni, al fine di individuare quelle che forniscono i risultati migliori per ciascun modello.

MLP (Random Search: $n = 15$)

- `input_dim`: dimensionalità input
- `hidden_sizes`: [(32,32), (64,64), (128,)]
- `dropout`: [0.1, 0.2, 0.5]
- `lr`: [10^{-3} , 10^{-4}]
- `batch_size`: 32
- `l2_lambda`: [0, 10^{-5} , 10^{-4} , 10^{-3}]

KAN (Random Search: $n = 6$)

- `input_dim`: dimensionalità input
- `width`: [(8,4), (16,8)]
- `grid`: [5, 10]
- `k`: [2, 4]
- `seed`: 0
- `lr`: 10^{-3}
- `batch_size`: 32
- `l2_lambda`: [0, 10^{-5} , 10^{-4} , 10^{-3}]

Random Forest (Random Search: $n = 32$)

- `n_estimators`: [100, 200, 300, 500]
- `max_depth`: [10, 20, 30]
- `min_samples_split`: [10, 20, 30]

- `min_samples_leaf`: [5, 10]
- `max_features`: ['sqrt', 'log2']
- `random_state`: 42

XGBoost (Random Search: $n = 677$)

- `n_estimators`: [100, 200, 300]
- `max_depth`: [3, 4, 5, 6]
- `learning_rate`: [0.01, 0.05, 0.1]
- `subsample`: [0.7, 0.8, 0.9]
- `colsample_bytree`: [0.7, 0.8, 0.9]
- `reg_alpha`: [0.5, 1.0, 2.0]
- `reg_lambda`: [2.0, 5.0, 10.0]
- `random_state`: 42

9.3.2 Scelte architetturali finali

Nella Tabella 9.1 vengono mostrate le scelte finali, dopo l'ottimizzazione degli iperparametri, utilizzate per training, valutazioni comparative e studio di ablazione.

Tabella 9.1: Configurazioni finali dei modelli usati per il Training, dopo aver effettuato l'ottimizzazione degli iperparametri.

MLP	input_dim = 1048, hidden_sizes = (128,), dropout = 0.1; ottimizzatore: Adam; lr = 1e-3; l2_lambda = 1e-5; batch_size = 32. Early stopping applicato.
KAN	input_dim = 1048, width = (16,8), grid = 10, k = 4, seed = 0; ottimizzatore: Adam; lr = 1e-3; l2_lambda = 1e-4; batch_size = 32. Early stopping applicato.
Random Forest	n_estimators = 100, max_depth = 30, min_samples_split = 20, min_samples_leaf = 5, max_features = 'sqrt', random_state = 42.
XGBoost	n_estimators = 300, max_depth = 6, learning_rate = 0.1, subsample = 0.8, colsample_bytree = 0.7, reg_alpha = 2.0, reg_lambda = 2.0, random_state = 42.

9.4 Valutazione dei modelli

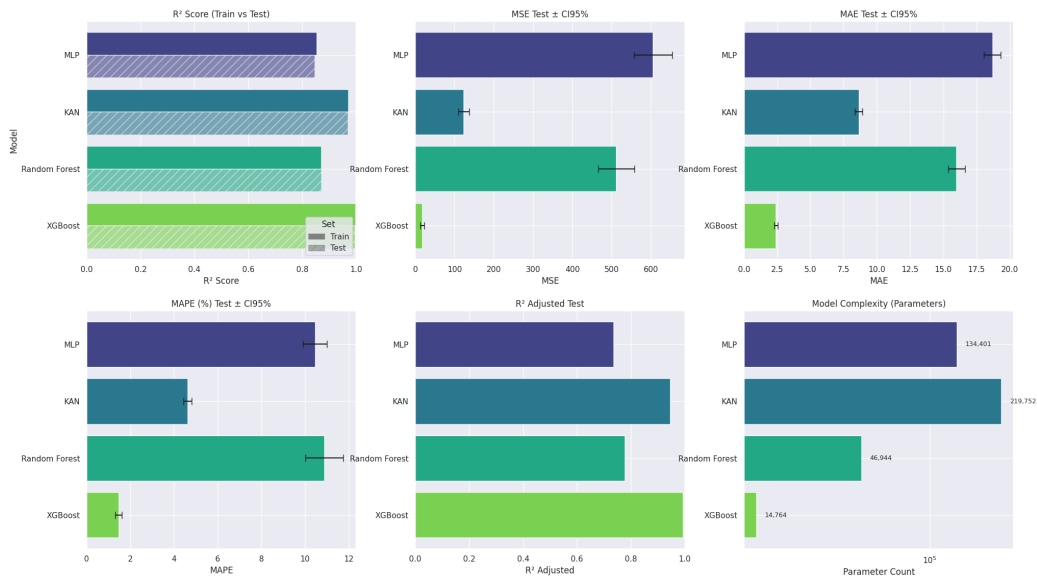


Figura 9.1: Confronto visivo della complessità e prestazioni dei modelli, utilizzando come metriche R^2 , MSE, MAE, MAPE e R^2 aggiustato.

Model	MSE Train (CI95%)	MSE Test (CI95%)	R^2 Train	R^2 Test	R^2 Adj. Test	MAE Test	MAPE Test (CI95%)	Max Error Test
MLP	589.72 (568.69 – 610.76)	605.28 (556.94 – 653.62)	0.8541	0.8467	0.8451	18.69	10.46 (9.91 – 11.01)	136.45
KAN	119.10 (112.34 – 125.86)	123.33 (110.11 – 136.54)	0.9705	0.9688	0.9687	8.63	4.63 (4.45 – 4.81)	108.35
RF	529.81 (510.10 – 549.51)	511.72 (465.44 – 558.00)	0.8717	0.8704	0.8654	15.98	10.89 (10.02 – 11.76)	134.95
XGBoost	12.42 (11.22 – 13.63)	17.52 (12.56 – 22.48)	0.9970	0.9956	0.9968	2.39	1.46 (1.31 – 1.62)	70.21

Tabella 9.2: Tabella riassuntiva delle prestazioni (MSE con CI95%, R^2 , MAE, MAPE con CI95%), Max Error e complessità.

9.4.1 Analisi dei risultati sperimentali

L'analisi dei risultati ha messo in evidenza differenze significative tra le reti neurali e i modelli ensemble, sia in termini di accuratezza sia di complessità computazionale.

Per quanto riguarda le reti neurali, la **MLP** ha ottenuto un R^2 in test pari a 0.8467 e un R^2 aggiustato di 0.8451, con un errore quadratico medio (MSE) di circa 605 e un errore assoluto medio (MAE) pari a 18.9. Il valore di MAPE si attesta attorno al 10.5%, mentre il massimo errore assoluto in test

ha raggiunto circa 136.5. Queste prestazioni sono accettabili ma non particolarmente competitive, soprattutto considerando il numero relativamente elevato di parametri del modello (circa 134k), che lo rende più pesante da addestrare e meno efficiente rispetto ad altre soluzioni. La **KAN** ha invece fornito risultati decisamente migliori, raggiungendo un R^2 test pari a 0.9688 e un R^2 aggiustato di 0.9687, con un MSE di 123 e un MAE di 9.0. Anche il MAPE, pari a circa 4.6%, conferma l'elevata accuratezza, mentre il massimo errore assoluto in test è stato di circa 108.3. Questo incremento di precisione è stato ottenuto al costo di una complessità ancora maggiore, pari a circa 220k parametri. Nel complesso, le reti neurali hanno mostrato un buon compromesso tra capacità di generalizzazione e accuratezza, ma con differenze marcate: l'MLP è risultato il meno competitivo, mentre KAN ha dimostrato una notevole capacità di modellare il problema, pur a fronte di un costo parametrico elevato.

I modelli ensemble hanno mostrato un comportamento altrettanto interessante. La **Random Forest** ha raggiunto un R^2 test pari a 0.8704 e un R^2 aggiustato di 0.8654, con MSE di 511 e MAE di 15.9. Il MAPE ha raggiunto un valore medio del 10.9%, mentre il massimo errore assoluto è stato di circa 134.9. Le sue prestazioni sono state quindi superiori a quelle dell'MLP, ma comunque meno accurate di quelle della KAN e soprattutto di XGBoost. La complessità del modello, stimata in circa 47k parametri, lo rende più leggero delle reti neurali, ma non sufficientemente performante da rappresentare la soluzione migliore. **XGBoost**, al contrario, si è distinto come il modello più efficace: il suo R^2 test ha raggiunto il valore di 0.9956 con un R^2 aggiustato di 0.9968, accompagnato da un MSE di appena 17.5 e un MAE pari a 2.5. Il MAPE medio in test si attesta intorno all'1.5%, e il massimo errore assoluto è stato limitato a circa 70.2, valori che lo rendono nettamente superiore agli altri modelli. Oltre all'elevata precisione, un ulteriore vantaggio di XGBoost è l'efficienza, poiché la sua complessità è di soli 15k parametri, rendendolo non solo il più accurato ma anche il più scalabile e pratico da utilizzare.

9.4.2 Selezione del miglior modello

I risultati hanno mostrato una chiara prevalenza del modello XGBoost su tutte le metriche ponderate. Nello specifico, XGBoost è risultato il modello vincente in Equal Weight (1:1), Complexity Weighted (1:2), Extreme Complexity (1:3) e Pareto Approach (40:60).

Di seguito la tabella riassuntiva con i valori principali usati per il ranking (valori ricavati dall'analisi finale):

Model	Param_Count	Perf_Score	Compl_Rank	Equal_Rank	Ext_Rank	Pareto_Rank
MLP	134,401	3.83	3	4	3	4
KAN	219,752	2.00	4	3	4	3
XGB	14,764	1.00	1	1	1	1
RF	46,944	3.17	2	2	2	2

Tabella 9.3: Riepilogo ranking: conteggio parametri, performance media (rank-based) e ranks per metodo di aggregazione.

9.4.3 Conclusioni

L'analisi ha portato ad individuare XGBoost come il più adatto al problema. Questo modello non solo ha ottenuto le migliori performance assolute su tutte le metriche di regressione, ma ha anche la complessità stimata più bassa, rendendolo la scelta ideale per un deployment operativo. La rete KAN ha raggiunto un buon compromesso in termini di performance, sebbene la sua complessità stimata sia risultata elevata in rapporto al beneficio predittivo assoluto. L'MLP, pur avendo il maggior numero di parametri di XGBoost, ha conseguito performance inferiori rispetto ad esso in termini di MSE e MAE. Il Random Forest ha fornito risultati stabili e prestazioni intermedie, ma non è riuscito a superare XGBoost nel compromesso tra performance e complessità.

La classifica dei tre migliori modelli, secondo il criterio *complexity-weighted* (dal migliore al peggiore), è la seguente:

1. XGBoost (Parametri: 14,764; MSE: 17.52)

2. Random Forest (Parametri: 46,944; MSE: 511.72)

3. MLP (Parametri: 134,401; MSE: 605.28)

9.5 Studio di ablazione

9.5.1 Ablation study: L1 pruning su MLP e KAN

Figura riassuntiva

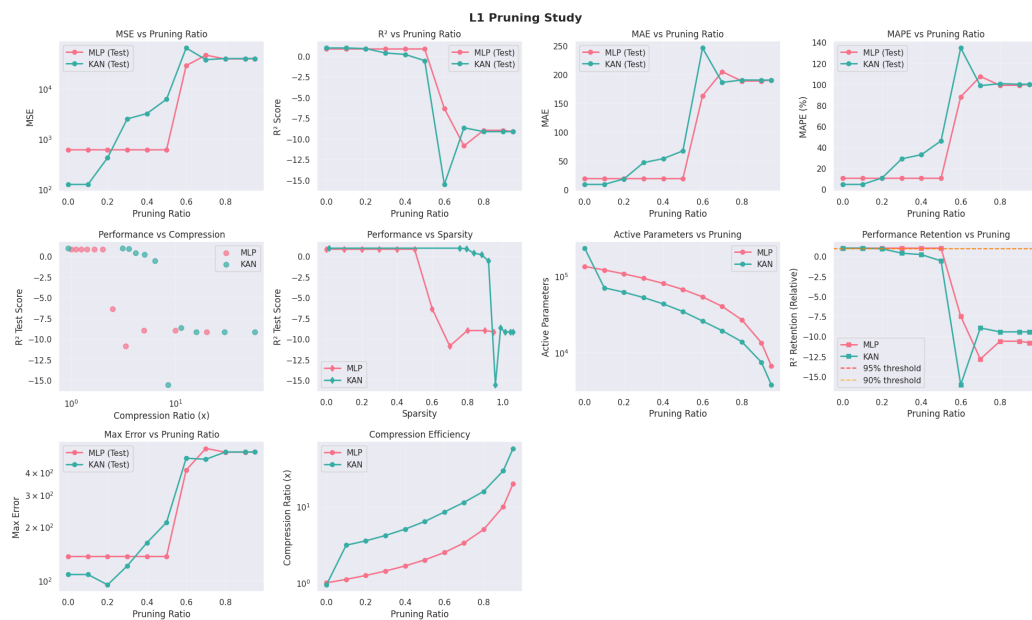


Figura 9.2: Risultati dello studio L1 pruning per MLP e KAN, utilizzando metriche principali e indicatori di compressione (MSE, R^2 , MAE, MAPE, max error, sparsità, parametri attivi, performance retention).

Risultati

Model	Total params	Baseline R^2	Best trade-off	Sign. degr.
MLP	134 401	0.8467	50% pruning (compression $\approx 2.0\times$)	60% pruning
KAN	219 752	0.9688	10% pruning (compression $\approx 3.1\times$)	20% pruning

Tabella 9.4: Riepilogo dei punti di trade-off e dei punti di degrado osservati nello studio L1 pruning.

La MLP ha una baseline $R^2 = 0.8467$. Fino a un pruning del 50%, non si osserva una perdita significativa in R^2 , rappresentando il miglior trade-off con una compressione di circa $2.0\times$. Il modello inizia a mostrare un degrado significativo intorno al 60% di pruning, dove MSE e MAE aumentano drasticamente e l' R^2 diventa negativo per pruning più aggressivi. La compressione massima sperimentata è di circa $20\times$ (pruning del 95%), ma a questo livello la performance è inutilizzabile.

La KAN, con una baseline $R^2 = 0.9688$, ha mostrato il suo miglior trade-off empirico con un pruning del 10% con una compressione di circa $3.1\times$, mantenendo la performance praticamente invariata. Il degrado significativo compare già a circa il 20% di pruning, oltre il quale MSE e MAE aumentano rapidamente. La massima compressione sperimentata è di circa $57.8\times$ (pruning del 95%), ma con una perdita di performance molto elevata.

9.5.2 Ablation study: ensemble pruning su Random Forest e XGBoost

Figura riassuntiva

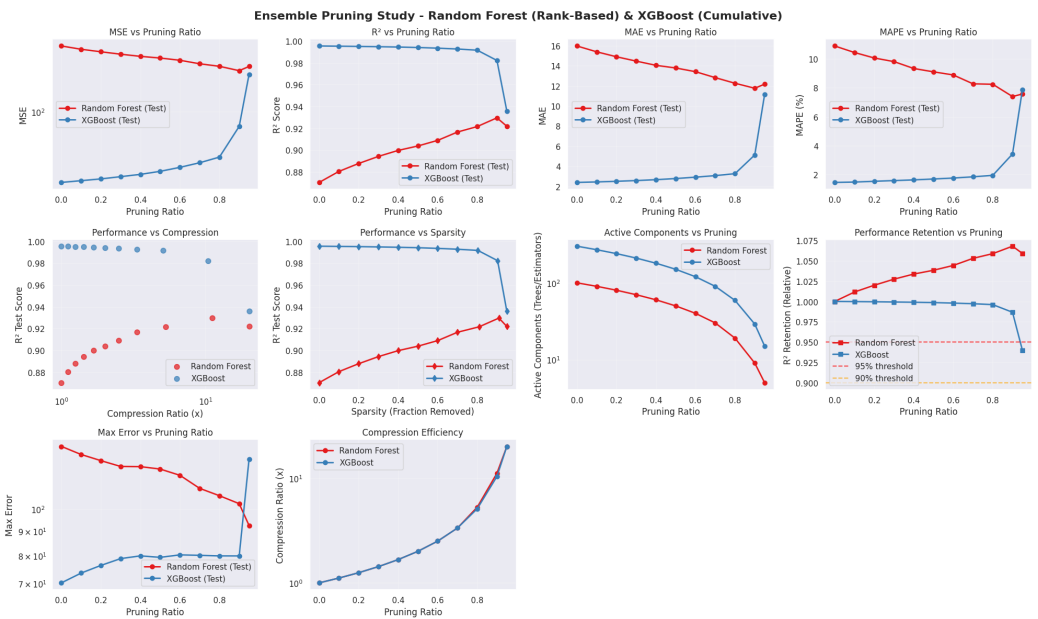


Figura 9.3: Risultati dello studio di pruning per Random Forest (rank-based) e XGBoost (cumulative), utilizzando metriche principali e indicatori di compressione (MSE, R^2 , MAE, MAPE, max error, sparsità, parametri attivi, performance retention).

Risultati

Model	Total components	Baseline (R^2)	Best trade-off	Sign. degr.
RF	100 trees	0.8704	95% (compression 20.0x)	no degr. rilevata
XGB	300 trees	0.9956	90% (compression 10.3x)	95% pruning

Tabella 9.5: Riepilogo sintetico dei punti di trade-off osservati per i due ensemble.

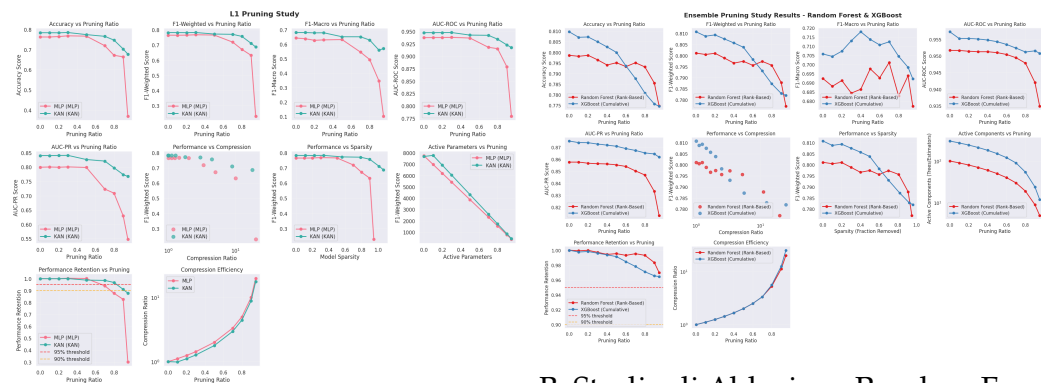
XGBoost ha dimostrato di essere molto robusto anche con pruning aggressivi, mantenendo una perdita trascurabile in R^2 fino ad una riduzione del

70-80% degli estimatori. Il miglior trade-off empirico si è verificato al 90% di pruning, mantenendo 29 alberi su 300, con una compressione di circa 10.3× ed una perdita relativa di R^2 di appena l'1.3%. Oltre questa soglia, le performance calano rapidamente.

Nel caso del Random Forest, il pruning rank-based ha mostrato una tendenza a mantenere o addirittura a migliorare leggermente la generalizzazione, riducendo l'overfitting. L'esperimento ha rivelato che il modello rimane stabile anche con pruning aggressivi: mantenendo solo 5 alberi su 100 (95% di pruning, compressione 20×), l' R^2 sul test set non peggiora e, in alcuni casi, migliora rispetto alla baseline.

9.5.3 Ablation study — Confronto complessivo (Neural Networks vs Ensemble)

Figure riassuntive



A: Studio di Ablazione MLP e KAN (L1 pruning).

B: Studio di Ablazione Random Forest (Rank-based) e XGBoost (cumulative).

Figura 9.4: Risultati sintetici degli studi di ablazione.

Riepilogo

Model	Baseline R^2	Best trade-off	Compression
XGB	0.9956	0.9823 @ 90% pr	$\approx 10.3\times$
RF	0.8704	0.9219 @ 95% pr	$\approx 20.0\times$
MLP	0.8467	0.8467 @ 50% pr	$\approx 2.0\times$
KAN	0.9688	0.9688 @ 10% pr	$\approx 3.1\times$

Tabella 9.6: Riepilogo sintetico dei principali punti di trade-off e compressione.

Confronto su soglie tipiche (30%, 50%, 70%, 90%)

Ad un pruning del 30%, Random Forest e XGBoost mantengono le prestazioni quasi invariate ($R^2 \approx 0.894$ e 0.995 rispettivamente), mentre la MLP rimane stabile ($R^2 = 0.8467$). La KAN, al contrario, degrada molto rapidamente ($R^2 \approx 0.364$).

Con un pruning del 50%, XGBoost conserva un $R^2 \approx 0.994$, e Random Forest migliora ulteriormente la generalizzazione ($R^2 \approx 0.904$). La MLP resta stabile, mentre la KAN crolla sotto lo 0.

Al 70% di pruning, XGBoost si dimostra ancora robusto ($R^2 \approx 0.993$, con una compressione di $3.3\times$). Il Random Forest continua a migliorare ($R^2 \approx 0.917$), mentre la MLP e la KAN subiscono un forte degrado, con un R^2 negativo. A un pruning del 90%, Random Forest mostra la migliore performance in termini di retention ($R^2 \approx 0.9296$) con una compressione di $11\times$. Anche XGBoost mantiene buone prestazioni ($R^2 \approx 0.9823$, compressione $10.3\times$), mentre MLP e KAN risultano completamente compromessi.

Conclusioni

1. XGBoost si conferma il modello più stabile sotto pruning aggressivo, mantenendo un $R^2 > 0.98$ ed una compressione di circa $10.3\times$ fino ad un pruning del 90%.

2. Per il Random Forest, il pruning rank-based porta ad un notevole miglioramento della generalizzazione, con un R^2 che sale a 0.9296 con un pruning del 90% ed una compressione di circa 11.1×.
3. La MLP tollera il pruning fino al 50% (compressione 2×) senza perdita di R^2 , ma degrada rapidamente oltre questa soglia.
4. La KAN è estremamente sensibile al pruning: già al 20% l' R^2 scende a circa 0.893, e crolla sotto lo 0 oltre il 40-50%.

Capitolo 10

Secondo Caso Studio: Classificazione di PM2.5

10.1 Introduzione

Il presente caso studio si occupa della classificazione dei livelli di PM2.5, polveri fini inquinanti, utilizzando un dataset nazionale di misurazioni orarie provenienti da 453 città indiane nel periodo 2010-2023, arricchito con variabili meteorologiche. I dati utilizzati sono stati forniti dal *Central Pollution Control Board* (CPCB), portale ufficiale del Governo indiano per il monitoraggio e il controllo dell'inquinamento, resi pubblicamente disponibili sul sito istituzionale (<https://cpcb.nic.in>). In questo caso di studio, le operazioni di caricamento, pulizia, gestione dei missing, individuazione degli outlier, ricampionamento ed aggregazione costituiscono parte integrante del workflow sperimentale e sono descritte nelle sezioni successive. Gli obiettivi del caso studio sono: (i) trasformare la previsione delle concentrazioni di PM2.5 in un problema di classificazione ordinata secondo le classi AQI (GOOD → HAZARDOUS) per garantire interpretabilità applicativa; (ii) confrontare i quattro modelli precedentemente definiti e valutare l'effetto delle scelte di preprocessing, del feature engineering temporale (lag, componenti cicliche), delle tecniche di bilanciamento e delle procedure di ottimizzazione sugli indicatori di performance. Per una stima robusta

della generalizzazione si utilizza una validazione con criterio temporale e si forniscono intervalli di confidenza per le principali metriche.

Nel capitolo vengono presentati, in sequenza: una nota sulle sorgenti e la composizione del dataset; la pipeline di preprocessing e le scelte di indicizzazione/aggregazione; le tecniche di feature engineering e la discretizzazione in classi AQI; la strategia di training e ottimizzazione; i risultati quantitativi (metriche aggregate con CI95% e confusion matrix); infine gli studi di ablazione e le considerazioni finali sul compromesso tra performance e complessità per scenari di deployment.

10.2 Data preparation

Questa sezione descrive in dettaglio le operazioni svolte per il caricamento, la normalizzazione, la pulizia e l'arricchimento del dataset utilizzato nel caso studio. Lo scopo principale della fase di Data preparation è trasformare i dati grezzi in una rappresentazione coerente, completa e utilizzabile per la successiva fase di allenamento.

10.2.1 Fonti e descrizione generale del dataset

Il dataset principale utilizzato nello studio raccoglie misurazioni relative alla qualità dell'aria in numerose stazioni di monitoraggio presenti in 453 città indiane per il periodo temporale 2010-2023. Le osservazioni includono sia concentrazioni di inquinanti sia variabili meteorologiche ed ambientali. Un file ausiliario, denominato `stations_info.csv`, contiene la mappatura tra i file contenenti le misure e informazioni di contesto (stato, città, agenzia responsabile, data di inizio rilevamento), permettendo in tal modo una gestione centralizzata dei metadati associati alle stazioni di monitoraggio.

10.2.2 Caricamento dati e organizzazione iniziale

Il caricamento del dataset è stato effettuato tramite la lettura dei file compressi estratti in una directory locale o su Google Colab. Per ogni file contenente

le misure di una singola stazione, sono state eseguite diverse operazioni. Innanzitutto, il contenuto di ciascun archivio ZIP è stato estratto in una directory strutturata per stato. In seguito, sono stati letti i metadati dal file `stations_info.csv` e sono state rimosse le colonne non necessarie per l'analisi, al fine di semplificare la tabella. Infine, è stato costruito un insieme aggregato: per ogni stato, sono stati individuati automaticamente tutti i file CSV con il prefisso identificativo dello stato, ogni file è stato letto e arricchito con le colonne "city" e "state", e tutte le tabelle sono state concatenate in un unico DataFrame nazionale.

Il risultato di questa fase è un DataFrame unico che contiene le misurazioni orarie con colonne per le concentrazioni degli inquinanti, le variabili meteorologiche e gli identificatori geografici.

10.2.3 Indicizzazione temporale

Nel dataset originale, le finestre temporali di misurazione sono definite dalle colonne "From Date" e "To Date". Per semplificare la gestione delle serie storiche, la colonna "From Date" è stata convertita in tipo `datetime`, quindi rinominata in "datetime" ed impostata come indice temporale del DataFrame. La colonna "To Date", ridondante per l'analisi, è stata rimossa. Questa trasformazione permette di sfruttare le funzionalità native di raggruppamento e ricampionamento temporale offerte dalle librerie per la manipolazione delle serie storiche.

10.2.4 Riduzione e unificazione di feature ridondanti

Durante l'esplorazione iniziale è emersa la presenza di colonne duplicate o varianti dello stesso nome, come "Ozone (ug/m3)" e "Ozone (ppb)". Per evitare rappresentazioni inconsistenti della stessa grandezza, sono state seguite alcune operazioni. In primo luogo, sono stati identificati i gruppi di colonne equivalenti tramite l'analisi grafica dei trend (andamento delle medie annue) ed il confronto delle statistiche di base. Successivamente, è stato definito un mapping di riduzione, ad esempio aggregando tutte le

varianti di Xilene in una singola colonna comune. Infine, per ogni gruppo, i valori non nulli provenienti dalle colonne secondarie sono stati trasferiti nella colonna principale, e le colonne ridondanti sono state eliminate. Il codice seguente mostra le funzioni usate per la riduzione e l'unificazione delle colonne ridondanti.

```
def plot_feature_similarities(
    dataframe, feature_groups, columns=2):
    rows = int((len(feature_groups)/columns)//1)
    fig, axes = plt.subplots(rows, columns, figsize=(13, 4*rows))
    fig.tight_layout(pad=3.0)

    row_num = 0
    col_num = 0
    for pos, group in enumerate(feature_groups):
        if pos % columns == 0 and pos != 0:
            row_num += 1
            col_num = 0

        for feature in feature_groups[group]:
            df_feature = dataframe[dataframe[feature].notnull()][feature]
            df_feature = df_feature.groupby([df_feature.index.year])
                .mean(numeric_only=True)
            sns.lineplot(
                data=df_feature,
                label=feature,
                ax=axes[row_num, col_num])
            axes[row_num, col_num].set_title(group)
            axes[row_num, col_num].set_xlabel=None)
            col_num += 1

    plt.plot()

groups = {
```

```

'Xylene': ['Xylene (ug/m3)', 'Xylene ()'],
'MP-Xylene': ['MP-Xylene (ug/m3)', 'MP-Xylene ()'],
'Wind Direction': ["WD (degree)", "WD (degree C)",
    "WD (deg)", "WD ()"],
'Ozone': ['Ozone (ug/m3)', 'Ozone (ppb)'],
'Nitrogen Oxides': ['NOx (ug/m3)', 'NOx (ppb)'],
'Relative humidity': ['RH (%)', 'RH ()'],
'Solar Radiation': ['SR (W/mt2)', 'SR ()'],
'Air Temperature': ['AT (degree C)', 'AT ()']
}

plot_feature_similarities(df_india, groups, columns=2)

reduction_groups = {
    "Xylene (ug/m3)": ["Xylene ()"],
    "MP-Xylene (ug/m3)": ["MP-Xylene ()"],
    "Benzene (ug/m3)": ["Benzene ()"],
    "Toluene (ug/m3)": ["Toluene ()"],
    "SO2 (ug/m3)": ["SO2 ()"],
    "NOx (ug/m3)": ["NOx (ppb)"],
    "Ozone (ug/m3)": ["Ozone (ppb)"],
    "AT (degree C)": ["AT ()"],
    "WD (degree)": ["WD (degree C)", "WD (deg)", "WD ()"],
    "WS (m/s)": ["WS ()"]
}

def merge_columns(dataframe, columns):
    for column, cols_to_merge in columns.items():
        if column not in dataframe.columns
            and any(name in dataframe.columns for name in cols_to_merge):
            dataframe[column] = np.nan

    for col_name in cols_to_merge:
        if col_name in dataframe.columns:
            dataframe[column] = dataframe[col_name]

```

```
        .fillna(dataframe[col_name])
    dataframe = dataframe.drop(columns=[col_name])

    return dataframe

df_india = merge_columns(df_india, reduction_groups)
```

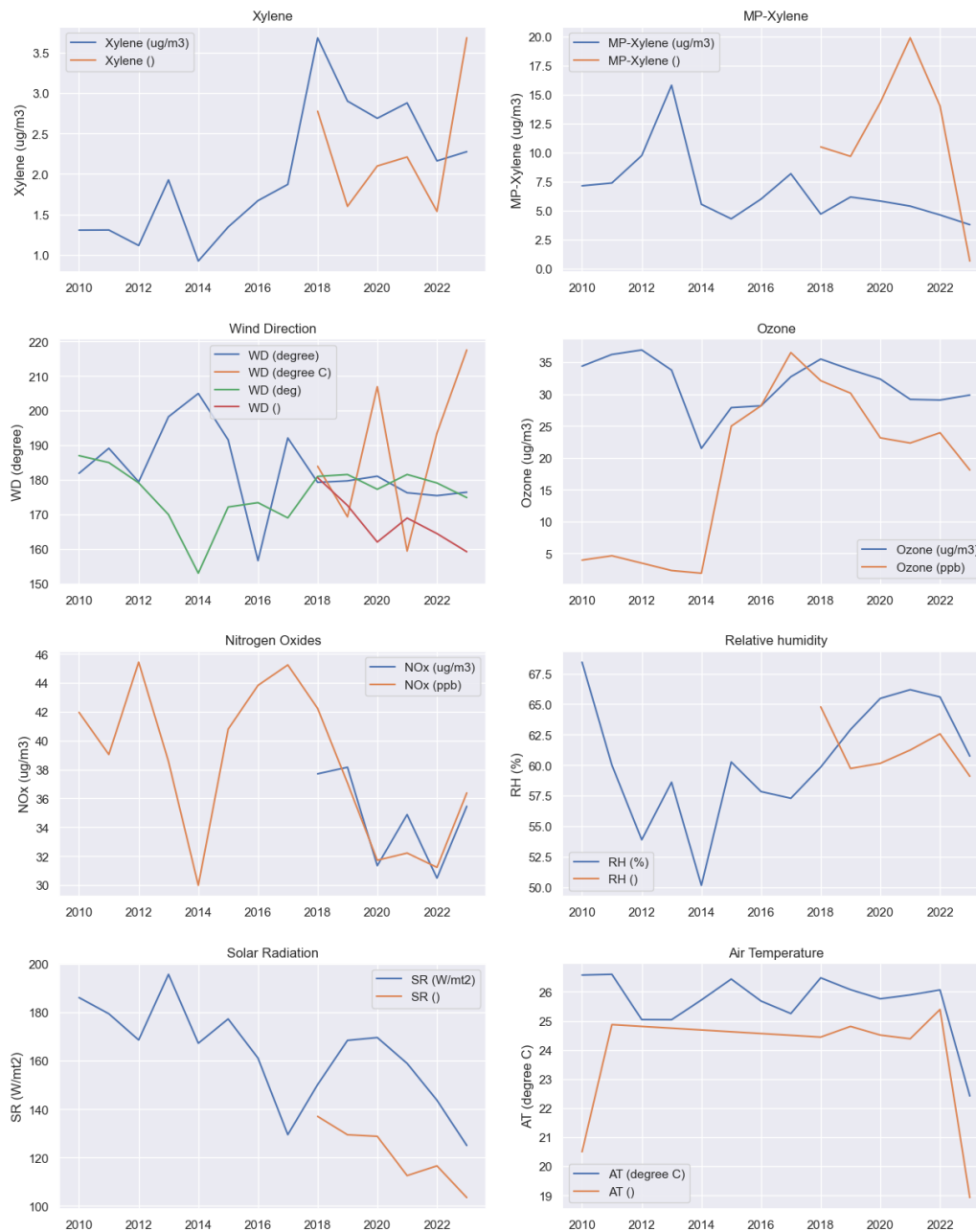


Figura 10.1: Similarità delle features - Analisi medie annue

10.2.5 Verifica e gestione dei valori mancanti

La quantificazione dei valori mancanti è stata effettuata calcolando il numero assoluto e la percentuale di missing per ogni variabile.

```
def get_null_info(dataframe):
    null_vals = dataframe.isnull().sum()

    df_null_vals = pd.concat(
        {'Null Count': null_vals,
         'Percentage of Missing Values (%)':
         round(null_vals * 100 / len(dataframe), 2)}, axis=1)

    return df_null_vals.sort_values(
        by=['Null Count'], ascending=False)

df_india_null_info = get_null_info(df_india)

plt.figure(figsize=(8, 10))
sns.barplot(
    data=df_india_null_info,
    x='Percentage of Missing Values (%)',
    y=df_india_null_info.index,
    orient='h',
    color='steelblue')
plt.show()
```


strategia è comunemente adottata nelle serie temporali, poiché preserva la dinamica locale dei segnali ed evita l'introduzione di discontinuità.

```
numeric_cols = df_india.select_dtypes(include='number')
    .columns
df_india[numeric_cols] = df_india[numeric_cols]
    .interpolate(method='pad')
df_india[numeric_cols] = df_india[numeric_cols]
    .fillna(df_india[numeric_cols].mean())
```

10.2.6 Analisi esplorativa e selezione delle feature rilevanti

Per esplorare le relazioni e le correlazioni tra le variabili, sono state eseguite diverse analisi. Sono state calcolate le medie su diverse frequenze temporali (giorno, mese, anno) per visualizzare i trend con grafici a linee. È stato utilizzato il pairplot per ispezionare le relazioni bivariate e le distribuzioni univariate. Inoltre, è stata costruita la matrice di correlazione e visualizzata tramite una heatmap per quantificare le correlazioni lineari, identificando come potenziali feature rilevanti quelle con una correlazione assoluta superiore a 0.4 con "PM2.5".

```
slice_groups = {
    'Group by Day': df_india.groupby(pd.Grouper(freq='1D'))
        .mean(numeric_only=True),
    'Group by Month': df_india.groupby(pd.Grouper(freq='1ME'))
        .mean(numeric_only=True),
    'Group by Year': df_india.groupby(pd.Grouper(freq='1YE'))
        .mean(numeric_only=True)
}

def plot_features_by_group(features, slice_groups):
    for feature in features:
        fig, ax = plt.subplots(1, 1, figsize=(12, 4))
        fig.suptitle(feature)
```

```

labels = []
for i, (group, group_df) in enumerate(slice_groups.items()):
    data_slice = group_df[
        group_df.columns.intersection(pollutants[feature])
    ]

    if feature == "Nitrogen Compounds":
        data_slice = data_slice.drop(
            ['NO (ug/m3)', 'NO2 (ug/m3)'], axis=1)

    data_slice.plot(kind="line", ax=ax)

    for column in data_slice.columns:
        labels.append(f'{column} [{group}]')

ax.set(xlabel=None)
ax.legend(labels)
plt.plot()
features_to_plot = [
    'Particulate Matter',
    'Carbon Monoxide',
    'Ozone Concentration',
    'Nitrogen Compounds']
plot_features_by_group(features_to_plot, slice_groups)
df_india.head()

sns.pairplot(slice_groups['Group by Month'])

corr = slice_groups['Group by Day']
    .corr(numeric_only=True).round(2)
mask = np.triu(np.ones_like(corr, dtype=bool))

```

```
plt.figure(figsize=(10,5))
sns.heatmap(
    data=corr,
    mask=mask,
    annot=True,
    cmap="rocket_r")
plt.show()
```

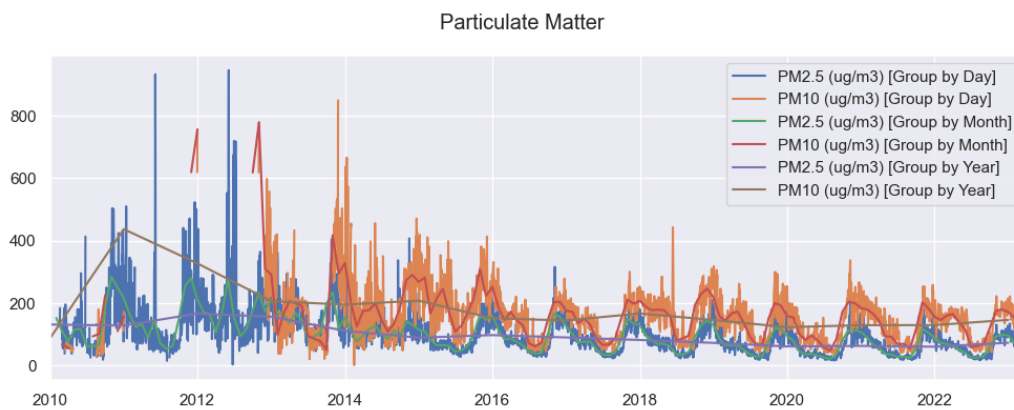


Figura 10.3: Analisi dei trend giornalieri, mensili e annuali per il Particolato.

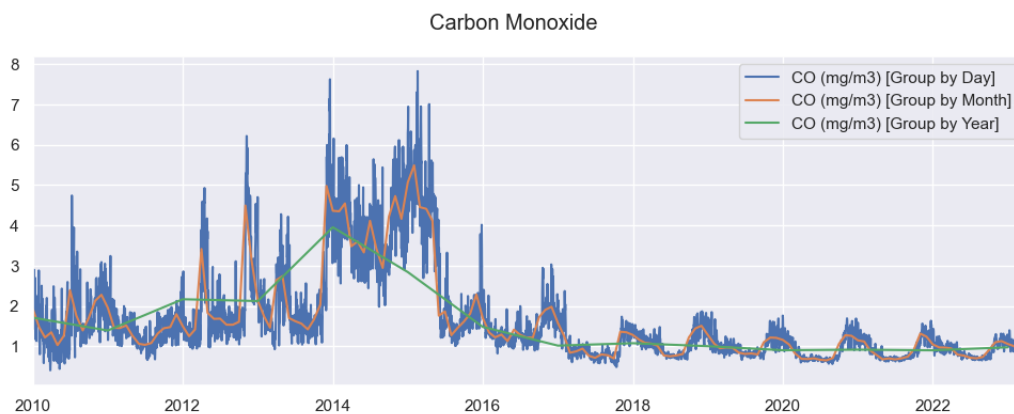


Figura 10.4: Analisi dei trend giornalieri, mensili e annuali per il Monossido di carbonio.

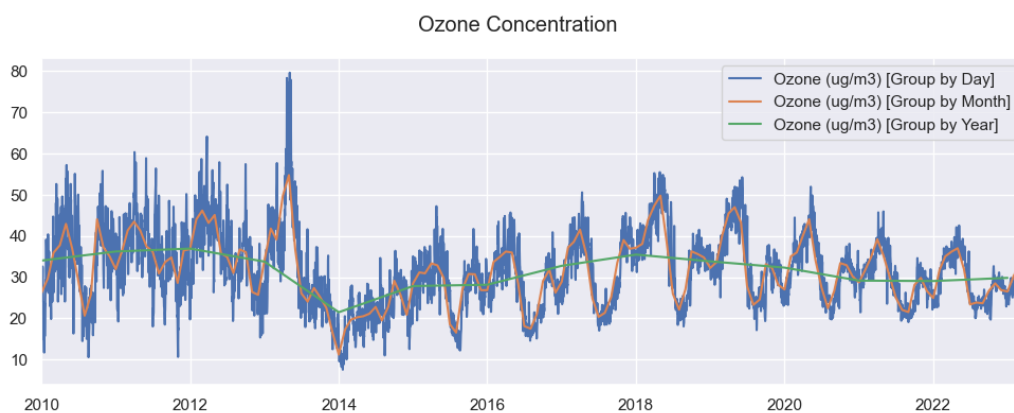


Figura 10.5: Analisi dei trend giornalieri, mensili e annuali per l'Ozono.

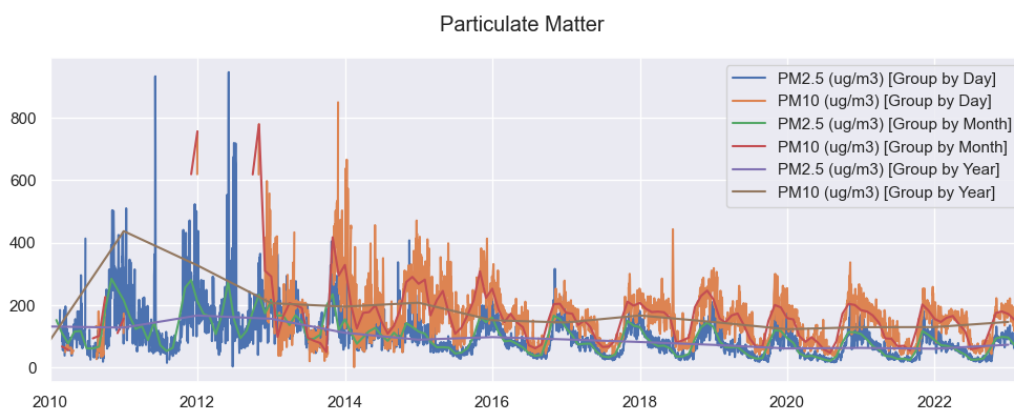


Figura 10.6: Analisi dei trend giornalieri, mensili e annuali per i Composti dell'azoto.

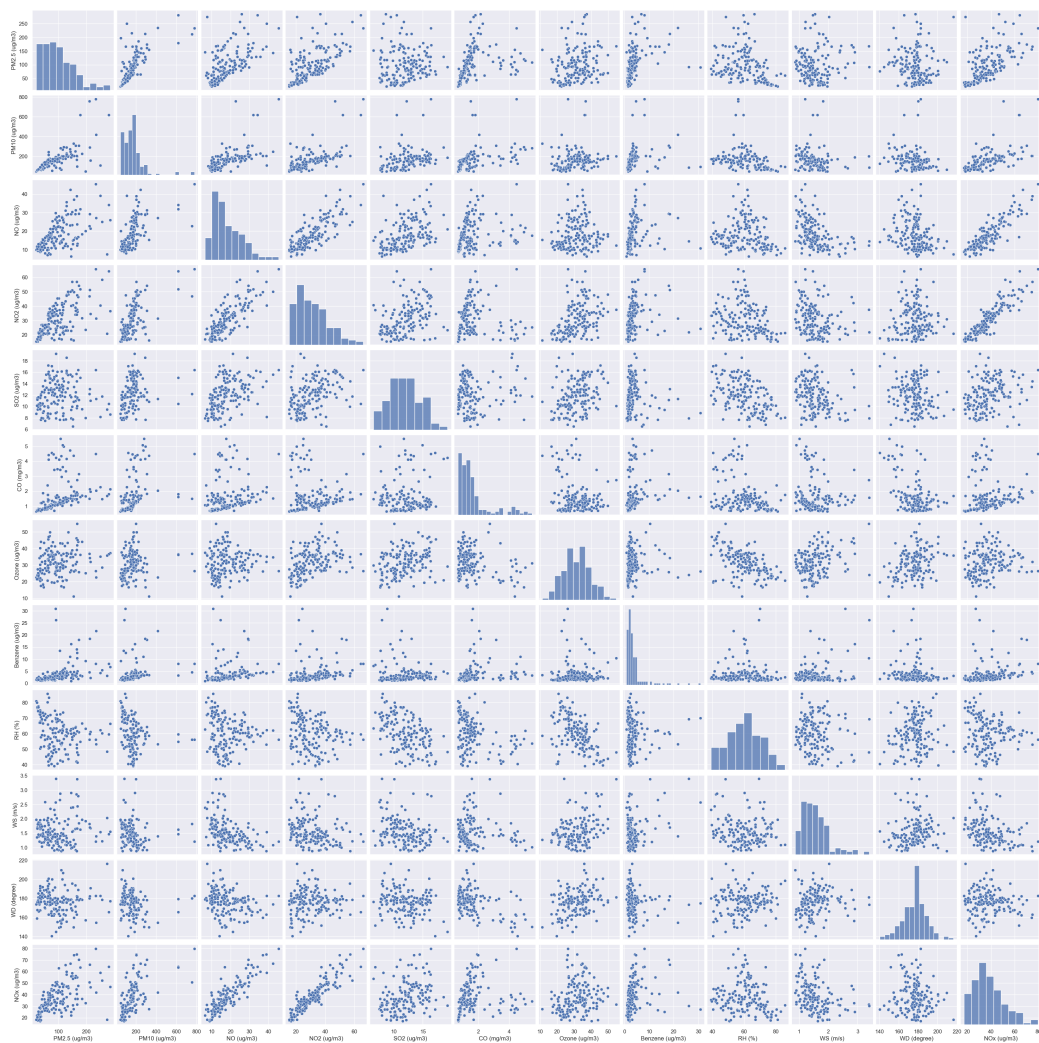


Figura 10.7: Analisi della relazione tra features e le loro distribuzioni univariate.

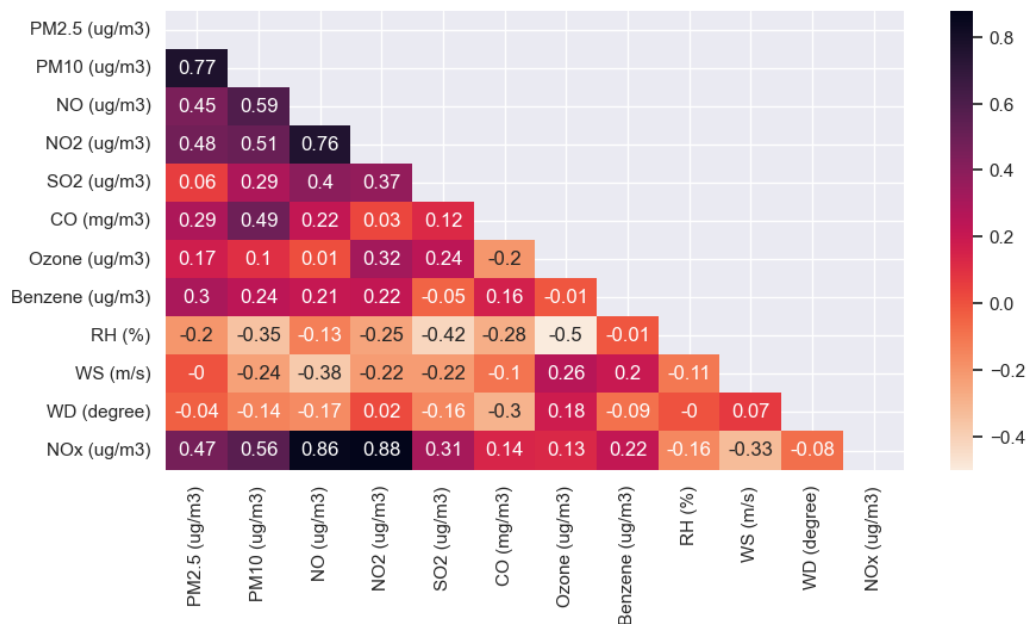


Figura 10.8: Analisi della correlazione tra variabili.

Dall'analisi, si é deciso di mantenere la variabile aggregata "NOx" e rimuovere le componenti ridondanti ("NO", "NO2") per evitare multicollinearità e semplificare i modelli successivi.

10.2.7 Ricampionamento ed aggregazione a livello statale

Poiché i dati aggregati includevano misurazioni provenienti da più stazioni all'interno dello stesso stato con lo stesso timestamp, si é deciso di ricampionare temporaneamente i dati a frequenza oraria e di aggregare le osservazioni a livello di stato tramite media aritmetica. La procedura implementata é la seguente:

```
df_resampled = (df_india
    .groupby('state')
    .resample('60min')
    .mean(numeric_only=True)
    .reset_index()
)
```

```
df_resampled = df_resampled.set_index('datetime')
df_india=df_resampled.copy()
```

10.2.8 Rilevamento e rimozione degli outlier

Per l'identificazione degli outlier nelle concentrazioni degli inquinanti, si è utilizzato l'algoritmo Isolation Forest, un metodo ensemble efficace per l'identificazione di valori anomali indipendentemente dalla distribuzione a priori dei dati [39]. I punti principali dell'approccio sono stati: la scelta di un insieme limitato di variabili ("PM2.5", "CO", "Ozone", "NOx"); l'inizializzazione del modello con un valore di `contamination` pari a 0.01 ed un `random_state` fisso per la riproducibilità; l'addestramento del modello e l'utilizzo del metodo `predict` per etichettare le istanze anomale (-1) e quelle normali (1); infine, la rimozione delle osservazioni etichettate come outlier e la verifica della distribuzione prima e dopo tramite istogrammi.

```
features = [
    'PM2.5 (ug/m3)',
    'CO (mg/m3)',
    'Ozone (ug/m3)',
    'NOx (ug/m3)']
df_india_features = df_india[features].copy()
iso = IsolationForest(
    contamination=0.01,
    random_state=42,
    n_jobs=-1)
iso.fit(df_india_features)
df_india['anomaly'] = iso.predict(df_india_features)
df_india_clean = df_india[df_india['anomaly'] == 1]
    .drop(columns='anomaly')

fig, axes = plt.subplots(4, 2, figsize=(14, 12))
for i, col in enumerate(features):
```

```

axes[i, 0].hist(df_india[col].dropna(), bins=100)
axes[i, 0].set_title(f"Originale: {col}")

axes[i, 1].hist(df_india_clean[col].dropna(), bins=100)
axes[i, 1].set_title(f"Pulita: {col}")
plt.tight_layout()
plt.show()

df_india = df_india_clean.copy()

```

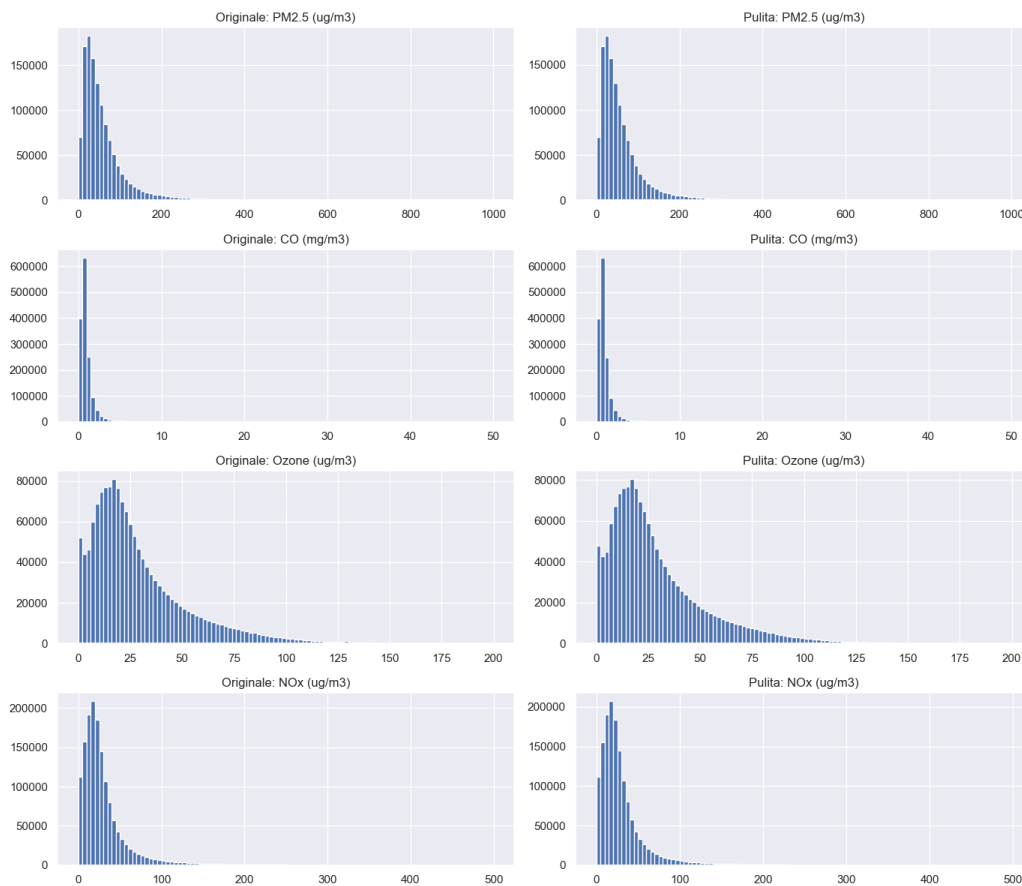


Figura 10.9: Visualizzazione della distribuzione, di ogni variabile esaminata, prima e dopo.

10.2.9 Feature engineering ed arricchimento temporale

Per catturare informazioni temporali rilevanti e migliorare la capacità predittiva dei modelli, sono state create variabili derivate dall'indice temporale quali: "hour", "dayofmonth", "dayofweek", "dayofyear", "weekofyear", "month", "quarter" e "year". Queste variabili servono a modellare stagionalità, ciclicità giornaliera e pattern settimanali/annuali tipici sia dei processi atmosferici sia delle attività umane che influenzano l'inquinamento atmosferico.

```
def create_features(df):
    df = df.copy()
    df['hour'] = df.index.hour
    df['dayofmonth'] = df.index.day
    df['dayofweek'] = df.index.dayofweek
    df['dayofyear'] = df.index.dayofyear
    df['weekofyear'] = df.index.isocalendar().week.astype("int64")
    df['month'] = df.index.month
    df['quarter'] = df.index.quarter
    df['year'] = df.index.year
    return df
df_india = create_features(df_india)

def plot_by_datetime(metric, time_groups):
    for time_group in time_groups:
        fig, ax = plt.subplots(figsize=(12, 4))
        sns.boxplot(
            data=df_india,
            x=time_group,
            y=metric,
            hue=time_group,
            palette="icefire",
            showfliers=False,
            legend=False)
```

```

ax.set_title(f'{metric} by {time_group}')
ax.set(xlabel=time_group)
plt.show()
plot_by_datetime('PM2.5 (ug/m3)', [
    'hour',
    'dayofmonth',
    'dayofweek',
    'weekofyear',
    'month',
    'quarter',
    'year'])

```

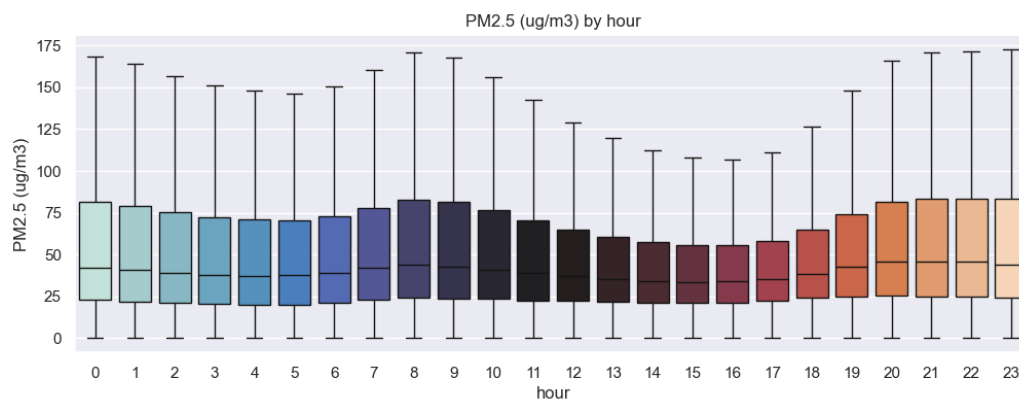


Figura 10.10: Visualizzazione della distribuzione di PM2.5 per ora.

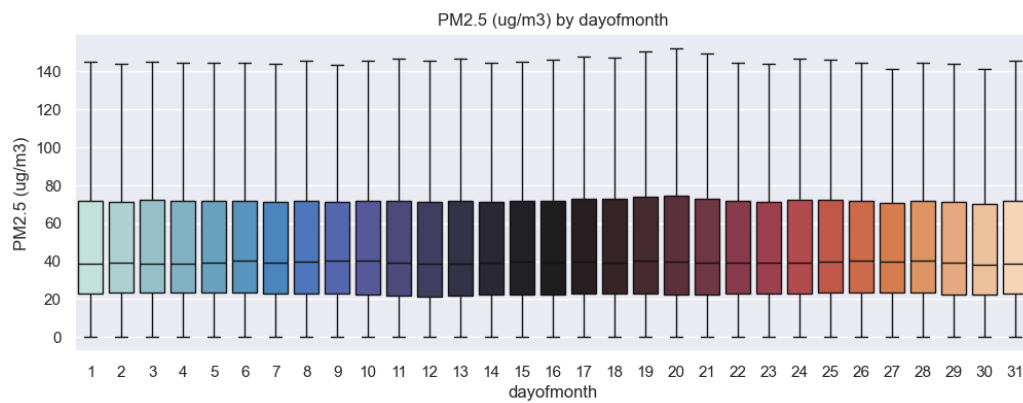


Figura 10.11: Visualizzazione della distribuzione di PM2.5 per giorno del mese.

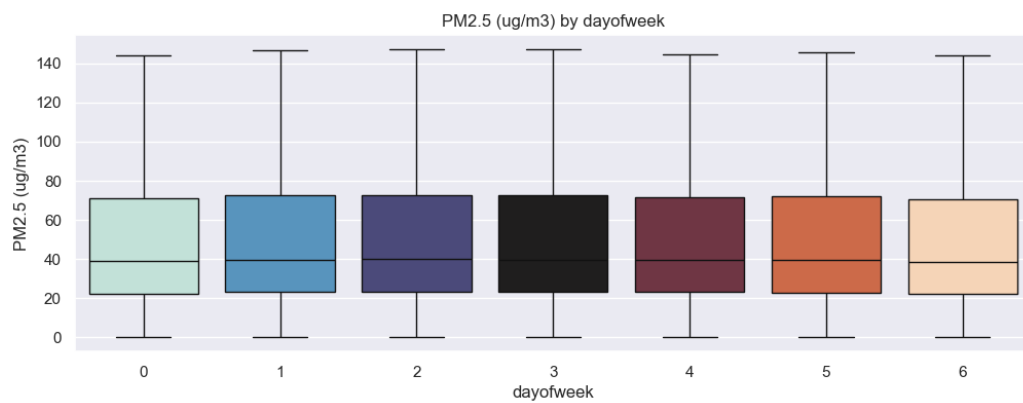


Figura 10.12: Visualizzazione della distribuzione di PM2.5 per giorno della settimana.

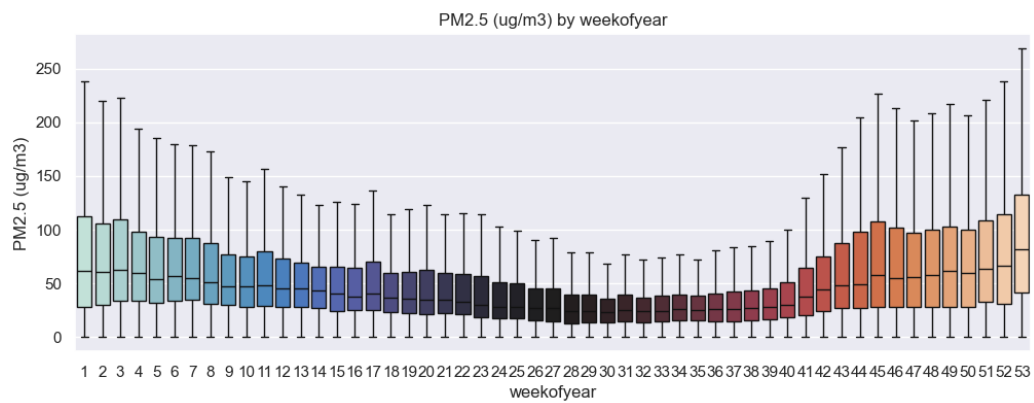


Figura 10.13: Visualizzazione della distribuzione di PM2.5 per settimana dell'anno.

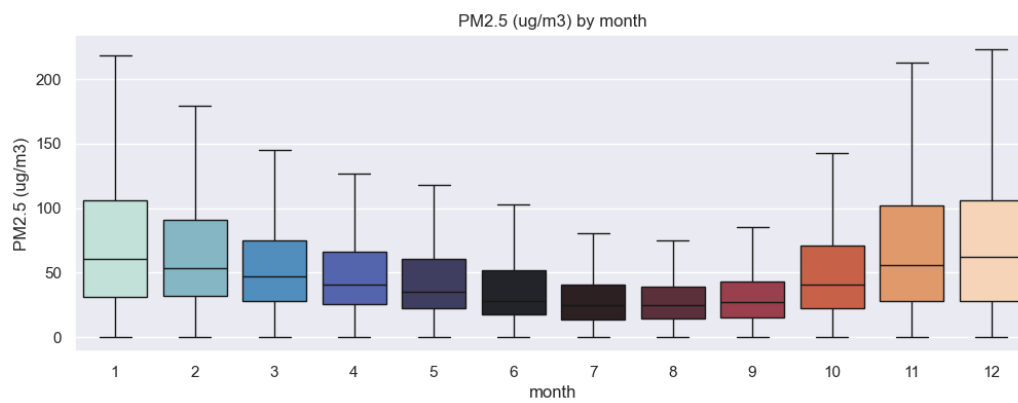


Figura 10.14: Visualizzazione della distribuzione di PM2.5 per mese dell'anno.

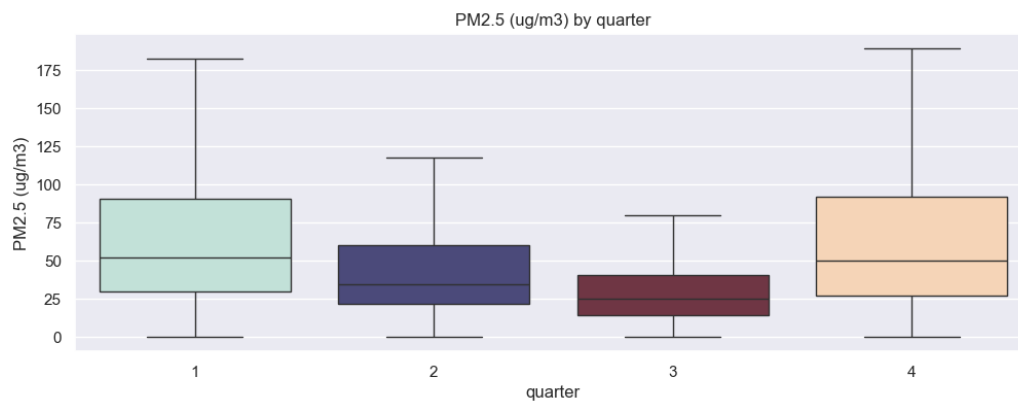


Figura 10.15: Visualizzazione della distribuzione di PM2.5 per trimestre dell'anno.

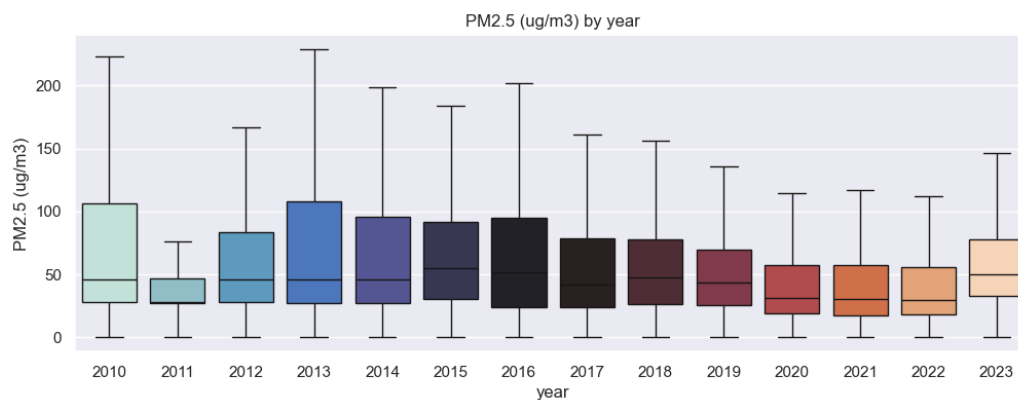


Figura 10.16: Visualizzazione della distribuzione di PM2.5 per anno.

10.2.10 Creazione di lag-features e categorizzazione di PM2.5

Per catturare l'informazione storica intrinseca nelle serie temporali, sono state create delle lag-features per alcune delle variabili più rilevanti, che consentono ai modelli di sfruttare dipendenze temporali (ad esempio stagionalità annua, effetto di breve periodo e persistenti condizioni mensili). La creazione di lag è una tecnica consolidata nella preparazione dei dati per serie temporali. In questo caso, sono state generate lag a lungo termine (1 e 2 anni), a medio termine (1 mese) ed a breve termine (1 settimana e ultimi 3 giorni).

Per trasformare la variabile target "PM2.5" da continua a discreta, con l'obiettivo di passare da un problema di regressione ad uno di classificazione, si é deciso di mappare i valori di PM2.5 in sei categorie dell'AQI (Good, Moderate, Unhealthy for Sensitive, Unhealthy, Very Unhealthy, Hazardous) usando le soglie standard corrispondenti ai breakpoints AQI per "PM2.5" (National Ambient Air Quality Standards for PM). Questo passaggio è motivato dalla volontà di concentrarsi sulla valutazione della qualità dell'aria in termini di rischio per la salute, piuttosto che sulla semplice previsione del valore esatto della concentrazione. Per molte applicazioni, è più utile sapere se la qualità dell'aria rientra in una categoria di rischio "pericoloso" o "buono", piuttosto che prevedere un valore numerico preciso, che potrebbe non essere immediatamente interpretabile.

Le soglie utilizzate, misurate in $\mu g/m^3$, sono le seguenti:

- **GOOD:** 0-9.0
- **MODERATE:** 9.1-35.4
- **UNHEALTHY FOR SENSITIVE:** 35.5-55.4
- **UNHEALTHY:** 55.5-125.4
- **VERY UNHEALTHY:** 125.5-225.4
- **HAZARDOUS:** 225.5+

Per facilitare l'analisi e l'addestramento dei modelli, le categorie sono state convertite in etichette intere da 1 (GOOD) a 6 (HAZARDOUS). Questo approccio non solo semplifica la gestione dei dati, ma mantiene anche la relazione ordinale tra le classi.

10.3 Addestramento dei modelli

Il dataset finale comprende le seguenti features:

- **Year:** anno della misurazione (numerica);

- Month: mese dell'anno (numerica);
- DayOfMonth: giorno del mese (numerica);
- DayOfWeek: giorno della settimana (numerica);
- DayOfYear: giorno dell'anno (numerica);
- WeekOfYear: settimana dell'anno (numerica);
- Quarter: trimestre dell'anno (numerica);
- State: stato di misurazione (categorica);
- PM_lag_1D, PM_lag_2D, PM_lag_3D, PM_lag_1W, PM_lag_1M, PM_lag_1Y: valori ritardati di PM2.5 rispettivamente di 1, 2, 3 giorni, 1 settimana, 1 mese e 1 anno (numeriche);
- CO_lag_1D, CO_lag_2D, CO_lag_3D, CO_lag_1W, CO_lag_1M, CO_lag_1Y: valori ritardati di CO rispettivamente di 1, 2, 3 giorni, 1 settimana, 1 mese e 1 anno (numeriche);
- O3_lag_1D, O3_lag_2D, O3_lag_3D, O3_lag_1W, O3_lag_1M, O3_lag_1Y: valori ritardati di O₃ rispettivamente di 1, 2, 3 giorni, 1 settimana, 1 mese e 1 anno (numeriche).

La variabile target da prevedere è una variabile discreta a 6 classi, corrispondenti ai livelli di qualità dell'aria per la concentrazione di PM2.5 definiti dalla scala *EPA* (Environmental Protection Agency, USA).

10.3.1 Strategia di training comune e griglie di iperparametri

La strategia di training è stata mantenuta coerente per tutti i modelli, con differenze mirate alle singole architetture. Per i modelli ensemble, la pipeline ha incluso un preprocessore per la standardizzazione delle variabili numeriche e la codifica one-hot di quelle categoriche, seguito dall'oversampling tramite SMOTE e dal classificatore. Per gestire i casi

in cui il numero di campioni minoritari risultava troppo basso rispetto al parametro `k_neighbors`, è stata implementata una versione adattiva di SMOTE, in grado di ridurre dinamicamente k . In aggiunta, è stato applicato il bilanciamento dei pesi di classe. Per le reti neurali, l'addestramento ha utilizzato l'ottimizzatore Adam, la funzione di perdita `CrossEntropyLoss`, una regolarizzazione L2 opzionale e il meccanismo di early stopping per ridurre l'overfitting. La configurazione finale per ciascun modello è stata selezionata in base alle prestazioni medie sui fold di validazione. [40]

```
class SmoteKNeighbors(SMOTE):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def _fit_resample(self, X, y):
        counts = np.bincount(y)
        minority_class = np.argmin(counts)
        n_samples_minority = counts[minority_class]

        if self.k_neighbors >= n_samples_minority:
            new_k_neighbors = max(1, n_samples_minority - 1)

            original_k_neighbors = self.k_neighbors
            self.k_neighbors = new_k_neighbors

            print(f"Warning: k_neighbors too high.
                  Adjusting from {original_k_neighbors}
                  to {self.k_neighbors} for this fold.")

        X_res, y_res = super()._fit_resample(X, y)

        self.k_neighbors = original_k_neighbors
```



```

        return X_res, y_res
    else:
        return super()._fit_resample(X, y)

model_ht_rf = ImbPipeline([
    ("preproc", preprocessor),
    ("sampler", SmoteKNeighbors(random_state=42)),
    ("tree", RandomForestClassifier(
        random_state=42,
        class_weight='balanced'))
])

gs_rf = RandomizedSearchCV(
    model_ht_rf, grid_rf, n_iter=49,
    cv=tscv, scoring='f1_weighted', n_jobs=-1,
    verbose=0, random_state=42)

model_ht_xgb = ImbPipeline([
    ("preproc", preprocessor),
    ("sampler", SmoteKNeighbors(random_state=42)),
    ("xgb", xgb.XGBClassifier(objective='multi:softprob',
        num_class=num_classes,
        use_label_encoder=False,
        eval_metric='mlogloss',
        n_jobs=-1,
        verbosity=0,
        random_state=42))
])

gs_xgb = RandomizedSearchCV(
    model_ht_xgb, grid_xgb, n_iter=98,
    cv=tscv, scoring='f1_weighted', n_jobs=-1,
    verbose=0, random_state=42
)

```

```

def random_search_neural(model_builder, param_dist, dataset,
    n_iter=10, cv_folds=5, early_patience=5,
    early_min_delta=1e-4, class_weights=None,
    smote_k_neighbors=6):
    train_keys = ['lr', 'l2_lambda']
    best_val_loss = float('inf')
    best_model_params, best_train_params = None, None
    best_model = None

    tscv = TimeSeriesSplit(n_splits=cv_folds)

    print("Avvio Random Search")

    smote = SMOTE(
        k_neighbors=smote_k_neighbors,
        random_state=RANDOM_STATE
    )

    for param_id, params in enumerate(ParameterSampler(
        param_dist, n_iter=n_iter, random_state=42)):
        print(f"Testing parameter set {param_id+1}/{n_iter}")

        model_params = {k: v for k, v in params.items()
            if k not in train_keys}
        train_params = {k: v for k, v in params.items()
            if k in train_keys}
        val_losses = []

        for fold_idx, (train_idx, val_idx) in
            enumerate(tscv.split(range(len(dataset)))):
            print(f"  Fold {fold_idx+1}/{cv_folds}")

```

```

train_features = dataset.tensors[0][train_idx]
    .cpu().numpy()
train_labels = dataset.tensors[1][train_idx]
    .cpu().numpy()

val_features = dataset.tensors[0][val_idx]
val_labels = dataset.tensors[1][val_idx]

try:
    unique_classes = np.unique(train_labels)
    if len(unique_classes) < 2:
        train_features_resampled = train_features
        train_labels_resampled = train_labels
    else:
        min_samples = min([np.sum(train_labels == cls)
            for cls in unique_classes])
        if min_samples <= smote_k_neighbors:
            print(f"    Warning:
                some classes have fewer than
                {smote_k_neighbors+1} samples.
                Reducing k_neighbors.")
            smote_fold = SMOTE(
                k_neighbors=min(min_samples-1, 1),
                random_state=42)
        else:
            smote_fold = smote

    train_features_resampled, train_labels_resampled =
        smote_fold.fit_resample(
            train_features,
            train_labels)

    print(f"    SMOTE applicato:

```

```

        {len(train_features)} ->
        {len(train_features_resampled)} campioni")

    unique, counts = np.unique(
        train_labels_resampled,
        return_counts=True)
    print(f"    Distribuzione post-SMOTE:
          {{{', '.join(f'{u}: {c}' for u, c in zip(unique, counts))}}}")

except Exception as e:
    print(f"    Warning: SMOTE fallito ({str(e)}).
          Uso dataset originale.")
    train_features_resampled = train_features
    train_labels_resampled = train_labels

train_features_tensor = torch.FloatTensor(
    train_features_resampled)
train_labels_tensor = torch.LongTensor(
    train_labels_resampled)

balanced_train_dataset = TensorDataset(
    train_features_tensor, train_labels_tensor)
val_dataset = TensorDataset(
    val_features, val_labels)

train_loader = DataLoader(
    balanced_train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(
    val_dataset, batch_size=32, shuffle=False)

model = model_builder(**model_params)
if hasattr(model, 'speed'):
    model.speed()

```

```

model.to(device)
optimizer = optim.Adam(
    model.parameters(),
    lr=train_params['lr']
)
criterion = nn.CrossEntropyLoss()
stopper = EarlyStopper(
    patience=early_patience,
    min_delta=early_min_delta)

for epoch in range(1000):
    train_loss = train_epoch(
        model,
        train_loader,
        optimizer,
        criterion,
        l2_lambda=train_params.get('l2_lambda', 0.0))
    val_loss = eval_loss(
        model, val_loader, criterion)

    if epoch % 10 == 0:
        print(f"    Epoch {epoch}:
              train_loss = {train_loss:.6f},
              val_loss = {val_loss:.6f}")

    if stopper.early_stop(val_loss):
        print(f"    Early stopping at epoch {epoch},
              best_val_loss: {stopper.best_loss:.6f}")
        break

final_val_loss = eval_loss(model, val_loader, criterion)
val_losses.append(final_val_loss)

```

```

mean_val = np.mean(val_losses)
print(f" Mean validation loss: {mean_val:.6f}")

if mean_val < best_val_loss:
    best_val_loss = mean_val
    best_model_params = model_params
    best_train_params = train_params
    best_model = model_builder(**best_model_params).to(device)
    best_model.load_state_dict(model.state_dict())
    print(f" New best validation loss: {best_val_loss:.6f}")

print(f"\nBest validation loss: {best_val_loss:.6f}")
return best_model, best_model_params, best_train_params

```

Di seguito sono riportate le griglie di iperparametri entro cui la Random Search esplora le combinazioni, al fine di individuare quelle che forniscono i risultati migliori per ciascun modello.

Random Forest (Random Search: $n = 49$)

- `n_estimators`: [100, 150, 200]
- `max_samples`: [0.5, 0.7, 0.9]
- `max_depth`: [5, 10, 15]
- `min_samples_split`: [2, 5]
- `min_samples_leaf`: [2, 5]
- `max_features`: ['sqrt', 'log2']

XGBoost (Random Search: $n = 98$)

- `max_depth`: [3, 5]
- `learning_rate`: [0.01, 0.05, 0.1]

- `n_estimators`: [100, 200, 300]
- `subsample`: [0.7, 0.9]
- `colsample_bytree`: [0.7, 0.9]
- `gamma`: [0, 0.2, 0.4]
- `min_child_weight`: [1, 5]

MLP (Random Search: $n = 26$)

- `hidden_sizes`: [(64,64), (128,), (128,64), (256,128), (512,256)];
- `dropout`: [0.0, 0.2, 0.5];
- `lr` (learning rate): [10^{-3} , 10^{-4}];
- `l2_lambda` (coefficiente L2): [0.0, 10^{-5} , 10^{-4} , 10^{-3}].

KAN (Random Search: $n = 43$)

- `width`: [(8,4), (16,8), (32,16), (64,32)] (struttura a livelli della rete);
- `grid`: [5, 10, 20] (dimensione della griglia interna);
- `k`: [2, 4] (ordine/complessità della combinazione);
- `seed`: [0] (per riproducibilità);
- `lr`: [10^{-3} , 10^{-4}];
- `l2_lambda`: [0.0, 10^{-3} , 10^{-4} , 10^{-5}].

10.3.2 Scelte architettureali finali

Nella Tabella 10.1 vengono mostrate le scelte finali, dopo l'ottimizzazione degli iperparametri, utilizzate per training, valutazioni comparative e studio di ablazione.

Tabella 10.1: Configurazioni finali dei modelli usati per il Training, dopo aver effettuato l'ottimizzazione degli iperparametri.

MLP	<code>input_dim = 49; hidden_sizes = (64, 64); dropout = 0.2; ottimizzatore = Adam; lr = 1e-04; l2_lambda = 1e-03; batch_size = 32. Early stopping applicato.</code>
KAN	<code>input_dim = 49; width = (16,8); grid = 5; k = 4, seed = 0; ottimizzatore = Adam; lr = 1e-04; l2_lambda = 1e-05; batch_size = 32. Early stopping applicato.</code>
Random Forest	<code>n_estimators = 100; max_depth = 15; min_samples_split = 5; min_samples_leaf = 2; max_features = 'sqrt'; max_samples = 0.7; random_state = 42.</code>
XGBoost	<code>n_estimators = 300; max_depth = 5; learning_rate = 0.05; subsample = 0.9; colsample_bytree = 0.7; min_child_weight = 1; gamma = 0.2; objective = 'multi:softprob'; eval_metric = 'mlogloss'; random_state = 42.</code>

10.4 Valutazione dei modelli

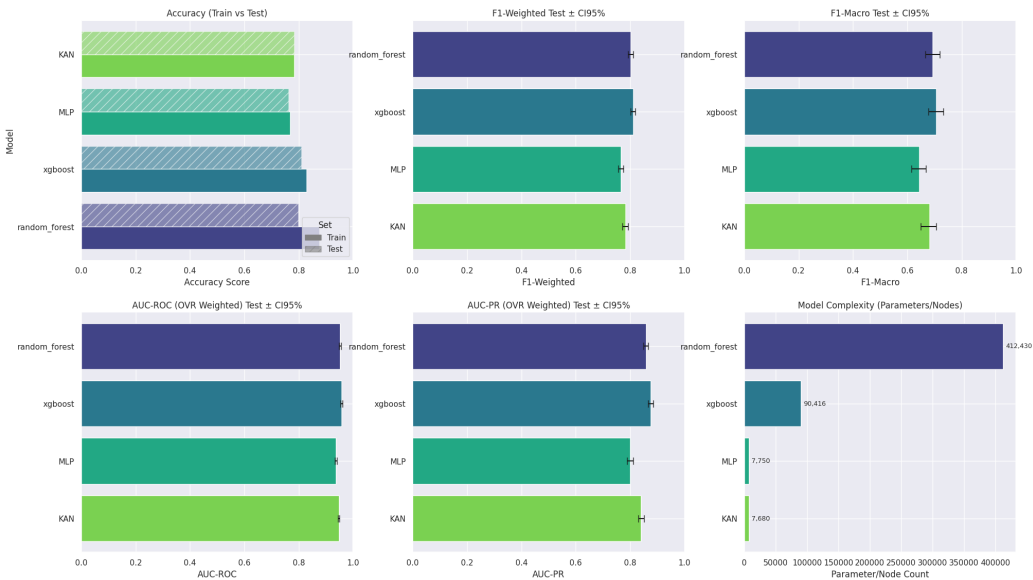


Figura 10.17: Confronto visivo delle prestazioni dei modelli (Accuracy train vs test, F1-weighted / F1-macro con CI95%, AUC-ROC e AUC-PR OVR weighted, e complessità in parametri/nodi).

Classification report e confusion matrix per modello

Random Forest

Numero di parametri / nodi: **412 430**.

Tabella 10.2: Classification report

Classe	precision	recall	f1-score	support
0	0.76	0.73	0.74	321
1	0.89	0.86	0.88	2 727
2	0.65	0.70	0.67	1 364
3	0.85	0.80	0.82	1 831
4	0.61	0.82	0.70	307
5	0.32	0.37	0.34	30
accuracy		0.80		6 580
macro avg	0.68	0.71	0.69	6 580
weighted avg	0.81	0.80	0.80	6 580

Tabella 10.3: Confusion matrix

$$\begin{bmatrix} 233 & 88 & 0 & 0 & 0 & 0 \\ 73 & 2350 & 286 & 18 & 0 & 0 \\ 1 & 196 & 953 & 213 & 1 & 0 \\ 0 & 9 & 222 & 1456 & 143 & 1 \\ 0 & 1 & 0 & 32 & 252 & 22 \\ 0 & 0 & 0 & 0 & 19 & 11 \end{bmatrix}$$

AUC-ROC (OVR, weighted): **0.952**

AUC-PR (OVR, weighted): **0.858**

XGBoost

Numero di parametri / nodi: **90 416**.

Tabella 10.4: Classification report

Classe	precision	recall	f1-score	support
0	0.76	0.79	0.77	321
1	0.89	0.87	0.88	2 727
2	0.67	0.68	0.68	1 364
3	0.83	0.85	0.84	1 831
4	0.69	0.70	0.70	307
5	0.32	0.43	0.37	30
accuracy		0.81		6 580
macro avg	0.69	0.72	0.71	6 580
weighted avg	0.81	0.81	0.81	6 580

Tabella 10.5: Confusion matrix

$$\begin{bmatrix} 253 & 68 & 0 & 0 & 0 & 0 \\ 79 & 2360 & 266 & 22 & 0 & 0 \\ 1 & 200 & 930 & 232 & 1 & 0 \\ 0 & 8 & 187 & 1556 & 78 & 2 \\ 0 & 1 & 0 & 64 & 216 & 26 \\ 0 & 0 & 0 & 0 & 17 & 13 \end{bmatrix}$$

AUC-ROC (OVR, weighted): **0.957**

AUC-PR (OVR, weighted): **0.875**

MLP

Numero di parametri: **7 750**.

Tabella 10.6: Classification report

Classe	precision	recall	f1-score	support
0	0.43	0.38	0.40	321
1	0.86	0.80	0.83	2 727
2	0.63	0.72	0.67	1 364
3	0.85	0.80	0.82	1 831
4	0.59	0.82	0.69	307
5	0.37	0.57	0.45	30
accuracy		0.76		6 580
macro avg	0.62	0.68	0.64	6 580
weighted avg	0.77	0.76	0.77	6 580

Tabella 10.7: Confusion matrix

$$\begin{bmatrix} 121 & 200 & 0 & 0 & 0 & 0 \\ 157 & 2189 & 360 & 21 & 0 & 0 \\ 4 & 160 & 977 & 220 & 3 & 0 \\ 0 & 5 & 204 & 1465 & 156 & 1 \\ 0 & 1 & 0 & 27 & 251 & 28 \\ 0 & 0 & 0 & 0 & 13 & 17 \end{bmatrix}$$

AUC-ROC (OVR, weighted): **0.938**

AUC-PR (OVR, weighted): **0.800**

KAN

Numero di parametri: **7 680**.

Tabella 10.8: Classification report

Classe	precision	recall	f1-score	support
0	0.65	0.36	0.46	321
1	0.85	0.85	0.85	2 727
2	0.63	0.72	0.67	1 364
3	0.84	0.82	0.83	1 831
4	0.74	0.69	0.71	307
5	0.51	0.60	0.55	30
accuracy		0.78		6 580
macro avg	0.70	0.67	0.68	6 580
weighted avg	0.79	0.78	0.78	6 580

Tabella 10.9: Confusion matrix

$$\begin{bmatrix} 116 & 204 & 1 & 0 & 0 & 0 \\ 61 & 2329 & 320 & 17 & 0 & 0 \\ 2 & 198 & 985 & 179 & 0 & 0 \\ 0 & 7 & 261 & 1498 & 63 & 2 \\ 0 & 1 & 0 & 79 & 212 & 15 \\ 0 & 0 & 0 & 0 & 12 & 18 \end{bmatrix}$$

AUC-ROC (OVR, weighted): **0.948**

AUC-PR (OVR, weighted): **0.840**

10.4.1 Analisi dei risultati sperimentali

L'analisi dei classification report e delle confusion matrix (riportati nella sottosezione precedente) mostra che, a livello di metriche aggregate, le differenze tra i modelli testati sono complessivamente contenute: accuratezza, F1-score, AUC-ROC e AUC-PR si collocano su valori molto simili, con scarti percentuali spesso marginali. La discriminante più evidente è la complessità architetturale, che diventa il fattore operativo principale per decisioni di deployment, latenza e consumo di memoria.

Per quanto riguarda le reti neurali, la **MLP** ha ottenuto un'accuratezza in test pari al 76%, con un F1-weighted di 0.7654, un F1-macro di 0.7421, un AUC-ROC di 0.938 e un AUC-PR di 0.800. L'analisi per classe evidenzia una sensibilità ridotta sulle classi meno rappresentate (ad es. la classe **0** è spesso confusa), sebbene il modello mantenga buone prestazioni sulla classe maggioritaria (1). Con circa 7,750 parametri la MLP è una soluzione leggera e facilmente scalabile, ma meno competitiva rispetto ad altre alternative sul piano delle performance complessive.

La **KAN** ha mostrato un comportamento più equilibrato: accuratezza in test del 78%, F1-weighted 0.7823, F1-macro 0.7608, AUC-ROC 0.948 e AUC-PR 0.840. KAN presenta inoltre una migliore capacità di trattare le classi meno rappresentate rispetto alla MLP (si osservano recall e F1 mediamente più alti su alcune classi difficili), mantenendo un numero di parametri molto

contenuto (circa 7,680). Queste caratteristiche rendono KAN una candidata interessante quando si richiede un buon compromesso tra accuratezza e parsimonia computazionale.

I modelli ensemble si differenziano principalmente per il trade-off tra performance leggermente migliori e complessità superiore. La **Random Forest**, con circa 412k nodi, ha raggiunto un'accuratezza dell'80%, F1-weighted 0.7997, F1-macro 0.7750, AUC-ROC 0.952 e AUC-PR 0.858. Le confusion matrix mostrano che Random Forest è robusta nel predire la classe maggioritaria e riduce alcune specifiche confusioni osservate nelle reti neurali, ma il costo in termini di complessità è molto elevato.

XGBoost è risultato il modello con le metriche aggregate più alte: accuratezza in test 81%, F1-weighted 0.8107, F1-macro 0.7901, AUC-ROC 0.957 e AUC-PR 0.875. Tuttavia i miglioramenti rispetto a KAN e Random Forest sono marginali. Con circa 90k parametri XGBoost occupa una posizione intermedia: molto più leggero della Random Forest ma sensibilmente più complesso rispetto alle architetture neurali leggere come MLP e KAN. Dal confronto delle confusion matrix emerge come XGBoost migliori leggermente il recall sulle classi meno rappresentate rispetto a Random Forest e MLP, pur restando i guadagni contenuti in termini assoluti.

Va inoltre segnalato che, per mitigare lo sbilanciamento delle classi, nel training sono state impiegate sia la ponderazione degli errori per classe (class weights) sia una procedura di oversampling basata su SMOTE. Queste contromisure hanno portato a miglioramenti locali nei punteggi di recall sulle classi minority (riducendo alcuni falsi negativi), ma non hanno eliminato completamente le difficoltà associate alle classi con supporto molto basso.

10.4.2 Selezione del miglior modello

I risultati di questa procedura di ranking multi-criterio mostrano che XGBoost è il migliore secondo il criterio di Equal Weight (1:1), mentre la KAN risulta vincente nei criteri Complexity Weighted (1:2), Extreme

Complexity (1:3) e Pareto Approach (40:60). I valori principali utilizzati per il ranking sono riassunti nella tabella seguente:

Tabella 10.10: Riepilogo ranking aggiornato: conteggio parametri, performance media (rank-based) e ranks per metodo di aggregazione.

Model	Param_Count	Perf_Score	Compl_Rank	Equal_Rank	Ext_Rank	Pareto_Rank
KAN	7,680	3.0	1	1	1	1
XGBoost	90,416	1.0	2	1	2	2
MLP	7,750	4.0	3	3	2	3
RF	412,430	2.0	4	3	4	4

10.4.3 Conclusioni

Dalla procedura di ranking multi-criterio e dall'analisi delle metriche aggregate, XGBoost presenta le performance assolute più elevate sulle metriche di classificazione (in particolare l'F1-weighted), ma richiede una complessità molto maggiore rispetto alle reti neurali testate. KAN offre un eccellente compromesso fra accuratezza e parsimonia e, nelle strategie che penalizzano la complessità, risulta il modello raccomandato per il deployment. La MLP mantiene un buon rapporto performance/complessità, ma è lievemente inferiore a KAN, secondo il criterio complexity-weighted. Il Random Forest, pur competitivo in termini di metriche assolute, è meno interessante quando la parsimonia del modello è un vincolo operativo.

La classifica dei tre migliori modelli, basata sul criterio "complexity-weighted" (dal migliore al peggiore), è la seguente:

1. KAN (Params: 7,680; F1-Weighted: 0.7823)
2. XGBoost (Params: 90,416; F1-Weighted: 0.8107)
3. MLP (Params: 7,750; F1-Weighted: 0.7654)

10.5 Studio di ablazione

10.5.1 Ablation study: L1 pruning su MLP e KAN

Figura riassuntiva

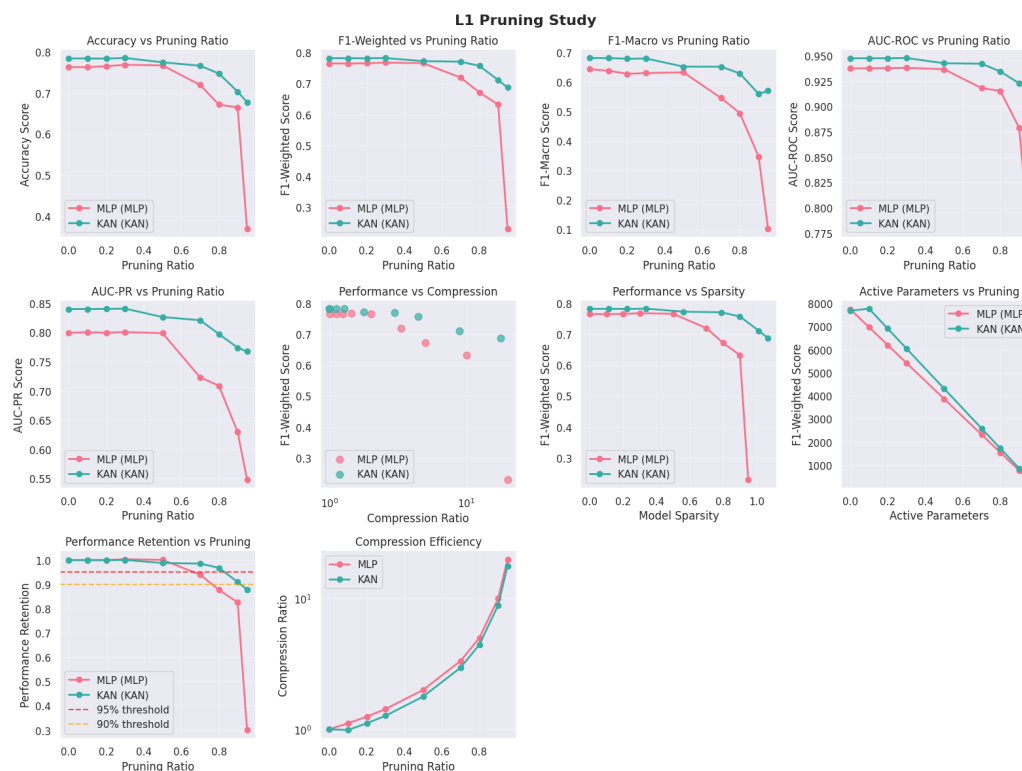


Figura 10.18: Risultati dello studio L1 pruning per MLP e KAN, utilizzando metriche principali e indicatori di compressione (Accuracy, F1-Weighted, F1-Macro, AUC-ROC, AUC-PR, sparsità, parametri attivi, performance retention).

Risultati

Model	Total params	Baseline F1	Best trade-off	Sign. degr.
MLP	7,750	0.7654	50% pruning (2.0× compression)	70% pruning
KAN	7,680	0.7823	70% pruning (3.0× compression)	90% pruning

Tabella 10.11: Riepilogo dei punti di trade-off e dei punti di degrado osservati nello studio L1 pruning (valori aggiornati).

La MLP, con baseline F1-Weighted = **0.7654**, mostra il miglior trade-off al **50%** di pruning: in questa condizione la compressione è di circa 2× e la variazione relativa in F1-weighted è trascurabile (ossia $\leq 0.2\%$ in termini assoluti, nel nostro run si è misurata una variazione leggermente positiva rispetto alla baseline). Il punto di degrado significativo per la MLP si evidenzia a partire da circa il **70%** di pruning, oltre il quale la perdita di performance diventa marcata; a pruning estremi ($\geq 90\%$) le prestazioni decadono drasticamente.

La KAN, con baseline F1-Weighted = **0.7823**, risulta più robusta per pruning moderati: il miglior trade-off osservato è al **70%** di pruning, corrispondente a una compressione di circa 3× e a una perdita relativa in F1-weighted dell'ordine di $\approx 1.4\%$. La soglia di degrado significativo per KAN si trova invece intorno al **90%** di pruning.

La compressione massima raggiungibile nei test è stata dell'ordine di $\sim 20\times$ per la MLP e $\sim 17.8\times$ per la KAN (pruning 95%), ma tali livelli estremi comportano perdite di prestazione troppo elevate per essere considerati praticabili in un contesto di deployment senza ulteriori tecniche (ad esempio, retraining post-pruning).

10.5.2 Ablation study: ensemble pruning su Random Forest e XGBoost

Figura riassuntiva

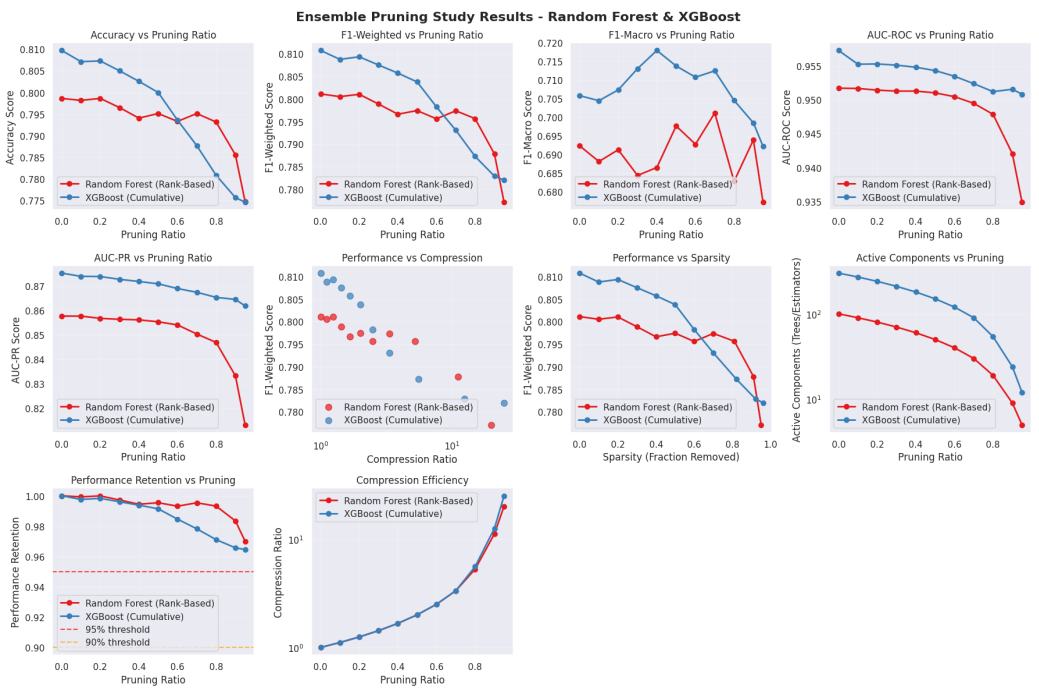


Figura 10.19: Risultati dello studio di pruning per Random Forest (rank-based) e XGBoost (cumulative), utilizzando metriche principali e indicatori di compressione (Accuracy, F1-Weighted, F1-Macro, AUC-ROC, AUC-PR, sparsità, alberi rimanenti, performance retention).

Risultati

Model	Total trees	Baseline F1	Best trade-off (pruning)	Sign. degr.
RF	100	0.8011	90% (11.1× compression)	nessuna degr. rilevata
XGB	300	0.8107	60% (2.5× compression)	nessuna degr. rilevata

Tabella 10.12: Riepilogo sintetico dei punti di trade-off osservati per i due ensemble (valori aggiornati).

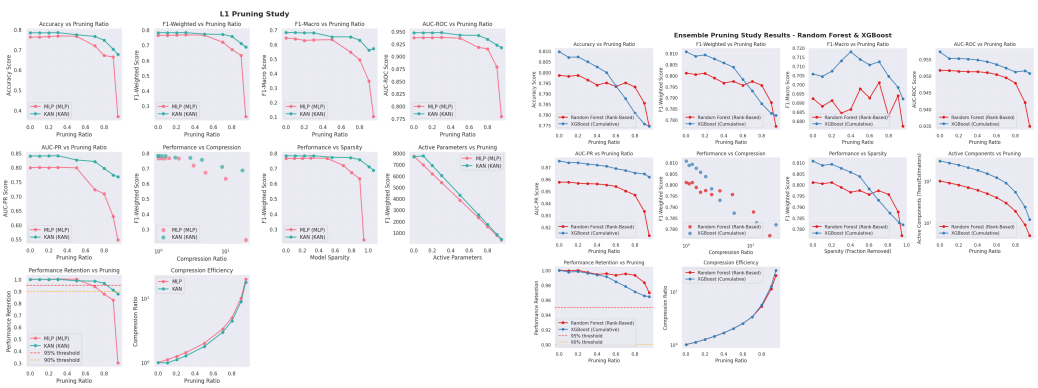
XGBoost mantiene buona parte delle prestazioni riducendo il numero di rounds. Il miglior compromesso osservato nel run corrente è stato al 60% di pruning, con i primi 120 rounds su 300, corrispondente a una compressione di $\approx 2.5\times$ ed ad una perdita relativa in F1-weighted di circa 1.5% rispetto alla baseline.

Random Forest dimostra elevata robustezza: l'approccio rank-based permette di rimuovere un gran numero di alberi mantenendo metriche stabili; nel nostro esperimento il best trade-off si è verificato al 90% di pruning (9 alberi rimanenti), con una compressione $\approx 11.11\times$ ed una perdita relativa in F1-weighted di circa 1.7%. In nessuno dei due casi si è osservata una perdita relativa superiore al 5% nell'intervallo di pruning testato.

La compressione massima sperimentata è stata dell'ordine di $\sim 20\times$ per Random Forest e $\sim 25\times$ per XGBoost (pruning 95%), tuttavia tali livelli estremi comportano perdite di prestazione significativamente maggiori e risultano poco praticabili senza interventi aggiuntivi (es. fine-tuning post-pruning, distillazione o quantizzazione).

10.5.3 Ablation study — Confronto complessivo (Neural Networks vs Ensemble)

Figure riassuntive



A: Studio di ablazione MLP e KAN (L1 pruning). B: Studio di ablazione Random Forest (rank-based) e XGBoost (cumulative).

Figura 10.20: Risultati sintetici degli studi di ablazione.

Riepilogo

Model	Baseline F1 (weighted)	Best trade-off (pruning)	Compression
MLP	0.7654	0.7662 @ 50% pr	≈ 2.0×
KAN	0.7823	0.7710 @ 70% pr	≈ 3.0×
RF	0.8011	0.7878 @ 90% pr	≈ 11.1×
XGBoost	0.8107	0.7982 @ 60% pr	≈ 2.5×

Tabella 10.13: Riepilogo sintetico dei principali punti di trade-off e compressione (valori aggiornati).

Confronto su soglie tipiche (30%, 50%, 70%, 90%)

Al 30% di pruning, tutti i modelli mantengono o presentano leggeri miglioramenti rispetto alla baseline nelle metriche principali: MLP mostra un lieve incremento dell’F1-weighted, mentre KAN e XGBoost mantengono prestazioni molto elevate. A 50% di pruning la MLP raggiunge il suo miglior

trade-off operativo (compressione $\sim 2\times$ con F1-weighted sostanzialmente invariata rispetto alla baseline), mentre KAN e gli ensemble rimangono stabili. A 70% di pruning XGBoost continua a mostrare una ritenzione delle prestazioni accettabile (nelle soglie tra il 50% ed il 70% la perdita relativa di F1 è inferiore al 2%), KAN risulta robusto fino a questo livello e la MLP inizia a evidenziare un degrado più marcato oltre certe soglie di sparsity. A 90% di pruning le reti neurali presentano una degradazione significativa (più pronunciata per MLP), mentre gli ensemble (in particolare Random Forest con pruning rank-based) conservano una retention relativamente migliore fino a livelli molto elevati di riduzione.

Conclusioni

1. XGBoost è la scelta consigliata quando l'obiettivo primario è massimizzare la performance assoluta (F1-weighted baseline: 0.8107). Ridurre i rounds fino a circa il 50% – 60% ha permesso di ottenere un buon compromesso (compressione $\sim 2.0-2.5\times$) con una perdita contenuta in F1 (tipicamente $< 2\%$).
2. Le reti MLP e KAN rispondono bene all'L1 pruning: la MLP ottiene il miglior trade-off operativo al 50% di pruning (compressione $\sim 2\times$ con F1 invariata o leggermente superiore), mentre KAN risulta più robusta fino al 70% (compressione $\sim 3\times$ con perdita relativa contenuta $\sim 1\%-1.5\%$).
3. Random Forest presenta un comportamento interessante con il rank-based pruning: è possibile rimuovere gran parte degli alberi mantenendo performance stabili; il best trade-off è risultato ad un pruning molto elevato (90%), con 9 alberi rimasti (compressione $\approx 11.1\times$) ed una perdita relativa in F1 dell'ordine di 1%-2%. Questo rende la RF particolarmente utile quando si desidera una forte riduzione del modello senza operare ri-addestramenti complessi.

Capitolo 11

Terzo Caso Studio: Classificazione di fasce d'età tramite immagini

11.1 Introduzione

Il presente caso studio affronta il problema della classificazione di fasce d'età a partire da immagini facciali. Il dataset utilizzato si chiama UTKFace (<https://susanqq.github.io/UTKFace/>) ed è un insieme di volti annotati con età, genere ed etnia, che copre un ampio intervallo di età (da 0 a 116 anni) e mostra un'ampia variabilità in termini di posa, espressione, illuminazione e qualità delle immagini.

Gli obiettivi del caso di studio sono: innanzitutto si trasforma il compito di stima continua dell'età in un problema di classificazione multi-classe mediante la definizione di fasce d'età interpretabili (*child, young, adult, senior*), bilanciate quando possibile per garantire stabilità statistica; in secondo luogo viene valutata e confrontata l'efficacia di CNN con classificatore MLP e KAN, con particolare attenzione alla quantificazione del trade-off prestazioni/complessità; inoltre si analizza l'impatto delle scelte di pre-processing, comprendenti la pulizia delle etichette estratte dai nomi file, la normalizzazione e le strategie di data augmentation, così come delle politiche di campionamento e bilanciamento.

Nel capitolo vengono presentati, in sequenza, la composizione del dataset;

la pipeline di preprocessing e le motivazioni alla base della discretizzazione delle età; la strategia di training e le procedure di ottimizzazione degli iperparametri; la descrizione delle architetture finali impiegate per i confronti; i risultati quantitativi corredati da intervalli di confidenza, confusion matrix ed analisi per classe.

11.2 Data preparation

La preparazione dei dati è stata condotta seguendo un protocollo riproducibile e tracciabile, volto a trasformare il repository grezzo di immagini facciali e le informazioni codificate nei nomi dei file in insiemi di dati pronti per la fase di addestramento, valutazione e studio di ablazione. Il workflow adottato comprende: acquisizione e normalizzazione dei percorsi delle immagini, estrazione e pulizia delle etichette (età, genere, etnia), definizione di classi d'età coerenti con l'obiettivo sperimentale, bilanciamento numerico del dataset, controllo di integrità dei file immagine ed un'analisi esplorativa finalizzata all'identificazione di bias e anomalie.

11.2.1 Pre-processing e costruzione delle etichette

I nomi dei file seguono la convenzione <age>_<gender>_<ethnic>_...jpg e sono stati analizzati per estrarre le tre etichette primarie: età, genere ed etnia. Le righe contenenti metadati non conformi o non numerici sono state scartate e le colonne rimanenti sono state convertite ai tipi più appropriati per l'elaborazione.

Per migliorare la leggibilità e la coerenza delle analisi successive, i codici numerici relativi a genere ed etnia sono stati mappati su etichette testuali (ad esempio *Male/Female* per il genere; tassonomia semantica per i gruppi etnici). Le colonne relative a genere ed etnia sono state trattate come variabili categoriche per ridurre l'occupazione di memoria e facilitare operazioni di aggregazione e raggruppamento.

La variabile età, originariamente numerica e molto dettagliata, è stata discretizzata in quattro fasce d'età: bambini ("child"), adolescenti ("young"),

adulti ("adult") e anziani ("senior"). Tale scelta nasce dall'esigenza di un compromesso tra risoluzione predittiva e robustezza statistica: una granularità eccessiva porta a classi con numerosità troppo basse, compromettendo la capacità dei modelli di generalizzare.

Scelta della dimensione target e procedura di campionamento

Per ottenere un dataset sperimentale con distribuzioni più bilanciate tra le fasce d'età, è stata definita una dimensione complessiva target del dataset e, a partire da questa, il numero target di esempi per ciascuna classe. Indichiamo con N_{tot} la dimensione target complessiva e con G il numero di gruppi (fasce d'età); il numero target per gruppo è calcolato come

$$N_{\text{target}} = \left\lfloor \frac{N_{tot}}{G} \right\rfloor,$$

con gestione del resto tramite possibile assegnazione di una o più istanze aggiuntive ad alcune classi. I campioni per ogni classe vengono concatenati ed infine rimescolati con seed fissato, ottenendo così il dataset sperimentale definitivo. Tale procedura controllata consente di limitare il bias dovuto a classi fortemente sbilanciate mantenendo riproducibilità e semplicità di implementazione.

Di seguito, il codice responsabile della generazione del grafico a torta che mostra il bilanciamento tra classi nel dataset modificato.

```
class_counts = df['age_group'].value_counts()
print(class_counts)

age_group_labels = {
    0: "Child [1;12]",
    1: "Young [13;18]",
    2: "Adult [19;60]",
    3: "Senior [60;inf]"}
pie_labels = [age_group_labels[i]
               for i in class_counts.index]
```



```
plt.figure(figsize=(8, 8))
plt.pie(
    class_counts,
    labels=pie_labels,
    autopct='%1.1f%%',
    startangle=140)
plt.title('Distribution of Classes')
plt.show()
```

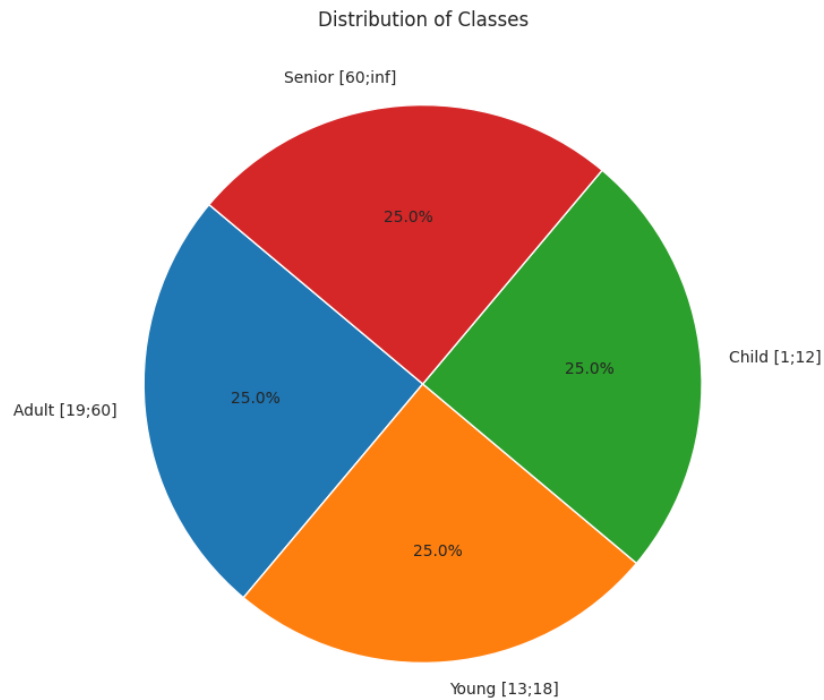


Figura 11.1: Distribuzione delle classi in percentuale.

Controllo di integrità dei file e politica di fallback

Prima dell'estrazione finale dei tensori con immagini all'interno, si è eseguito un controllo di integrità su tutti i percorsi indicati nel DataFrame. Per

ogni percorso, in caso di mancata corrispondenza tra entry e file reale, si sostituisce con un altro esempio appartenente alla stessa fascia d'età, selezionato a campione tra le istanze disponibili di quella fascia.

Trasformazioni immagine e criteri di normalizzazione

Per la compatibilità con le architetture CNN impiegate, tutte le immagini sono state ridimensionate e ritagliate centralmente a risoluzione 224×224 pixel, convertite in tensori e normalizzate canale per canale con parametri di media e deviazione standard comunemente adottati nelle reti pre-addestrate. Durante la fase di addestramento è stata valutata l'applicazione di data augmentation controllata (flip orizzontale, rotazioni leggere, leggere variazioni di luminosità e contrasto); l'ampiezza delle trasformazioni è stata limitata per non alterare i dettagli del viso che sono informazioni importanti per il compito di stima dell'età.

11.2.2 Data exploration

L'analisi esplorativa è stata finalizzata a identificare squilibri ed anomalie nei dati grezzi e a produrre una serie di visualizzazioni da inserire nella tesi per documentare lo stato del dataset.

La prima visualizzazione è una rappresentazione circolare (pie chart) della distribuzione percentuale delle età; le età con frequenza inferiore alla soglia dell'1% sono state aggregate nella categoria "Others" al fine di migliorare chiarezza e leggibilità.

```
age_counts = df.age.value_counts()
total_count = age_counts.sum()
age_percentages = (age_counts / total_count) * 100

other_ages = age_percentages[age_percentages < 1]
other_percentage = other_ages.sum()
```

```

plot_data = age_percentages[age_percentages >= 1]
if other_percentage > 0:
    plot_data['Others'] = other_percentage

sns.set_style("whitegrid")

plt.figure(figsize=(10, 10))
plt.title("Distribution of Ages in Percentage")
plot_data.plot.pie(autopct='%1.1f%%', startangle=90)
plt.ylabel('')
plt.figtext(0.5, 0.01,
            "The 'Others' category includes ages  

            with a distribution of less than 1%.",
            ha="center",
            fontsize=10,
            bbox={"facecolor":"orange", "alpha":0.5, "pad":5})
plt.show()

```

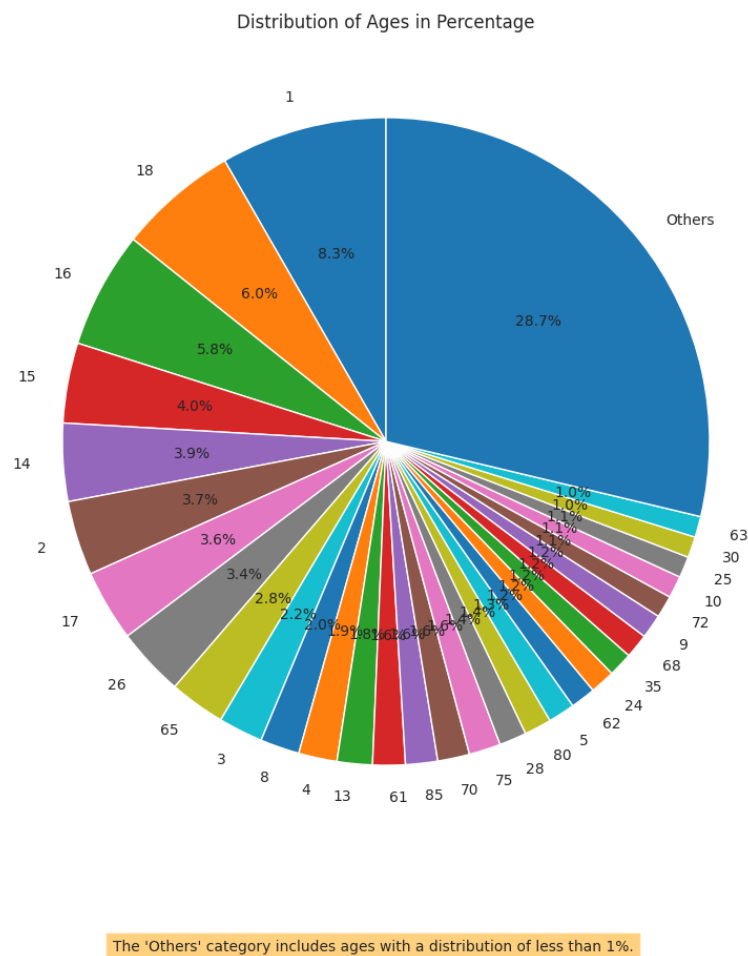


Figura 11.2: Distribuzione dell'età in percentuale.

A complemento della panoramica generale, sono stati prodotti tre grafici a barre posti in parallelo che mostrano la distribuzione per età del dataset complessivo, dei soli individui di genere maschile e femminile. Questa tripletta di visualizzazioni consente di osservare in maniera immediata eventuali bias di genere associati a singole età o fasce d'età.

```
sns.set_style("whitegrid")
fig, axes = plt.subplots(3, 1, figsize=(40, 30))
fig.suptitle("Distribution of Ages by Gender", fontsize=40)
```

```

age_counts = df.age.value_counts()
sns.barplot(
    x=age_counts.index,
    y=age_counts.values,
    ax=axes[0], color='black')
axes[0].set_title("Distribution of All Ages", fontsize=20)
axes[0].set_xlabel("Age", fontsize=18)
axes[0].set_ylabel("Count", fontsize=18)

male_age_counts = df[df.gender == 'Male'].age.value_counts()
sns.barplot(
    x=male_age_counts.index,
    y=male_age_counts.values,
    ax=axes[1], color='blue')
axes[1].set_title("Distribution of Ages for Males", fontsize=20)
axes[1].set_xlabel("Age", fontsize=18)
axes[1].set_ylabel("Count", fontsize=18)

female_age_counts = df[df.gender == 'Female'].age.value_counts()
sns.barplot(
    x=female_age_counts.index,
    y=female_age_counts.values,
    ax=axes[2], color='orange')
axes[2].set_title("Distribution of Ages for Females", fontsize=20)
axes[2].set_xlabel("Age", fontsize=18)
axes[2].set_ylabel("Count", fontsize=18)

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()

```

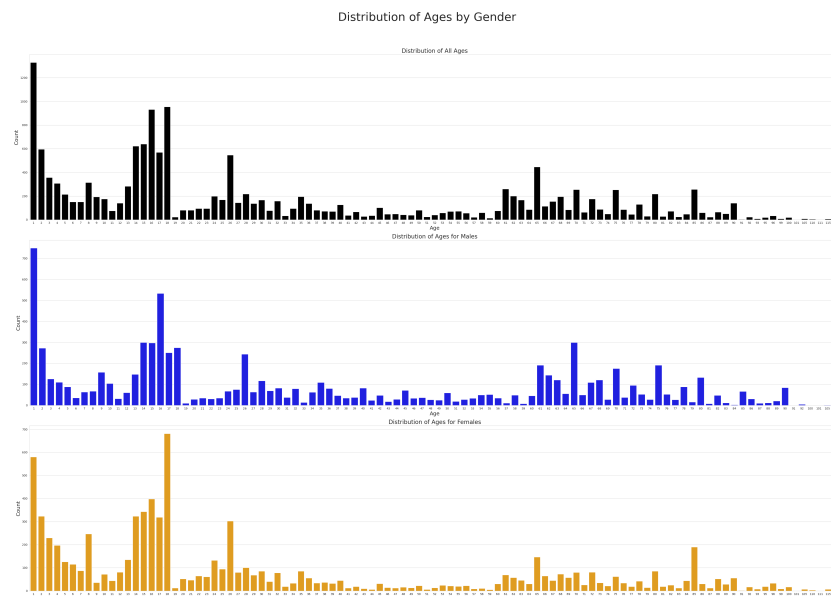


Figura 11.3: Distribuzione dell'età per genere.

In aggiunta alla distribuzione delle fasce d'età, è stata analizzata la composizione di genere del dataset, che evidenzia la quasi perfetta parità tra soggetti maschili e femminili, condizione che riduce il rischio di bias sistematici legati al genere.

```
sns.set_style("whitegrid")
plt.figure(figsize=(8,8))
plt.title("Distribution of Genders")
plt.pie(
    df.gender.value_counts(),
    labels=df.gender.value_counts().index,
    autopct='%1.1f%%')
plt.show()
```

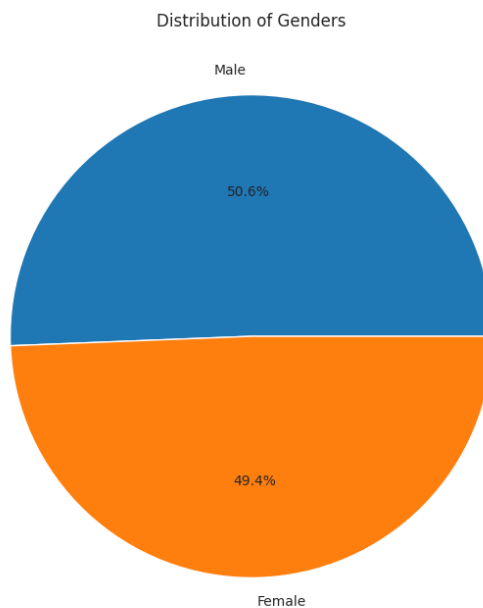


Figura 11.4: Distribuzione percentuale dei generi.

Per indagare la relazione tra età e genere è stato inoltre prodotto un grafico a barre che mostra, per ciascuna età, la percentuale di individui maschi e femmine, che evidenzia come la distribuzione sia in generale bilanciata, pur mostrando variazioni locali dovute a squilibri demografici.

```
age_gender_counts = df.groupby(['age', 'gender'], observed=True)
    .size().unstack(fill_value=0)
age_gender_percentages = age_gender_counts
    .apply(lambda x: x / x.sum(), axis=1)
```

```
ax = age_gender_percentages.plot(
    kind='bar',
    stacked=True,
    figsize=(20, 10),
    color=['blue', 'orange'])
```

```
plt.title('Percentage of Males and Females by Age', fontsize=20)
plt.xlabel('Age', fontsize=15)
```

```

plt.ylabel('Percentage', fontsize=15)
plt.xticks(fontsize=8)
plt.yticks(fontsize=10)
plt.legend(title='Gender')

for p in ax.patches:
    width, height = p.get_width(), p.get_height()
    if height > 0:
        x, y = p.get_xy()
        ax.text(
            x + width / 2,
            y + height / 2,
            '{:.1f}%'.format(height * 100),
            horizontalalignment='center',
            verticalalignment='center',
            fontsize=6,
            color='white'
        )

plt.tight_layout()
plt.show()

```

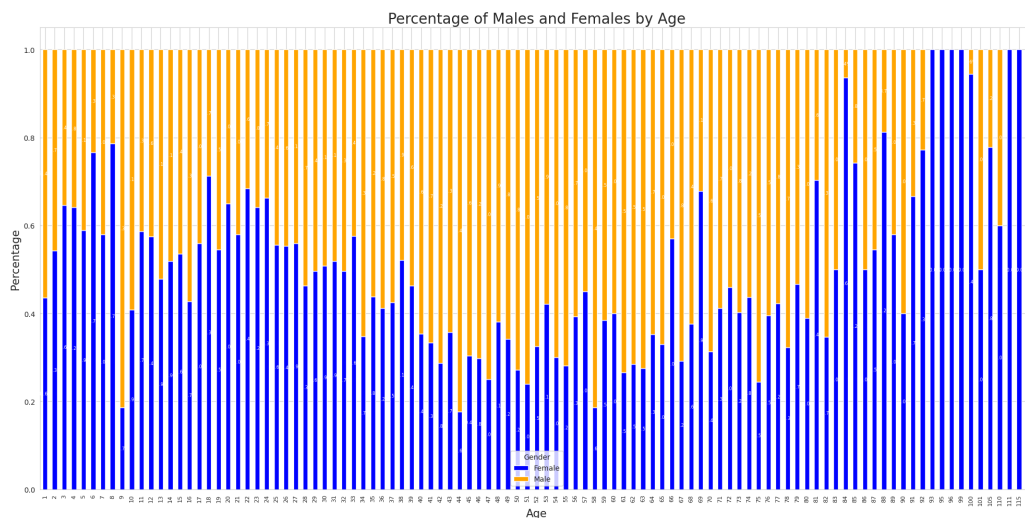



Figura 11.5: Distribuzione percentuale di maschi e femmine, per ciascuna età.

La ripartizione etnica è stata analizzata mediante grafici a torta e conteggi per età distinti per ciascun gruppo etnico, in modo da mettere in evidenza potenziali disparità demografiche all'interno delle fasce d'età.

```
sns.set_style("whitegrid")
plt.figure(figsize=(8,8))
plt.title("Distribution of Ethnic groups")
plt.pie(
    df.ethnic.value_counts(),
    labels=df.ethnic.value_counts().index,
    autopct='%1.1f%%')
plt.show()

for ethnic_group in df['ethnic'].unique():
    plt.figure(figsize=(20, 10))
    ethnic_df = df[df['ethnic'] == ethnic_group]
    sns.countplot(data=ethnic_df, x='age')
    plt.title(f'Age Distribution for {ethnic_group}')
```

```
plt.xlabel('Age')  
plt.ylabel('Count')  
plt.xticks(rotation=90)  
plt.show()
```

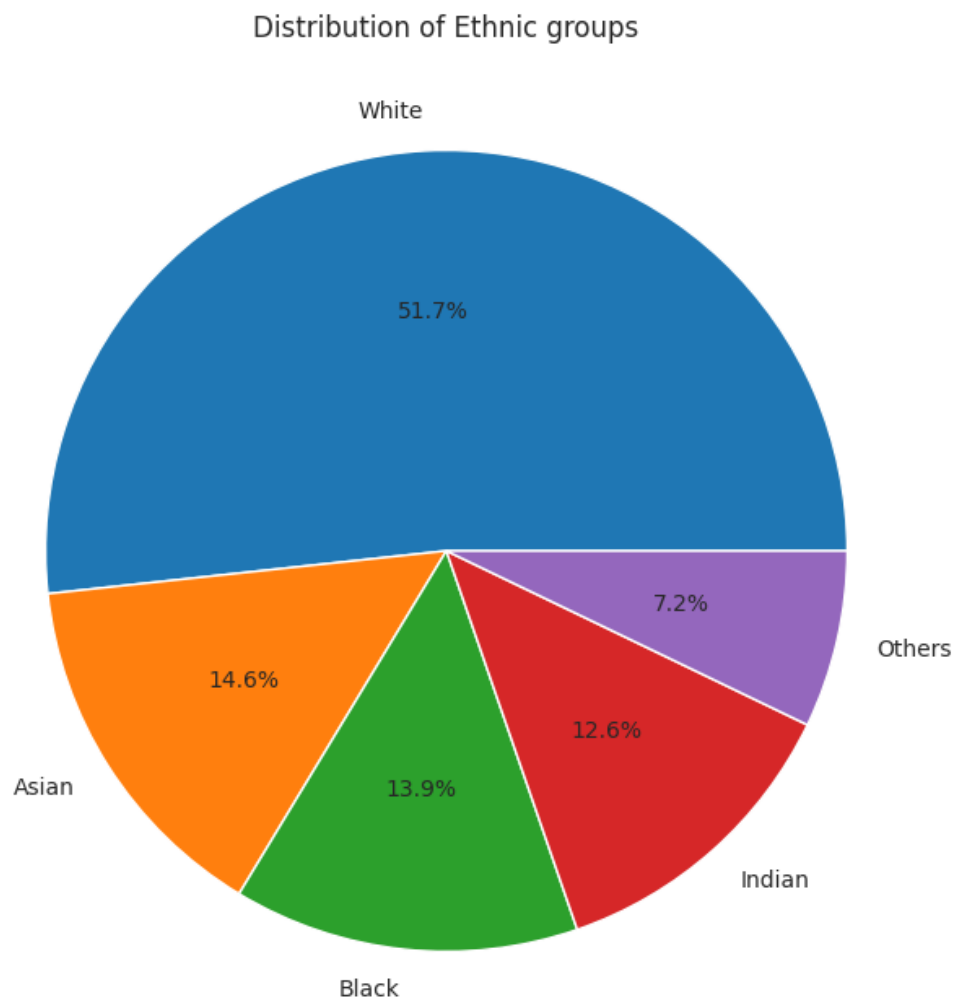


Figura 11.6: Distribuzione delle etnie in percentuale.

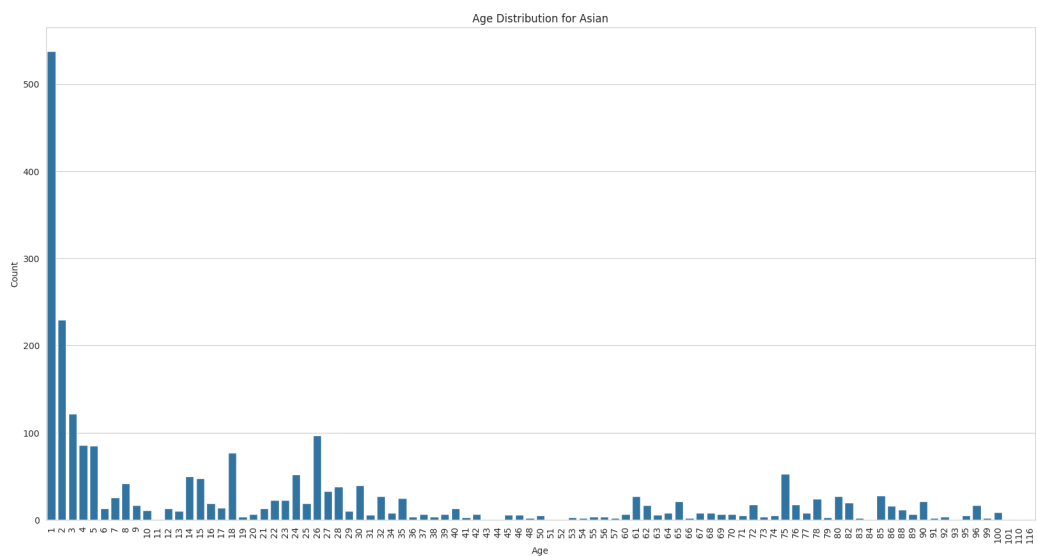


Figura 11.7: Distribuzione dell'etnia asiatica per età.

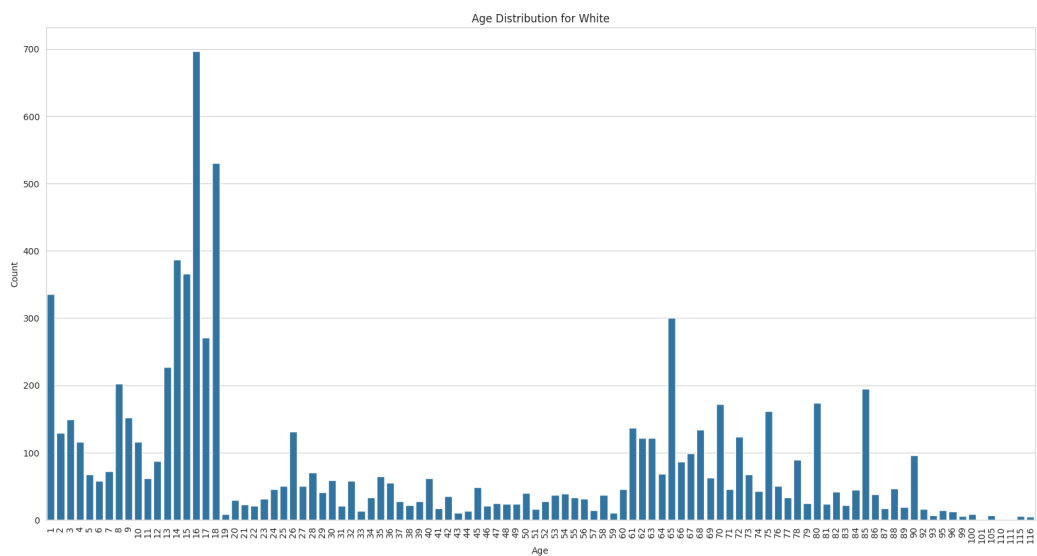


Figura 11.8: Distribuzione dell'etnia con persone di carnagione bianca per età.

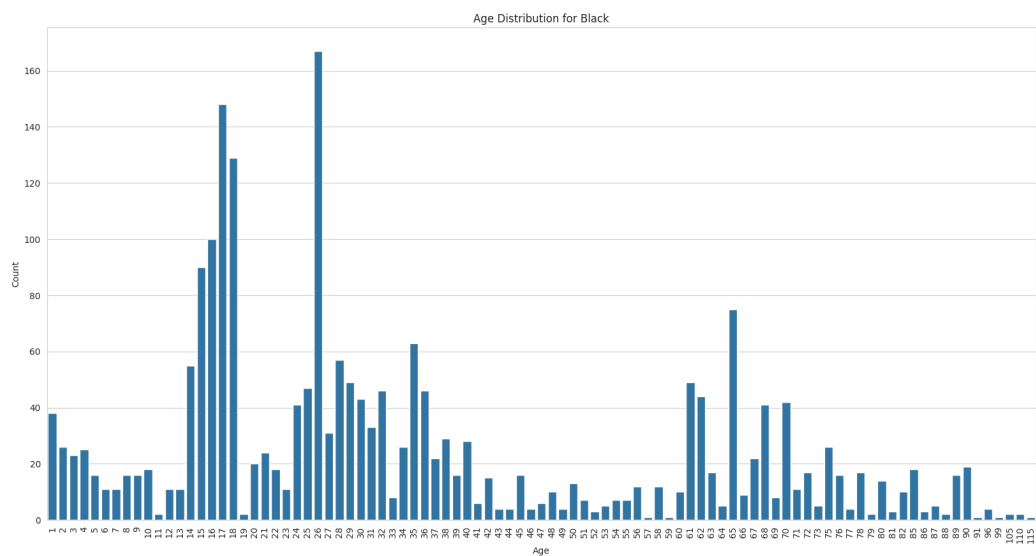


Figura 11.9: Distribuzione dell'etnia con persone di carnagione scura per età..

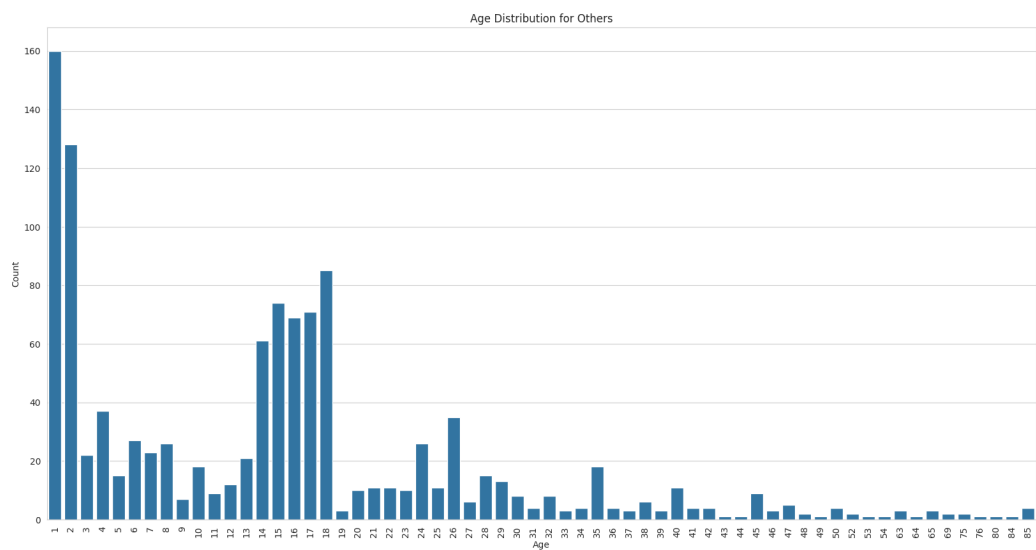


Figura 11.10: Distribuzione di altre etnie per età.

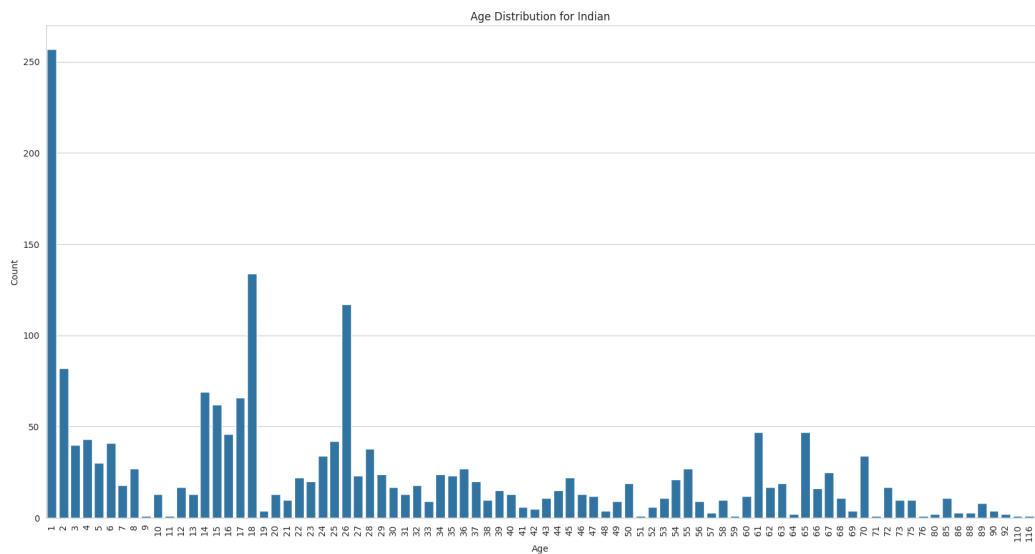


Figura 11.11: Distribuzione dell'etnia indiana per età.

L'interazione tra etnia e genere è stata ulteriormente indagata con un grafico a barre con suddivisione per genere, utile per rilevare concentrazioni di genere in specifici cluster etnici. Per ricapitolare la composizione delle classi d'età, è stato infine elaborato un grafico a torta riepilogativo che sintetizza la quota relativa di ciascuna fascia definita nella fase di preprocessing.

```
plt.figure(figsize=(10, 6))
sns.countplot(data=df, x='ethnic', hue='gender')
plt.title('Distribution of Gender by Ethnicity')
plt.xlabel('Ethnicity')
plt.ylabel('Count')
plt.legend(title='Gender')
plt.show()
```

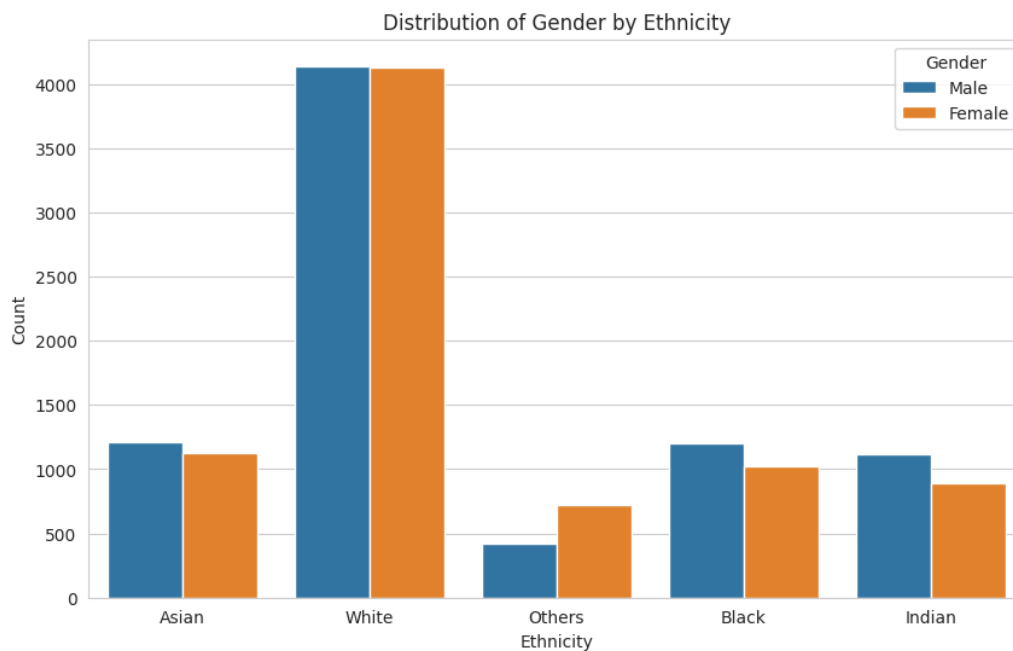


Figura 11.12: Distribuzione dei generi per etnia.

Per l'ispezione qualitativa, sono state generate delle griglie di immagini in formato 5×5 per triple "età–genere-etnia" specifiche. Tali griglie sono state impiegate per valutare variabilità intrinseca al dataset (pose, espressioni, illuminazione, qualità) e per motivare eventuali scelte di pulizia addizionale.

```
print(f'Choose one Age in:
      {sorted([int(age) for age in df["age"].unique()])}\n')
print(f'Choose one Gender in:
      {df["gender"].unique()}\n')
print(f'Choose one Ethnic (or "All") in:
      {df["ethnic"].unique()}\n')

age = 26
gender = 'Male'
ethnic = 'All'

if ethnic == 'All':
```

```

files = df.loc[(df['gender'] == gender)
               & (df['age'] == age)
              ]
else:
    files = df.loc[(df['gender'] == gender)
                   & (df['ethnic'] == ethnic)
                   & (df['age'] == age)
                  ]

plt.figure(figsize=(20,20))
for i, (index, row) in enumerate(files.head(25).iterrows()):
    plt.subplot(5,5, i+1)
    img = Image.open(row['image'])
    img = np.array(img)
    plt.imshow(img)
    plt.title(f"Age: {row['age']} ; Gender: {row['gender']}")
    plt.axis('off')

plt.show()

```



Figura 11.13: Esempio di Griglia di Immagini, in formato 5×5 , di maschi con 26 anni di qualsiasi etnia.

11.3 Addestramento dei modelli

11.3.1 Strategia di training comune e criteri di arresto

La strategia di addestramento è stata mantenuta omogenea per entrambe le architetture, con ottimizzatore Adam, perdita CrossEntropyLoss e batch size configurabile (tipicamente 32). Per prevenire l'overfitting è stato adottato

l'early stopping basato sulla validation loss, parametrizzato da patience e min_delta. La ricerca degli iperparametri è stata effettuata tramite Random Search abbinata a K-Fold cross-validation. La gestione dei dati immagine è centralizzata in una classe ImageDataset che si occupa di trasformazioni standard (resize, center crop, normalizzazione).

```
class CNNFeatureExtractor(nn.Module):
    def __init__(self, input_channels=3):
        super(CNNFeatureExtractor, self).__init__()
        self.conv1 = nn.Conv2d(input_channels, 6, 3, 1)
        self.conv2 = nn.Conv2d(6, 16, 3, 1)
        self.feature_dim = 16 * 54 * 54

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, self.feature_dim)
        return x

class CNN_MLP(nn.Module):
    def __init__(self, input_channels=3,
                 hidden_sizes=[120, 84, 20],
                 dropout=0.0, num_classes=6,
                 device='cpu'):
        super(CNN_MLP, self).__init__()
        self.cnn_features = CNNFeatureExtractor(input_channels)

        layers = []
        dim = self.cnn_features.feature_dim
        for hs in hidden_sizes:
            layers.append(nn.Linear(dim, hs))
```

```

        layers.append(nn.ReLU())
        layers.append(nn.Dropout(dropout))
        dim = hs
        layers.append(nn.Linear(dim, num_classes))
        self.mlp = nn.Sequential(*layers)

    def forward(self, x):
        features = self.cnn_features(x)
        return self.mlp(features)

class CNN_KAN(nn.Module):
    def __init__(self, input_channels=3, width=[8, 4],
        grid=5, k=3, num_classes=6,
        seed=0, device='cpu'):
        super(CNN_KAN, self).__init__()
        self.cnn_features = CNNFeatureExtractor(input_channels)

        kan_width = [self.cnn_features.feature_dim] +
            list(width) +
            [num_classes]
        self.kan = KAN(
            width=kan_width,
            grid=grid,
            k=k,
            seed=seed,
            device=device
        )

    def forward(self, x):
        features = self.cnn_features(x)
        return self.kan(features)

```

```

class ImageDataset(torch.utils.data.Dataset):
    def __init__(self, image_paths, labels, transform=None):
        self.image_paths = image_paths
        self.labels = labels
        self.transform = transform

        if self.transform is None:
            self.transform = transforms.Compose([
                transforms.Resize(224),
                transforms.CenterCrop(224),
                transforms.ToTensor(),
                transforms.Normalize([0.485, 0.456, 0.406],
                                     [0.229, 0.224, 0.225])
            ])

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        try:
            img = Image.open(self.image_paths[idx]).convert('RGB')
            img = self.transform(img)
            label = self.labels[idx]
            return img, label
        except Exception as e:
            print(f"Errore nel caricamento dell'immagine  

                  {self.image_paths[idx]}: {e}")
            img = torch.zeros(3, 224, 224)
            return img, self.labels[idx]

def create_image_train_test_sets(image_paths, labels, split=0.8):
    dataset_size = len(image_paths)
    train_size = int(dataset_size * split)

```

```

train_paths = image_paths[:train_size]
test_paths = image_paths[train_size:]
train_labels = labels[:train_size]
test_labels = labels[train_size:]

return train_paths, test_paths, train_labels, test_labels

def random_search_cnn(model_builder, param_dist,
    train_paths, train_labels,
    n_iter=10, cv_folds=5, batch_size=32,
    early_patience=5, early_min_delta=1e-4,
    class_weights=None, device='cpu'):
    train_keys = ['lr', 'l2_lambda']
    best_val_loss = float('inf')
    best_model_params, best_train_params = None, None
    best_model = None

    kf = KFold(n_splits=cv_folds, shuffle=True, random_state=42)

    print("Starting Random Search CNN with KFold Cross Validation...")

    for param_id, params in enumerate(
        ParameterSampler(
            param_dist, n_iter=n_iter, random_state=42)):
        print(f"Testing parameter set {param_id+1}/{n_iter}")

        model_params = {k: v for k, v in params.items()
            if k not in train_keys}
        train_params = {k: v for k, v in params.items()
            if k in train_keys}
        val_losses = []

```

```

indices = np.arange(len(train_paths))

for fold_idx, (train_idx, val_idx) in enumerate(
    kf.split(indices)):
    print(f"  Fold {fold_idx+1}/{cv_folds}")

    fold_train_paths = [train_paths[i] for i in train_idx]
    fold_train_labels = [train_labels[i] for i in train_idx]
    fold_val_paths = [train_paths[i] for i in val_idx]
    fold_val_labels = [train_labels[i] for i in val_idx]

    train_dataset = ImageDataset(fold_train_paths, fold_train_labels)
    val_dataset = ImageDataset(fold_val_paths, fold_val_labels)

    if class_weights is not None:
        sample_weights = np.array([class_weights.get(label, 1.0)
            for label in fold_train_labels])
        sampler = WeightedRandomSampler(
            weights=sample_weights,
            num_samples=len(sample_weights),
            replacement=True
        )
        train_loader = DataLoader(
            train_dataset,
            batch_size=batch_size,
            sampler=sampler)
    else:
        train_loader = DataLoader(
            train_dataset,
            batch_size=batch_size,
            shuffle=True)

```

```

val_loader = DataLoader(
    val_dataset,
    batch_size=batch_size,
    shuffle=False)

if 'width' in model_params:
    model = model_builder(**model_params, device=device)
    if isinstance(model, CNN_KAN):
        model.kan.speed()
else:
    model = model_builder(**model_params)
model.to(device)

optimizer = torch.optim.Adam(
    model.parameters(), lr=train_params['lr'])
criterion = nn.CrossEntropyLoss()
stopper = EarlyStopper(
    patience=early_patience, min_delta=early_min_delta)

for epoch in range(100):
    train_loss = train_cnn_epoch(
        model, train_loader, optimizer,
        criterion, device,
        l2_lambda=train_params.get('l2_lambda', 0.0))
    val_loss = eval_cnn_loss(
        model, val_loader, criterion, device)

    if epoch % 10 == 0:
        print(f"    Epoch {epoch}:
            train_loss = {train_loss:.6f},
            val_loss = {val_loss:.6f}")

    if stopper.early_stop(val_loss):

```

```

        print(f"    Early stopping at
              epoch {epoch},
              best_val_loss: {stopper.best_loss:.6f}")
        break

    final_val_loss = eval_cnn_loss(
        model, val_loader, criterion, device)
    val_losses.append(final_val_loss)

    mean_val = np.mean(val_losses)
    print(f"  Mean validation loss: {mean_val:.6f}")

    if mean_val < best_val_loss:
        best_val_loss = mean_val
        best_model_params = model_params
        best_train_params = train_params
        if 'width' in best_model_params:
            best_model = model_builder(
                **best_model_params,
                device=device).to(device)
        else:
            best_model = model_builder(
                **best_model_params).to(device)

    best_model.load_state_dict(model.state_dict())
    print(f"  New best validation loss:
          {best_val_loss:.6f}")

    print(f"\nBest validation loss: {best_val_loss:.6f}")
    return best_model, best_model_params, best_train_params

```

Di seguito sono riportate le griglie di iperparametri entro cui la Random

Search esplora le combinazioni, al fine di individuare quelle che forniscono i risultati migliori per ciascun modello.

CNN + MLP (Random Search: $n = 11$) Il modello adotta una rete convoluzionale come feature extractor, composta da due blocchi di convoluzione + pooling. Per un input di dimensione 224×224 , la prima convoluzione con kernel 3×3 e successivo max pooling riduce la dimensione spaziale a 111×111 . La seconda convoluzione seguita da max pooling produce una mappa finale di $16 \times 54 \times 54$, corrispondente a un vettore flatten di dimensione $\text{feature_dim} = 46,656$.

Questo vettore viene passato ad una MLP configurabile, in cui sono stati esplorati diversi insiemi di iperparametri, quali:

- **hidden_sizes**: [(120, 84, 20), (64, 32), (128, 64, 32)];
- **dropout**: [0.0, 0.2, 0.5];
- **lr**: [10^{-3} , 10^{-4}];
- **l2_lambda**: [0.0, 10^{-4} , 10^{-5}];

CNN + KAN (Random Search: $n = 15$) La variante CNN+KAN utilizza la stessa CNN di base per garantire comparabilità. Le feature estratte sono fornite come input ad una KAN. Questo vettore viene passato ad una KAN configurabile, in cui sono stati esplorati diversi insiemi di iperparametri, quali:

- **width**: [(8,4), (16,8), (32,16)];
- **grid**: [5, 10];
- **k**: [2, 3];
- **lr**: [10^{-3} , 10^{-4}];
- **l2_lambda**: [0.0, 10^{-4} , 10^{-5}];
- **seed**: 0 per riproducibilità.

11.3.2 Scelte architettureali finali

Nella Tabella 11.1 vengono mostrate le scelte finali, dopo l'ottimizzazione degli iperparametri, utilizzate per training, valutazioni comparative e studio di ablazione.

Tabella 11.1: Configurazioni finali dei modelli usati per il Training, dopo aver effettuato l'ottimizzazione degli iperparametri.

CNN + MLP	<code>input_channels = 3; hidden_sizes = (128, 64, 32); dropout = 0.5; lr = 10⁻³; l2_lambda = 10⁻⁴; num_classes = 4.</code> Early stopping applicato.
CNN + KAN	<code>input_channels = 3; width = (32, 16); grid = 10; k = 2, seed = 0; lr = 10⁻⁴; l2_lambda = 10⁻⁵; num_classes = 4.</code> Early stopping applicato.

11.4 Valutazione dei modelli

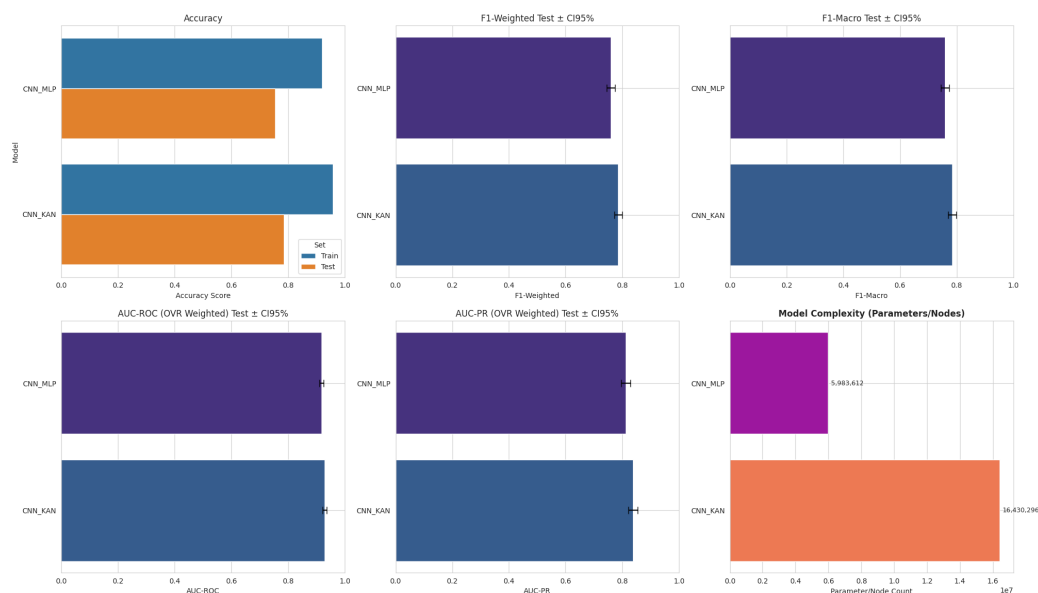


Figura 11.14: Confronto visivo delle prestazioni dei modelli (Accuracy train vs test, F1-weighted/macro, AUC-ROC e AUC-PR OVR weighted con CI95%, e complessità in parametri).

Classification report e confusion matrix per modello

CNN + MLP

Numero di parametri / nodi: **5 983 612**.

Tabella 11.2: Classification report

Classe	precision	recall	f1-score	support
0	0.85	0.71	0.78	775
1	0.91	0.80	0.85	815
2	0.57	0.75	0.65	807
3	0.78	0.75	0.76	803
accuracy		0.75		3 200
macro avg	0.78	0.75	0.76	3 200
weighted avg	0.78	0.75	0.76	3 200

Tabella 11.3: Confusion matrix

$$\begin{bmatrix} 553 & 21 & 158 & 43 \\ 22 & 649 & 126 & 18 \\ 49 & 38 & 607 & 113 \\ 28 & 5 & 168 & 602 \end{bmatrix}$$

AUC-ROC (OVR, weighted): **0.918**

AUC-PR (OVR, weighted): **0.813**

CNN + KAN

Numero di parametri / nodi: **16 430 296**.

Tabella 11.4: Classification report

Classe	precision	recall	f1-score	support
0	0.83	0.75	0.79	775
1	0.85	0.89	0.87	815
2	0.66	0.68	0.67	807
3	0.80	0.81	0.81	803
accuracy		0.78		3 200
macro avg	0.79	0.78	0.78	3 200
weighted avg	0.79	0.78	0.78	3 200

Tabella 11.5: Confusion matrix

$$\begin{bmatrix} 582 & 28 & 124 & 41 \\ 21 & 727 & 53 & 14 \\ 70 & 85 & 547 & 105 \\ 30 & 13 & 106 & 654 \end{bmatrix}$$

AUC-ROC (OVR, weighted): **0.928**

AUC-PR (OVR, weighted): **0.837**

11.4.1 Analisi dei risultati sperimentali

L'analisi complessiva delle prestazioni mostra differenze contenute ma significative tra le due architetture considerate. La **CNN+KAN** ottiene le migliori prestazioni aggregate sul test set, con un'accuratezza attorno a 0.78, un F1-weighted pari a circa 0.78 ed un F1-macro vicino a 0.78. I valori di AUC risultano anch'essi a favore della CNN+KAN (AUC-ROC OVR weighted = 0.93, AUC-PR OVR weighted = 0.84). La **CNN+MLP** presenta prestazioni leggermente inferiori sul test set con un'accuratezza = 0.75, con F1-weighted e macro intorno a 0.76, AUC-ROC = 0.92 e AUC-PR = 0.81. La differenza di performance, pur costante, rimane nell'ordine di pochi punti percentuali; è pertanto necessario pesare questi guadagni rispetto ai vincoli computazionali e di memoria.

L'analisi per classe, utilizzando i classification report e le confusion matrix, mostra che la classe 1 è la più stabile in entrambe le architetture, con precision e recall elevate. La classe 2 è quella che presenta il maggior compromesso precision/recall: nella CNN+MLP si osserva una tendenza all'over-prediction su tale classe, mentre la CNN+KAN riduce questo effetto migliorando sia precision sia recall. Le confusioni più frequenti coinvolgono classi adiacenti (fenomeno atteso per la classificazione d'età), suggerendo che molte errate assegnazioni sono dovute a sovrapposizioni tra fasce contigue piuttosto che a errori casuali.

11.4.2 Selezione del miglior modello

Tramite il calcolo del ranking multi-criterio, CNN_MLP é stato definito come il modello migliore su tutte le metriche ponderate, anche se a livello di pure performance CNN_KAN rimane il migliore. Nello specifico, CNN_MLP è risultato il modello vincente in Equal Weight (1:1), Complexity Weighted (1:2), Extreme Complexity (1:3) e Pareto Approach (40:60).

Di seguito la tabella riassuntiva con i valori principali usati per il ranking:

Tabella 11.6: Riepilogo ranking: conteggio parametri, performance media (rank-based) e ranks per metodo di aggregazione.

Model	Param_Count	Perf_Score	Compl_Rank	Equal_Rank	Ext_Rank	Pareto_Rank
CNN_MLP	5,983,612	2.0	1	1	1	1
CNN_KAN	16,430,296	1.0	2	1	2	2

11.4.3 Conclusioni

Dalla procedura di ranking multi-criterio e dall'analisi delle metriche aggregate emerge che la rete **CNN+KAN** raggiunge le migliori prestazioni complessive sul test set (F1 e AUC sensibilmente più alti), mentre la **CNN+MLP** presenta una complessità nettamente inferiore (circa 6M vs 16.4M parametri). I risultati sintetizzati nella Tabella di selezione mostrano chiaramente questo trade-off tra qualità predittiva e complessità computazionale.

Se l'obiettivo primario è massimizzare le performance e le risorse computazionali non sono vincolanti, la scelta preferibile è la **CNN+KAN** per il suo livello di accuratezza e per la maggiore robustezza osservata su alcune classi difficili. Se invece il vincolo dominante è la complessità, la **CNN+MLP** rappresenta il modello raccomandato: offre prestazioni ancora competitive con costi di deployment significativamente inferiori e quindi un miglior compromesso per scenari reali a bassa latenza o risorse limitate.

La classifica, basata sul criterio "complexity-weighted" (dal migliore al peggiore), è la seguente:

1. CNN_MLP (Params: 5,983,612; F1-Weighted: 0.7595)
2. CNN_KAN (Params: 16,430,296; F1-Weighted: 0.7843)

Conclusioni

Sintesi dei contributi scientifici e metodologici

La presente tesi ha condotto un'indagine metodologica ed applicativa approfondita su quattro paradigmi di apprendimento automatico: le reti neurali classiche (MLP), una loro recente evoluzione (KAN) e due modelli ensemble basati su alberi (Random Forest e XGBoost). L'obiettivo primario era duplice: da un lato, fornire un'analisi teorica esaustiva di ciascun modello, esaminandone architettura, fondamenti matematici e funzionamento operativo; dall'altro, valutare sperimentalmente le loro prestazioni su tre distinti casi di studio del mondo reale, che comprendevano problemi di regressione su dati tabellari, di classificazione su serie storiche e di classificazione su immagini. Un aspetto centrale della ricerca è stata l'implementazione di un rigoroso framework sperimentale volto a garantire stime robuste e non distorte della capacità di generalizzazione dei modelli. Sono state impiegate metodologie di validazione avanzate, come la Nested Cross-Validation per i dati non temporali e la Time Series Cross-Validation per i dati con dipendenza temporale, al fine di evitare il data leakage e simulare in modo fedele lo scenario di deployment. Per la selezione degli iperparametri è stato scelto il Random Search, ritenuto il miglior compromesso tra efficienza esplorativa e scalabilità in contesti con spazi di ricerca ampi. A completamento, è stato condotto un esteso studio di ablazione post-addestramento, che ha valutato il compromesso tra la complessità e le prestazioni dei modelli attraverso tecniche di pruning mirate.

Risultati dei casi di studio

I risultati empirici hanno evidenziato che la scelta del modello ottimale non è universale, ma dipende strettamente dalla natura del problema e dai vincoli operativi, in particolare il compromesso tra performance e complessità.

Primo Caso Studio: Regressione su emissioni di automobili

In questo task di regressione su dati tabellari eterogenei, il modello XGBoost ha dimostrato una superiorità schiacciante, ottenendo un R^2 in test di 0.9956 ed un errore quadratico medio (MSE) di appena 17.52. Ha superato nettamente la Random Forest ($R^2 = 0.8704$), che a sua volta ha sovraperformato le reti neurali. Questo risultato conferma l'eccezionale efficacia degli algoritmi di *gradient boosting* su dati strutturati, dove la loro natura additiva e sequenziale, che corregge progressivamente gli errori, si dimostra particolarmente adatta a modellare relazioni complesse e non lineari. Il ranking multi-criterio ha confermato la dominanza di XGBoost, che è risultato il migliore in tutte le strategie di ponderazione (Equal Weight, Complexity Weighted, ecc.), grazie all'ottimale equilibrio tra performance ed un numero di parametri molto contenuto (circa 15k).

Secondo Caso Studio: Classificazione di PM2.5

Per il problema di classificazione su serie storiche, le performance dei modelli si sono allineate su valori aggregati molto simili. Sebbene XGBoost abbia mantenuto il primato per le metriche assolute più elevate (F1-weighted di 0.8107), l'analisi del ranking multi-criterio ha rivelato un'altra dinamica. Penalizzando la complessità del modello, le KAN sono emerse come la scelta raccomandata. Con un numero di parametri di circa 7,680 contro i 90,416 di XGBoost ed i 412,430 di Random Forest, le KAN hanno fornito un F1-weighted di 0.7823, un valore leggermente inferiore ma in un modello drasticamente più leggero. Questo risultato dimostra come in scenari reali, dove vincoli di memoria, velocità e scalabilità sono cruciali, un modello con un rapporto favorevole tra performance e complessità parametrica possa

essere preferibile rispetto ad un modello che offre un guadagno marginale di accuratezza ad un costo computazionale molto superiore.

Terzo Caso Studio: Classificazione di Fasce d'Età tramite Immagini

Questo caso di studio ha richiesto un'architettura ibrida, utilizzando una CNN come estrattore di feature, abbinata ad un classificatore finale basato su MLP o KAN. L'architettura CNN+KAN ha ottenuto una performance leggermente superiore (F1-weighted di 0.7843) rispetto alla CNN+MLP (F1-weighted di 0.7595). Tuttavia, questa piccola differenza è stata ottenuta ad un costo parametrico significativo, con la KAN che ha richiesto circa 16.4M di parametri contro i 6M della MLP. L'analisi di ranking multi-criterio, che ha privilegiato la parsimonia, ha quindi indicato la CNN+MLP come il modello raccomandato. Questa scoperta suggerisce che, pur con i suoi vantaggi teorici, il costo parametrico della KAN, specialmente in architetture complesse, può renderla una soluzione meno pratica e scalabile rispetto a un classificatore più convenzionale e leggero.

Sintesi finale: Trade-off, Robustezza e Pruning

I risultati di questo lavoro possono essere riassunti in una serie di raccomandazioni pratiche basate sul problema e sui vincoli operativi. Per ottenere la massima accuratezza con dati tabellari, XGBoost è emerso come la scelta ottimale. Quando si affrontano problemi più complessi, come le serie storiche, e si ha un vincolo di complessità, le KAN offrono un equilibrio superiore. Infine, per l'analisi di immagini, l'architettura CNN+MLP rappresenta un'alternativa più efficiente, a livello computazionale, rispetto ad una configurazione più complessa come la CNN+KAN.

Lo studio di ablazione ha rivelato differenze profonde nella robustezza e comprimibilità dei modelli. Gli ensemble, in particolare la Random Forest, hanno mostrato una straordinaria resilienza: il modello poteva essere ridotto del 90% mantenendo o persino migliorando leggermente

le prestazioni di generalizzazione. Similmente, XGBoost ha mostrato una perdita trascurabile di performance dopo un pruning del 90% nel primo caso studio. Questa notevole comprimibilità suggerisce un alto grado di ridondanza strutturale e di complementarità tra i weak learner, rendendoli ideali per l'ottimizzazione in ambienti con risorse limitate. Al contrario, le reti neurali si sono dimostrate più fragili. La MLP ha tollerato un pruning fino al 50% senza un degrado significativo, ma oltre questa soglia ha manifestato una rapida perdita di performance. Le KAN, sebbene più resistenti rispetto alle MLP, hanno mostrato un degrado più marcato con pruning superiore al 70%. La minore tolleranza al pruning suggerisce che la densa parametrizzazione delle reti neurali, pur essendo efficiente per l'approssimazione funzionale, potrebbe non contenere lo stesso grado di ridondanza sfruttabile per la compressione che si ritrova negli ensemble.

Considerazioni finali

Questa tesi ha fornito un quadro di valutazione basato su dati empirici che supporta la scelta del modello in diversi scenari applicativi, dimostrando che l'efficacia di un algoritmo va ben oltre la sua performance grezza. L'analisi integrata di accuratezza, complessità e comprimibilità offre una guida concreta per il deployment in contesti operativi. In particolare, il lavoro ha messo in luce che, mentre XGBoost è lo standard per eccellenza su dati tabellari, le KAN offrono una valida e leggera alternativa con un equilibrio unico in termini di performance e complessità. I risultati sullo studio di ablazione sottolineano un vantaggio cruciale degli ensemble, ovvero la loro intrinseca robustezza alla compressione, un fattore determinante per l'adozione di modelli su dispositivi edge.

Bibliografia

- [1] Popescu et al. , *Multilayer perceptron and neural networks*, 2009. 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
- [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] Liu et al. , *Kan: Kolmogorov-arnold networks*, 2024. <https://arxiv.org/pdf/2404.19756>
- [7] 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.
- [8] V.I. Arnold, *On functions of three variables*, *Doklady Akademii Nauk SSSR*, vol. 141, no. 4, 1963.
- [9] A. Chaudhuri, *B-Splines*, 2021. <https://arxiv.org/pdf/2108.06617>

- [10] J. Henseler, *Back Propagation*, 1995. <https://link.springer.com/chapter/10.1007/BFb0027022>
- [11] 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.
- [12] R. Yu, *KAN or MLP: A Fairer Comparison*, 2024. <https://arxiv.org/pdf/2407.16674v1>
- [13] L. Breiman, *Bagging Predictors*, vol. 24, 1996. <https://link.springer.com/article/10.1007/BF00058655>
- [14] L. Breiman, *Random Forests*, vol. 45, 2001. <https://link.springer.com/article/10.1023/A:1010933404324>
- [15] 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.
- [16] J. H. Friedman, *Greedy function approximation: A gradient boosting machine*, 2001. <https://jerryfriedman.su.domains/ftp/trebst.pdf>
- [17] T. Chen, C. Guestrin, *XGBoost: A Scalable Tree Boosting System*, 2016. <https://arxiv.org/pdf/1603.02754>
- [18] S. Fatima, *XGBoost and Random Forest Algorithms: An in Depth Analysis*, 2023. https://www.researchgate.net/publication/377135877_XGBoost_and_Random_Forest_Algorithms_An_in_Depth_Analysis
- [19] A. E. Hoerl, R. W. Kennard, *Ridge regression: Biased estimation for nonorthogonal problems*, 1970. <https://homepages.math.uic.edu/~lreyzin/papers/ridge.pdf>

- [20] 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)
- [21] H. Zou, T. Hastie, *Regularization and variable selection via the elastic net*, 2005. <https://academic.oup.com/jrsssb/article-abstract/67/2/301/7109482>
- [22] N. Srivastava et al. , *Dropout: A simple way to prevent neural networks from overfitting*, 2014. <https://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf>
- [23] K. O'Shea, R. Nash, *An Introduction to Convolutional Neural Networks*, 2015. <https://arxiv.org/pdf/1511.08458>
- [24] A. Krizhevsky et al. , *ImageNet Classification with Deep Convolutional Neural Networks*, 2012. https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf
- [25] K. Simonyan, A. Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2015. <https://arxiv.org/pdf/1409.1556>
- [26] S. Ioffe, C. Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015. <https://arxiv.org/pdf/1502.03167>
- [27] J.L. Ba et al. , *Layer Normalization*, 2016. <https://arxiv.org/pdf/1607.06450>
- [28] Y. Wu, K. He, *Group Normalization*, 2018. <https://arxiv.org/pdf/1803.08494>
- [29] R. Poojary, *Effect of data-augmentation on fine-tuned CNN model performance*, 2020. https://www.researchgate.net/publication/347437393_Effect_of_data-augmentation_on_fine-tuned_CNN_model_performance

- [30] J. Bergstra , Y. Bengio, *Random Search for Hyper-Parameter Optimization*, 2012. <https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf>
- [31] L. Franceschi et al. , *Hyperparameter Optimization in Machine Learning*, 2024. <https://arxiv.org/pdf/2410.22854>
- [32] J. Snoek et al. , *Practical Bayesian Optimization of Machine Learning Algorithms*, 2012 <https://arxiv.org/pdf/1206.2944>
- [33] H. Alibrahim, S.A. Ludwig, *Hyperparameter Optimization: Comparing Genetic Algorithm against Grid Search and Bayesian Optimization*, 2021. <https://web.cs.ndsu.nodak.edu/~siludwig/Publish/papers/CEC2021.pdf>
- [34] J. Wainer, G. Cawley, *Nested cross-validation when selecting classifiers is overzealous for most practical applications*, 2018. <https://arxiv.org/pdf/1809.09446>
- [35] A. Deng, *Time series cross validation: A theoretical result and finite sample performance*, 2023. <https://www.sciencedirect.com/science/article/abs/pii/S0165176523003944>
- [36] G. Tsoumakas, *An Ensemble Pruning Primer*, 1970. https://www.researchgate.net/publication/225606257_An_Ensemble_Pruning_Primer
- [37] S. Vadera, S. Ameen, *Methods for Pruning Deep Neural Networks*, 2021. <https://arxiv.org/pdf/2011.00241>
- [38] R. Meyes, *Ablation Studies in Artificial Neural Networks*, 2019. https://www.researchgate.net/publication/330672937_Ablation_Studies_in_Artificial_Neural_Networks
- [39] F.T. Liu et al. , *Isolation Forest*, 2009. https://www.researchgate.net/publication/224384174_Isolation_Forest

- [40] N.W. Chawla et al. , *SMOTE: Synthetic Minority Over-sampling Technique*, 2011.<https://arxiv.org/pdf/1106.1813>
- [41] P. Olivi et al. , *Car Emissions Dataset*, 2024. <https://github.com/pietroolivi/dia-project>