Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Dipartimento di Fisica e Astronomia
Corso di Laurea in Fisica

Reti Neurali di Kolmogorov-Arnold
Un'Introduzione

Relatore:
Prof. Mirko Degli Esposti

Presentata da:
Luca Vincenzo Spallanzani

Anno Accademico 2023/2024

Sommario

Questo lavoro presenta le reti neurali di Kolmogorov-Arnold (KANs) come alternativa alle reti neurali multistrato (MLPs), approfondendone potenzialità e limiti attraverso un'analisi comparativa. Viene evidenziata l'importanza teorica del teorema di Kolmogorov-Arnold (KART) sia per la matematica e la fisica, che come base formale per le KANs, consentendo di affrontare sfide comuni nelle MLPs, come la *curse of dimensionality* e la *catastrophic forgetting*. A supporto di questa iniziale disanima teorica, viene infine presentato e commentato del codice che, oltre a rafforzare la comprensione di queste reti con esempi concreti, ne evidenzia l'intuitività e l'interpretabilità. Attraverso una rassegna critica della letteratura e l'analisi di implementazioni pratiche, la tesi mette in luce le prerogative delle KANs, che potrebbero migliorare le capacità e l'efficienza degli attuali modelli. Questo trattazione vuole offrire, pertanto, un'introduzione dettagliata e critica a queste reti, mettendo in risalto sia le loro potenzialità che le loro attuali limitazioni.

Indice

Sommario **1**

Indice **2**

Introduzione **3**

0.1 Il Contesto . . . 4
0.2 KANs, la Nuova Speranza . . . 5
0.3 Obiettivi e Struttura del Lavoro . . . 6

1 Il Teorema di Kolmogorov-Arnold **8**

1.1 Genesi del Teorema . . . 8
1.2 Enunciato . . . 9
1.3 Rilevanza ed Applicazioni . . . 11
1.4 Il KART e le Reti Neurali . . . 13

2 Architettura e funzionamento delle KANs **17**

2.1 Introduzione . . . 17
2.2 Architettura di Base delle KANs . . . 18
2.3 Funzioni di Attivazione Basate sui B-splines . . . 21
2.4 La Grid ed il suo Ruolo nella Parametrizzazione delle Funzioni di Attivazione 24
2.5 Altre Funzioni di Attivazione . . . 26
2.6 Addestramento delle KANs . . . 29
2.7 Possibili Estensioni e Varianti delle KANs . . . 32

3 KANs ed MLPs: un Confronto **33**

3.1 Multi-Layer Perceptrons . . . 33
3.2 MLPs vs KANs . . . 36
3.3 Problematiche delle MLPs . . . 38
3.4 Parametri Tipici delle Reti Neurali . . . 41

4 Implementazione in Python **47**

4.1 Struttura del Capitolo . . . 47
4.2 Function Fitting . . . 48
4.2.1 Introduzione al Problema . . . 48
4.2.2 Commento al Codice . . . 48
4.3 PDEs Resolution . . . 58
4.3.1 Introduzione al Problema . . . 58
4.3.2 Commento al Codice . . . 59

Conclusioni **70**

Bibliografia **72**

Introduzione

0.1 Il Contesto

Negli ultimi anni, l'intelligenza artificiale (AI, dall'inglese "artificial intelligence") è divenuta una presenza tangibile e pervasiva nelle nostre vite quotidiane. Sebbene la ricerca sull'AI abbia radici profonde, che risalgono a diversi decenni fa, è solo in tempi recentissimi che le sue applicazioni hanno iniziato a permeare l'immaginario collettivo e a influenzare in modo diretto la vita delle persone: non riguardando più unicamente gli "addetti ai lavori", ma sempre di più anche la quotidianeità delle persone comuni. Tecnologie come il riconoscimento vocale, gli assistenti virtuali e l'automazione nei campi più svariati stanno ridefinendo il modo in cui interagiamo con il mondo, dimostrando l'impatto significativo che l'AI già possiede sulla società odierna.

La storia dello sviluppo dell'intelligenza artificiale è caratterizzata da momenti paradigmatici che ne hanno segnato l'evoluzione. Uno dei primi contributi significativi fu il "perceptron", sviluppato da Frank Rosenblatt alla fine degli anni '50. Questo modello, ispirato alla struttura del neurone umano, rappresentò un primo tentativo di sviluppare un sistema capace di apprendere autonomamente attraverso l'esperienza. Nonostante le limitazioni tecniche dell'epoca, il *perceptron* aprì la strada ad una nuova era di ricerca nel campo delle reti neurali.

Successivamente, nel corso degli anni '80, l'introduzione delle reti neurali multistrato (MLPs, dall'inglese "multi-layer perceptron") e lo sviluppo dell' "algoritmo di backpropagation", segnò un altro punto di svolta: questo algoritmo, introdotto da Rumelhart, Hinton e Williams, permise per la prima volta alle reti neurali di "apprendere"efficacemente, superando molte delle difficoltà che avevano afflitto i modelli precedenti. Le MLPs rappresentarono una svolta fondamentale, consentendo l'emulazione di processi cognitivi umani complessi.

Nonostante la ricerca sull'AI e sulle MLPs sia sempre stata prolifica, i veri progressi sono arrivati soltanto negli ultimi due decenni, grazie all'aumento della potenza di calcolo e alla disponibilità di enormi quantità di dati. Tuttavia, nonostante i successi ottenuti, le MLPs presentano diverse limitazioni, il che ha spinto i ricercatori a esplorare nuove possibili soluzioni all'interno della ricerca sulle reti neurali [1].

In questo contesto, le reti neurali di Kolmogorov-Arnold (KANs, dove "N"viene dall'inglese per "network", ovvero appunto "rete") emergono come una possibile soluzione innovativa, con l'obiettivo di poter risolvere alcune delle sfide più importanti presentate dalle MLPs. Basate su principi matematici differenti, le KANs promettono di offrire nuove prospettive per il futuro dell'intelligenza artificiale.

0.2 KANs, la Nuova Speranza

Le reti neurali di Kolmogorov-Arnold rappresentano una nuovissima e promettente architettura nel panorama delle reti neurali. Alla base delle KANs vi è il teorema di Kolmogorov-Arnold (KART, dall'inglese "Kolmogorov-Arnold representation theorem"), che soppianta il teorema di rappresentazione tradizionalmente utilizzato nelle reti neurali multistrato. Il KART, formulato negli anni '50 dai matematici russi Andrey Kolmogorov e Vladimir Arnold, ha rappresentato un risultato matematico di grande rilevanza, introducendo la possibilità di rappresentare funzioni complesse con un approccio matematico diverso da quello adottato all'interno delle MLPs. Nonostante l'importanza teorica del KART, i tentativi iniziali di applicarlo all'intelligenza artificiale non portarono ai risultati sperati. Le difficoltà tecniche e computazionali incontrate nell'implementazione di questo teorema spinsero i ricercatori a scartare ben presto questa linea di ricerca a favore delle MLPs, che, pur con le loro limitazioni, offrivano soluzioni maggiormente praticabili per l'apprendimento automatico ai suoi albori.

La situazione potrebbe però cambiare a seguito della recente pubblicazione dell'articolo *"Kolmogorov-Arnold Neural Networks: A New Paradigm for Function Approximation"*, ad aprile di quest'anno [2]. Questo lavoro ha introdotto tecniche innovative per l'ottimizzazione e la gestione delle reti in oggetto, risolvendo molte delle problematiche che avevano impedito l'adozione del KART per le reti neurali in passato. È stato proprio questo lavoro a rappresentare l'epicentro dell'enorme entusiasmo generatosi in questi ultimi mesi attorno alle KANs. Tuttavia, dopo così poco tempo da questa pubblicazione, la ricerca attorno a questa nuova tecnologia è inevitabilmente in fase embrionale. Questo significa che ancora ci troviamo nella fase in cui i ricercatori cercano di verificare se le KANs possano effettivamente rivoluzionare il mondo dell'intelligenza artificiale, o se si tratta meramente di un'illusione destinata a svanire.

La novità delle KANs non risiede solo nel teorema che ne sta alla base, ma anche nella loro struttura unica, dove le funzioni di attivazione giocano un ruolo rinnovato e fondamentale. Questo aspetto differenzia le KANs dalle MLPs, suggerendo potenziali miglioramenti nella capacità di rappresentare funzioni complesse. Ciononostante, per quanto riguarda l'apprendimento, le KANs fanno ancora affidamento su tecniche già consolidate nello studio delle MLPs, come l'*algoritmo di backpropagation*, che migliora iterativamente le prestazioni della rete attraverso l'ottimizzazione dei suoi parametri.

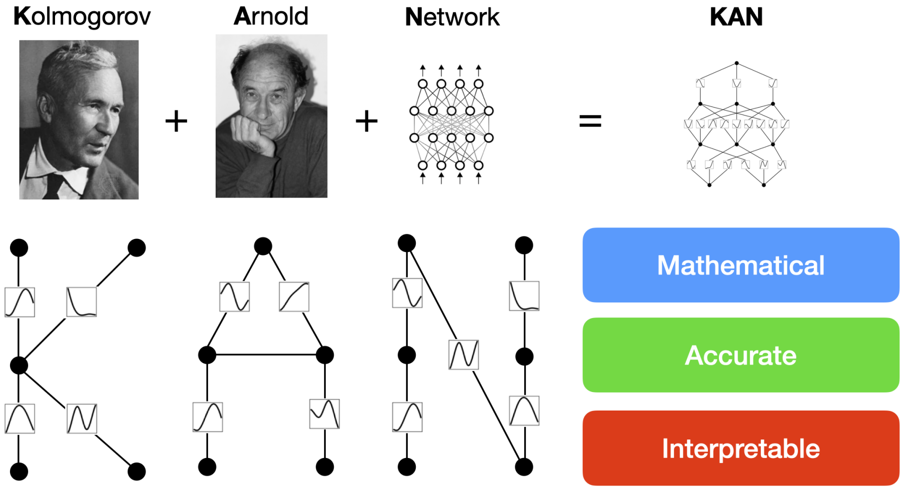


Figura 1: *Caratteristiche delle KANs; in foto i padri del teorema. Illustrazione da: [2].*

Le MLPs, pur essendo state al centro della ricerca sull'intelligenza artificiale per decenni, presentano, come già accennato, diversi problemi intrinseci, come la "curse of dimensionality"(che si potrebbe tradurre in italiano come "maledizione della dimensionalità"), la difficoltà ad apprendere funzioni altamente non lineari, e la cosiddetta "catastrophic forgetting"(che si potrebbe tradurre in italiano come "dimenticanza catastrofica")[1]. Negli ultimi anni, i ricercatori hanno cercato di migliorare l'efficienza delle MLPs, ma senza riuscire a risolvere completamente queste problematiche di fondo. Proprio per questo motivo, esiste da tempo un forte desiderio nella comunità scientifica di trovare un nuovo paradigma che possa superare i limiti di queste reti tradizionali. L'enorme aspettativa attorno alle KANs è alimentata dunque dalla speranza che queste reti possano rappresentare quella tanto attesa svolta, capace di fornire soluzioni innovative ai problemi sopracitati.

0.3 Obiettivi e Struttura del Lavoro

La presente tesi si propone di esplorare le reti neurali di Kolmogorov-Arnold, delineandone i fondamenti teorici e confrontando questa innovativa architettura con le reti neurali multistrato. L'intento è fornire una visione d'insieme delle KANs, sia mettendone in risalto i punti di forza che mostrandone gli aspetti richiedenti ulteriore indagine. In particolare, la trattazione si articolerà nei seguenti punti fondamentali:

- Analisi del teorema di Kolmogorov-Arnold dal punto di vista matematico, con una discussione delle sue applicazioni all'interno delle KANs e non solo.

[1] Questi problemi saranno approfonditi più avanti nel corso del lavoro.

- Disanima dell'architettura e dell'efficienza delle KANs, il tutto in confronto alle MLPs.
- Esplorazione dei campi di applicazione che, allo stato attuale delle ricerche, appaiono più promettenti per questo nuovo tipo di reti neurali.
- Presentazione dei dibattiti attuali nella comunità scientifica riguardanti il possibile ruolo delle KANs nel superare le limitazioni delle MLPs, e analisi delle future direzioni di ricerca necessarie per confermare la superiorità di questa architettura per l'intero mondo dell'AI.

Il lavoro è organizzato in cinque capitoli principali, ciascuno dei quali si concentra su un aspetto specifico delle KANs e/o del loro confronto con le MLPs:

1. Introduzione al teorema di Kolmogorov-Arnold, con un'analisi matematica del relativo enunciato e una discussione delle sue principali applicazioni in fisica.
2. Descrizione della struttura e del funzionamento delle KANs, con particolare attenzione ai nuovi ruoli rivestiti da nodi, archi e funzioni di attivazione.
3. Confronto tra KANs e MLPs, con un'analisi delle differenze in termini di efficienza, velocità di apprendimento, *catastrophic forgetting*, *curse of dimensionality*, interpretabilità e interattività.
4. Presentazione e commento di codice sviluppato in Python per l'implementazione di semplici reti KAN per problemi di *functiong fitting* e risoluzione di PDEs.
5. Conclusioni, con una sintesi dei principali spunti esplorati nel corso del lavoro, ed una breve riflessione sul possibile futuro delle KANs.

Capitolo 1

Il Teorema di Kolmogorov-Arnold

1.1 Genesi del Teorema

Il teorema di Kolmogorov-Arnold rappresenta una delle pietre miliari nella teoria delle funzioni, stabilendo risultati sorprendenti riguardo alla rappresentazione di funzioni continue di più variabili[1]. Formulato negli anni '50 dal matematico sovietico Andrey Kolmogorov e successivamente perfezionato dal suo allievo Vladimir Arnold, questo teorema ha rivoluzionato la comprensione delle strutture funzionali, dimostrando che qualsiasi funzione continua di più variabili può essere espressa come somma di funzioni univariate. Questo risultato ha avuto implicazioni profonde non solo nel campo della matematica pura, ma anche in discipline applicate, contribuendo a sviluppare metodi più efficienti per l'approssimazione di funzioni complesse.

Il contesto storico in cui è nato il teorema di Kolmogorov-Arnold è fondamentale per comprenderne l'importanza. Durante il periodo di intenso sviluppo della matematica in Unione Sovietica, che ebbe luogo principalmente tra gli anni '30 e '60 del XX secolo[2], la ricerca si concentrava su problemi di rappresentazione delle funzioni e approssimazione, in particolare nel contesto di funzioni definite in spazi ad alta dimensionalità[3]. Andrey Kolmogorov, già noto per i suoi contributi fondamentali alla teoria della probabilità, si dedicò allo studio della rappresentazione delle funzioni in questi spazi complessi. Uno dei suoi risultati preliminari più significativi fu la dimostrazione che qualsiasi funzione continua di due variabili può essere rappresentata come la somma di tre funzioni continue

[1]La rappresentazione di funzioni si riferisce alla possibilità di esprimere una funzione in una forma che ne semplifichi l'analisi e il calcolo, spesso scomponendola in componenti più semplici, che, come vedremo tra poco, è esattamente quello che fa il KART.

[2]Tale periodo è spesso considerato l'epoca d'oro della matematica sovietica, durante il quale emersero numerosi matematici di spicco e furono sviluppate teorie fondamentali in vari campi della matematica.

[3]Le funzioni in spazi ad alta dimensionalità si riferiscono a funzioni che dipendono da molte variabili indipendenti. In questi contesti, la complessità aumenta esponenzialmente con il numero di dimensioni, rendendo difficile la loro rappresentazione e approssimazione.

di una sola variabile. Tuttavia, fu Vladimir Arnold, suo brillante allievo, a generalizzare questo risultato, dimostrando che il teorema poteva essere esteso a funzioni di un numero arbitrario di variabili, portando così alla formulazione completa del teorema che oggi porta il nome di entrambi [3,4]. Questa generalizzazione ha permesso di superare le limitazioni imposte dalle rappresentazioni tradizionali, aprendo nuove strade per l'analisi di sistemi complessi.

Questo teorema non solo ha aperto nuove prospettive nella teoria delle funzioni, ma ha anche avuto un impatto significativo in numerosi altri campi, tra cui la fisica teorica, l'analisi numerica e, più recentemente, l'intelligenza artificiale. In particolare, l'applicazione di questo teorema nelle reti KANs rappresenta una delle innovazioni più interessanti degli ultimi anni, offrendo nuove soluzioni ai problemi di rappresentazione e apprendimento nelle reti neurali tradizionali [5]. Le potenzialità del teorema continuano a essere esplorate, con nuove applicazioni che emergono costantemente, dimostrando la versatilità e l'importanza duratura di questo risultato matematico.

1.2 Enunciato

Il **teorema di rappresentazione di Kolmogorov-Arnold** stabilisce che per ogni funzione continua $f : \mathbb{R}^n \to \mathbb{R}$, esistono funzioni continue univariate $\phi_{ij}, g_j : \mathbb{R} \to \mathbb{R}$ tali che:

$$f(x_1, x_2, \ldots, x_n) = \sum_{j=1}^{2n+1} g_j \left(\sum_{i=1}^{n} \phi_{ij}(x_i) \right) \tag{1.1}$$

dove:

- $\phi_{ij}(x_i)$ sono funzioni continue univariate che trasformano ciascuna variabile x_i in un valore reale $\phi_{ij}(x_i)$. L'indice i varia da 1 a n, mentre j varia da 1 a $2n + 1$.
- La somma interna $\sum_{i=1}^{n} \phi_{ij}(x_i)$ rappresenta una combinazione lineare dei valori $\phi_{ij}(x_i)$, dove per ogni j, i valori di i vanno da 1 a n. Questa somma produce un singolo valore reale per ogni j.
- $g_j(\cdot)$ sono funzioni continue univariate che mappano il risultato della somma interna in un valore reale $g_j : \mathbb{R} \to \mathbb{R}$. Ogni funzione g_j agisce su una combinazione lineare delle $\phi_{ij}(x_i)$.
- La somma esterna $\sum_{j=1}^{2n+1} g_j$ combina i risultati delle g_j per ottenere il valore finale della funzione $f(x_1, x_2, \ldots, x_n)$.

Questa formulazione mostra come una funzione f di n variabili possa essere decomposta in una somma di $2n+1$ funzioni univariate, ciascuna delle quali dipende da combinazioni lineari delle variabili originali.

Spiegazione Intuitiva

Il teorema di Kolmogorov-Arnold suggerisce che, nonostante la complessità di una funzione di più variabili, è possibile scomporla in componenti più semplici, ciascuna delle quali dipende solo da una combinazione lineare delle variabili originali. Questo processo di scomposizione permette di trattare funzioni complesse in modo più gestibile, facilitando la loro analisi e approssimazione.

Intuitivamente, la somma interna $\sum_{i=1}^{n} \phi_{ij}(x_i)$ può essere vista come un modo per combinare le informazioni provenienti dalle diverse variabili $x_1, x_2, \ldots, x_n$. Ogni funzione $\phi_{ij}(x_i)$ trasforma una singola variabile in un valore che ne cattura un aspetto specifico, e la somma di queste trasformazioni fornisce un unico valore che rappresenta una combinazione delle variabili originali.

La somma esterna $\sum_{j=1}^{2n+1} g_j$ raccoglie tutte queste combinazioni lineari, applicando a ciascuna una funzione g_j che ne determina l'influenza finale sulla funzione complessiva $f(x_1, x_2, \ldots, x_n)$. Questo approccio consente di scomporre il problema originale in sottoproblemi più semplici, dove ogni combinazione lineare delle variabili è gestita separatamente.

In pratica, anziché considerare tutte le variabili simultaneamente, il teorema consente di lavorare con combinazioni più semplici, migliorando l'efficienza computazionale e l'interpretabilità del problema.

Esempio: Funzione Quadratica

Consideriamo la seguente funzione quadratica di due variabili:

$$f(x_1, x_2) = x_1^2 + x_2^2 + x_1 x_2$$

Utilizzando il teorema di Kolmogorov-Arnold, possiamo scomporre questa funzione in termini di funzioni univariate. Il teorema ci suggerisce di rappresentare $f(x_1, x_2)$ come una somma di funzioni che dipendono da combinazioni lineari delle variabili x_1 e x_2.

Una possibile scomposizione è:

$$f(x_1, x_2) = \left(\frac{x_1 + x_2}{\sqrt{3}}\right)^2 + \left(\frac{x_1 - x_2}{\sqrt{6}}\right)^2$$

In questa rappresentazione, possiamo identificare le funzioni $\phi_{ij}(x_i)$ del teorema:

- $\phi_{11}(x_1) = \frac{x_1}{\sqrt{3}}$ e $\phi_{12}(x_2) = \frac{x_2}{\sqrt{3}}$ contribuiscono al primo termine $\frac{x_1+x_2}{\sqrt{3}}$.
- $\phi_{21}(x_1) = \frac{x_1}{\sqrt{6}}$ e $\phi_{22}(x_2) = -\frac{x_2}{\sqrt{6}}$ contribuiscono al secondo termine $\frac{x_1-x_2}{\sqrt{6}}$.

Quindi, la scomposizione può essere riscritta in termini di funzioni g_j e ϕ_{ij} come:

$$f(x_1, x_2) = g_1\left(\phi_{11}(x_1) + \phi_{12}(x_2)\right) + g_2\left(\phi_{21}(x_1) + \phi_{22}(x_2)\right)$$

Dove:

- $g_1(\xi) = \xi^2$ è la funzione che agisce sul risultato della somma interna $\phi_{11}(x_1) + \phi_{12}(x_2)$.
- $g_2(\xi) = \xi^2$ è la funzione che agisce sul risultato della somma interna $\phi_{21}(x_1) + \phi_{22}(x_2)$.

La prima combinazione lineare rappresenta la somma delle variabili x_1 e x_2, normalizzata da un fattore $\frac{1}{\sqrt{3}}$. La seconda combinazione lineare rappresenta la differenza delle variabili x_1 e x_2, normalizzata da $\frac{1}{\sqrt{6}}$.

Questo esempio illustra come una funzione quadratica, che include un termine di interazione tra le variabili, possa essere scomposta in termini di componenti univariate, ciascuna dipendente da una combinazione lineare delle variabili originali. Le funzioni ϕ_{ij} rappresentano le trasformazioni di base delle variabili, mentre le funzioni g_j operano su queste combinazioni per produrre il risultato finale.

1.3 Rilevanza ed Applicazioni

Il teorema di Kolmogorov-Arnold ha avuto un impatto significativo nella matematica pura e nelle sue applicazioni, specialmente nel contesto dell'analisi numerica[4] e della fisica teorica. La sua capacità di semplificare la rappresentazione di funzioni complesse ha aperto nuove possibilità di studio e applicazione in vari campi scientifici.

Applicazioni in Matematica e Analisi Numerica

Nella matematica, il teorema di Kolmogorov-Arnold ha fornito una base teorica per lo sviluppo di metodi avanzati di approssimazione. La scomposizione di funzioni multivariate in componenti univariate ha facilitato l'approccio a problemi che coinvolgono spazi ad alta dimensionalità, notoriamente complessi da gestire a causa della cosiddetta *curse of dimensionality*[5].

[4] L'analisi numerica è una branca della matematica che si occupa dello sviluppo e dell'analisi di algoritmi per la risoluzione approssimata di problemi matematici complessi, come l'integrazione, la risoluzione di equazioni differenziali e l'ottimizzazione.

[5] La *curse of dimensionality* si riferisce al fenomeno per cui l'aumento del numero di dimensioni in un problema rende esponenzialmente più difficile la sua risoluzione. Questo effetto si manifesta in molteplici contesti, tra cui l'analisi dei dati, l'ottimizzazione e la simulazione numerica.

Un'applicazione significativa del teorema si trova nell'interpolazione numerica e nella risoluzione di equazioni differenziali. Quando si affrontano equazioni in spazi altodimensionali, la possibilità di decomporre la funzione target in componenti univariate riduce drasticamente la complessità del problema. Questo approccio ha permesso di migliorare l'efficienza degli algoritmi numerici, specialmente in contesti dove è necessario approssimare soluzioni in modo rapido e accurato.

Ad esempio, nel caso dell'integrazione numerica, la scomposizione di una funzione multivariata in termini univariati permette di applicare metodi di quadratura[6] a problemi che altrimenti sarebbero difficilmente trattabili. Questo ha portato a sviluppi significativi nei metodi di calcolo in vari ambiti della matematica applicata e delle scienze computazionali.

Applicazioni in Fisica

In fisica, il teorema di Kolmogorov-Arnold ha trovato applicazione in numerosi contesti, specialmente nella modellizzazione di sistemi dinamici complessi. Uno degli ambiti in cui questo teorema ha avuto un impatto rilevante è la risoluzione di equazioni alle derivate parziali (PDEs, dall'inglese "partial differential equations"), che descrivono una vasta gamma di fenomeni fisici, tra cui la propagazione delle onde, la dinamica dei fluidi e i processi termodinamici.

Un esempio concreto è l'applicazione del KART nella meccanica quantistica ([6]), dove le funzioni d'onda, che descrivono lo stato quantico di un sistema, possono essere estremamente complesse e dipendere da molte variabili. La capacità di scomporre queste funzioni complesse in componenti più semplici ha facilitato la comprensione e la simulazione di sistemi quantistici, contribuendo allo sviluppo di modelli più accessibili e gestibili. Un altro ambito di applicazione è la dinamica dei fluidi, dove le equazioni di Navier-Stokes ([7]), che governano il movimento dei fluidi, sono spesso risolte numericamente. La rappresentazione di queste equazioni in termini di funzioni univariate ha migliorato l'efficienza delle simulazioni, permettendo di ottenere risultati più accurati con risorse computazionali limitate.

Il KART nella Fisica dei Sistemi Complessi

Il teorema di Kolmogorov-Arnold ha avuto anche un impatto significativo nella fisica dei sistemi complessi[7]. Le evoluzioni di questi sistemi sono difficilmente prevedibili dalle

[6] I metodi di quadratura univariata sono tecniche di integrazione numerica utilizzate per calcolare l'integrale di una funzione di una variabile su un intervallo. Questi metodi includono, ad esempio, la regola del trapezio e la regola di Simpson, e sono fondamentali per risolvere problemi che richiedono l'integrazione su intervalli continui.

[7] Branca della fisica che studia i comportamenti emergenti da interazioni tra molteplici componenti, spesso non lineari, all'interno di un sistema.

proprietà delle loro singole componenti, richiedendo strumenti matematici avanzati per essere modellizzati.

Nell'ambito dei sistemi non lineari, dove piccole perturbazioni possono causare grandi cambiamenti nel comportamento globale del sistema, il teorema è stato utilizzato per sviluppare modelli semplificati che catturano l'essenza delle dinamiche complesse. Un esempio rilevante è lo studio della turbolenza nei fluidi, un fenomeno altamente complesso che è stato descritto attraverso modelli basati sulla scomposizione in componenti univariate.

Un altro esempio è la dinamica delle reti neuronali, campo interdisciplinare che combina elementi di fisica, biologia e informatica per studiare come i comportamenti collettivi emergano dall'interazione di singoli neuroni. Qui, il teorema di Kolmogorov-Arnold ha permesso di sviluppare modelli che semplificano la rappresentazione delle interazioni tra neuroni, facilitando lo studio dei processi di apprendimento e memoria nelle reti complesse [8].

1.4 Il KART e le Reti Neurali

Il KART gioca, come già preannunciato, un ruolo centrale nella struttura delle reti neurali di Kolmogorov-Arnold, offrendo una soluzione teorica per la rappresentazione esatta delle funzioni continue di più variabili. Questa caratteristica distingue le KANs dalle reti neurali multistrato, che invece si basano sul teorema di rappresentazione universale per approssimare funzioni complesse.

KART vs UAT

Le KANs utilizzano la scomposizione enunciata nel KART per semplificare l'apprendimento delle funzioni, migliorando la loro efficienza e interpretabilità rispetto alle reti neurali multistrato. D'altra parte, le MLPs si basano sul teorema di rappresentazione universale (UAT, dall'inglese "universal approximation theorem"), il quale garantisce che una rete neurale, con almeno uno strato nascosto ed una funzione di attivazione non lineare, può approssimare qualsiasi funzione continua su un intervallo chiuso. In altri termini, il teorema afferma che, per qualsiasi funzione continua $f(x)$ definita su uno spazio compatto $\mathcal{X} \subset \mathbb{R}^n$, e per ogni $\epsilon > 0$, esiste una rete neurale come sopra, con una data configurazione di pesi e bias, tale che la funzione approssimata dalla rete differisca dalla funzione target per meno di ϵ; formalmente, si scrive:

$$f(x) \approx \sum_{i=1}^{N(\epsilon)} a_i \sigma(w_i \cdot x + b_i)$$

dove a_i, w_i, e b_i sono i parametri della rete, e $N(\epsilon)$ rappresenta il numero di nodi necessari per raggiungere la precisione ϵ. Importante da notare è come in questo caso si tratti solo di un'approssimazione, e non di un'uguaglianza (cfr. (1.2)).

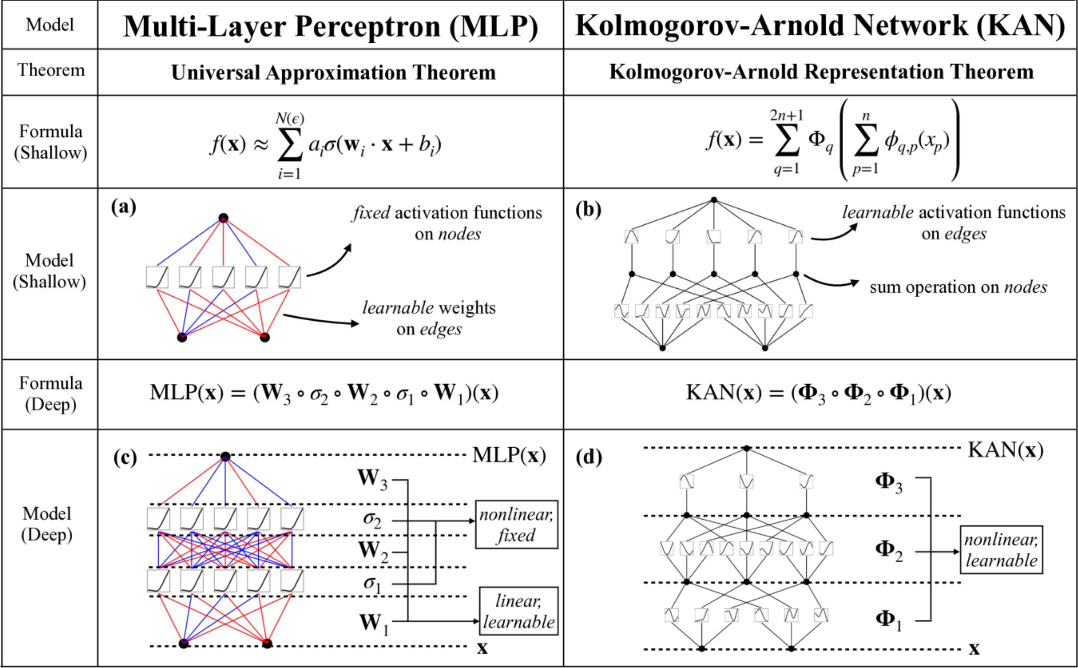


Figura 1.1: *Confronto schematico tra le due tipologie di rete. Immagine presa da [2].*

Accenno Storico ed Applicazioni

Il teorema di rappresentazione universale fu dimostrato per la prima volta negli anni '80, in modo indipendente da diversi ricercatori, tra cui George Cybenko (1989) e Kurt Hornik (1991). Questo risultato fu rivoluzionario per lo sviluppo delle reti neurali, poiché stabilì una base teorica per l'uso delle MLPs nell'approssimazione di funzioni complesse, aprendo la strada al loro utilizzo in molte applicazioni pratiche. Oltre alle MLPs, il teorema ha trovato applicazione anche in altri modelli di reti neurali, inclusi i modelli convoluzionali e ricorrenti[8], dove la capacità di approssimare funzioni non lineari è fondamentale per il trattamento di dati complessi, come immagini, serie temporali e segnali. A differenza del KART, che fornisce una scomposizione esatta delle funzioni multivariate, il teorema di rappresentazione universale delle MLPs offre, come già accennato, solo un'approssimazione della funzione target. Questo approccio, sebbene potente, non specifica come la rete debba essere configurata per ottenere l'approssimazione né garantisce l'efficienza computazionale del processo [9].

[8]Le reti neurali convoluzionali (Convolutional Neural Networks, CNNs) sono un tipo di rete neurale particolarmente efficace per l'elaborazione di dati strutturati in griglie, come le immagini. Utilizzano operazioni di convoluzione per estrarre caratteristiche locali dalle immagini e ridurre la dimensionalità dei dati, mantenendo al contempo informazioni rilevanti per la classificazione o il riconoscimento. Le reti neurali ricorrenti (Recurrent Neural Networks, RNNs), invece, sono un tipo di rete neurale progettata per elaborare dati sequenziali, come serie temporali o sequenze di testo. Le RNNs utilizzano connessioni cicliche tra i nodi per mantenere una memoria interna, che consente di catturare dipendenze temporali e contestuali nei dati sequenziali.

Ritardo nell'Applicazione del KART alle Reti Neurali

Nonostante il KART sia stato formulato negli anni '60, la sua applicazione alle reti neurali ha subito un ritardo significativo a causa di una serie di sfide tecniche e concettuali. Una delle principali problematiche era la complessità computazionale associata alla scomposizione delle funzioni multivariate in componenti univariate, come previsto dal teorema. In pratica, implementare tale scomposizione richiedeva risorse computazionali che erano fuori portata per le tecnologie dell'epoca. Un altro ostacolo era rappresentato dalla difficoltà nell'ottimizzare le funzioni univariate risultanti, che non si prestavano facilmente alle tecniche di ottimizzazione standard utilizzate nelle reti neurali tradizionali.

La mancanza di algoritmi efficienti per l'apprendimento e la parametrizzazione di queste funzioni univariate rendeva difficile sfruttare appieno il potenziale del KART. Inoltre, l'assenza di una struttura di rete standard per implementare il KART rendeva complesso integrare questo teorema nelle architetture neurali esistenti. Le prime reti neurali, come le MLPs, erano fortemente influenzate da modelli biologici e da approcci che privilegiavano l'interconnessione tra neuroni su più livelli, piuttosto che la scomposizione diretta delle funzioni in componenti più semplici. Solo con il progredire delle tecnologie computazionali e l'emergere delle limitazioni delle MLPs, i ricercatori hanno iniziato a esplorare l'applicazione del KART nelle reti neurali, culminando appunto nella recente implementazione delle KANs. Questa nuova architettura sfrutta appieno le potenzialità del teorema, offrendo una rappresentazione esatta delle funzioni complesse e promettendo di superare, come approfondiremo in seguito, alcune delle principali limitazioni delle MLPs.

Implicazioni Filosofiche e Concettuali del Teorema di Kolmogorov-Arnold

Il teorema di Kolmogorov-Arnold rappresenta non solo un risultato matematico di notevole importanza, ma anche una prospettiva concettuale significativa nel contesto dell'intelligenza artificiale e della teoria delle funzioni. La sua capacità di fornire una rappresentazione esatta delle funzioni continue multivariate mediante una somma finita di funzioni offre un approccio che enfatizza l'ordine e la riduzione della complessità. In questo senso, il KART si inserisce in una tradizione che cerca di decodificare la complessità attraverso principi fondamentali e rigorosi. Questa prospettiva suggerisce che esista un insieme di leggi precise che, se comprese e applicate correttamente, permettono di decifrare la complessità apparente del mondo matematico e computazionale. Il KART, quindi, può essere visto come una potente espressione di questa visione, dove la rappresentazione esatta e deterministica delle funzioni diventa un obiettivo potenzialmente raggiungibile tramite la comprensione delle leggi sottostanti.

D'altra parte, il teorema di rappresentazione universale delle MLPs (UAT) adotta un approccio più pragmatico: l'UAT afferma, infatti, che le MLPs possono approssimare qualsiasi funzione continua, sebbene non in modo esatto, ma con un grado di appros-

simazione che può essere reso arbitrariamente piccolo. Questo riflette una filosofia che riconosce i limiti della conoscenza e della rappresentazione esatta, enfatizzando l'importanza dell'approssimazione come strumento pratico nell'intelligenza artificiale. Le MLPs rappresentano un metodo che integra l'approssimazione e l'incertezza nel processo di apprendimento e rappresentazione. Questa filosofia è profondamente radicata nell'intelligenza artificiale contemporanea, dove l'obiettivo è spesso quello di trovare soluzioni che, pur non essendo esatte, risultano sufficientemente accurate e praticabili per affrontare problemi complessi. In questo contesto, l'UAT riflette un approccio che accetta l'approssimazione come parte integrante del processo di modellazione e risoluzione dei problemi.

La tensione tra il determinismo del KART e l'approssimazione dell'UAT è manifesta di una dicotomia significativa nell'evoluzione dell'intelligenza artificiale. Da un lato, le KANs, basate sul KART, potrebbero aprire nuove strade verso lo sviluppo di reti neurali che non solo siano più interpretabili, ma che possano offrire una precisione e un rigore matematico maggiori. Dall'altro lato, le MLPs e l'approccio approssimativo che esse incarnano continueranno a essere dominanti in quei contesti in cui la flessibilità, la scalabilità e l'efficienza sono essenziali.

In conclusione, questa dicotomia tra KART e UAT non rappresenta solo una differenza tecnica tra due approcci matematici, ma riflette anche una divisione concettuale che potrebbe influenzare le direzioni future della ricerca nell'AI.

Capitolo 2

Architettura e funzionamento delle KANs

2.1 Introduzione

Le reti neurali di Kolmogorov-Arnold (KANs) rappresentano un'evoluzione significativa rispetto alle architetture tradizionali, in particolare riguardo alla nuova posizione e ruolo delle funzioni di attivazione. A differenza delle MLPs, dove le funzioni di attivazione sono generalmente collocate in ogni nodo della rete per introdurre non-linearità, nelle KANs queste funzioni assumono una posizione centrale e strategica, in linea con la decomposizione funzionale prevista dal teorema di Kolmogorov-Arnold.

L'architettura delle KANs si basa su una rete di nodi e archi che, pur mantenendo la struttura gerarchica tipica delle reti neurali tradizionali, introduce innovazioni nella disposizione e nell'interazione tra i vari componenti. La struttura complessiva della rete può essere suddivisa, analogamente a quanto fatto tradizionalmente, in tre tipologie di strati principali, ciascuno con un ruolo ben definito:

1. **Strati di Input:** Questi strati sono deputati a ricevere le variabili in ingresso, che costituiscono i veri e propri dati del problema. Prima di essere elaborate dagli strati successivi, le variabili di input possono essere trasformate o normalizzate, preparandole per un trattamento più approfondito.

2. **Strati Intermedi:** È qui che si manifesta la principale innovazione delle KANs. Ogni nodo in questi strati è responsabile del calcolo di una funzione univariata, secondo la scomposizione funzionale dettata dal KART. Gli archi che collegano i nodi riflettono la somma additiva delle funzioni, integrando i risultati in modo da formare una rappresentazione più complessa. La posizione delle funzioni di attivazione in questi nodi non è più solo uno strumento per introdurre non-linearità, ma diventa

fondamentale per la corretta decomposizione e ricostruzione delle informazioni in ingresso.

3. **Strati di Output:** Lo strato di output raccoglie e combina i risultati ottenuti dagli strati intermedi. Gli archi in questo strato sono responsabili della somma pesata delle funzioni univariate calcolate, con l'obiettivo di produrre l'output finale della rete. Questa sintesi finale rappresenta l'essenza della capacità delle KANs di rappresentare fedelmente funzioni complesse attraverso la somma di componenti univariati.

L'aspetto più rivoluzionario di questa architettura risiede, quindi, nella collocazione e nel ruolo delle funzioni di attivazione. Queste non sono più semplici elementi di supporto, ma diventano il fulcro attorno al quale ruota l'intera struttura della rete. Questa configurazione permette alle KANs di affrontare con maggiore efficacia problemi complessi, grazie alla loro capacità di scomporre le informazioni in parti gestibili e di ricombinarle in modo coerente.

Approfondiremo in maggiore dettaglio queste componenti nelle prossime sezioni del capitolo, analizzando per esempio come la scelta delle funzioni di attivazione influisca sull'efficienza e sulla precisione delle KANs.[1]

2.2 Architettura di Base delle KANs

Nelle Kolmogorov-Arnold Networks, ogni strato k della rete esegue una trasformazione sull'input proveniente dal livello precedente, che rappresentiamo con il vettore $\mathbf{a}_{k-1}$. Questa trasformazione è descritta da una matrice di funzioni univariate ϕ^k, detta "matrice di trasformazione", che agisce sull'input per produrre un nuovo output $\mathbf{a}_k$.

Trasformazione in un Generico Strato: Notazione Matriciale

Consideriamo dunque uno strato generico k con input $\mathbf{a}_{k-1}$ di dimensione n, e output $\mathbf{a}_k$ di dimensione m. La trasformazione operata da questo strato può essere rappresentata in notazione matriciale come:

$$\mathbf{z}_k = \phi^k \mathbf{a}_{k-1} = \begin{bmatrix} \varphi_{11} & \varphi_{12} & \cdots & \varphi_{1n} \\ \varphi_{21} & \varphi_{22} & \cdots & \varphi_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_{m1} & \varphi_{m2} & \cdots & \varphi_{mn} \end{bmatrix} \begin{bmatrix} a_{k-1,1} \\ a_{k-1,2} \\ \vdots \\ a_{k-1,n} \end{bmatrix}$$

In questa espressione, n rappresenta il numero di input (o nodi nello strato precedente) e m il numero di output (o nodi nello strato corrente); si ottiene pertanto una matrice di trasformazione ϕ^k con m righe e n colonne (formalmente una matrice $m \times n$). Ogni

[1] Quanto esposto in questo capitolo si basa principalmente sull'architettura e funzionamento delle reti neurali di Kolmogorov-Arnold come presentate nell'articolo originario di Ziming Liu [2].

elemento φ_{ij} della matrice ϕ^k è una funzione univariata generica attraverso cui si parametrizzano le funzioni di attivazione della rete neurale; ognuna di esse verrà poi applicata all'input specifico $a_{k-1,j}$ nel prodotto riga-colonna.

In Notazione Estesa: Prodotto Righe-Colonne

L'espressione sopra può essere espansa, mostrando il risultato del prodotto tra le righe della matrice delle funzioni e il vettore degli input dello strato k-esimo in esame :

$$\mathbf{z}_k = \begin{bmatrix} \varphi_{11} & \varphi_{12} & \cdots & \varphi_{1n} \\ \varphi_{21} & \varphi_{22} & \cdots & \varphi_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_{m1} & \varphi_{m2} & \cdots & \varphi_{mn} \end{bmatrix} \begin{bmatrix} a_{k-1,1} \\ a_{k-1,2} \\ \vdots \\ a_{k-1,n} \end{bmatrix} = \begin{bmatrix} \varphi_{11}(a_{k-1,1}) + \varphi_{12}(a_{k-1,2}) + \cdots + \varphi_{1n}(a_{k-1,n}) \\ \varphi_{21}(a_{k-1,1}) + \varphi_{22}(a_{k-1,2}) + \cdots + \varphi_{2n}(a_{k-1,n}) \\ \vdots \\ \varphi_{m1}(a_{k-1,1}) + \varphi_{m2}(a_{k-1,2}) + \cdots + \varphi_{mn}(a_{k-1,n}) \end{bmatrix}$$

Questa espansione mostra chiaramente come ogni elemento $z_{k,i}$ del vettore $\mathbf{z}_k$ sia il risultato della somma delle trasformazioni applicate agli elementi dell'input. In alternativa, possiamo esprimere ogni singolo elemento $z_{k,i}$ usando la notazione con sommatoria:

$$z_{k,i} = \sum_{j=1}^{n} \varphi_{ij}(a_{k-1,j})$$

Si ottiene pertanto questa forma condensata della trasformazione, che riassume in un'unica espressione il processo di somma ponderata che genera ogni elemento dell'output.

Composizione degli Strati e Output Finale

L'output finale $\mathbf{y}$ della rete è ottenuto a partire dall'input iniziale x componendo le trasformazioni di tutti gli strati, fino all'ultimo degli L livelli. Questa composizione è riassumibile come:

$$\mathbf{y} = KAN(\mathbf{x}) = \phi^L\left(\phi^{L-1}\left(\cdots \phi^2\left(\phi^1(\mathbf{x})\right)\cdots\right)\right)$$

Dove ϕ^k è la matrice di trasformazione del livello k e $\mathbf{y}$ rappresenta appunto l'output finale della rete.

Esempio Concreto: KAN con un Singolo Strato Intermedio

Consideriamo una KAN con un input layer di due nodi, x_1 e x_2, un singolo strato intermedio con cinque nodi, e un output layer con un singolo nodo. L'obiettivo è determinare $y = f(x_1, x_2)$.

Passaggio 1: Trasformazione dell'Input

Gli input dati da x_1 e x_2 passano ognuno attraverso cinque funzioni univariate: rispettivamente $\phi_{11}, \phi_{12}, \ldots, \phi_{15}$ e $\phi_{21}, \phi_{22}, \ldots, \phi_{25}$. Questi producono i seguenti output per ogni nodo nello strato intermedio[2].:

$$\mathbf{z}_1 = [\phi_{11}(x_1), \phi_{12}(x_1), \ldots, \phi_{15}(x_1)]$$
$$\mathbf{z}_2 = [\phi_{21}(x_2), \phi_{22}(x_2), \ldots, \phi_{25}(x_2)]$$

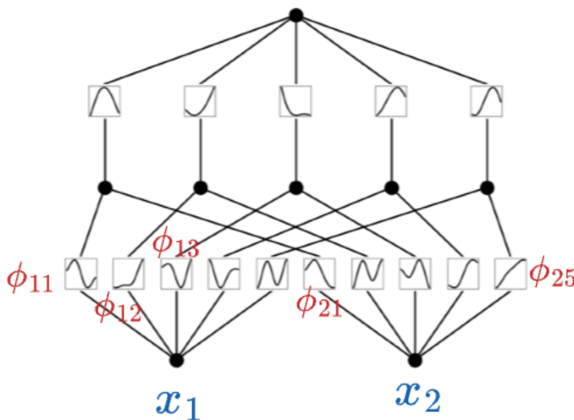


Figura 2.1: *Illustrazione della trasformazione dell'input*

Tali risultati vengono poi sommati nodo per nodo per ottenere l'output dello strato intermedio:

$$\mathbf{z} = \mathbf{z}_1 + \mathbf{z}_2 = [\phi_{11}(x_1) + \phi_{21}(x_2), \phi_{12}(x_1) + \phi_{22}(x_2), \ldots, \phi_{15}(x_1) + \phi_{25}(x_2)]$$

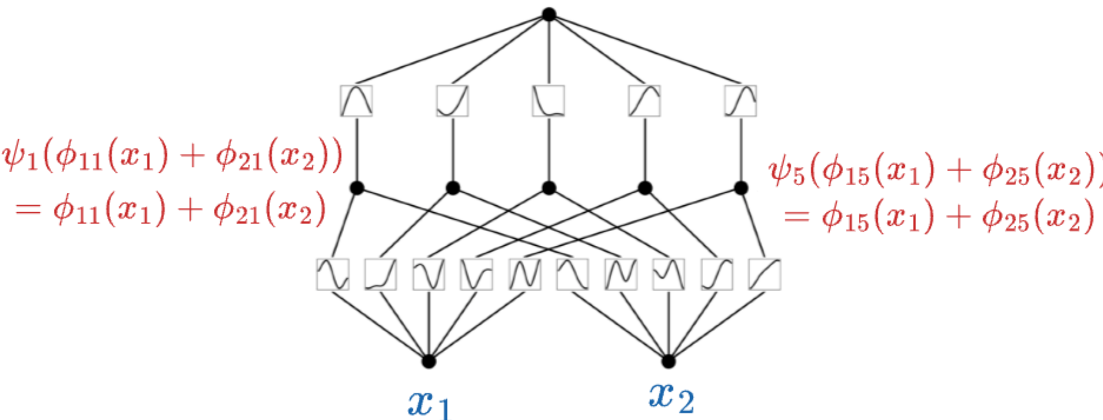


Figura 2.2: *Illustrazione della somma dei risultati per ciascun nodo dello strato intermedio.*

[2] Le tre immagini riportate in questa sotto-sezione sono prese da [10].

Passaggio 2: Aggregazione e Output Finale

L'output dello strato intermedio viene poi trasformato da una seconda serie di funzioni $\hat{\phi}_1, \hat{\phi}_2, \ldots, \hat{\phi}_5$ per produrre l'output finale y:

$$y = \sum_{i=1}^{5} \hat{\phi}_i \left(\phi_{i1}(x_1) + \phi_{i2}(x_2)\right)$$

In questo esempio, abbiamo usato una KAN con un singolo strato intermedio per determinare una funzione y di due variabili x_1 e x_2. Le funzioni di attivazione univariate ϕ vengono applicate individualmente ai singoli input, i risultati vengono aggregati e successivamente trasformati nuovamente per ottenere l'output finale.

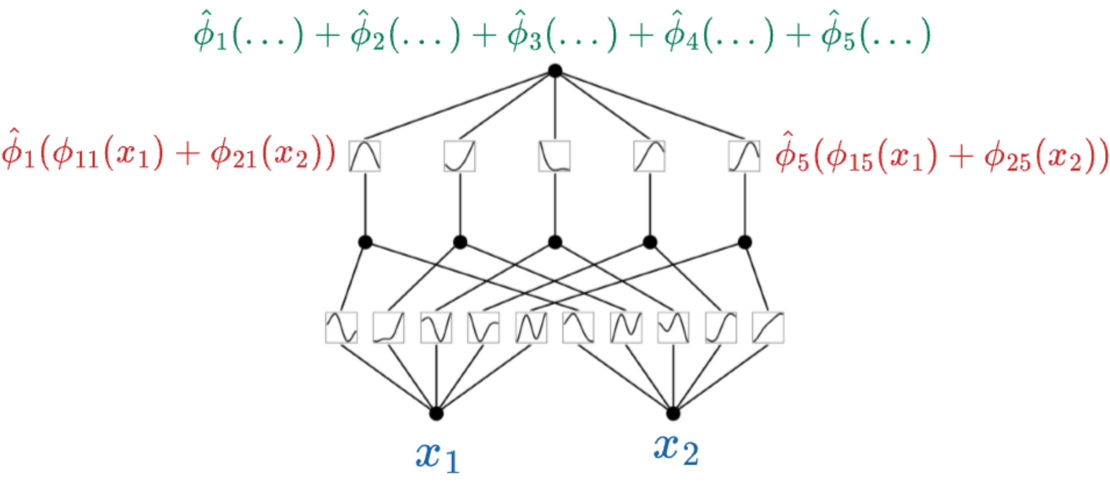


Figura 2.3: *Illustrazione dell'aggregazione e output finale.*

2.3 Funzioni di Attivazione Basate sui B-splines

Gli splines ([11]) sono funzioni composte da segmenti di polinomi a tratti definiti su intervalli consecutivi, con condizioni di continuità imposte alle giunzioni, chiamate nodi. I B-splines ("Basis splines") sono una base per lo spazio delle funzioni spline e ne rappresentano una classe specifica, offrenti vantaggi come la località, la stabilità numerica e la flessibilità.

Formulazione Matematica dei B-splines

Consideriamo un B-spline di grado d con $n + 1$ punti di controllo $P_0, P_1, \ldots, P_n$ e un vettore di nodi $T = \{t_0, t_1, \ldots, t_m\}$, dove $m = n + d + 1$:

- **Punti di controllo**: I punti di controllo P_i determinano la forma della curva B-spline, influenzando il tratto polinomiale associato. Ogni tratto polinomiale è

influenzato da $d+1$ punti di controllo, il che significa che il numero di punti di controllo è strettamente legato al grado dello spline.

- **Vettore dei nodi**: Il vettore dei nodi T è una sequenza ordinata di valori che definisce gli intervalli su cui ciascun segmento polinomiale è valido. I nodi sono i punti dell'intervallo complessivo in cui si congiungono i polinomi a tratti che costituiscono lo spline. In corrispondenza di ciascun nodo, le funzioni B-spline garantiscono continuità fino alla derivata di ordine $d-1$.

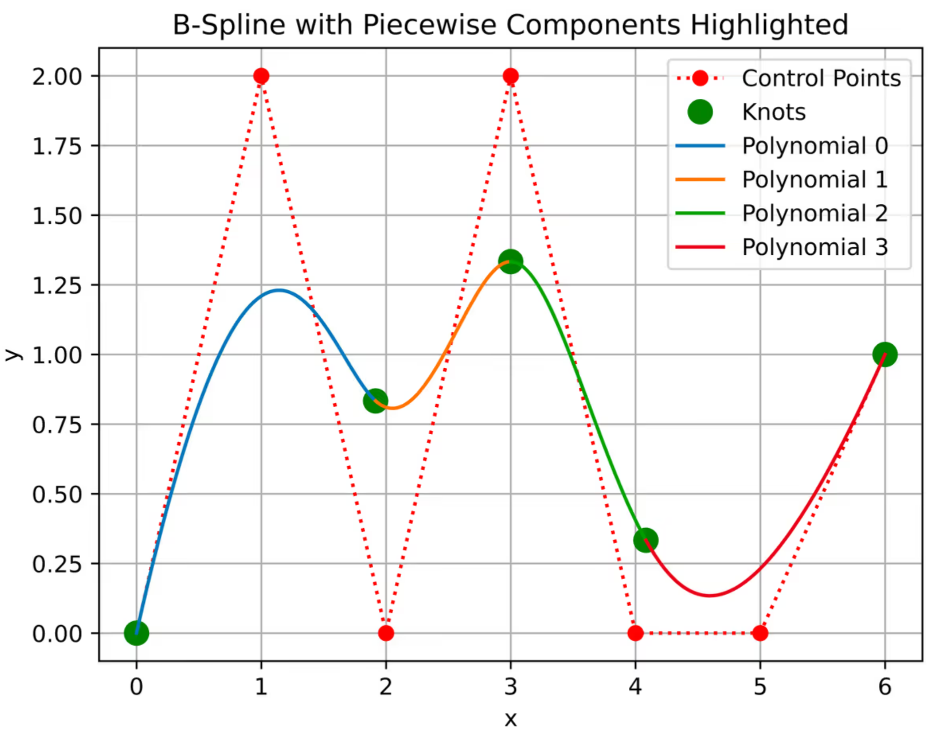


Figura 2.4: *Illustrazione presa da [12]: "Esempio di B-spline in cui sono evidenziati i punti di controllo ("Control Points"), i nodi ("Knots") e i polinomi a tratti ("Polinomial i")."*

La curva B-spline $C(t)$, che è una curva parametrizzata con t come parametro, può essere espressa come:

$$C(t) = \sum_{i=0}^{n} N_{i,d}(t) P_i$$

Qui, $N_{i,d}(t)$ sono le funzioni di base B-spline ("B-splines basis functions") di grado d, definite come:

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

Il dominio del parametro t è l'intervallo definito dal vettore dei nodi T. Per $d > 0$, le funzioni di base B-spline vengono calcolate iterativamente usando la seguente formula di de Boor-Cox ([13]):

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

Questa ricorsione consente di costruire B-splines di ordine superiore a partire da funzioni di base di ordine inferiore, garantendo la continuità delle funzioni spline e delle loro derivate fino all'ordine $d - 1$ [14].

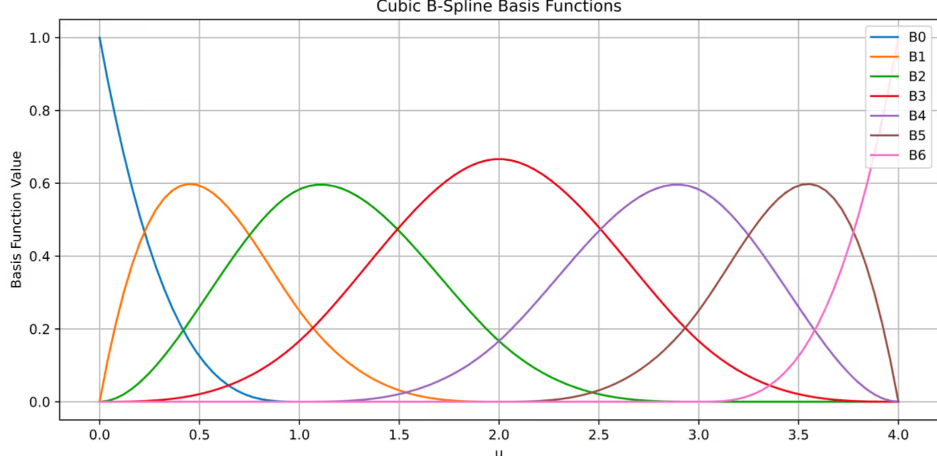


Figura 2.5: *Illustrazione presa da [12]: "Funzioni base di un B-Spline cubico (ordine 3) date da un vettore di nodi uniforme. Si noti che la prima e l'ultima funzione assumono valore 1 agli estremi, mentre tutte le altre hanno valore 0 in quei punti, implicando che i punti iniziale e finale esercitano un'influenza completa sul B-Spline finale."*

Uso dei B-splines nelle KANs

Nelle Kolmogorov-Arnold Networks, i B-splines vengono utilizzati come funzioni di attivazione: ogni arco nella rete è associato ad un B-spline che trasforma l'input in modo non lineare, permettendo alla rete di catturare dettagli locali della funzione da approssimare. Questo approccio, utilizzato in [2], sfrutta appieno le proprietà dei B-splines, come la località e la capacità di modellare funzioni complesse con una rappresentazione compatta.

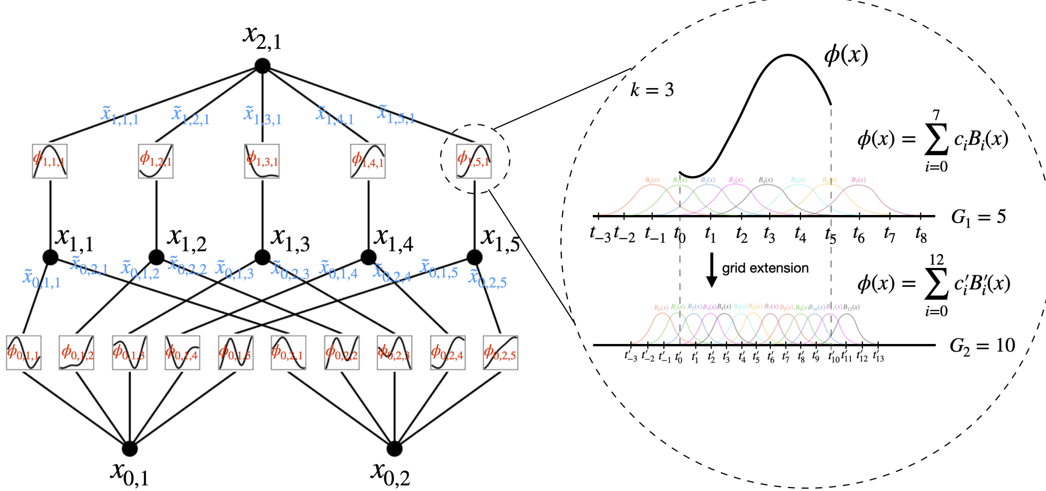


Figura 2.6: *Rappresentazione schematica di una KAN con le stesse dimensioni viste in Esempio Concreto: KAN con un Singolo Strato Intermedio, con un dettaglio su una funzione di attivazione parametrizzata tramite B-Splines. Immagine presa da [2].*

2.4 La Grid ed il suo Ruolo nella Parametrizzazione delle Funzioni di Attivazione

Nelle KANs, la parametrizzazione delle funzioni di attivazione tramite B-splines è strettamente legata alla definizione della "grid". La grid è un insieme ordinato di nodi $T = \{t_0, t_1, \ldots, t_m\}$ che divide l'intervallo di definizione delle funzioni in sotto-intervalli, ognuno dei quali controlla un segmento della funzione B-spline.

Matematicamente, la grid determina l'intervallo su cui ogni B-spline è definito, controllando sia la posizione che l'estensione dei singoli segmenti polinomiali della funzione di attivazione. Il numero di nodi m e la loro distribuzione influenzano direttamente la complessità e la precisione della funzione B-spline risultante. Se indichiamo la grid come intervallo $[a, b]$ suddiviso in m sotto-intervalli, possiamo definire la lunghezza di ciascun sotto-intervallo come $\Delta t_i = t_{i+1} - t_i$, con $i = 0, 1, \ldots, m-1$. La scelta di Δt_i influenza la capacità della KANs di adattarsi alle caratteristiche locali della funzione target: una grid più densa (Δt_i piccolo) permette di catturare dettagli più fini, mentre una grid più larga (Δt_i grande) è più adatta per catturare variazioni globali.

Per comprendere intuitivamente il ruolo della grid, possiamo immaginarla come un sistema di coordinate su cui viene disegnata la funzione di attivazione. Ogni nodo nella grid rappresenta un punto in cui la funzione può cambiare direzione o curvatura. Pertanto, una grid ben calibrata consente alla funzione di attivazione di adattarsi meglio alla complessità della funzione target che la KAN deve approssimare. Una metafora utile potrebbe essere quella di una tela su cui si disegna un grafico: la grid rappresenta i punti

di riferimento lungo gli assi, e la qualità del disegno (cioè la precisione della funzione B-spline) dipende da quanto finemente è suddivisa la tela su cui si lavora.

Nel contesto delle KANs, la grid non è solo un elemento matematico, ma un parametro di progettazione fondamentale. La sua configurazione deve essere accuratamente scelta per bilanciare la capacità di apprendimento della rete e la complessità computazionale richiesta per addestrarla. La disposizione dei nodi influenza la flessibilità delle funzioni di attivazione e, di conseguenza, l'efficacia della rete nell'apprendimento di funzioni complesse. L'importanza della grid è particolarmente evidenziata nel lavoro di Ziming Liu [2], dove viene utilizzata per ottimizzare la capacità delle KANs di approssimare funzioni target complesse.

Manipolazione della Grid

Nell'articolo appena citato, viene infatti discussa l'importanza della manipolazione della grid per migliorare l'efficienza delle KANs. Gli autori dimostrano che variando la densità dei nodi lungo l'intervallo di definizione, la rete può allocare risorse computazionali in modo più efficace, adattandosi meglio alla complessità della funzione target.

Densità della Grid Consideriamo un intervallo di definizione $[a, b]$ suddiviso in m sottointervalli con nodi $T = \{t_0, t_1, \ldots, t_m\}$. Quando la funzione target varia rapidamente in una regione $[t_i, t_{i+1}]$, è vantaggioso aumentare la densità dei nodi in quella regione, riducendo la lunghezza dell'intervallo Δt_i corrispondente, per esempio attraverso:

$$\Delta t_i = \frac{b-a}{m} \cdot z(t_i)$$

dove $z(t_i)$ è una funzione che misura la complessità locale della funzione target. Questo approccio permette di ottenere una maggiore precisione nelle aree critiche senza aumentare inutilmente la complessità computazionale associata.

Estensione della Grid ai Bordi per i Punti di Controllo Esterni Un altro aspetto fondamentale trattato nell'articolo riguarda l'estensione della grid oltre i bordi dell'intervallo principale. Per garantire la corretta definizione dei B-splines, è infatti necessario aggiungere nodi esterni $[t_{-d}, \ldots, t_{-1}]$ e $[t_{m+1}, \ldots, t_{m+d}]$, dove d è il grado scelto per lo spline , andando a riscrivere dunque:

$$C(t) = \sum_{i=-d}^{n+d} N_{i,d}(t) P_i$$

Questi nodi esterni assicurano la continuità e la regolarità della funzione di attivazione ai bordi, evitando discontinuità e oscillazioni indesiderate.

Uniformità del Vettore dei Nodi Il vettore dei nodi T può essere definito come uniforme, per cui la distanza tra due nodi consecutivi è costante, cioè $\Delta t_i = \Delta t_{i+1}$ per ogni i, oppure non uniforme, per cui la distanza non è sempre uguale. Un vettore di nodi

non uniforme consente di aumentare la densità dei nodi in regioni specifiche, migliorando l'accuratezza dell'approssimazione in quelle aree, senza aumentare il numero totale di nodi. In questo caso, possiamo avere $\Delta t_i \neq \Delta t_{i+1}$, consentendo una flessibilità maggiore nella rappresentazione delle funzioni di attivazione.

Manipolazione della Grid durante l'Apprendimento Durante il processo di apprendimento, la grid stessa può essere adattata in risposta ai dati. In particolare, la distribuzione dei nodi può essere ottimizzata in modo iterativo, concentrando la densità dei nodi nelle regioni in cui la rete rileva errori maggiori o variazioni significative. Questo approccio adattivo, se ben implementato, può ulteriormente migliorare l'efficienza della rete, riducendo la necessità di interventi manuali nella scelta della grid.

2.5 Altre Funzioni di Attivazione

All'interno delle Kolmogorov-Arnold Networks, oltre ai B-splines, si potrebbe considerare l'utilizzo di altre funzioni di attivazione basate su polinomi. Una possibile scelta è l'uso di polinomi generici di grado arbitrario, grazie alla loro semplicità di implementazione e alla facilità con cui possono essere manipolati matematicamente. I polinomi generici di grado n assumono la forma:

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

dove gli a_i rappresentano i coefficienti del polinomio e n è il grado del polinomio. Questi coefficienti a_i determinano l'importanza relativa di ciascun termine del polinomio, controllando così la forma della funzione complessiva $P_n(x)$. La semplicità di questa formulazione, unita alla facilità di calcolo dei termini polinomiali, potrebbe far pensare che l'uso di polinomi generici come funzioni di attivazione sia una scelta naturale e conveniente. Tuttavia, ci sono significativi limiti legati all'uso di polinomi generici in contesti di approssimazione, specialmente su intervalli ampi.

Un problema ben noto è il cosiddetto *"Runge's phenomenon"* ([15]), che si manifesta come una crescente oscillazione alle estremità dell'intervallo di approssimazione all'aumentare del grado del polinomio. Questo fenomeno si osserva, ad esempio, quando si utilizza un polinomio di grado elevato per interpolare una funzione in n punti equidistanti su un intervallo $[a, b]$. Il risultato è che l'errore di interpolazione può crescere significativamente vicino ai bordi, rendendo l'approssimazione poco accurata. Matematicamente, possiamo esprimere l'errore di approssimazione $R(x)$ come:

$$R(x) = f(x) - P_n(x)$$

dove $f(x)$ è la funzione target che si desidera approssimare. Nel contesto in esame, il Runge's phenomenon si caratterizza per l'aumento dell'errore $R(x)$ quando x si avvicina ai bordi a e b dell'intervallo; matematicamente:

$$\lim_{x \to a^+} |R(x)| \to \infty \quad \text{e} \quad \lim_{x \to b^-} |R(x)| \to \infty$$

Questa divergenza dell'errore ai bordi compromette inevitabilmente l'accuratezza globale dell'approssimazione cercata.

Oltre al Runge's phenomenon, i polinomi generici soffrono di un'altra limitazione significativa: la mancanza di località. In altre parole, ogni coefficiente a_i ha un'influenza globale sulla funzione totale nell'intervallo $[a, b]$, il che significa che una piccola variazione di ogni singolo coefficiente modifica la funzione $P_n(x)$ su tutto l'intervallo. Questa mancanza di controllo locale rende difficile adattare la funzione di attivazione alle caratteristiche locali dell'input, riducendo l'efficacia dell'approssimazione.

Al contrario, i B-splines risolvono efficacemente queste problematiche. Essi sono funzioni di base a tratti, definite localmente su sotto-intervalli determinati dalla grid dei nodi. Questa località implica che la modifica di un singolo coefficiente di un B-spline influisca solo su un sotto-intervallo specifico, senza alterare il resto della funzione. Inoltre, i B-splines non soffrono del fenomeno di Runge, grazie alla loro costruzione a tratti che evita oscillazioni indesiderate. A questo proposito, infine, la località dei B-splines è strettamente collegata al concetto di *catastrophic forgetting*, un problema comune nelle reti neurali tradizionali, ma che le KANs riescono a mitigare efficacemente. Questo aspetto sarà approfondito più avanti nel corso della presente trattazione.

Polinomi di Legendre e Chebyshev come Funzioni di Attivazione

Oltre ai B-splines, che come visto sono di gran lunga preferibili alla semplice approssimazione polinomiale, alcuni studiosi hanno già provato a implementare KANs che parametrizzano le funzioni di attivazione utilizzando altre tipologie di polinomi. In particolare, i polinomi di Legendre e i polinomi di Chebyshev hanno attirato l'attenzione per le loro proprietà matematiche uniche e le rispettive potenziali applicazioni [16, p. 185-192].

Polinomi di Legendre

I polinomi di Legendre ([17]), indicati nel seguito con $P_n(x)$, sono una famiglia di polinomi ortogonali rispetto alla distribuzione uniforme sull'intervallo $[-1, 1]$[3]. Essi sono definiti dalla relazione di ricorrenza:

$$P_0(x) = 1, \quad P_1(x) = x, \quad P_{n+1}(x) = \frac{2n+1}{n+1} x P_n(x) - \frac{n}{n+1} P_{n-1}(x)$$

La formula chiusa per i polinomi di Legendre è invece data da:

$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} \left[(x^2 - 1)^n \right]$$

Questi polinomi sono utilizzati principalmente nella fisica quantistica, specialmente nella soluzione dell'equazione di Schrödinger indipendente dal tempo in coordinate sferiche.

[3] Ciò significa che l'integrale del prodotto di due polinomi **distinti** in questa famiglia, pesato dalla distribuzione uniforme, è pari a zero su quell'intervallo. In altre parole, $\int_{-1}^{1} P_m(x) P_n(x) dx = 0$ per $m \neq n$.

Quando si separano le variabili dell'equazione di Schrödinger per un potenziale centrale, come per esempio si fa nel caso degli atomi idrogenoidi, la parte angolare della funzione d'onda può essere espressa in termini dei polinomi di Legendre, che riflettono la simmetria sferica del sistema:

$$Y_\ell^m(\theta, \phi) = P_\ell^m(\cos\theta)e^{im\phi}$$

dove $P_\ell^m(x)$ sono per l'appunto i polinomi associati di Legendre, che rappresentano la parte angolare dell'equazione di Schrödinger in coordinate sferiche.

Tuttavia, anche i polinomi di Legendre, nonostante la loro ortogonalità e le proprietà che li rendono adatti per problemi con simmetria sferica, possono essere soggetti al *Runge's phenomenon* se utilizzati su intervalli ampi. Questo perché, come nel caso dei polinomi generici, l'errore di interpolazione può crescere significativamente vicino ai bordi dell'intervallo. Inoltre, i polinomi di Legendre, essendo definiti globalmente, soffrono anch'essi di una mancanza di località, influenzando tutta la funzione su $[-1, 1]$ con la variazione di ogni singolo loro coefficiente.

Polinomi di Chebyshev

I polinomi di Chebyshev ([18]), che indichiamo con $T_n(x)$, sono un'altra famiglia di polinomi ortogonali, definiti dalla relazione di ricorrenza:

$$T_0(x) = 1, \quad T_1(x) = x, \quad T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

La corrispondente formula chiusa è:

$$T_n(x) = \cos(n\cos^{-1}(x))$$

Questi polinomi sono particolarmente noti per la loro proprietà di minimizzare l'errore di approssimazione rispetto alla funzione coseno, il che li rende ideali per l'approssimazione numerica su intervalli ampi[4]. In matematica e fisica, i polinomi di Chebyshev trovano applicazione nella discretizzazione spettrale di equazioni differenziali, dove si cerca di approssimare una funzione complessa mediante una somma pesata di tali polinomi. Ad esempio, nell'ambito dell'elettrostatica, nella risoluzione dell'equazione di Poisson:

$$\nabla^2\phi = \rho$$

dove ϕ rappresenta il potenziale elettrico e ρ la densità di carica, i polinomi di Chebyshev vengono utilizzati per rappresentare il potenziale $\phi(x)$ come una serie di polinomi ortogonali:

$$\phi(x) \approx \sum_{n=0}^{N} c_n T_n(x)$$

[4]Questo accade perché l'approssimazione tramite la funzione coseno garantisce che l'errore di approssimazione sia minimizzato in senso uniforme sull'intervallo, grazie alla distribuzione uniforme dei punti di massimo e minimo dell'errore. I polinomi di Chebyshev, derivanti, come si vede nella formula chiusa, dalla funzione coseno, riescono così a evitare le grandi oscillazioni (fenomeno di Runge) che tipicamente compromettono l'accuratezza delle approssimazioni polinomiali su intervalli estesi, garantendo una maggiore stabilità e precisione.

dove c_n sono i coefficienti della serie, determinati in modo da minimizzare l'errore di approssimazione.

Rispetto al *Runge's phenomenon*, i polinomi di Chebyshev offrono un miglior comportamento rispetto ai polinomi generici e di Legendre, grazie alla loro capacità di distribuire meglio i punti di interpolazione lungo l'intervallo, riducendo così le oscillazioni ai bordi. Tuttavia, anche i polinomi di Chebyshev soffrono della stessa mancanza di località che caratterizza i polinomi visti finora, poiché ogni variazione dei coefficienti influenza la funzione su tutto l'intervallo considerato. Questo li rende meno adatti rispetto ai B-splines in contesti dove è necessario un certo grado di località [19].

2.6 Addestramento delle KANs

Il processo di addestramento delle Kolmogorov-Arnold Networks condivide le idee di base con quello delle reti neurali tradizionali. In particolare, anche nelle KANs si fa uso della retropropagazione dell'errore (*backpropagation*) per ottimizzare i pesi della rete e ridurre la funzione di perdita[5] ([20]). Tuttavia, l'approccio specifico per le KANs presenta alcune particolarità, soprattutto per quanto riguarda le funzioni di attivazione parametrizzate tramite B-splines[6].

Idee Principali del KAN-Learning

L'idea centrale sottostante all'apprendimento delle KANs è quella di rendere le posizioni dei punti di controllo delle funzioni di attivazione apprendibili durante questo processo, consentendo al modello di apprendere le attivazioni che meglio si adattano ai dati. In altre parole, le KANs permettono al modello di plasmare dinamicamente le funzioni di attivazione, modellando forme complesse e arbitrarie. Matematicamente, nelle KANs, ogni funzione di attivazione $\phi(x)$ è definita come:

$$\phi(x) = w\left(b(x) + spline(x)\right)$$

dove:

- $b(x)$ rappresenta una funzione di base, tipicamente una funzione semplice e fissa come la sigmoid $b(x) = \frac{x}{1+e^{-x}}$, che fornisce una prima approssimazione della non linearità necessaria per la funzione di attivazione. In questo contesto, $b(x)$ stabilisce una base uniforme e consistente su cui viene aggiunta la complessità fornita dalla componente splines, contribuendo a garantire che la funzione di attivazione mantenga un comportamento desiderato anche prima che i B-splines siano completamente addestrati.

[5] La *funzione di perdita* (o in inglese "*loss function*") è una misura che quantifica quanto il risultato prodotto dalla rete neurale si discosti dal risultato atteso. Durante l'addestramento della rete, l'obiettivo è minimizzare questa funzione, che rappresenta l'errore commesso dalla rete nelle sue previsioni. Funzioni di perdita comuni includono l'errore quadratico medio (MSE, dall'inglese "mean squared error") per problemi di regressione e l'entropia incrociata ("cross-entropy") per problemi di classificazione.

[6] Quanto presentato in questa sezione prende spunto dalla trattazione in [10]

- $spline(x)$ è la componente splines della funzione di attivazione, che è parametrizzata dai coefficienti c_i e dalle funzioni $B_i(x)$. La formula per $spline(x)$ è quindi:

$$spline(x) = \sum_i c_i B_i(x)$$

dove c_i sono i parametri apprendibili che determinano la posizione dei punti di controllo dei B-splines $B_i(x)$.

- w è un parametro scalare che controlla il peso complessivo della funzione di attivazione. Sebbene descritto come teoricamente ridondante, w viene mantenuto per fornire un controllo aggiuntivo sull'ampiezza delle funzioni di attivazione durante l'addestramento.

Fase di Inizializzazione

Durante l'inizializzazione, ogni funzione di attivazione $\phi(x)$ viene inizializzata imponendo $spline(x) \approx 0$. Questo viene fatto assegnando ai coefficienti c_i dei B-splines valori estratti da una distribuzione normale[7] con media nulla e varianza ridotta ($\sigma^2 \approx 0.1$):

$$c_i \sim \mathcal{N}(0, \sigma^2)$$

Inoltre, il parametro w viene inizializzato ricorrendo alla Xavier Initialization[8], che differenzia tra distribuzione uniforme e normale:

- **Distribuzione uniforme**[9]:

$$w \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right)$$

dove:

- $\mathcal{U}$ significa che si sta campionando su una distribuzione uniforme.
- n_{in} è la dimensione dell'input.
- n_{out} è la dimensione dell'output.

[7]La distribuzione normale, o gaussiana, è una distribuzione di probabilità continua caratterizzata dalla funzione densità: $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$, dove μ è la media e σ è la deviazione standard. È utilizzata per modellare fenomeni naturali che tendono a concentrarsi attorno a un valore medio.

[8]Tecnica utilizzata per inizializzare i pesi di una rete neurale in modo tale da mantenere l'output dei neuroni all'interno di un intervallo ragionevole durante l'addestramento. Questa tecnica bilancia la varianza dei pesi in modo che sia costante tra i layer, facilitando la convergenza.

[9]La distribuzione uniforme è una distribuzione di probabilità in cui ogni risultato in un intervallo specificato ha la stessa probabilità di verificarsi. La distribuzione uniforme continua è caratterizzata dalla funzione densità $f(x) = \frac{1}{b-a}$, per x nell'intervallo $[a, b]$.

- **Distribuzione normale**:

$$w \sim \mathcal{N}\left(0, \frac{2}{n_{in} + n_{out}}\right)$$

dove:

- $\mathcal{N}$ significa che si sta campionando su una distribuzione normale.
- n_{in} è la dimensione dell'input.
- n_{out} è la dimensione dell'output.

Addestramento

L'addestramento delle KANs procede attraverso i seguenti passaggi:

1. Inizializzazione delle Matrici ϕ^k
Per ogni layer k della rete, le matrici ϕ^k vengono inizializzate come segue:

$$\phi^k = \begin{bmatrix} w\left(b(x) + \sum_i c_i B_i(x)\right) & \phi_{12}(\cdot) & \dots & \phi_{1n}(\cdot) \\ w\left(b(x) + \sum_i c_i B_i(x)\right) & \phi_{22}(\cdot) & \dots & \phi_{2n}(\cdot) \\ \vdots & \vdots & \ddots & \vdots \\ w\left(b(x) + \sum_i c_i B_i(x)\right) & \phi_{m2}(\cdot) & \dots & \phi_{mn}(\cdot) \end{bmatrix}$$

In forma compatta, questa matrice può essere rappresentata come:

$$\phi^k(x) = w\left(b(x) + \sum_i c_i B_i(x)\right)$$

2. "Forward Pass"
Si esegue il passaggio in avanti attraverso la rete, calcolando l'output finale:

$$KAN(x) = \phi^L\left(\phi^{L-1}\left(\dots\left(\phi^2\left(\phi^1(x)\right)\right)\right)\right)$$

3. Calcolo della Funzione di Perdita e Backpropagation
Si calcola la funzione di perdita e si utilizza la *backpropagation* per aggiornare i parametri apprendibili, inclusi i coefficienti c_i, le posizioni dei punti di controllo e il parametro w.

2.7 Possibili Estensioni e Varianti delle KANs

In questa sezione accenneremo a diverse varianti ed estensioni delle KANs come discusse finora, che sono emerse nella ricerca recente, evidenziando la versatilità di questa struttura[10].

Una delle varianti più promettenti è l'integrazione delle KANs con reti convoluzionali, portando alla nascita delle *"convolutional KANs"* (ConvKANs). In questa architettura, le trasformazioni lineari tipiche dei layer convoluzionali sono sostituite da funzioni di attivazione non lineari parametrizzate in ciascun pixel, permettendo una rappresentazione più ricca e dettagliata dei dati visivi. Questo potrebbe risultare particolarmente utile in applicazioni come il riconoscimento delle immagini e l'elaborazione video.

Un altro progetto interessante è *GraphKAN*, che esplora l'uso delle KANs in combinazione con *reti neurali su grafi*[11]. In GraphKAN, le trasformazioni delle funzioni di attivazione delle KANs vengono applicate ai nodi di un grafo, migliorando la capacità della rete di catturare relazioni complesse e strutture all'interno di dati rappresentabili sotto forma di grafo, come nelle reti sociali o nei sistemi di raccomandazione.

Ricollegandoci a quanto discusso precedentemente, ci sono implementazioni delle KANs che sfruttano polinomi di Chebyshev e, in alcuni casi, anche di Legendre, come funzioni di attivazione al posto dei B-splines. Queste varianti, note nel primo caso come *"ChebyKANs"*, cercano di sfruttare le proprietà ortogonali dei polinomi per migliorare l'efficienza computazionale e la precisione dell'approssimazione[12].

Infine, è interessante menzionare i progetti che integrano le KANs con modelli di transformer[13], come *"KAN-GPT"*, che adatta le KANs all'architettura dei transformers, utilizzata nei modelli di linguaggio di grandi dimensioni come ChatGPT[14]. In questi progetti, le KANs vengono utilizzate per sostituire le componenti tradizionali dei transformers, migliorando potenzialmente l'efficienza e la capacità di apprendimento di tali modelli.

Queste estensioni e varianti dimostrano come la struttura delle KANs possa essere adattata a diversi contesti, aprendo la strada a nuove applicazioni e miglioramenti nei modelli di deep learning di cui disponiamo oggi.

[10] Repository GitHub: [21]

[11] Le *reti neurali su grafi* ("graph neural networks", GNN) sono una classe di reti neurali progettate per lavorare su dati rappresentati sotto forma di *grafi*. Un *grafo* è una struttura composta da nodi (o vertici) collegati da archi, utilizzata per rappresentare relazioni tra elementi.

[12] Repositories Github delle citate implementazioni disponibili rispettivamente tramite i links in [22] e [23].

[13] I *transformers* sono un'architettura di rete neurale utilizzata principalmente per compiti di elaborazione del linguaggio naturale.

[14] I *modelli di linguaggio di grandi dimensioni* ("large language models", LLM) sono modelli di deep learning addestrati su enormi quantità di testo, utilizzati per generare testo in linguaggio naturale.

Capitolo 3

KANs ed MLPs: un Confronto

Alla luce di quanto esposto nei capitoli precedenti, risulta chiaro come le reti neurali multistrato (le già introdotte MLPs) abbiano costituito, per decenni, un pilastro fondamentale nello sviluppo delle tecnologie di intelligenza artificiale, grazie alla loro versatilità e capacità di approssimare funzioni complesse in vari contesti applicativi. Tuttavia, le limitazioni strutturali e operative che caratterizzano le MLPs hanno spinto i ricercatori a cercare delle alternative, di cui una delle più innovative e potenzialmente rivoluzionarie è rappresentata, come detto, dalle reti neurali di Kolmogorov-Arnold.

Nel presente capitolo verrà delineato un confronto tra le MLPs e le KANs, con l'obiettivo di introdurre non solo le prerogative di queste ultime, ma anche di individuare gli ambiti in cui esse possono superare le MLPs, offrendo miglioramenti significativi in termini di efficienza, velocità di apprendimento e gestione della dimensionalità. Parallelamente, verranno esplorati anche gli aspetti in cui le KANs mostrano ancora delle carenze rispetto alle reti neurali tradizionali, con l'intento di offrire una valutazione critica e bilanciata delle potenzialità e dei limiti di questa nuova tecnologia. In questo contesto, si cercherà inoltre di identificare gli ambiti applicativi più promettenti per queste reti neurali, valutandone possibili implicazioni nel panorama attuale dell'intelligenza artificiale.

3.1 Multi-Layer Perceptrons

Le reti neurali multistrato (MLPs) costituiscono l'architettura più consolidata e studiata nel campo dell'intelligenza artificiale. La loro struttura è basata su una sequenza di strati di neuroni organizzati in modo gerarchico, in cui ogni livello elabora gli output del livello precedente. La rappresentazione delle informazioni e la capacità di apprendere funzioni complesse derivano dall'interazione di queste trasformazioni tra strati adiacenti.

Architettura

Consideriamo una MLP composta da L strati, dove il livello l (con $l = 1, 2, \ldots, L$) contiene n_l neuroni. Il vettore iniziale di input $\mathbf{x} \in \mathbb{R}^{n_0}$ viene trasformato in un output $\mathbf{y} \in \mathbb{R}^{n_L}$ attraverso una serie di trasformazioni lineari seguite da funzioni di attivazione non lineari.

La trasformazione che avviene tra lo strato $l-1$ e lo strato l può essere formalizzata come:

$$\mathbf{z}^{(l)} = \sigma\left(\mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}\right) = \sigma\left(\begin{bmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \dots & w_{1n_{l-1}}^{(l)} \\ w_{21}^{(l)} & w_{22}^{(l)} & \dots & w_{2n_{l-1}}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_l1}^{(l)} & w_{n_l2}^{(l)} & \dots & w_{n_ln_{l-1}}^{(l)} \end{bmatrix} \begin{bmatrix} z_1^{(l-1)} \\ z_2^{(l-1)} \\ \vdots \\ z_{n_{l-1}}^{(l-1)} \end{bmatrix} + \begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_{n_l}^{(l)} \end{bmatrix}\right)$$

dove: - $\mathbf{W}^{(l)}$ è la matrice dei pesi che connette lo strato $l-1$ allo strato l, - $\mathbf{z}^{(l-1)}$ è il vettore degli output dallo strato $l-1$, - $\mathbf{b}^{(l)}$ è il vettore dei bias associato allo strato l, - $\sigma(\cdot)$ rappresenta la funzione di attivazione non lineare, applicata "element-wise"[1].

Possiamo inoltre, con una formula più compatta, rappresentare l'output di un singolo neurone j nel livello l come:

$$z_j^{(l)} = \sigma\left(\sum_{i=1}^{n_{l-1}} w_{ji}^{(l)} z_i^{(l-1)} + b_j^{(l)}\right)$$

Combinando queste trasformazioni su tutti i livelli, l'output finale $\mathbf{y}$ della rete può essere espresso come:

$$\mathbf{y} = \sigma^{(L)}\left(\mathbf{W}^{(L)}\sigma^{(L-1)}\left(\mathbf{W}^{(L-1)}\cdots\sigma^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(L-1)}\right) + \mathbf{b}^{(L)}\right)$$

Questa formulazione mostra chiaramente come ogni livello elabori e trasformi l'informazione restituita da quello precedente in modo incrementale, permettendo alla rete di apprendere rappresentazioni complesse dei dati.

Funzioni di Attivazione

Le funzioni di attivazione $\sigma(\cdot)$ sono essenziali per introdurre non linearità nella rete: senza queste funzioni, le MLPs si ridurrebbero a una semplice combinazione lineare degli input, limitando drasticamente la loro capacità di modellare relazioni non lineari tra le variabili. Le funzioni di attivazione più comuni includono:

Sigmoide	**Tangente iperbolica (tanh)**	**Rectified Linear Unit (ReLU)**
$\sigma(x) = \frac{1}{1+e^{-x}}$	$\sigma(x) = \tanh(x)$	$\sigma(x) = \max(0, x)$

Ognuna di queste funzioni ha caratteristiche specifiche che influenzano il comportamento della rete. Ad esempio, la sigmoide è spesso utilizzata per problemi di classifica-

[1] La *matrice dei pesi* tra due strati in una MLP è una matrice che contiene i coefficienti utilizzati per ponderare le connessioni tra i neuroni di uno strato e quelli del successivo. Ogni elemento della matrice rappresenta il peso di una connessione specifica. Il *vettore dei bias* associato a uno strato è invece un vettore che contiene i valori di bias aggiunti ai neuroni di quello strato. Entrambi vengono calibrati durante la fase di apprendimento della rete.

zione binaria[2], mentre la ReLU è preferita nelle reti profonde[3] per la sua efficienza computazionale.

Addestramento

L'addestramento delle MLPs avviene tramite un processo iterativo chiamato retropropagazione del gradiente (ovvero il già citato *algoritmo di backpropagation* ([24])). In questo processo, l'errore tra l'output prodotto dalla rete e l'output desiderato viene calcolato e utilizzato per aggiornare i pesi $\mathbf{W}^{(l)}$ e i bias $\mathbf{b}^{(l)}$ della rete, minimizzando una funzione di costo.

Quest'ultima, che possiamo indicare con $E(\Theta)$, dove Θ rappresenta l'insieme di tutti i parametri della rete, misura la differenza tra l'output predetto $\mathbf{y}$ e l'output target $\mathbf{y}_{\text{target}}$. Una delle funzioni di costo più comuni è l'errore quadratico medio:

$$E(\Theta) = \frac{1}{2} \sum_{i=1}^{N} (y_i - y_{i,\text{target}})^2$$

dove N è il numero di esempi nel dataset di addestramento.

L'aggiornamento dei pesi avviene secondo la regola del *"gradient descent"*[4]:

$$\Theta^{(t+1)} = \Theta^{(t)} - \eta \nabla_\Theta E(\Theta)$$

dove η è il *"learning rate"*, ovvero il fattore di scala che determina quanto velocemente o lentamente il modello aggiorna i suoi parametri in risposta al gradiente calcolato: un valore troppo grande può causare instabilità, mentre un valore troppo piccolo può rallentare notevolmente l'addestramento. Nell'equazione, $\nabla_\Theta E(\Theta)$ rappresenta il *gradiente della funzione costo* rispetto ai parametri della rete: si tratta di un vettore contenente le derivate parziali di $E(\Theta)$ rispetto a ciascun parametro della rete, indicato collettivamente come Θ. Dato che, come accennato appena sopra, l'obiettivo del processo di ottimizzazione è minimizzare la funzione di costo, ad ogni iterazione, i parametri Θ della rete vengono aggiornati sottraendo il prodotto tra il *learning rate* η e il gradiente $\nabla_\Theta E(\Theta)$ della loss function. Questo processo permette alla rete di adattarsi gradualmente ai dati con cui è addestrata, migliorando progressivamente le sue prestazioni e riducendo l'errore di predizione [1].

[2]Tipo di problemi in cui il modello deve decidere tra due classi distinte, ad esempio, se un'e-mail è spam o non spam.

[3]Le reti profonde ("deep networks"in inglese) sono reti neurali con un gran numero di strati nascosti, che permettono di apprendere rappresentazioni più complesse e astratte dei dati.

[4]Algoritmo di ottimizzazione utilizzato per minimizzare la funzione di costo durante l'addestramento delle reti neurali. Consiste nel muoversi nella direzione opposta al gradiente di tale funzione rispetto ai parametri della rete, in modo da trovare il minimo locale della stessa.

3.2 MLPs vs KANs

KANs: un Breve Recap

Nel secondo capitolo sono state delineate le caratteristiche fondamentali delle reti neurali di Kolmogorov-Arnold, evidenziando come esse rappresentino un significativo allontanamento dalle reti neurali multistrato tradizionali. Le KANs si basano su un'architettura che trae origine dal teorema di Kolmogorov-Arnold, offrendo un nuovo approccio nella rappresentazione delle funzioni multivariate. In particolare, la rivoluzione introdotta dalle KANs risiede nell'utilizzo di funzioni di attivazione parametrizzate, come i B-splines, che denotiamo con ϕ o g, e che, a differenza delle funzioni di attivazione nelle MLPs, non sono fisse, ma vengono apprese dalla rete stessa durante il processo di addestramento. In una KAN ogni strato della rete esegue una trasformazione che coinvolge sia la somma pesata degli input sia l'applicazione di funzioni di attivazione parametrizzate. Queste ultime non sono predefinite a priori come nel caso delle MLPs, ma, come appena ricordato, vengono apprese durante l'addestramento, rendendo questa nuova tipologia di rete neurale estremamente adattabile e capace di modellare con precisione le caratteristiche specifiche dei dati.

Analisi Comparativa

Nodi ed Archi

Nelle MLPs la struttura della rete è costituita da nodi (neuroni) e archi (connessioni pesate) organizzati in strati sequenziali. In ogni strato, ciascun nodo riceve input da tutti i nodi del livello precedente, applica una funzione di attivazione sui valori ricevuti, e poi trasmette l'output a tutti i nodi del livello successivo. In questo schema, le funzioni di attivazione sono poste sui nodi, determinando come gli input combinati vengono trasformati prima di essere passati allo strato successivo.

Nelle KANs, invece, l'architettura si distingue in modo significativo. Qui, i nodi non rappresentano neuroni che applicano funzioni di attivazione, ma piuttosto funzioni univariate che vengono combinate attraverso una rete complessa di archi. La caratteristica distintiva delle KANs è che le funzioni di attivazione non sono associate ai nodi, bensì agli archi della rete. Questi archi non rappresentano semplici connessioni pesate, ma operano come composizioni di funzioni. Ogni arco applica una funzione di attivazione apprendibile tra le funzioni univariate rappresentate dai nodi che collega, permettendo una rappresentazione più ricca e flessibile della funzione target. Questo approccio consente alle KANs di modellare relazioni non lineari con una maggiore efficacia rispetto alle MLPs tradizionali, sfruttando la combinazione di funzioni univariate in modo altamente adattabile e specifico per il problema in esame.

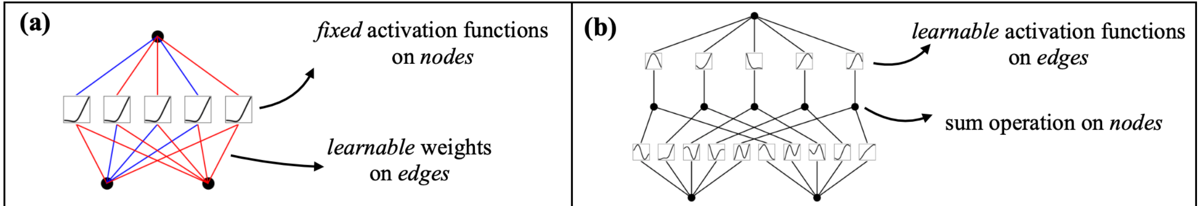


Figura 3.1: *Confronto tra una KAN ed una MLP dalle stesse dimensioni (Immagini adattate da [2]).*

Funzioni di Attivazione

Nelle MLPs, le funzioni di attivazione sono fissate e applicate dopo ogni trasformazione lineare dei dati, con lo scopo di introdurre non linearità nella rete. Queste funzioni, come la ReLU o la sigmoide, sono scelte a priori e non sono soggette ad apprendimento. Nelle KANs, al contrario, le funzioni di attivazione, che idealmente, come visto, assumono la forma di B-splines, non sono predefinite ma rappresentano parametri liberi che la rete apprende durante il processo di addestramento. Questa capacità di apprendere le funzioni di attivazione conferisce alle KANs una flessibilità notevolmente superiore.

Trasformazioni

Per le MLPs, la trasformazione che avviene in un generico strato l è data da (ricorrendo alla notazione già introdotta in questo capitolo):

$$\mathbf{z}^{(l)} = \sigma\left(\mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}\right)$$

Nelle KANs, la trasformazione in un generico strato l è invece esprimibile come:

$$\mathbf{z}^{(l)} = \phi^{(l)}\left(\mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}\right)$$

dove $\phi^{(l)}$ rappresenta una funzione di attivazione appresa, come i B-splines, e $\mathbf{W}^{(l)}$ è la matrice dei pesi associata allo strato l.

Un confronto diretto tra le formule relative a una rete deep per le due tipologie di rete è il seguente:

$$\begin{aligned}
\text{MLP(x)} &= \left(\mathbf{W}_{L-1} \circ \sigma \circ \mathbf{W}_{L-2} \circ \sigma \circ \cdots \circ \mathbf{W}_1 \circ \sigma \circ \mathbf{W}_0\right)(x) \\
\text{KAN(x)} &= \left(\Phi_{L-1} \circ \Phi_{L-2} \circ \cdots \circ \Phi_1 \circ \Phi_0\right)(x)
\end{aligned}$$

È evidente che nelle MLPs le trasformazioni lineari e le non linearità vengono trattate separatamente come $\mathbf{W}$ e σ, mentre nelle KANs queste vengono integrate insieme nelle funzioni Φ.

Un Punto in Comune: l'Addestramento

Nonostante le profonde differenze strutturali tra le MLPs e le KANs, entrambe le architetture condividono principi di addestramento simili. In particolare, entrambe possono essere addestrate utilizzando algoritmi di retropropagazione del gradiente, con l'obiettivo di minimizzare una funzione di costo che come già esposto misura la discrepanza tra l'output della rete e i valori target. Tuttavia, nelle KANs, il processo di apprendimento si estende non solo ai pesi, ma anche alle funzioni di attivazione stesse, aggiungendo un ulteriore livello di complessità e flessibilità al modello.

3.3 Problematiche delle MLPs

Le reti neurali multistrato hanno rappresentato negli ultimi decenni il paradigma dominante dell'apprendimento automatico. Tuttavia, nonostante la ricerca continua e numerosi affinamenti, queste reti presentano limitazioni architetturali che ne compromettono l'efficacia, specialmente in scenari complessi e ad alta dimensionalità. Le MLPs sono strutturate in modo da utilizzare funzioni di attivazione predefinite e non adattabili, il che porta a due problematiche principali: la *curse of dimensionality* e la *catastrophic forgetting*. Questi limiti derivano dalla rigidità strutturale di queste reti, le quali richiedono un numero elevato di parametri per rappresentare adeguatamente le relazioni complesse tra le variabili e mostrano difficoltà nel mantenere informazioni apprese in precedenza quando vengono introdotti nuovi compiti.

A fronte di queste problematiche, le reti neurali di Kolmogorov-Arnold si propongono come un'alternativa più flessibile, grazie in primis all'uso di funzioni di attivazione apprese durante l'addestramento, che consentono una migliore gestione delle complessità dei dati. Nelle prossime pagine, esamineremo in dettaglio le limitazioni delle MLPs e come le KANs promettano di riuscire ad affrontare queste sfide in modo più efficiente.

The Curse of Dimensionality

La *curse of dimensionality* è una delle problematiche più insidiose nelle reti neurali multistrato, in particolare quando si affrontano problemi ad alta dimensionalità: ci si riferisce infatti al fenomeno per cui l'aumento delle dimensioni di un dataset rende esponenzialmente più difficile l'analisi e l'apprendimento dei modelli. Questo fenomeno si verifica quando l'incremento delle dimensioni nello spazio dei dati richiede un numero esponenzialmente maggiore di parametri per modellare accuratamente le relazioni tra le variabili.

Consideriamo, per esempio, un'architettura MLP con un input di dimensione d, un singolo strato nascosto con h neuroni, e un output di dimensione k. Il numero totale di parametri P_{MLP} da ottimizzare nel corso dell'addestramento, uguale alla somma tra i

pesi e i bias, è dato da[5]:

$$P_{\text{MLP}} = d \cdot h + h \cdot k + h + k$$

Questa quantità aumenta rapidamente con il crescere di d, rendendo difficile l'addestramento e la generalizzazione della rete. Ad esempio, in applicazioni di visione artificiale[6], dove le immagini ad alta risoluzione richiedono input di grandi dimensioni (ad esempio $d = 1024 \times 1024$), il numero di parametri può diventare ingestibile. Un altro esempio si verifica nei modelli di elaborazione del linguaggio naturale[7], dove l'uso di vettori di parole di grandi dimensioni (d dell'ordine delle migliaia) può portare a una crescita esponenziale dei parametri, pesando inevitabilmente sul costo computazionale e aggravando il fenomeno dell'overfitting[8].

In contrasto, le KANs affrontano la *curse of dimensionality* in modo più efficiente. Anziché parametrizzare esplicitamente tutte le connessioni tra strati, le KANs sfruttano la scomposizione delle funzioni multivariate in funzioni univariate, riducendo drasticamente il numero di parametri. Se consideriamo una KAN con lo stesso numero di input d, un output univariato, e una griglia parametrizzata di m nodi di controllo per le funzioni di attivazione[9], il numero totale di parametri P_{KAN} può essere approssimato da[10]:

$$P_{\text{KAN}} = m \cdot (d + 1)$$

Per esempio, se $m = 20$, $d = 1000$, e $k = 10$, si ottengono 20,020 parametri per una KAN. In confronto, per una MLP con architettura e dimensioni paragonabili, ricorrendo alla formula vista appena sopra, il numero di parametri P_{MLP} risulterebbe pari circa 505,510, un ordine di grandezza superiore rispetto a P_{KAN}. Questa differenza permette alle KANs di gestire con maggiore efficacia l'aumento delle dimensioni, mantenendo una buona capacità di generalizzazione e limitando il rischio di overfitting.

[5] Questa formula si applica a una rete completamente connessa. Il termine $d \cdot h$ rappresenta il numero di pesi tra l'input e lo strato nascosto, mentre $h \cdot k$ rappresenta il numero di pesi tra lo strato nascosto e lo strato di output. I termini h e k rappresentano rispettivamente i bias associati ai neuroni dello strato nascosto e dello strato di output.

[6] La visione artificiale (dall'inglese "computer vision"), è un campo dell'intelligenza artificiale che si occupa di sviluppare modelli e tecniche per interpretare e decodificare immagini e video.

[7] L'elaborazione del linguaggio naturale (NLP, dall'inglese "natural language processing") è un campo dell'AI che si occupa dell'interazione tra computer e linguaggio umano.

[8] Fenomeno che si verifica quando un modello di machine learning diventa eccessivamente complesso e inizia a "memorizzare"il rumore o le caratteristiche specifiche del set di addestramento, piuttosto che apprendere pattern generalizzabili. Questo porta a prestazioni eccellenti sul set di addestramento originario, ma a una scarsa capacità di generalizzare su dati nuovi e mai visti prima, compromettendo l'efficacia complessiva del modello.

[9] Ci si riferisce al numero di nodi di controllo utilizzati per definire i B-splines, ovvero le funzioni di attivazione della rete. Questi nodi di controllo determinano la forma delle funzioni univariate, e il loro numero influenza direttamente la complessità del modello.

[10] La formula si applica assumendo che ogni funzione di attivazione nella KAN sia rappresentata da un B-spline parametrizzato da m punti di controllo. Qui, d rappresenta la dimensione dell'input, e il termine aggiuntivo 1 rappresenta il bias associato. Ogni funzione di attivazione univariata dipende quindi da $d + 1$ parametri, moltiplicati appunto per m punti di controllo.

L'adattabilità delle KANs in spazi ad alta dimensione rappresenta dunque un vantaggio significativo. La loro capacità di modellare funzioni complesse con un numero ridotto di parametri non solo migliora l'efficienza computazionale, ma facilita anche l'applicazione delle KANs a problemi di grande scala, rendendo le KANs una soluzione più robusta per affrontare la *curse of dimensionality* [25].

Catastrophic Forgetting

Il fenomeno della *catastrophic forgetting* è l'altra grande criticità associata alle MLPs, ed è particolarmente evidente quando queste reti vengono addestrate sequenzialmente su diversi compiti. Durante l'addestramento, le MLPs tendono ad adattare i loro parametri in modo tale che le nuove informazioni acquisite sovrascrivano quelle precedentemente apprese. Questo porta alla perdita delle conoscenze accumulate, riducendo significativamente l'efficacia della rete in contesti di apprendimento continuo (o "lifelong learning").

Le reti neurali di Kolmogorov-Arnold offrono un potenziale rimedio a questo problema attraverso diverse caratteristiche architettoniche e funzionali. Un elemento cruciale che permette alle KANs di mitigare la *catastrophic forgetting* è l'uso dei B-splines come funzioni di attivazione. I B-splines possiedono la proprietà di località, per cui la modifica di un dato punto di controllo interessa solo il tratto associato della funzione di attivazione. Questa caratteristica permette di adattare la rete a nuovi dati senza compromettere le informazioni già apprese, evitando quindi che l'intera funzione di attivazione venga alterata globalmente. Al contrario, le funzioni di attivazione standard utilizzate nelle MLPs, come ReLU o sigmoide, non offrono questa località, portando a un adattamento globale della rete e a un'inevitabile perdita di conoscenze passate. Consideriamo nuovamente una generica funzione di attivazione rappresentata da un B-spline:

$$\phi(x) = \sum_{i=1}^{m} c_i B_i(x), \quad \text{dove} \quad B_i(x) = \sum_{j=0}^{n} P_{i,j} N_{i,j}(x)$$

In queste formule, c_i rappresentano i coefficienti associati a ciascun B-spline $B_i(x)$, risultanti dalla combinazione delle B-splines basis functions $N_{i,j}(x)$, e $P_{i,j}$ sono i punti di controllo che determinano la forma di queste basis functions. La proprietà di località dei B-splines implica che la modifica di un punto di controllo $P_{i,j}$ influenzi solo il tratto di funzione ad esso associato, preservando così la stabilità delle altre sezioni della funzione.

I vantaggi delle KANs in scenari dove la conservazione della memoria è cruciale sono particolarmente evidenti in contesti come l'apprendimento incrementale[11] e l'addestramento su serie temporali. Ad esempio, in applicazioni di monitoraggio continuo dove è fondamentale mantenere traccia di eventi passati pur aggiornando il modello con nuovi

[11] L'apprendimento incrementale si riferisce a un approccio in cui un modello è addestrato in modo iterativo su piccoli batch di dati, aggiornando costantemente i parametri.

dati, le KANs possono offrire una maggiore stabilità rispetto alle MLPs, riducendo il rischio di perdita di informazioni essenziali [26].

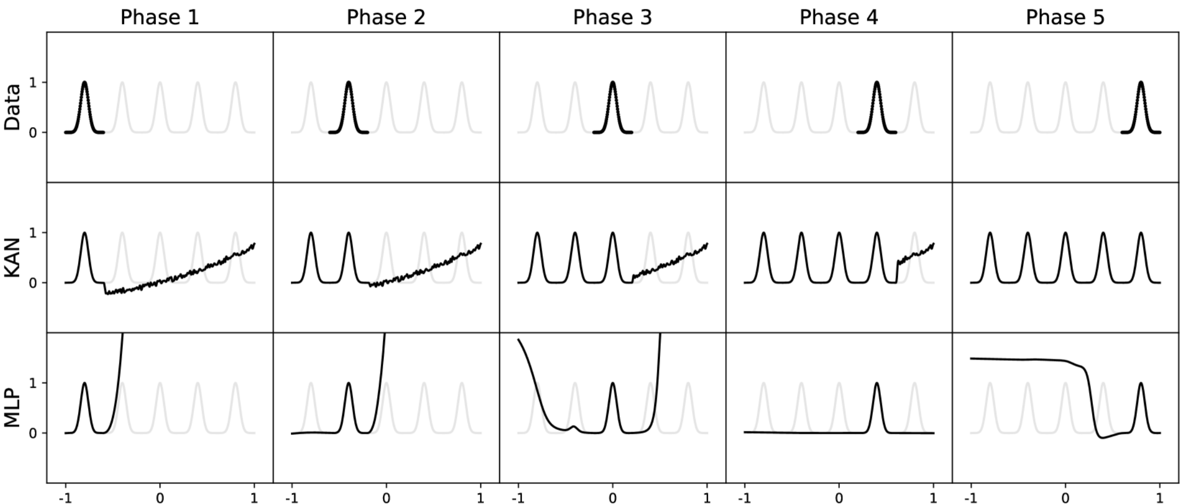


Figura 3.2: *Grafici dei risultati ottenuti in [2] riguardo alla catastrophic forgetting: "Un Problema di Apprendimento Continuo di Tipo Semplificato. Il dataset considerato è un problema di regressione monodimensionale caratterizzato dalla presenza di cinque picchi gaussiani (riga superiore). I dati associati a ciascun picco vengono presentati in maniera sequenziale, anziché simultanea, alle KANs e alle MLPs. Le KANs (riga centrale) dimostrano la capacità di evitare completamente la catastrophic forgetting, mentre le MLPs (riga inferiore) evidenziano un grave fenomeno di catastrophic forgetting."*

3.4 Parametri Tipici delle Reti Neurali

Efficienza

Il concetto di efficienza in una rete neurale può essere definito come la capacità del modello di ottenere risultati accurati con il minimo impiego di risorse computazionali. L'efficienza si misura attraverso vari indicatori, tra cui il numero di operazioni necessarie per l'addestramento e l'inferenza, la memoria richiesta, e il tempo di elaborazione. Una metrica comunemente utilizzata per valutare l'efficienza è il cosiddetto "flops"(dall'inglese "floating point operations per second"), che misura il numero di operazioni in virgola

mobile che la rete deve eseguire per elaborare un singolo input[12]:

$$\text{FLOPs} = \sum_{l=1}^{L} (2 \cdot d_l \cdot h_l)$$

dove L è il numero di strati della rete, d_l è il numero di dimensioni dell'input allo strato l, e h_l è il numero di neuroni nello strato l. Un valore di FLOPs inferiore indica una maggiore efficienza, a parità di prestazioni[13].

Le MLPs, caratterizzate da una struttura densa e un numero elevato di parametri, richiedono una quantità significativa di risorse computazionali. Per esempio, in una MLP con d dimensione di input, uno strato nascosto con h neuroni e k dimensione di output, il numero totale di operazioni per un passaggio di forward propagation è dell'ordine di $O(d \cdot h + h \cdot k)$. In applicazioni come la modellazione predittiva di dati genomici[14], il numero di operazioni richieste cresce rapidamente, influenzando negativamente l'efficienza della rete.

In contrapposizione, le KANs sfruttano la scomposizione delle funzioni multivariate, riducendo significativamente il numero di operazioni necessarie. Per una KAN con funzioni di attivazione parametrizzate tramite B-splines, il numero di operazioni richieste può essere espresso come $O(m \cdot (d + 1))$, dove m è il numero di nodi nella griglia parametrizzata. In scenari con elevata dimensionalità dell'input, questa riduzione del numero di operazioni si traduce in un miglioramento significativo dell'efficienza computazionale.

Le MLPs richiedono anche una maggiore quantità di memoria per immagazzinare i pesi, specialmente quando la rete è profonda e include molti neuroni per strato. Questo comporta un elevato consumo di memoria, particolarmente critico in applicazioni come il riconoscimento facciale su grandi dataset. D'altro canto, le KANs, grazie alla loro architettura più snella, riducono il numero complessivo di parametri, con un conseguente risparmio di memoria. I parametri, oltre a dover essere appresi, devono anche essere memorizzati. Come abbiamo visto nella sezione precedente, le MLPs, avendo un numero maggiore di parametri rispetto alle KANs, richiedono quindi una quantità di memoria significativamente superiore rispetto a queste ultime [27].

[12] Le operazioni in virgola mobile ("floating point operations") sono operazioni aritmetiche eseguite appunto sui numeri in virgola mobile ("floating point numbers"), che permettono di trattare una vasta gamma di valori reali, che possono dunque avere una parte decimale, inclusi numeri molto grandi e molto piccoli, con un certo grado di precisione.

[13] La formula riflette il numero di operazioni aritmetiche (moltiplicazioni e somme) necessarie per elaborare un singolo input in una rete neurale. Ogni strato l della rete esegue $2 \cdot d_l \cdot h_l$ operazioni in virgola mobile. Il fattore 2 è dovuto al fatto che ogni neurone effettua una moltiplicazione (del peso per l'input) e un'addizione (del bias). Sommando queste operazioni su tutti gli strati L, si ottiene il numero totale di FLOPs.

[14] La modellazione predittiva di dati genomici si riferisce all'uso di modelli per prevedere esiti basati su sequenze di DNA o altre informazioni genetiche.

Interpretabilità

L'interpretabilità è un aspetto cruciale nella valutazione della trasparenza e della comprensibilità di un modello di intelligenza artificiale, soprattutto in settori critici come la sanità, la finanza e l'ingegneria. Un modello interpretabile consente agli utenti di comprendere il processo decisionale del sistema, identificare eventuali bias, e garantire che le decisioni siano giustificabili e trasparenti. In questo contesto, il confronto tra l'interpretabilità delle MLPs e delle KANs risulta particolarmente informativo.

Le MLPs, a causa della loro complessità e del gran numero di strati nascosti, sono spesso considerate delle "scatole nere"(dall'inglese "black box"). La difficoltà principale nell'interpretare queste reti risiede nel fatto che i pesi appresi nei vari strati non hanno un significato intuitivo o direttamente collegabile alle caratteristiche del problema in esame. Ad esempio, in una rete neurale progettata per il riconoscimento di immagini, i pesi appresi nei diversi strati risultano difficili da interpretare in termini di caratteristiche visive specifiche. Questo rende complesso il processo di spiegazione delle decisioni prese dal modello, limitandone la trasparenza.

Al contrario, le KANs offrono un maggiore livello di interpretabilità grazie alla loro struttura modulare. La scomposizione delle funzioni multivariate in funzioni univariate permette di isolare e analizzare l'impatto delle singole variabili indipendenti. Inoltre, l'uso dei B-splines come funzioni di attivazione contribuisce ulteriormente all'interpretabilità del modello, poiché le modifiche ai punti di controllo possono essere direttamente correlate a variazioni specifiche nel comportamento del sistema. Un altro aspetto che accresce l'interpretabilità delle KANs è la possibilità di visualizzare e analizzare le funzioni di attivazione apprese. A differenza delle funzioni di attivazione standard nelle MLPs, come ReLU o sigmoide, che sono statiche e uniformi per tutti i neuroni di uno strato, le funzioni di attivazione nelle KANs possono variare e adattarsi in base ai dati specifici. Questa flessibilità consente di tracciare e comprendere meglio come le variazioni nei dati di input influenzano l'output del modello, facilitando l'individuazione di eventuali anomalie o bias.

L'interpretabilità riveste particolare importanza non solo in applicazioni sensibili come per esempio la diagnostica medica, ma anche nelle scienze in generale, dove è fondamentale comprendere direttamente le relazioni funzionali che intercorrono tra i dati. In tali contesti, l'uso delle KANs può offrire un vantaggio significativo rispetto alle MLPs, non solo per la loro capacità di mantenere elevate prestazioni, ma anche per la trasparenza e la fiducia che offrono agli utenti finali. Inoltre, la maggiore interpretabilità delle KANs permette anche di avere dei vantaggi dal punto di vista pedagogico, rendendole strumenti didattici efficaci, facilitando la comprensione del funzionamento del machine learning e promettendo di superare il problema della "black box"[15].

[15] Questa sezione e le successive si rifanno principalmente, come già detto, all'articolo di Ziming Liu [2], a meno che non sia diversamente indicato.

Interattività

L'interattività è un altro aspetto importante di una rete neurale: essa rappresenta la capacità del modello di adattarsi e rispondere efficacemente ai cambiamenti nei dati di input in tempo reale, mantenendo prestazioni ottimali anche in contesti dinamici. Questa proprietà è fondamentale in applicazioni dove l'ambiente varia rapidamente e richiede un aggiornamento continuo del modello, come nel monitoraggio di sistemi critici o nelle interazioni uomo-macchina. In questo contesto, il paragone tra KANs e MLPs mostra differenze significative che meritano di essere approfondite.

Le MLPs, pur essendo potenti e flessibili in molti contesti, mostrano limitazioni significative quando si tratta di interagire dinamicamente con ambienti in evoluzione. Questo è dovuto principalmente alla loro struttura rigida e al fatto che i parametri delle funzioni di attivazione sono fissi una volta addestrata la rete. In situazioni in cui i dati di input cambiano frequentemente, le MLPs possono richiedere un ri-addestramento completo o parziale per adattarsi ai nuovi dati, il che comporta un elevato costo computazionale e ritardi nella risposta.

Al contrario, le KANs sono progettate per essere più interattive grazie alla loro capacità di aggiornare non solo i pesi, ma anche i parametri delle funzioni di attivazione. Questa flessibilità permette alle KANs di adattarsi rapidamente ai cambiamenti nei dati senza necessità di un ri-addestramento completo. Ad esempio, in un sistema di controllo adattivo[16], dove è fondamentale reagire rapidamente a nuove condizioni operative, le KANs possono regolare i loro parametri interni in tempo reale, mantenendo un elevato livello di performance senza la necessità di interruzioni prolungate.

L'interattività delle KANs è particolarmente vantaggiosa nel contesto dell'interazione uomo-macchina, dove la capacità di adattarsi alle esigenze e ai comandi dell'utente è cruciale. Uno degli esempi più rilevanti è l'uso delle KANs in contesti di ricerca scientifica. Un ricercatore può infatti manipolare direttamente le funzioni di attivazione delle KANs, specialmente quelle che gestiscono trasformazioni di cui è già a conoscenza, per scoprire nuove relazioni “nascoste”nei dati. Questa possibilità di intervenire in modo interattivo sulle funzioni di attivazione permette di vagliare ipotesi in maniera più efficiente e intuitiva, migliorando la capacità di interpretare i dati e di facilitare nuove possibili scoperte scientifiche.

Infine, l'uso di B-splines come funzioni di attivazione nelle KANs contribuisce ulteriormente all'interattività qui discussa. La possibilità di modificare i punti di controllo dei B-splines permette alle KANs di adattarsi localmente ai cambiamenti nei dati, influenzando solo le porzioni rilevanti di una data funzione di attivazione. Questo approccio riduce il rischio di destabilizzare l'intero modello quando si apportano modifiche, migliorando la robustezza e la capacità di interazione della rete.

[16]Sistema che modifica automaticamente i suoi parametri per mantenere prestazioni ottimali in un ambiente in evoluzione.

Velocità di Apprendimento

Infine, un ruolo cruciale nell'addestramento delle reti neurali è rivestito dalla velocità di apprendimento: questa metrica valuta quanto rapidamente un modello può convergere verso una soluzione ottimale. Confrontando le KANs con le MLPs, emergono differenze significative che sono legate alla loro architettura e alla complessità delle funzioni di attivazione.

Le MLPs tendono ad avere un tempo di apprendimento più rapido rispetto alle KANs. Questa differenza può essere illustrata attraverso la funzione di costo $E(\theta)$ che viene minimizzata durante l'addestramento, dove θ rappresenta il vettore dei parametri del modello. Per una MLP, il vettore dei parametri θ è tipicamente composto dai pesi w e dai bias b, espressi esplicitamente come:

$$\theta_{\text{MLP}} = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{mn} \\ b_1 & b_2 & \dots & b_m \end{pmatrix}$$

Nel caso delle KANs, il vettore dei parametri θ include sia i pesi w che i parametri c delle funzioni di attivazione parametrizzate dai B-splines, espressi invece in questo caso come:

$$\theta_{\text{KAN}} = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{mn} \\ c_1 & c_2 & \dots & c_k \end{pmatrix}$$

Nelle MLPs, l'aggiornamento dei parametri segue la regola di discesa del gradiente:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \nabla_\theta E(\theta)$$

dove, come già specificato in precedenza, η è il learning rate e $\nabla_\theta E(\theta)$ rappresenta il gradiente della funzione di costo rispetto ai parametri θ. Questa operazione è relativamente semplice nelle MLPs, dove le funzioni di attivazione standard, come ReLU o sigmoide, permettono un calcolo dei gradienti meno complesso, facilitando una rapida convergenza.

Le KANs, per contro, richiedono l'ottimizzazione non solo dei pesi, ma anche dei parametri delle funzioni di attivazione, aumentando la complessità del processo di apprendimento. Questo richiede un numero maggiore di operazioni per ogni iterazione, il che rallenta inevitabilmente l'intero processo. Ad esempio, l'aggiornamento dei parametri in una KAN può essere espresso da:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot (\nabla_w E(\theta) + \nabla_c E(\theta))$$

dove ∇_w e ∇_c rappresentano rispettivamente i gradienti rispetto ai pesi e ai parametri delle funzioni di attivazione. Quanto detto introduce un livello di complessità aggiuntivo che può ritardare la convergenza rispetto alle MLPs: si stima che, a parità di condizioni, le KANs richiedano un tempo di apprendimento circa 10 volte maggiore rispetto alle MLPs[17].

Le differenze architettoniche tra MLPs e KANs si riflettono dunque direttamente sulla rapidità di apprendimento. Le MLPs, con le loro funzioni di attivazione fisse e semplici, offrono una velocità di apprendimento superiore, anche se ciò può avvenire a scapito della capacità di rappresentare relazioni complesse. Le KANs, sebbene più lente nell'apprendimento, offrono una flessibilità maggiore, capace di modellare relazioni intricate nei dati, rendendole preferibili in contesti dove la qualità del modello finale è prioritaria rispetto alla velocità di addestramento.

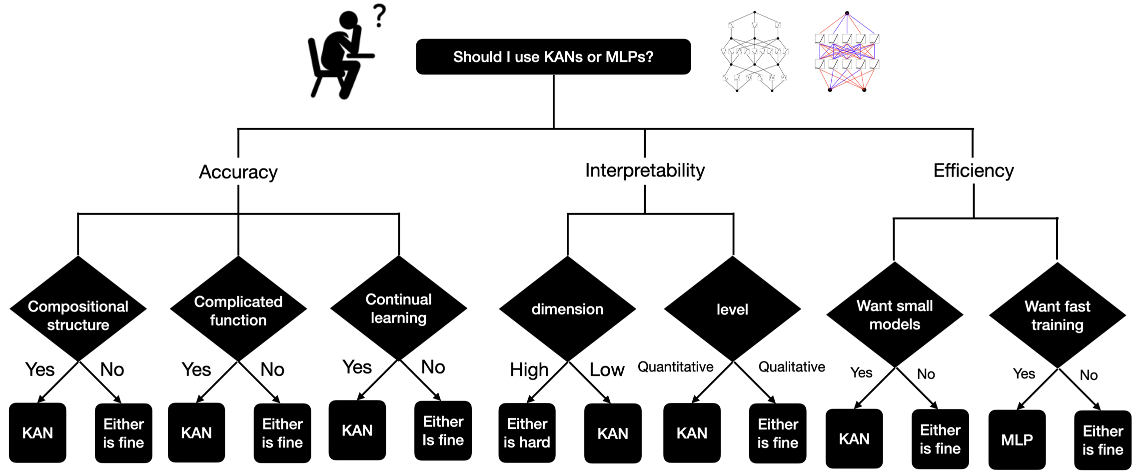


Figura 3.3: *Decision-Tree proposto in [2] per la scelta tra le due tipologie di rete in base alle necessità implementative.*

[17]Si veda [2] per un'analisi dettagliata delle differenze nei tempi di apprendimento tra KANs e MLPs.

Capitolo 4

Implementazione in Python

4.1 Struttura del Capitolo

Il presente capitolo è dedicato all'analisi di due notebooks Jupyter, ciascuno dei quali è composto da sequenze di celle di codice Python, seguite dagli output generati e da commenti esplicativi in formato markdown[1]. All'inizio di ogni notebook è riportata una breve introduzione del problema affrontato, che fornisce il contesto necessario per comprendere l'approccio implementativo adottato. Il primo notebook si concentra sul "function fitting", ovvero sull'approssimazione di funzioni complesse, utilizzando una rete neurale di Kolmogorov-Arnold. Il secondo notebook, invece, è dedicato alla risoluzione di un'equazione alle derivate parziali (PDE, dall'inglese "partial differential equation"), illustrando come una KAN possa essere impiegata per risolvere tali problemi seguendo un approccio numerico e iterativo. Entrambi i notebooks prendono spunto dal codice disponibile nella repository GitHub che correda i due articoli originali sulle KANs[2], e che rappresenta una fonte di riferimento fondamentale per l'implementazione di queste nuove reti. Tuttavia, si sottolinea che il codice presentato nei notebooks è stato parzialmente modificato per meglio adattarsi agli obiettivi di questa tesi. Inoltre, i commenti in codice markdown presenti nei notebooks sono stati scritti personalmente per provare a fornire una spiegazione chiara ed intuitiva degli esempi implementativi. Infine, va evidenziato che non è stata effettuata un'analisi dettagliata dell'implementazione in Python originale degli autori, poiché l'attenzione è stata posta principalmente sulla comprensione e sull'esposizione concettuale dei due esempi presentati, piuttosto che sulla disamina tecnica del codice. Ho scelto di riportare questi due notebooks in particolare perché mi sono parsi particolarmente rappresentativi ed esplicativi delle caratteristiche e delle potenzialità delle KANs.

[1] I notebooks Jupyter ([28]) sono stati utilizzati per eseguire ed analizzare il codice Python ([29]). Successivamente, i notebook in formato `.ipynb` sono stati convertiti in file `.tex` tramite `nbconvert` ([30]), un applicativo di conversione che fa parte del progetto Jupyter.

[2] Per la repository ed i due articoli, in ordine: [31] e [2], [32].

4.2 Function Fitting

4.2.1 Introduzione al Problema

In questo notebook si implementa una KAN con l'obiettivo di approssimare una funzione target specifica definita come $f(x_1, x_2) = |x_1 - \cos(x_2)|$. La KAN viene configurata con una griglia di interpolazione specifica e funzioni di attivazione basate su B-splines, ed è ottimizzata per migliorare la precisione dell'apprendimento.

Il processo di *function fitting* consiste nel trovare una funzione $f : \mathbb{R}^n \to \mathbb{R}$, rappresentata dalla rete, che approssimi al meglio la funzione target. Gli input della rete $\mathbf{X} \in \mathbb{R}^{N \times 2}$ sono costituiti da coppie di variabili x_1 e x_2, mentre l'output $\hat{\mathbf{Y}} \in \mathbb{R}^N$ rappresenta le predizioni della rete, che vengono confrontate con i valori reali $\mathbf{Y} \in \mathbb{R}^N$ corrispondenti alla funzione target.

Il processo di addestramento mira a minimizzare una funzione di perdita $\mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}}) = \frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2$, che quantifica la differenza tra le predizioni della rete e i valori target. La KAN viene raffinata attraverso un'ottimizzazione iterativa, che include la ricerca del valore ottimale della griglia (`grid`) per migliorare l'accuratezza dell'approssimazione.

Il notebook guida anche attraverso l'assegnazione delle funzioni di attivazione simboliche negli archi della rete, sia manualmente che automaticamente. Queste funzioni simboliche $\phi(x)$ sono cruciali per rappresentare fedelmente la funzione target, contribuendo a ridurre ulteriormente l'errore e a raggiungere una precisione numerica elevata.

Alla fine del processo, il modello apprende una rappresentazione della funzione target e viene valutato sia visivamente, attraverso rappresentazioni schematiche, che numericamente, con il calcolo delle perdite di addestramento e di test. L'obiettivo finale è ottenere $\hat{f}(x_1, x_2)$ tale che $\mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}})$ sia minimizzata, assicurando che la KAN fornisca un'accurata approssimazione della funzione ricercata.

4.2.2 Commento al Codice

Inizializzazione della KAN

[1]:
```python
from kan import *
torch.set_default_dtype(torch.float64)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)

model = KAN(width=[2,4,1], grid=3, k=4, seed=19, device=device)
```

```
cpu
checkpoint directory created: ./model
saving model version 0.0
```

In questa cella, viene importata la libreria KAN, che contiene le funzioni necessarie per definire e addestrare le reti neurali di Kolmogorov-Arnold. Viene impostato il tipo di dato predefinito in PyTorch su `float64`, il che significa che tutte le operazioni con tensori utilizzeranno numeri in virgola mobile a 64 bit per una maggiore precisione. Successivamente, viene specificato se il calcolo verrà eseguito sulla GPU (`cuda`) o sulla CPU, a seconda della disponibilità della GPU sul sistema in uso, e il dispositivo selezionato viene stampato a schermo.

Infine, viene istanziata una semplice KAN con la seguente struttura:

- `width=[2,4,1]`: la rete avrà 3 layers; 2 nodi di input, 4 nodi nel livello nascosto e 1 nodo di output.
- `grid=5`: definisce la griglia di interpolazione e fissa il numero di intervalli in cui sarà suddivisa.
- `k=4`: indica il grado dei B-splines utilizzati.
- `seed`: viene impostato per garantire la riproducibilità dei risultati.
- `device=device`: Specifica se il modello deve essere eseguito sulla GPU o sulla CPU.

Questa configurazione prepara la KAN per l'addestramento e l'inferenza successivi.

Creazione del Dataset

[2]:
```
from kan.utils import create_dataset

# create dataset f(x,y) = abs(x-cos(pi*y))
f = lambda x: torch.abs((x[:, [0]]) - torch.cos(x[:, [1]]))
dataset = create_dataset(f, n_var=2, train_num=3000, device=device)
dataset['train_input'].shape, dataset['train_label'].shape
```

[2]:
```
(torch.Size([3000, 2]), torch.Size([3000, 1]))
```

In questa cella, viene creato un dataset che sarà utilizzato per l'addestramento della KAN appena definita. Il dataset è generato a partire dalla seguente funzione target, che combina operazioni trigonometriche e algebriche: `f = lambda x: torch.abs((x[:, [0]]) - torch.cos(x[:, [1]]))`. Questa funzione target rappresenta il comportamento che il modello KAN dovrà apprendere durante il processo di function fitting. In questa funzione:

- `x[:, [0]]` corrisponde alla variabile di input x_1.
- `x[:, [1]]` corrisponde alla variabile di input x_2.

In forma simbolica, la funzione che vogliamo la nostra rete ricostruisca è:

$$f(x_1, x_2) = |x_1 - \cos(x_2)|$$

Essa calcola il valore assoluto della differenza tra la prima variabile di input (x_1) e il coseno della seconda variabile (x_2).

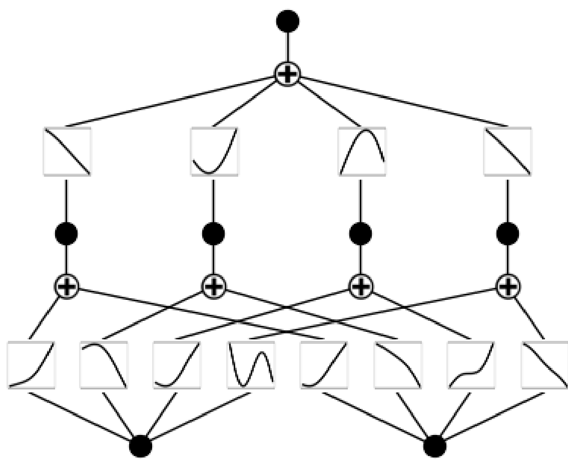
Successivamente, viene utilizzata la funzione `create_dataset` per generare un dataset sintetico basato sulla funzione target `f`. Questo dataset sarà composto da coppie di input e output che il modello utilizzerà per l'addestramento. Come argomenti appaiono:

- `n_var=2`: il numero di variabili di input è 2, corrispondente ai due argomenti della funzione target.
- `train_num=3000`: vengono generati 3000 campioni di addestramento, sufficienti per fornire al modello una varietà di dati da cui apprendere.
- `device=device`: il dataset viene creato sul dispositivo selezionato (GPU o CPU), garantendo che i dati siano già pronti per essere utilizzati nel contesto di addestramento.

Infine, viene verificata la dimensione degli input di addestramento e delle etichette del dataset, assicurandosi che il dataset sia stato creato correttamente e che le dimensioni siano conformi alle aspettative.

Plot della KAN all'Inizializzazione

[3]:
```
model(dataset['train_input']);
model.plot(beta=100)
```



Il modello KAN appena definito viene applicato agli input di addestramento per visualizzare l'architettura di inizializzazione della rete. In particolare:

- `model(dataset['train_input'])`: questo comando passa gli input di addestramento attraverso il modello KAN senza eseguire un vero e proprio addestramento. Tale step serve a preparare la rete per la visualizzazione.
- `model.plot()`: questo comando genera una rappresentazione grafica della struttura della KAN, mostrando i nodi, le connessioni e le operazioni che verranno effettuate durante l'elaborazione degli input. L'argomento `beta=100` serve per specificare la trasparenza degli archi rappresentati, che in questo caso fissiamo essere per tutti uguali al massimo.

Tale immagine di output generata permette dunque di osservare la configurazione della rete prima dell'addestramento, evidenziando come sono disposti i vari nodi ed archi.

Refinement della Grid

[4]:
```
grids = [3, 5, 10, 20]

best_grid = None
best_train_loss = float('inf')
best_test_loss = float('inf')

train_rmse = []
test_rmse = []

for i in range(len(grids)):
    model = model.refine(grids[i])
    results = model.fit(dataset, opt="LBFGS", steps=25,␣
 ↪stop_grid_update_step=30)

    current_train_loss = results['train_loss'][-1].item()
    current_test_loss = results['test_loss'][-1].item()

    train_rmse.append(current_train_loss)
    test_rmse.append(current_test_loss)

# if the current test_loss is the lowest found up till now, update␣
 ↪best_grid
    if current_test_loss < best_test_loss:
        best_grid = grids[i]
        best_train_loss = current_train_loss
        best_test_loss = current_test_loss

# print the best result
print(f"Il miglior valore di grid è: {best_grid}")
print(f"Con train_loss: {best_train_loss:.6f} e test_loss:␣
 ↪{best_test_loss:.6f}")

# apply the best found grid value to the original model
model = model.refine(best_grid)
```

```
saving model version 0.1

| train_loss: 9.44e-03 | test_loss: 9.89e-03 | reg: 1.05e+01 | : 100%|█|␣
 ↪25/25
[00:06<00:00,  3.62it

saving model version 0.2
saving model version 0.3
```

```
| train_loss: 5.38e-03 | test_loss: 6.12e-03 | reg: 1.09e+01 | : 100%|█|␣
↪25/25
[00:07<00:00,  3.22it

saving model version 0.4
saving model version 0.5

| train_loss: 3.27e-03 | test_loss: 4.44e-03 | reg: 1.09e+01 | : 100%|█|␣
↪25/25
[00:09<00:00,  2.58it

saving model version 0.6
saving model version 0.7

| train_loss: 2.16e-03 | test_loss: 3.67e-03 | reg: 1.10e+01 | : 100%|█|␣
↪25/25
[00:14<00:00,  1.72it

saving model version 0.8
Il miglior valore di grid è: 20
Con train_loss: 0.002165 e test_loss: 0.003672
saving model version 0.9
```

In questa cella, l'obiettivo è trovare la finezza ottimale della griglia (`grid`) per la KAN che permette di ottenere il miglior apprendimento, cioè la minore `train_loss` e `test_loss`. Questo processo è fondamentale per garantire che la rete sia capace di generalizzare bene sui dati non visti, minimizzando così l'errore di addestramento e di test.

Il vettore `grids = [3, 5, 10, 20]` contiene diversi valori possibili per la finezza della griglia. Un valore di `grid` più alto rappresenta una griglia più fine, quindi un'eventuale migliore capacità di approssimazione, ma potrebbe anche aumentare la complessità e il rischio di overfitting.

Il ciclo `for` scorre attraverso i valori di `grid` definiti in `grids`. Per ogni valore di `grid`, il modello KAN viene raffinato usando il metodo `model.refine()`, che adatta la struttura della rete alla nuova griglia. Successivamente, il modello viene addestrato sui dati utilizzando l'ottimizzatore `LBFGS`.

Dopo ogni fase di addestramento, si registrano i valori di `train_loss` e `test_loss` per confrontare i risultati ottenuti con diversi valori di griglia. Se il valore corrente di `test_loss` è il più basso finora ottenuto, il valore di `grid` corrispondente viene salvato come `best_grid`, insieme alle perdite associate.

Alla fine del ciclo, viene stampato il miglior valore di `grid` trovato, insieme ai corrispondenti valori di `train_loss` e `test_loss`. Infine, questo miglior valore di `grid` viene applicato definitivamente al modello originale tramite il metodo `refine`, affinché la rete possa essere utilizzata in ulteriori fasi di addestramento o inferenza con la configurazione ottimale.

[5]:
```python
model.fit(dataset, opt="LBFGS", steps=50, lamb=0.001);
```

```
| train_loss: 6.00e-03 | test_loss: 5.80e-03 | reg: 4.64e+00 | : 100%|■|␣
↪50/50
[00:35<00:00,  1.41it
```

```
saving model version 0.10
```

In questa cella viene eseguito il primo addestramento “isolato” del modello KAN (dato che infatti già abbiamo eseguito dei primi training nella cella precedente), utilizzando il valore ottimale di grid trovato appena sopra. Il modello viene addestrato sui dati forniti nel dataset creato in precedenza.

Il comando `model.fit(dataset, opt="LBFGS", steps=50, lamb=0.001);` specifica i dettagli dell’addestramento:

- `opt="LBFGS"`: Viene utilizzato l’ottimizzatore L-BFGS (“Limited-memory Broyden-Fletcher-Goldfarb-Shanno”), un metodo di ottimizzazione quasi-Newtoniano [3] particolarmente efficace per problemi di apprendimento con un numero elevato di parametri. Questo ottimizzatore è noto per la sua capacità di convergere rapidamente verso un minimo locale.
- `steps=50`: L’addestramento viene eseguito per 50 iterazioni, o passi. Durante ogni passo, i pesi del modello vengono aggiornati in modo da ridurre la funzione di perdita.
- `lamb=0.001`: Viene applicato un termine di regolarizzazione L2 [4] con un valore di 0.001. Questo termine penalizza i pesi del modello con valori elevati, aiutando a prevenire l’overfitting e a migliorare la generalizzazione del modello sui dati di test.

L’obiettivo di questa fase è ottimizzare il modello KAN con la configurazione di griglia ottimale, riducendo la funzione di perdita (loss function) il più possibile, e preparando il modello per le fasi successive di validazione e inferenza. Come già visto nella cella precedente, il metodo .fit stampa a schermo le grandezze fondamentali:

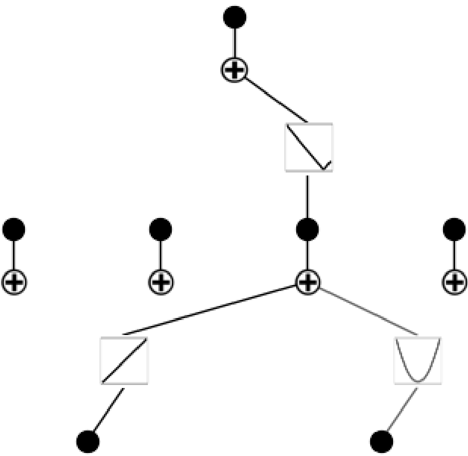
- `train_loss`: rappresenta la perdita calcolata sui dati di addestramento. Questo valore indica quanto bene il modello sta apprendendo dai dati su cui viene addestrato.
- `test_loss`: indica la perdita calcolata sui dati di test, che non sono stati utilizzati durante l’addestramento. Questo valore è un indicatore della capacità del modello di generalizzare su dati non visti. Un valore di `test_loss` vicino al `train_loss` suggerisce che il modello non sta overfittando.
- `reg`: è il termine di regolarizzazione, che serve a penalizzare modelli troppo complessi durante l’addestramento. La regolarizzazione aiuta a prevenire l’overfitting aggiungendo un costo al modello basato sulla complessità, incoraggiando soluzioni più semplici e generalizzabili.

[3] I metodi quasi-Newtoniani sono una classe di algoritmi di ottimizzazione che utilizzano un’approssimazione della matrice Hessiana, riducendo la complessità computazionale rispetto ai metodi Newtoniani pur mantenendo una rapida convergenza.

[4] La regolarizzazione L2 è una tecnica che aggiunge un termine alla funzione di perdita proporzionale alla somma dei quadrati dei pesi del modello, con l’obiettivo di mantenere i pesi piccoli e limitare la complessità del modello.

Plot della KAN Addestrata

[6]:
```
model.plot()
```

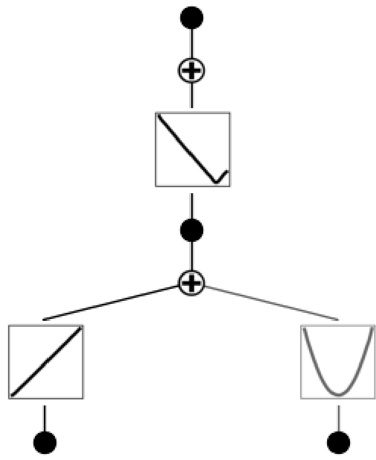


A questo punto richiamiamo il metodo `.plot()` senza fornirgli nessun argomento: in questo modo, di default, il programma disegna gli archi con una trasparenza proporzionale alla loro "robustezza" a seguito del training appena fatto. Come si vede dall'immagine generata, solamente due archi sono visibili dopo questo addestramento.

Pruning della KAN

[7]:
```
model = model.prune()
model.plot()
```

```
saving model version 0.11
```



In questa cella viene eseguita un'operazione di "pruning" sulla rete KAN tramite il comando `model.prune()`. Il pruning è una tecnica utilizzata per ridurre la complessità del modello, eliminando i nodi e i collegamenti che hanno un impatto minimo sull'output fi-

nale. Questo processo può portare a una rete più semplice e interpretabile, migliorando al contempo l'efficienza computazionale del modello senza compromettere significativamente le prestazioni.

Successivamente, il modello viene visualizzato nuovamente con `model.plot()`, mostrando la struttura semplificata della rete KAN dopo l'operazione di pruning.

Ulteriori Training e Plotting

[8]:
```
model.fit(dataset, opt="LBFGS", steps=50);
```

```
| train_loss: 3.39e-03 | test_loss: 3.24e-03 | reg: 5.86e+00 | : 100%|■|␣
↪50/50
[00:11<00:00,  4.28it

saving model version 0.12
```

Ulteriore training con gli stessi parametri di prima: da notare però come a seguito del pruning si raggiunga un addestramento migliore della rete: come già accennato, infatti, eseguendo il pruning sulla rete si ottiene un'architettura più snella, il che permette miglior training e rappresentazione dei dati.

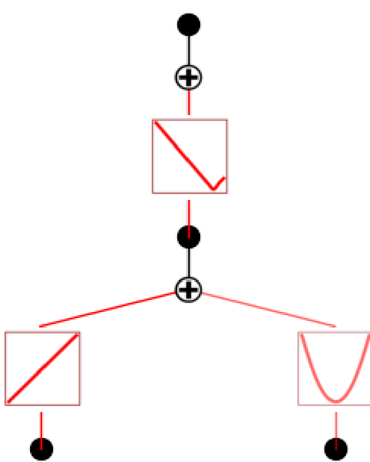
Riscrittura delle Funzioni di Attivazione con Formule Simboliche

[9]:
```
mode = "auto" # "manual"

if mode == "manual":
    # manual mode
    model.fix_symbolic(0,0,0,'x')
    model.fix_symbolic(0,1,0,'cos')
    model.fix_symbolic(1,0,0,'abs')
elif mode == "auto":
    # automatic mode
    lib =␣
↪['x','x^2','x^3','x^4','exp','log','sqrt','tanh','sin','cos','abs']
    model.auto_symbolic(lib=lib)

model.plot()
```

```
fixing (0,0,0) with x, r2=0.9999988786671576, c=1
fixing (0,1,0) with cos, r2=0.9999701260437224, c=2
fixing (1,0,0) with abs, r2=0.9999696147038063, c=3
saving model version 0.13
```



Implementazione del processo di assegnazione delle funzioni di attivazione simboliche sugli archi della rete KAN. È possibile scegliere tra due modalità: manuale o automatica:

- Se la modalità è impostata su `manual`, vengono specificate esplicitamente dall'utente le funzioni di attivazione per determinati archi utilizzando il metodo `fix_symbolic`. Ad esempio, `model.fix_symbolic(0,0,0,'x')` associa la funzione `x` al nodo specificato[5].
- Se la modalità è impostata su `auto`, viene utilizzato il metodo `auto_symbolic`, che sceglie automaticamente le funzioni di attivazione più appropriate da una libreria di funzioni predefinite. La libreria include funzioni algebriche e trigonometriche come x, x^2, *exp*, *log*, *sqrt*, *tanh*, *sin*, *cos*, *abs* e altre.

Dopo aver eseguito la cella, viene stampato quali funzioni sono state associate a ciascun nodo, insieme al coefficiente di correlazione (`r2`) che misura quanto bene la funzione simbolica approssima il comportamento effettivo del nodo. Infine si richiama il metodo plot, ottenendo uno schema della rete in cui si può vedere come il programma di default disegni di rosso gli archi le cui funzioni di attivazione associate sono state fissate.

Training fino alla Machine Precision

[10]:
```
model.fit(dataset, opt="LBFGS", steps=50);
```

```
| train_loss: 4.93e-14 | test_loss: 2.88e-15 | reg: 0.00e+00 | : 100%|■|␣
↪50/50
[00:03<00:00, 16.31it

saving model version 0.14
```

[5]Ogni funzione di attivazione è indicizzata da (l, i, j) dove l è l'indice del layer, i è l'indice del neurone di input, e j è l'indice del neurone di output. Tutti gli indici partono da 0. Ad esempio, la funzione associata al nodo in basso a sinistra nella nostra rete è $(0, 0, 0)$.

In questa cella viene eseguito l'ultimo addestramento del modello KAN, utilizzando i parametri ottimali determinati in precedenza e dopo aver associato le funzioni di attivazione simboliche agli archi della rete.

L'obiettivo di questo addestramento è raggiungere la "machine precision": si tratta del limite numerico oltre il quale un computer non è più in grado di distinguere tra due numeri distinti. In termini di precisione, su una macchina con numeri a 64-bit (float64), la precisione tipica è dell'ordine di $\epsilon \approx 2.22 \times 10^{-16}$. Quando un modello raggiunge questa precisione durante l'addestramento, significa che ulteriori miglioramenti della funzione di perdita (*loss function*) sono numericamente insignificanti, poiché si è raggiunto il limite delle capacità di calcolo della macchina.

L'associazione simbolica delle funzioni di attivazione alle varie componenti della rete, effettuata nella cella precedente, gioca un ruolo cruciale nel raggiungere questo livello di precisione. Assegnando funzioni simboliche ottimali, il modello è in grado di rappresentare più fedelmente la funzione target, riducendo così gli errori di approssimazione e migliorando la capacità del modello di convergere rapidamente a un minimo locale durante l'addestramento. Questa precisione nella rappresentazione simbolica contribuisce in modo significativo al raggiungimento di una loss estremamente bassa, come evidenziato dai risultati finali stampati a schermo al termine del processo.

Stampa della Formula Simbolica

[11]:
```
from kan.utils import ex_round
from sympy import simplify, init_printing, symbols, Abs, cos

init_printing()

symbolic_formula = ex_round(model.symbolic_formula()[0][0], 1)
simplified_formula = simplify(symbolic_formula)

simplified_formula
```

[11]:

$1.0\,|x_1 - \cos(1.0x_2)|$

Nell'ultima cella viene estratta la formula simbolica appresa dal modello KAN, oltre ad essere semplificata per ottenere una rappresentazione più leggibile e comprensibile.

La formula è stata ottenuta con il comando `ex_round()`, con cui si specifica il numero di cifre decimali dei coefficienti che si vogliono stampare, e successivamente semplificata con `simplify(symbolic_formula)`. Viene utilizzata la funzione `simplify` per ridurre la complessità dell'espressione simbolica ottenuta inizialmente dal modello. Questa operazione è necessaria in questo caso specifico perché la linearità della funzione più esterna della nostra funzione target, ovvero il valore assoluto, ha permesso all'algoritmo di aggiungere coefficienti che in realtà non compaiono nella funzione target. La semplificazione rimuove questi coefficienti superflui, fornendo una formula finale più accurata e rappresentativa della funzione target originale.

4.3 PDEs Resolution

4.3.1 Introduzione al Problema

L'obiettivo di questo codice è determinare numericamente la soluzione di un'equazione di Poisson bidimensionale. L'equazione differenziale parziale (PDE) da risolvere è: $\nabla^2 f(x,y) = 2\pi^2 \sin(\pi x)\sin(\pi y)$, con le condizioni al contorno: $f(-1,y) = f(1,y) = f(x,-1) = f(x,1) = 0$. La soluzione esatta, o "ground truth", di questa equazione è nota ed è data da: $f(x,y) = -\sin(\pi x)\sin(\pi y)$.

Il codice implementa una rete neurale di Kolmogorov-Arnold per approssimare la soluzione di questa PDE. La rete neurale viene addestrata minimizzando una funzione di perdita che combina due termini principali:

- "Interior Loss"(PDE Loss): misura quanto bene la soluzione proposta dalla rete soddisfa l'equazione di Poisson all'interno del dominio.
- "Boundary Loss"(BC Loss): misura la differenza tra la soluzione proposta dalla rete e le condizioni al contorno note.

L'approccio utilizzato in questo contesto è basato sulle "Physics-Informed Neural Networks"(PINNs), un metodo per risolvere le equazioni differenziali parziali (PDEs) combinando i principi delle reti neurali con le leggi fisiche che governano il problema studiato ([33]). A differenza delle reti neurali tradizionali, che vengono addestrate utilizzando dati etichettati, le PINNs utilizzano l'equazione differenziale stessa e le condizioni al contorno come vincoli nel processo di addestramento.

Input e Output del Modello

Nelle PINNs, il modello riceve come input le coordinate spaziali (x,y) all'interno del dominio di interesse. L'output della rete neurale è una funzione scalare $f(x,y)$ che rappresenta la soluzione approssimata della PDE in quei punti. L'obiettivo è addestrare la rete affinché l'output $f(x,y)$ soddisfi l'equazione differenziale e le condizioni al contorno imposte.

Funzione di Perdita e Target

La funzione di perdita nelle PINNs è composta da due termini principali:

1. *PDE Loss* (Interior Loss): Questo termine misura quanto l'output della rete $f(x,y)$ soddisfa l'equazione differenziale. Per calcolare questo termine, le derivate dell'output rispetto agli input spaziali (x,y) vengono determinate attraverso la "differenziazione automatica": questo processo calcola le derivate prime e seconde risalendo attraverso tutte le trasformazioni della rete neurale a partire dagli input ([34]). Ad esempio, la derivata seconda f_{xx} rispetto a x è ottenuta in modo automatico e confrontata con il termine sorgente noto della PDE. La PDE Loss rappresenta quindi la differenza quadratica media tra il laplaciano $\nabla^2 f(x,y)$ calcolato dalla rete e il termine sorgente $\nabla^2 f(x,y) = 2\pi^2 \sin(\pi x)\sin(\pi y)$.

2. *Boundary Loss* (BC Loss): Questo termine confronta l'output della rete nei punti di confine con i valori esatti imposti dalle condizioni al contorno. Se le condizioni richiedono che $f(x, y)$ sia zero sui bordi del dominio, la Boundary Loss penalizza il modello quando l'output non rispetta questa condizione, vincolando così la rete a rispettare le condizioni al contorno della PDE.

Addestramento del Modello

L'addestramento del modello avviene minimizzando la funzione di perdita totale, che, come approfondiremo nei prossimi commenti, è una combinazione pesata della *PDE Loss* e della *Boundary Loss*. Durante l'addestramento, l'algoritmo di ottimizzazione aggiorna iterativamente i parametri della rete per ridurre questa perdita, facendo in modo che il modello apprenda una soluzione che sia coerente sia con l'equazione differenziale che con le condizioni al contorno imposte. Questo approccio consente al modello di convergere verso la soluzione esatta della PDE, garantendo che tutte le condizioni fisiche imposte siano rispettate.

4.3.2 Commento al Codice

Definizione della KAN, del Dataset, e del Training

[1]:
```
# libraries
from kan import *
import matplotlib.pyplot as plt
from torch import autograd
from tqdm import tqdm

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)

# define dimensions and KAN
dim = 2
np_i = 21 # number of interior points (along each dimension)
np_b = 21 # number of boundary points (along each dimension)
ranges = [-1, 1]

model = KAN(width=[2,2,1], grid=5, k=5, seed=19, device=device)

# define jacobian
def batch_jacobian(func, x, create_graph=False):
    # x in shape (Batch, Length)
    def _func_sum(x):
        return func(x).sum(dim=0)
    return autograd.functional.jacobian(_func_sum, x,␣
 ↪create_graph=create_graph).permute(1,0,2)
```

```
# define solution
sol_fun = lambda x: -torch.sin(torch.pi*x[:,[0]])*torch.sin(torch.pi*x[:
↪,[1]])
source_fun = lambda x: 2*torch.pi**2 * torch.sin(torch.pi*x[:
↪,[0]])*torch.sin(torch.pi*x[:,[1]])

# interior
sampling_mode = 'random' # 'random' or 'mesh'

x_mesh = torch.linspace(ranges[0],ranges[1],steps=np_i)
y_mesh = torch.linspace(ranges[0],ranges[1],steps=np_i)
X, Y = torch.meshgrid(x_mesh, y_mesh, indexing="ij")
if sampling_mode == 'mesh':
    #mesh
    x_i = torch.stack([X.reshape(-1,), Y.reshape(-1,)]).permute(1,0)
else:
    #random
    x_i = torch.rand((np_i**2,2))*2-1

x_i = x_i.to(device)

# boundary, 4 sides
helper = lambda X, Y: torch.stack([X.reshape(-1,), Y.reshape(-1,)]).
↪permute(1,0)
xb1 = helper(X[0], Y[0])
xb2 = helper(X[-1], Y[0])
xb3 = helper(X[:,0], Y[:,0])
xb4 = helper(X[:,0], Y[:,-1])
x_b = torch.cat([xb1, xb2, xb3, xb4], dim=0)

x_b = x_b.to(device)

steps = 20
alpha = 0.01
log = 1

# training
def train():
    optimizer = LBFGS(model.parameters(), lr=1, history_size=10,␣
↪line_search_fn="strong_wolfe", tolerance_grad=1e-32,␣
↪tolerance_change=1e-32, tolerance_ys=1e-32)

    pbar = tqdm(range(steps), desc='description', ncols=100)
```

```
    for _ in pbar:
        def closure():
            global pde_loss, bc_loss
            optimizer.zero_grad()
            # interior loss
            sol = sol_fun(x_i)
            sol_D1_fun = lambda x: batch_jacobian(model, x,␣
↪create_graph=True)[:,0,:]
            sol_D1 = sol_D1_fun(x_i)
            sol_D2 = batch_jacobian(sol_D1_fun, x_i, create_graph=True)[:
↪,:,:]
            lap = torch.sum(torch.diagonal(sol_D2, dim1=1, dim2=2),␣
↪dim=1, keepdim=True)
            source = source_fun(x_i)
            pde_loss = torch.mean((lap - source)**2)

            # boundary loss
            bc_true = sol_fun(x_b)
            bc_pred = model(x_b)
            bc_loss = torch.mean((bc_pred-bc_true)**2)

            loss = alpha * pde_loss + bc_loss
            loss.backward()
            return loss

        if _ % 5 == 0 and _ < 50:
            model.update_grid_from_samples(x_i)

        optimizer.step(closure)
        sol = sol_fun(x_i)
        loss = alpha * pde_loss + bc_loss
        l2 = torch.mean((model(x_i) - sol)**2)

        if _ % log == 0:
            pbar.set_description("pde loss: %.2e | bc loss: %.2e | l2: %.
↪2e " % (pde_loss.cpu().detach().numpy(), bc_loss.cpu().detach().
↪numpy(), l2.cpu().detach().numpy()))

train()
```

```
cpu
checkpoint directory created: ./model
saving model version 0.0
```

```
pde loss: 7.91e-01 | bc loss: 4.49e-04 | l2: 1.68e-03 : 100%|███████|␣
↪20/20
[00:20<00:00,  1.02s/it]
```

 # libraries

Vengono importate le librerie necessarie per il progetto: la libreria `kan` contiene le implementazioni delle KAN, `matplotlib` serve per la visualizzazione dei risultati, `torch.autograd` per il calcolo delle derivate, e `tqdm` per creare barre di avanzamento visibili durante l'addestramento.

Viene anche definito il dispositivo utilizzato per il calcolo (`device`), che può essere la GPU (`cuda`) o la CPU, a seconda della disponibilità.

 # define dimensions and KAN

Successivamente vengono definite le dimensioni del problema e il numero di punti interni e di contorno per la discretizzazione del dominio.

- `dim = 2`: definisce che il problema è bidimensionale (2D).
- `np_i = 21`: numero di punti interni lungo ciascuna dimensione.
- `np_b = 21`: numero di punti di contorno lungo ciascuna dimensione.
- `ranges = [-1, 1]`: intervallo spaziale per le variabili x e y.

Contestualmente viene inizializzata una rete KAN (`model = KAN(width=[2,2,1], grid=5, k=5, seed=19, device=device)`),con 2 nodi di input, 2 nodi nel layer nascosto e 1 nodo di output (dunque 3 layers in tutto).

 # define jacobian

A questo punto, viene definita la funzione `batch_jacobian`, che calcola il jacobiano della funzione data `func` rispetto alle variabili indicate con `x`. Si tratta di una funzione ausiliaria che sarà utilizzata per calcolare le derivate necessarie nella formulazione della PDE.

 # define solution

Fatto ciò vengono poi definite le funzioni `sol_fun` e `source_fun`. `sol_fun` definisce la soluzione esatta $f(x,y) = \sin(\pi x)\sin(\pi y)$, che verrà confrontata con l'output della rete KAN per calcolare l'errore durante l'addestramento. `source_fun` rappresenta, invece, il termine sorgente dell'equazione, $-2\pi^2 \sin(\pi x)\sin(\pi y)$, che viene utilizzato per calcolare la *PDE Loss*. Entrambe queste funzioni sono essenziali per guidare l'addestramento della rete verso la corretta soluzione dell'equazione di Poisson.

 # interior

In questa sezione viene generato un insieme di punti interni al dominio, necessari per studiare la PDE. I punti possono essere campionati in due modalità: `random` o `mesh`. Nella modalità `mesh`, i punti sono disposti regolarmente su una griglia, mentre nella modalità `random` vengono generati in modo casuale all'interno del dominio $[-1,1] \times [-1,1]$. La scelta della modalità avviene tramite la variabile `sampling_mode`, e i punti generati vengono successivamente trasferiti sul dispositivo (`device`) per l'elaborazione.

boundary, 4 sides

In questa sezione vengono generati i punti che rappresentano i quattro lati del dominio per soddisfare le condizioni al contorno della PDE. La funzione `helper` trasforma le coordinate dei bordi in un formato utilizzabile, creando i punti `xb1`, `xb2`, `xb3`, e `xb4` per ciascun lato del dominio. Questi punti vengono poi concatenati in `x_b`, una tensor di dimensione `(4*np_b, 2)` che rappresenta tutti i punti di confine. Il tensor viene poi trasferito al dispositivo (`device`) per il calcolo.

Infine, vengono definite le variabili `steps`, `alpha`, e `log`. La variabile `steps` controlla il numero di iterazioni di addestramento che la rete eseguirà. La variabile `alpha` è un iperparametro che bilancia il peso tra il termine di perdita della PDE (*PDE Loss*) e il termine di perdita delle condizioni al contorno (*Boundary Loss*); un valore più alto di `alpha` dà maggiore importanza alla soluzione della PDE rispetto alle condizioni al contorno. La variabile `log` determina infine la frequenza con cui vengono stampate le informazioni di log durante l'addestramento, che includono dettagli come la perdita attuale della PDE e delle condizioni al contorno, e il livello di errore rispetto alla soluzione esatta.

training

La funzione `train()` gestisce il processo di addestramento del modello KAN. Viene inizializzato un ottimizzatore `LBFGS`, adatto per problemi di ottimizzazione non lineare come questo, con specifici parametri che controllano la convergenza, tra cui:

- `model.parameters()`: passa i parametri del modello KAN all'ottimizzatore, specificando quali variabili devono essere aggiornate durante l'addestramento.
- `lr=1`: specifica il learning rate.
- `history_size=10`: determina quante iterazioni passate devono essere memorizzate dall'ottimizzatore per valutare la direzione della discesa nella minimizzazione.
- `tolerance_grad=1e-32`, `tolerance_change=1e-32`, `tolerance_ys=1e-32`: questi parametri definiscono le tolleranze per la convergenza; `tolerance_grad` stabilisce la soglia per la norma del gradiente, `tolerance_change` la soglia per i cambiamenti nei parametri, e `tolerance_ys` per i cambiamenti nel passo della discesa. Valori estremamente piccoli come `1e-32` indicano che l'ottimizzazione deve essere molto precisa prima di terminare.

Il modello aggiorna la griglia di campionamento ogni 5 iterazioni con `model.update_grid_from_samples(x_i)` per migliorare l'efficienza dell'apprendimento, adattando la griglia ai dati campionati. Infine, la barra di avanzamento (`tqdm`) fornisce un feedback visivo del progresso, mostrando il numero di iterazioni (`steps`) e le perdite attuali (`pde loss`, `bc loss`, e `l2`), permettendo di monitorare il processo di addestramento in tempo reale.

interior loss

All'interno del ciclo `for`, viene definita la funzione `closure()`, che calcola la perdita interna (*PDE Loss*). Per calcolare quest'ultima, viene innanzitutto determinata la soluzione prevista dalla rete per i punti interni `x_i` utilizzando `sol_fun(x_i)`.

Successivamente, la funzione `batch_jacobian()` viene impiegata per calcolare le derivate prime della soluzione rispetto alle coordinate, e quindi le derivate seconde. Il laplaciano della soluzione è ottenuto sommando queste derivate seconde lungo le direzioni principali.

Il termine sorgente `source_fun(x_i)` rappresenta la parte destra dell'equazione di Poisson, con cui il laplaciano viene confrontato. La *PDE Loss* è quindi calcolata come la media dell'errore quadratico tra il laplaciano calcolato dalla rete e il termine sorgente, esprimendo quanto la soluzione della rete si discosta dalla soluzione esatta imposta dall'equazione differenziale.

 # boundary loss

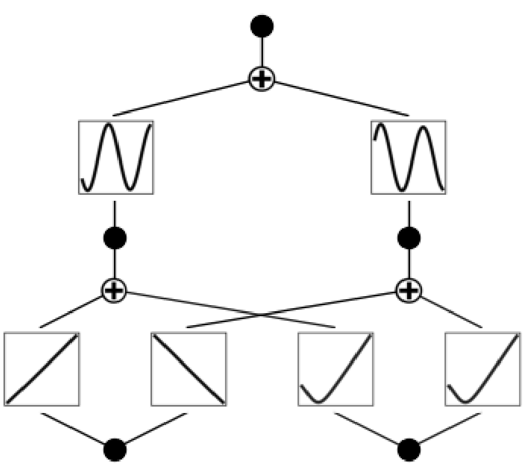
Oltre alla *PDE Loss*, viene calcolata anche la perdita delle condizioni al contorno (*Boundary Loss*). Questa perdita misura quanto la soluzione prevista dalla rete sui punti di contorno `x_b` si discosta dalla soluzione esatta `sol_fun(x_b)`, che rappresenta i valori corretti che la soluzione deve assumere lungo i bordi del dominio.

Il processo avviene confrontando l'output del modello `model(x_b)` con i valori esatti di riferimento, e calcolando l'errore quadratico medio tra questi due set di valori. Questa *Boundary Loss* viene combinata con la *PDE Loss* calcolata in precedenza per formare una funzione di perdita totale `loss = alpha * pde_loss + bc_loss`, dove `alpha` bilancia l'importanza tra le due perdite. Questa funzione di perdita totale viene quindi utilizzata per aggiornare i pesi della rete tramite backpropagation.

Infine, la barra di avanzamento (`tqdm`) viene aggiornata regolarmente con i valori attuali delle perdite (`pde loss`, `bc loss`, `l2`).

Plotting della KAN Addestrata

[2]:
```
model.plot()
```



Stampa di Suggerimenti per le Funzioni di Attivazione del Primo Strato

[3]:
```
lib = ['x','x^2','x^3','x^4','exp','log','sqrt','tanh','sin','tan','abs']

model.suggest_symbolic(0,0,0, lib=lib, topk=3, weight_simple=1.0);
model.suggest_symbolic(0,0,1, lib=lib, topk=3, weight_simple=1.0);

model.suggest_symbolic(0,1,0, lib=lib, topk=3, weight_simple=1.0);
model.suggest_symbolic(0,1,1, lib=lib, topk=3, weight_simple=1.0);
```

```
  function  fitting r2    r2 loss  complexity  complexity loss  total loss
0        x    0.999830 -12.436436           1                1         1.0
1      x^2    0.999996 -16.155940           2                2         2.0
2      exp    0.999996 -16.131041           2                2         2.0
  function  fitting r2    r2 loss  complexity  complexity loss  total loss
0        x    0.999957 -14.191402           1                1         1.0
1      x^2    0.999992 -15.762684           2                2         2.0
2      exp    0.999991 -15.720287           2                2         2.0
  function  fitting r2   r2 loss  complexity  complexity loss  total loss
0        x    0.822581 -2.494690           1                1         1.0
1      x^2    0.967146 -4.927331           2                2         2.0
2      exp    0.932774 -3.894628           2                2         2.0
  function  fitting r2   r2 loss  complexity  complexity loss  total loss
0        x    0.822338 -2.492709           1                1         1.0
1      x^2    0.966700 -4.907891           2                2         2.0
2      exp    0.931971 -3.877495           2                2         2.0
```

Il metodo `suggest_symbolic` viene utilizzato per ottenere le formule simboliche che meglio approssimano le funzioni di attivazione sugli archi della rete KAN: questi sono specificati dai tre indici passati come primi argomenti al metodo. Le formule vengono selezionate dalla libreria qui definita `lib`, che include funzioni polinomiali, esponenziali, logaritmiche e trigonometriche.

L'argomento `topk=3` specifica che devono essere restituite le tre migliori formule per ogni arco, mentre `weight_simple=1.0` è un iperparametro favorisce formule più semplici, riducendo la complessità delle espressioni proposte. Questo approccio consente al modello di suggerire rappresentazioni simboliche che bilanciano sia l'accuratezza della rappresentazione (r2) sia la semplicità strutturale. L'output include una valutazione delle formule suggerite in termini di accuratezza, complessità, e una perdita totale combinata, permettendo una scelta informata delle funzioni di attivazione più appropriate.

Fix delle Funzioni di Attivazione del Primo Strato a Funzioni Lineari

[4]:
```
model.fix_symbolic(0,0,0,'x')
model.fix_symbolic(0,0,1,'x')
model.fix_symbolic(0,1,0,'x')
```

```
model.fix_symbolic(0,1,1,'x')
```

```
r2 is 0.9998295903205872
saving model version 0.1
r2 is 0.9999565482139587
saving model version 0.2
r2 is 0.82258141040802
r2 is not very high, please double check if you are choosing the correct
symbolic function.
saving model version 0.3
r2 is 0.8223376870155334
r2 is not very high, please double check if you are choosing the correct
symbolic function.
saving model version 0.4
```

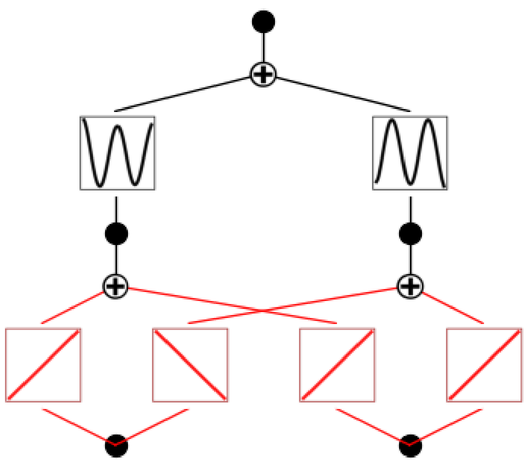
[4]:
```
tensor(0.8223)
```

In questa cella, viene utilizzato il metodo `fix_symbolic` per fissare le funzioni di attivazione sugli archi specificati: i tre indici passati come primi argomenti al metodo identificano, come già visto, tali archi, mentre il quarto argomento `'x'` indica che la funzione di attivazione che si vuole fissare su questi archi sarà la funzione identità, rappresentata dalla variabile `x`; nella cella precedente si è infatti ottenuta questa funzione come primo suggerimento. Questo procedimento blocca le funzioni di attivazione interessate, impedendo ulteriori modifiche durante gli addestramenti successivi.

[5]:
```
train()
```

```
pde loss: 1.24e+00 | bc loss: 1.34e-03 | l2: 1.30e-03 : 100%|███████|␣
↪20/20
[00:20<00:00,  1.01s/it]
```

[6]:
```
model.plot()
```



Stampa di Suggerimenti per le Funzioni di Attivazione del Secondo Strato

[7]:
```
lib = ['x','x^2','x^3','x^4','exp','log','sqrt','tanh','sin','tan','abs']

model.suggest_symbolic(1,0,0, lib=lib, topk=3, weight_simple=0.2);
model.suggest_symbolic(1,1,0, lib=lib, topk=3, weight_simple=0.2);
```

```
  function  fitting r2   r2 loss  complexity  complexity loss  total loss
0      sin    0.758869 -2.052054           2                2   -1.241643
1        x    0.001325 -0.001898           1                1    0.198482
2      abs    0.263978 -0.442159           3                3    0.246273
  function  fitting r2   r2 loss  complexity  complexity loss  total loss
0      sin    0.290982 -0.496086           2                2    0.003131
1      abs    0.391909 -0.717618           3                3    0.025906
2        x    0.000372 -0.000522           1                1    0.199583
```

Fix delle Funzioni di Attivazione del Secondo Strato a Funzioni Sinusoidali

[8]:
```
model.fix_symbolic(1,0,0,'sin')
model.fix_symbolic(1,1,0,'sin')
```

```
Best value at boundary.
r2 is 0.7588694095611572
r2 is not very high, please double check if you are choosing the correct
symbolic function.
saving model version 0.5
Best value at boundary.
r2 is 0.2909822165966034
r2 is not very high, please double check if you are choosing the correct
symbolic function.
saving model version 0.6
```

[8]:
```
tensor(0.2910)
```

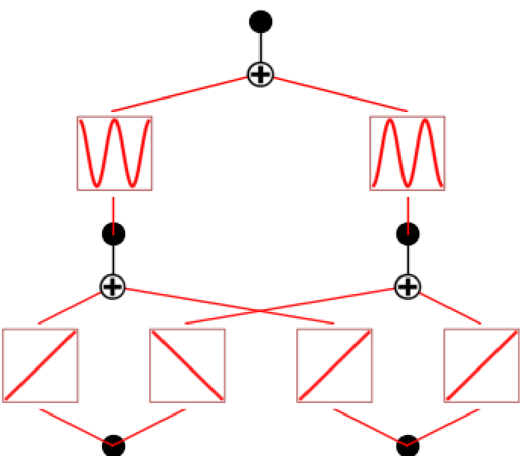
In queste due ultime celle si è ripetuto quanto fatto per le funzioni di attivazione del primo layer: in questo caso le due funzioni del secondo layer sono state fissate, come suggerito dal metodo `suggest_symbolic` , alla funzione seno (`sin`).

Ultimi Training e Plotting della Rete

[9]:
```
train()
```

```
pde loss: 1.15e-11 | bc loss: 2.95e-14 | l2: 3.09e-14 : 100%|████████|␣
↪20/20
[00:08<00:00,  2.39it/s]
```

[10]:
```
model.plot()
```



Da notare come in quest'ultima illustrazione tutti gli archi della rete sono in rosso, dato che abbiamo fissato tutte le funzioni di attivazione come suggerito dal metodo `suggest_symbolic`. Inoltre, dall'output stampato a schermo durante il training, notiamo come con l'ultimo training anche per questa task raggiungiamo valori di accuratezza prossimi alla *machine precision*, il che è ovviamente buona indicazione dell'efficacia del modello.

Stampa della Formula Simbolica per la Soluzione

[11]:
```
formula = model.symbolic_formula()[0][0]
ex_round(formula,3)
```

[11]:

$$-0.5\sin\left(-3.142x_1 + 3.142x_2 + 1.571\right) - 0.5\sin\left(3.142x_1 + 3.142x_2 - 1.571\right)$$

Per concludere, si procede ad estrarre la formula simbolica ottenuta dalla nostra KAN per la soluzione della PDE in esame, attraverso gli stessi metodi già discussi nell'esempio implementativo del Function Fitting (sezione 4.2).

La formula originale a cui è giunto il nostro modello è dunque:

$$f(x_1, x_2) = -0.5\sin(-3.142x_1 + 3.142x_2 + 1.571) - 0.5\sin(3.142x_1 + 3.142x_2 - 1.571)$$

Tuttavia, occorre ora svolgere una serie di passaggi algebrici di natura trigonometrica per ricondursi da tale espressione alla soluzione nella forma standard che abbiamo riportato all'inizio del notebook.

Come primo passo, sostituiamo i valori numerici con le corrispondenti frazioni di π: $3.142 \approx \pi$, $1.571 \approx \frac{\pi}{2}$. Quindi, l'espressione diventa:

$$f(x_1, x_2) = -0.5\sin(-\pi x_1 + \pi x_2 + \frac{\pi}{2}) - 0.5\sin(\pi x_1 + \pi x_2 - \frac{\pi}{2})$$

Sfruttiamo ora l'identità trigonometrica: $\sin(\theta \pm \frac{\pi}{2}) = \pm\cos(\theta)$, applicandola a ciascun termine:

- Per il primo termine: $\sin(-\pi x_1 + \pi x_2 + \frac{\pi}{2}) = \cos(-\pi x_1 + \pi x_2) = \cos(\pi x_2 - \pi x_1) = \cos(\pi(x_2 - x_1))$
- Per il secondo termine: $\sin(\pi x_1 + \pi x_2 - \frac{\pi}{2}) = -\cos(\pi x_1 + \pi x_2)$

Quindi, la funzione diventa:

$$f(x_1, x_2) = -0.5\cos(\pi(x_2 - x_1)) + 0.5\cos(\pi(x_1 + x_2))$$

Utilizziamo quindi l'identità trigonometrica per trasformare la differenza tra coseni: $\cos A - \cos B = -2\sin\left(\frac{A+B}{2}\right)\sin\left(\frac{A-B}{2}\right)$, dove $A = \pi(x_2 - x_1)$ e $B = \pi(x_1 + x_2)$. Si ottiene:

$$f(x_1, x_2) = -0.5\left[-2\sin\left(\frac{\pi(x_2 - x_1) + \pi(x_1 + x_2)}{2}\right)\sin\left(\frac{\pi(x_2 - x_1) - \pi(x_1 + x_2)}{2}\right)\right]$$

E, semplificando gli argomenti:

$$\frac{\pi(x_2 - x_1) + \pi(x_1 + x_2)}{2} = \frac{\pi x_2 - \pi x_1 + \pi x_1 + \pi x_2}{2} = \pi x_2$$

$$\frac{\pi(x_2 - x_1) - \pi(x_1 + x_2)}{2} = \frac{\pi x_2 - \pi x_1 - \pi x_1 - \pi x_2}{2} = -\pi x_1$$

l'espressione diventa: $f(x_1, x_2) = -\sin(\pi x_2)\sin(\pi x_1)$

Possiamo, pertanto, riscrivere semplicemente la funzione come:

$$f(x, y) = -\sin(\pi x)\sin(\pi y)$$

Dunque, attraverso una serie di banali passaggi algebrici, si è verificato che la rete KAN addestrata nel presente notebook approssima correttamente la soluzione della PDE studiata con le condizioni al contorno specificate.

Conclusioni

In questo lavoro ci si è proposti di fornire un'analisi sufficientemente approfondita delle reti neurali di Kolmogorov-Arnold, esaminandone il potenziale rispetto alle reti neurali multistrato nel contesto dell'odierna intelligenza artificiale. Attraverso una rassegna critica della letteratura e l'analisi di implementazioni pratiche, si è cercato di mettere in luce i miglioramenti che questa nuova architettura potrebbe apportare nel panorama del *machine learning*.

Uno degli obiettivi principali della tesi è stato quello di evidenziare la profondità e la rilevanza del teorema di Kolmogorov-Arnold, non solo nel contesto delle KANs, ma anche nella sua portata in senso lato. Si è infatti cercato di evidenziare come il KART non solo offra una solida base teorica per le KANs, ma anche come esso rappresenti un contributo significativo nella matematica e nelle scienze in generale. La sua capacità di rappresentare funzioni multivariate complesse attraverso la decomposizione in funzioni univariate, oltre ad aver finalmente trovato fertile applicazione come nuovo possibile fondamento dell'apprendimento automatico, lo eleva a risultato teorico dalle notevolissime implicazioni sia concettuali, che applicative, in svariati ambiti. Parallelamente, si è voluto esplorare e discutere le principali difficoltà che affliggono le MLPs, come la *curse of dimensionality*, la *catastrophic forgetting* e la scarsa interpretabilità. Si è evidenziato come le KANs possano affrontare queste problematiche attraverso una struttura più flessibile e interpretabile, proponendosi come una possibile soluzione ai limiti che ancora caratterizzano le architetture standard. L'analisi fatta ha mostrato come, almeno in linea di principio, le KANs possano migliorare l'efficienza e la capacità di generalizzazione in contesti dove le MLPs mostrano particolarmente tutte le loro limitazioni.

Implicazioni e Contesto Attuale

Le implicazioni di quanto trattato sono significative, non solo dal punto di vista teorico, ma anche per le potenziali applicazioni pratiche. Le reti di Kolmogorov-Arnold, almeno inizialmente, sembravano promettere soluzioni innovative ai problemi più ostici dell'AI, suscitando un enorme entusiasmo nella comunità scientifica. Tuttavia, è importante notare che questo lavoro è stato realizzato pochi mesi dopo la pubblicazione del primo articolo ([2]), già citato diverse volte, epicentro del periodo di grande fervore e aspettative attorno a questa nuova tecnologia. Da allora, la situazione è evoluta rapidamente. Nel

tempo intercorso, la comunità scientifica ha iniziato a esaminare più criticamente le KANs, e nuove risorse e articoli disponibili sul web hanno analizzato queste reti in maniera più critica e dettagliata, mostrando che queste reti non sono necessariamente superiori alle MLPs in tutti i contesti inizialmente avanzati, come si credeva, o forse meglio, sperava, in un primo momento. Questo appunto non vuole sminuire l'importanza del KART o il potenziale delle KANs, ma sottolineare la necessità di ulteriori ricerche e un'analisi più approfondita per valutare con precisione in quali ambiti le KANs possano davvero eccellere rispetto alle MLPs. La comprensione di queste dinamiche in continua evoluzione resta pertanto essenziale.

Limitazioni e Pretese di questo Lavoro

Il presente lavoro non ha avuto altre pretese se non quella di provare ad introdurre la nuova architettura delle KANs, analizzandole principalmente dal punto di vista teorico. Il capitolo 4, dove è stato implementato del codice, rappresenta l'unica eccezione a questo approccio prevalentemente teorico. L'intento è stato quello di offrire una comprensione approfondita di come le KANs si collochino rispetto alle sfide attuali dell'intelligenza artificiale, in particolare rispetto alle MLPs, e di riflettere su se e come, almeno in linea di principio, queste nuove reti possano operare più efficientemente in alcuni contesti specifici.

I due problemi implementativi discussi nel capitolo 4 hanno avuto l'obiettivo di mostrare la grande intuitività implementativa delle KANs e la facilità con cui si possono interpretare i risultati con esse ottenuti. Questo è stato reso evidente, ad esempio, dalla possibilità di estrarre le funzioni di attivazione dal modello addestrato, di fissare alcuni archi della rete per eseguire addestramenti più "raffinati", e dalla capacità delle KANs di adattarsi rapidamente ai dati. L'integrazione di questi aspetti pratici ha permesso di evidenziare le potenzialità delle KANs non solo dal punto di vista teorico, ma anche nella loro applicabilità concreta.

Prospettive Future

Guardando avanti, è chiaro che le KANs rappresentino un campo di ricerca ancora in assoluta evoluzione. Le prospettive future includono, in prima battuta: l'ottimizzazione dell'architettura sottostante, l'esplorazione di nuovi contesti applicativi e la verifica della loro reale efficacia rispetto alle MLPs. Sarà interessante vedere se l'entusiasmo iniziale attorno alle KANs potrà tradursi in applicazioni concrete che giustifichino le aspettative o se, al contrario, le MLPs continueranno a mantenere la loro egemonia.
L'importanza del KART e delle KANs nel panorama dell'intelligenza artificiale rimane comunque un tema di grande interesse, con potenziali sviluppi che potrebbero portare a nuove e fondamentali direzioni di ricerca e applicazione.

Bibliografia

[1] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.

[2] Z. Liu, Y. Wang, S. Vaidya, F. Ruehle, J. Halverson, M. Soljačić, T. Y. Hou, and M. Tegmark, "Kan: Kolmogorov-arnold networks," *arXiv preprint arXiv:2404.19756*, 2024.

[3] 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, pp. 953–956, 1957.

[4] V. I. Arnold, "On functions of three variables," *Doklady Akademii Nauk SSSR*, vol. 141, no. 4, pp. 793–796, 1963.

[5] A. Pinkus, "Approximation theory of the mlp model in neural networks," *Acta Numerica*, vol. 8, pp. 143–195, 1999.

[6] G. P. Brandino, J.-S. Caux, and R. M. Konik, "Glimmers of a quantum kam theorem: Insights from quantum quenches in one-dimensional bose gases," *Phys. Rev. X*, vol. 5, p. 041043, Dec 2015. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevX.5.041043

[7] N. Smaoui, A. El-Kadri, and M. Zribi, "On the control of the 2d navier–stokes equations with kolmogorov forcing," *Complexity*, vol. 2021, no. 1, p. 3912014, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1155/2021/3912014

[8] T. Poggio and F. Girosi, "Networks for approximation and learning," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1481–1497, 1990.

[9] Towards AI. (2024) Kan (kolmogorov-arnold networks): A starter guide. [Online]. Available: https://towardsai.net/p/ml/kan-kolmogorov-arnold-networks-a-starter-guide

[10] Daily Dose of Data Science, "A beginner friendly introduction to kolmogorov-arnold networks (kan)," https://www.dailydoseofds.com/a-beginner-friendly-introduction-to-kolmogorov-arnold-networks-kan/#kan-vs-mlp, 2023.

[11] C. de Boor, “A practical guide to spline,” *Applied Mathematical Sciences, New York: Springer, 1978*, vol. Volume 27, 01 1978.

[12] R. Gautam, “(kans part 1) an introduction to b-splines,” *rohangautam.github.io*, Jun 2024. [Online]. Available: https://rohangautam.github.io/blog/b_spline_intro/

[13] M. G. COX, “The Numerical Evaluation of B-Splines*,” *IMA Journal of Applied Mathematics*, vol. 10, no. 2, pp. 134–149, 10 1972. [Online]. Available: https://doi.org/10.1093/imamat/10.2.134

[14] G. Beer, B. Marussig, and C. Duenser, “Basis functions, b-splines,” in *The Isogeometric Boundary Element Method*, ser. Lecture Notes in Applied and Computational Mechanics. Springer, Cham, 2020, vol. 90, pp. 39–73.

[15] S. Abumaryam, “The convergence of polynomial interpolation and runge phenomenon,” *Sirte University Scientific Journal*, vol. 8, no. 1, pp. 77–100, 2018.

[16] L. N. Trefethen, *Approximation Theory and Approximation Practice*. Philadelphia: SIAM, 2013.

[17] W. Schweizer, *Legendre Polynomials and Legendre Functions*. Cham: Springer International Publishing, 2021, pp. 33–63. [Online]. Available: https://doi.org/10.1007/978-3-030-64232-7_3

[18] X. Lv and S. Shen, “On chebyshev polynomials and their applications,” *Advances in Difference Equations*, vol. 2017, 10 2017.

[19] S. SS, K. AR, G. R, and A. KP, “Chebyshev polynomial-based kolmogorov-arnold networks: An efficient architecture for nonlinear function approximation,” 2024. [Online]. Available: https://arxiv.org/abs/2405.07200

[20] Q. Wang, Y. Ma, K. Zhao, and Y. Tian, “A comprehensive survey of loss functions in machine learning,” *Annals of Data Science*, pp. 1–26, 2020.

[21] Mintisan, “Awesome kolmogorov-arnold network (kan) repository,” 2024. [Online]. Available: https://github.com/mintisan/awesome-kan

[22] SynodicMonth, “Chebykan: Implementation of chebyshev polynomial-based kolmogorov-arnold networks,” https://github.com/SynodicMonth/ChebyKAN, 2024.

[23] Boris-73-TA, “Orthogpolykans: Orthogonal polynomial-based kolmogorov-arnold networks implementation,” https://github.com/Boris-73-TA/OrthogPolyKANs, 2024.

[24] J. Henseler, *Back Propagation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 37–66. [Online]. Available: https://doi.org/10.1007/BFb0027022

[25] D. L. Donoho, "High-dimensional data analysis: The curses and blessings of dimensionality," *AMS Math Challenges Lecture*, 2000, available at: https://statweb.stanford.edu/~donoho/Lectures/AMS2000/Curses.pdf.

[26] R. M. French, "Catastrophic forgetting in connectionist networks," *Trends in Cognitive Sciences*, vol. 3, no. 4, pp. 128–135, 1999.

[27] N. J. Schaub and N. Hotaling, "Assessing efficiency in artificial neural networks," *Applied Sciences*, vol. 13, no. 18, 2023. [Online]. Available: https://www.mdpi.com/2076-3417/13/18/10286

[28] P. J. Contributors, *Project Jupyter*, 2023. [Online]. Available: https://jupyter.org/

[29] P. S. Foundation, *Python: A dynamic, open source programming language*, 2023. [Online]. Available: https://www.python.org/

[30] J. Contributors, *nbconvert: Jupyter Notebook Conversion*, 2023. [Online]. Available: https://nbconvert.readthedocs.io/

[31] Z. Liu, "pykan: A library for kolmogorov-arnold neural networks," 2023, accessed: 2024-08-28. [Online]. Available: https://github.com/KindXiaoming/pykan

[32] Z. Liu, P. Ma, Y. Wang, W. Matusik, and M. Tegmark, "Kan 2.0: Kolmogorov-arnold networks meet science," 2024. [Online]. Available: https://arxiv.org/abs/2408.10205

[33] S. Cuomo, V. S. Di Cola, F. Giampaolo, G. Rozza, M. Raissi, and F. Piccialli, "Scientific machine learning through physics–informed neural networks: Where we are and what's next," *Journal of Scientific Computing*, vol. 92, no. 3, p. 88, 2022.

[34] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *Journal of Machine Learning Research*, vol. 18, no. 153, pp. 1–43, 2018. [Online]. Available: http://jmlr.org/papers/v18/17-468.html