Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Matematica

Algebra Lineare e Intelligenza Artificiale: Il Caso di AlphaTensor

Tesi di Laurea in Geometria

Relatore:
Chiar.mo Prof.
Giovanni Paolini

Presentata da:
Francesca Berti Ceroni

Anno Accademico 2024/2025

A mio papà e ad Antonio.

Introduzione

Qual è il modo più veloce possibile per moltiplicare due matrici? Dietro questa domanda si nasconde uno dei problemi aperti più significativi per l'algebra lineare moderna e per la teoria della complessità. L'importanza di tale quesito deriva dalla natura fortemente applicativa dell'operazione di moltiplicazione matriciale. Essa costituisce infatti il nucleo di una grande quantità di algoritmi di calcolo, fondamentali nel progresso scientifico e tecnologico. La moltiplicazione matriciale è onnipresente nella vita quotidiana di ognuno di noi: quando si guarda una foto o un video sul telefono, entra in gioco nell'elaborazione grafica delle immagini; quando si utilizza un servizio di traduzione automatica o si interagisce con un assistente virtuale, sostiene i modelli di intelligenza artificiale che rendono possibili tali applicazioni; nelle previsioni meteorologiche alimenta i modelli numerici che simulano l'evoluzione dell'atmosfera; anche nella logistica o nell'organizzazione di reti di trasporto, algoritmi basati su calcoli matriciali vengono impiegati per ottimizzare percorsi e risorse. La lista delle applicazioni potrebbe continuare a lungo; l'idea fondamentale è che un miglioramento anche minimo nell'efficienza della moltiplicazione di matrici si traduce in un impatto significativo nella riduzione di tempi di calcolo e di costi computazionali e di conseguenza nella qualità delle applicazioni stesse.

Il tema della moltiplicazione matriciale è qui trattato a partire dalle sue origini, con particolare attenzione alle sue evoluzioni teoriche nel corso dei secoli, fino ai più recenti risultati ottenuti grazie all'utilizzo di un sistema di intelligenza artificiale capace di produrre nuovi e più efficienti algoritmi: AlphaTensor. Il primo capitolo, diviso in quattro sezioni, ripercorre la storia del calcolo matriciale dalle prime rappresentazioni tabellari degli studiosi cinesi del 1000 a.C., al miglioramento dell'algoritmo di moltiplicazione matriciale standard ad opera del matematico tedesco Volker Strassen nel 1969. Approfondisce le tappe intermedie più importanti, tra cui le scoperte dei matematici indiani tra il V e il XII secolo, l'introduzione, nel IX secolo, del termine *algoritmo* dovuta al matematico arabo Al-khwarizmi e la formalizzazione dei termini e delle procedure di calcolo matriciale attribuita agli inglesi James Joseph Sylvester e Arthur Cayley. Nel secondo capitolo, dopo una breve introduzione al concetto di tensore, fondamentale per

la trattazione, vengono analizzati architettura e funzionamento di AlphaTensor, l'agente di intelligenza artificiale sviluppato dal gruppo di ricercatori di Google DeepMind nel 2022. L'idea degli sviluppatori è l'implementazione di un gioco *single–player*, il TensorGame, in cui il ruolo di giocatore spetta all'agente AlphaTensor e il tabellone di gioco è il tensore che rappresenta la moltiplicazione tra due matrici. Tramite il metodo di apprendimento per rinforzo profondo (*deep reinforcement learning*) descritto nella seconda sezione del capitolo, l'agente migliora di volta in volta le proprie prestazioni, trovando algoritmi sempre più efficienti. I risultati prodotti da AlphaTensor vengono illustrati e confrontati con i migliori algoritmi scoperti dai matematici negli anni precedenti al suo sviluppo nel capitolo conclusivo della trattazione. Grazie al contributo di AlphaTensor che ha permesso l'individuazione di nuove strategie di calcolo più efficienti di quelle finora conosciute, il problema di ottimizzazione della complessità computazionale degli algoritmi di moltiplicazione matriciale ha compiuto grandi passi avanti. Tuttavia, la ricerca della complessità ottimale per questo tipo di operazione continua a rappresentare una delle sfide centrali per la matematica e l'informatica teorica.

Indice

Capitolo 1

Moltiplicazione Matriciale: il Cuore del Calcolo Computazionale

1.1 Origini ed evoluzione

La moltiplicazione matriciale è attualmente una delle operazioni fondamentali del calcolo computazionale e della progettazione di algoritmi efficienti. La sua rilevanza, tuttavia, non si limita all'epoca contemporanea: tracce significative dell'impiego di strutture tabellari simili alle odierne matrici sono riscontrabili già in epoche precedenti alla formalizzazione dell'algebra lineare moderna.

Le prime testimonianze di manipolazioni di coefficienti riconducibili a operazioni matriciali giungono dall'antica Cina durante l'impero della dinastia Zhou [16, 3]. In particolare, nel trattato *Jiuzhang Suanshu* (*I nove capitoli sull'arte della matematica*), redatto tra gli anni 1000 e 200 a.C., è presente un capitolo interamente dedicato alla risoluzione di sistemi lineari tramite il metodo *fangcheng*. La procedura di tale metodo consiste nel disporre coefficienti e termini noti verticalmente, creando una tabella simile a una matrice trasposta rispetto alla notazione attuale, ed eseguire trasformazioni sulle colonne al fine di azzerare progressivamente i coefficienti, riducendo il sistema a forma triangolare. Si conclude risolvendo il sistema di equazioni tramite il metodo di sostituzione all'indietro [13]. Le tecniche applicate per giungere alla soluzione del problema consistevano dunque in operazioni elementari sulle colonne estremamente simili a quelle del metodo di eliminazione che oggi conosciamo come metodo di Gauss. Dunque tale metodo di risoluzione di sistemi lineari, sebbene con alcune notevoli differenze, era utilizzato in Cina già duemila anni prima degli studi di Carl Friedrich Gauss (1777–1855), il quale formalizzò l'omonimo algoritmo solo agli inizi del XIX secolo.

Numerose civiltà nella storia hanno fornito contributi significativi allo sviluppo di

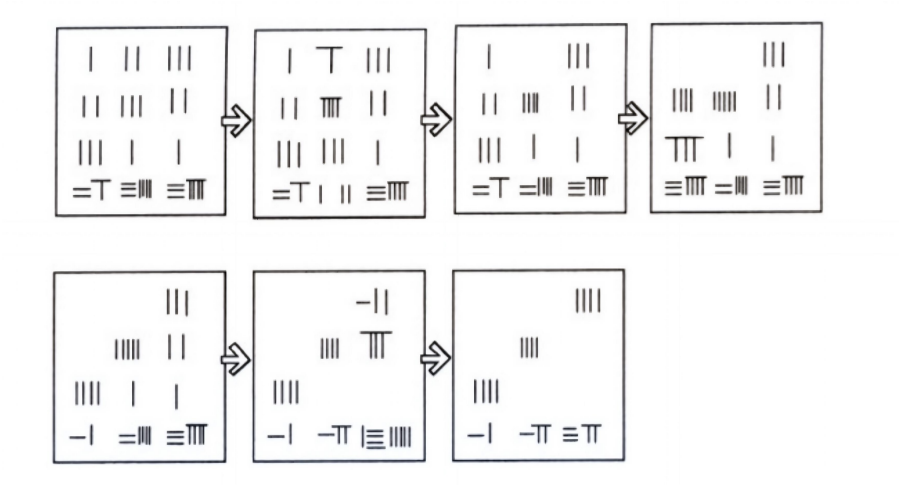


Figura 1.1: *Rappresentazione grafica del metodo fangcheng. L'immagine è tratta da [26].*

tecniche di calcolo algoritmico, anticipando concetti che sarebbero poi stati formalizzati con la nascita dell'algebra lineare.

Un altro esempio rilevante è dato dai matematici indiani che tra il V e il XII secolo si impegnarono nell'implementazione di procedure per sistemi di equazioni lineari e quadratiche, spesso legati alla risoluzione di problemi di natura astronomica. Una testimonianza di tale conoscenza è l'opera di Brahmagupta (VII secolo), che nel suo *Brahmasphuta Siddhanta* introdusse nuovi metodi per trattare tali sistemi di equazioni [17].

Un'eredità ancora più diretta si trova nella matematica araba, dove, durante il medioevo, si consolidò il concetto di algoritmo come processo sistematico di calcolo. Il matematico al-Khwarizmi (IX secolo), da cui deriva il termine *algoritmo*, fu tra i primi a formalizzare metodi per la risoluzione di equazioni lineari e quadratiche attraverso procedimenti operativi rigorosi [19]. Sebbene non si parlasse ancora di *matrice* come oggetto matematico, l'idea di rappresentare e manipolare relazioni numeriche attraverso schemi tabellari era assai presente nei suoi scritti e in quelli dei suoi successori.

Durante il XVIII secolo, numerosi matematici contribuirono allo sviluppo di strumenti che anticipavano la teoria delle matrici, pur senza una formulazione unificata. Tra questi si ricordano Colin Maclaurin, Gabriel Cramer, Étienne Bézout, Alexandre-Théophile Vandermonde, Pierre-Simon Laplace e Joseph-Louis Lagrange.

La svolta concettuale decisiva avvenne tuttavia nel XIX secolo, quando il termine *matrice* venne introdotto formalmente da James Joseph Sylvester nel 1848. Poco dopo, nel 1858, Arthur Cayley pubblicò il *Memoir on the Theory of Matrices*, all'interno del quale fornì la prima vera definizione di matrice. Cayley, inoltre, formalizzò il concetto di

prodotto matriciale come operazione algebrica e in particolare dimostrò come numerose operazioni già note, tra cui la moltiplicazione tra sistemi lineari, i determinanti e le trasformazioni geometriche, siano casi particolari di quest'ultima. Introdusse anche il concetto di matrice inversa, ponendo così le basi della teoria oggi nota come teoria delle matrici.

Già da prima della metà del Novecento, la moltiplicazione matriciale era considerata una tecnica di calcolo di fondamentale importanza sia nella ricerca che in numerosi ambiti applicativi. Nonostante l'algoritmo risultasse particolarmente oneroso in termini di operazioni richieste, non si avvertiva ancora l'urgenza di ottimizzarne l'esecuzione: la sua struttura veniva ritenuta naturale e, per certi versi, definitiva. L'avvento dei computer modificò radicalmente questa percezione, rendendo evidente la necessità di metodi più efficienti.

Con il rapido progresso tecnologico e la conseguente introduzione di nuovi numerosissimi ambiti applicativi – dalla matematica finanziaria alla statistica, dall'elaborazione delle immagini digitali fino al *deep learning*, fulcro della presente trattazione – la moltiplicazione matriciale si rivelò una delle operazioni più importanti e dunque diventò oggetto di numerose ricerche volte alla sua ottimizzazione.

In virtù della sua capacità di modellare trasformazioni lineari e sistemi di equazioni, la moltiplicazione matriciale rappresenta oggi un punto di convergenza tra matematica teorica e applicazioni avanzate, confermandosi come una delle operazioni più centrali e onnipresenti del calcolo moderno.

1.2 Il metodo classico

In questa sezione verrà presentata la definizione formale della moltiplicazione matriciale secondo la formulazione introdotta da Arthur Cayley a metà del XIX secolo, oggi considerata il riferimento teorico fondamentale nell'ambito dell'algebra lineare.

Dopo aver descritto l'algoritmo standard, ne verrà analizzato il costo computazionale, ponendo l'attenzione sul numero di operazioni necessarie e sul conseguente impatto in termini di efficienza. Tale analisi metterà in luce le principali criticità dell'algoritmo, in particolare in presenza di matrici di grandi dimensioni, e fornirà la base per comprendere la necessità di sviluppare metodi più rapidi.

D'ora in avanti assumeremo che i coefficienti della matrice appartengano al campo dei numeri reali $\mathbb{K} = \mathbb{R}$, scelta che consente di porre l'attenzione sugli aspetti computazionali evitando complicazioni legate a ambienti di studio più complessi.

Definizione 1.2.1. Sia $A \in \mathbb{R}^{m\times n}$ e $B \in \mathbb{R}^{n\times p}$. Allora il prodotto riga per colonna $C = AB$ è definito come la matrice $C \in \mathbb{R}^{m\times p}$ tale che

$$C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}, \quad \text{per ogni } 1 \leq i \leq m,\ 1 \leq j \leq p.$$

Valutiamo ora il costo computazionale di tale operazione. Supponiamo $A, B \in \mathbb{R}^{n\times n}$ matrici quadrate di ordine n. Gli elementi di $C = AB$ sono n^2 e per ogni elemento dobbiamo effettuare $n-1$ somme e n prodotti. Dunque il **costo computazionale** richiesto per la moltiplicazione tra due matrici quadrate di ordine n è $n^2(n-1+n) = n^2(2n-1) = 2n^3 - n^2 = \mathcal{O}(n^3)$.

Ormai da diversi decenni, di pari passo con l'evoluzione tecnologica e informatica, i matematici di tutto il mondo stanno cercando una risposta a una domanda di fondamentale importanza: *è possibile implementare un algoritmo per il calcolo del prodotto matriciale di complessità computazionale pari ad* $\mathcal{O}(n^2)$*?*

L'esigenza di diminuire i costi di tale operazione scaturisce dalla rilevanza che l'operazione di prodotto tra matrici ha all'interno di ogni ambito applicativo in cui viene utilizzata.

Un esempio interessante è la generazione di immagini digitali. Supponiamo che la Figura 1.2 debba essere trasmessa da un satellite meteorologico a un laboratorio sulla Terra. Per una questione di accuratezza del risultato, il satellite genera l'immagine in alta risoluzione, ossia con un numero elevatissimo di *pixel*. Supponiamo che tale numero sia nell'ordine di un milione e che dunque le dimensioni dell'immagine siano 1000×1000. Possiamo immaginare ogni singolo pixel come uno dei 10^6 elementi di una matrice a valori interi tra 0 e 255 (range di codifica scala di grigi standard), ciascuno rappresentante una particolare sfumatura di colore. Dunque, per trasmettere l'immagine, il satellite dovrebbe inviare al laboratorio sulla terra un milione di numeri, e poiché stiamo analizzando il lavoro di un satellite utile alle previsioni meteorologiche, questo dovrebbe avvenire quotidianamente se non più volte durante la giornata. I satelliti comunicano con le stazioni sulla Terra tramite radiofrequenze, e hanno una larghezza di banda limitata, dunque più dati devono essere trasmessi, più tempo richiede la comunicazione. Trasmettere un milione di numeri (tipicamente in formato binario a 8, 16 o 32 bit) richiede tempo e larghezza di banda preziosi. Tramite la moltiplicazione matriciale (per esempio tramite la *decomposizione in valori singolari*, SVD) possiamo ridurre notevolmente il numero di dati da gestire e rendere più efficiente e veloce la comunicazione [22]. La decomposizione SVD consiste nel scomporre la matrice A, nel nostro caso l'immagine del satellite, nel prodotto di tre matrici $A = U\Lambda V^T$ con U e V ortogonali. Grazie al Teorema di Eckart–Young che

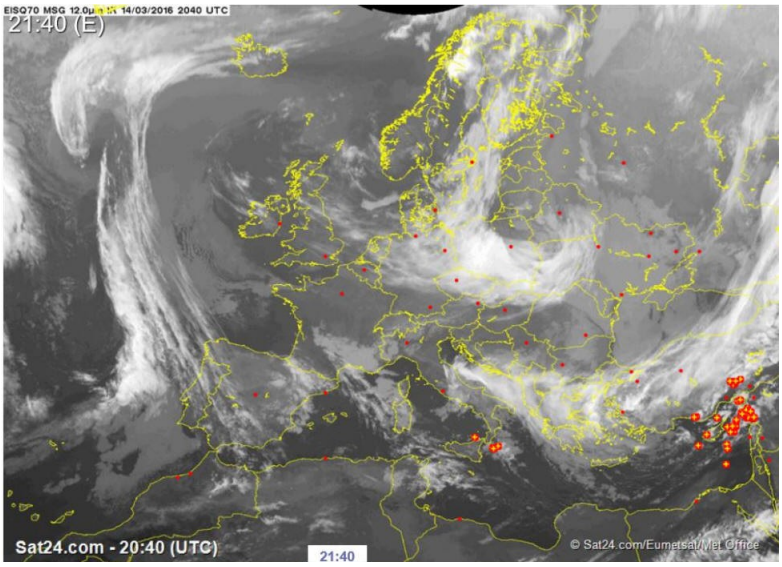


Figura 1.2: *Meteosat 10 (MSG3)*

ci assicura che troncare la SVD dopo i primi k valori assicura la miglior approssimazione (ossia la matrice di rango k che minimizza la distanza da A in norma di Frobenius) a rango k della matrice A, possiamo comprimere l'immagine mantenendone le informazioni più significative (i valori singolari) ed eliminando gli elementi trascurabili. [12]. Per esempio, fissato $k = 20$, è sufficiente che il satellite invii le prime venti colonne di U e di V e i primi venti elementi Λ_{ii} per un totale di soli $20 \cdot 1000 + 20 \cdot 1000 + 20 = 40020$ numeri che devono essere trasmessi al laboratorio. Una riduzione da un milione a quarantamila numeri implica una trasmissione venticinque volte più veloce. Così, tramite un'applicazione della moltiplicazione matriciale, abbiamo ottimizzato un processo di trasmissione dati altrimenti complicato e costoso.

Oltre a presentare una delle numerose applicazioni del prodotto di matrici, questo esempio evidenzia un altro dato fondamentale: anche per indagini di frequenza quotidiana come la previsione di eventi meteorologici è necessario gestire e manipolare un numero di dati, ordinati in strutture matriciali, di enormi dimensioni.

1.2.1 Limiti dell'algoritmo

L'algoritmo classico diventa rapidamente inadeguato quando si ha a che fare con tali ordini di grandezza. Come già discusso precedentemente, la complessità computazionale di questo metodo è di $\mathcal{O}(n^3)$: ciò comporta che, all'aumentare della dimensione n della matrice, il numero di operazioni richieste cresce cubicamente. Di conseguenza, anche un incremento modesto nel numero di elementi comporta un aumento notevole del tempo di calcolo. Oltre al tempo di esecuzione, anche il consumo di memoria ad accesso

casuale (RAM) e l'efficienza energetica diventano fattori critici. La larghezza di banda della memoria, il numero di operazioni in virgola mobile per secondo (flops) che la macchina è in grado di sostenere e il costo energetico per eseguire un'istruzione elementare influenzano drasticamente le prestazioni. Per questo motivo, la moltiplicazione tra matrici di grandi dimensioni rappresenta oggi uno dei principali colli di bottiglia nel calcolo computazionale.

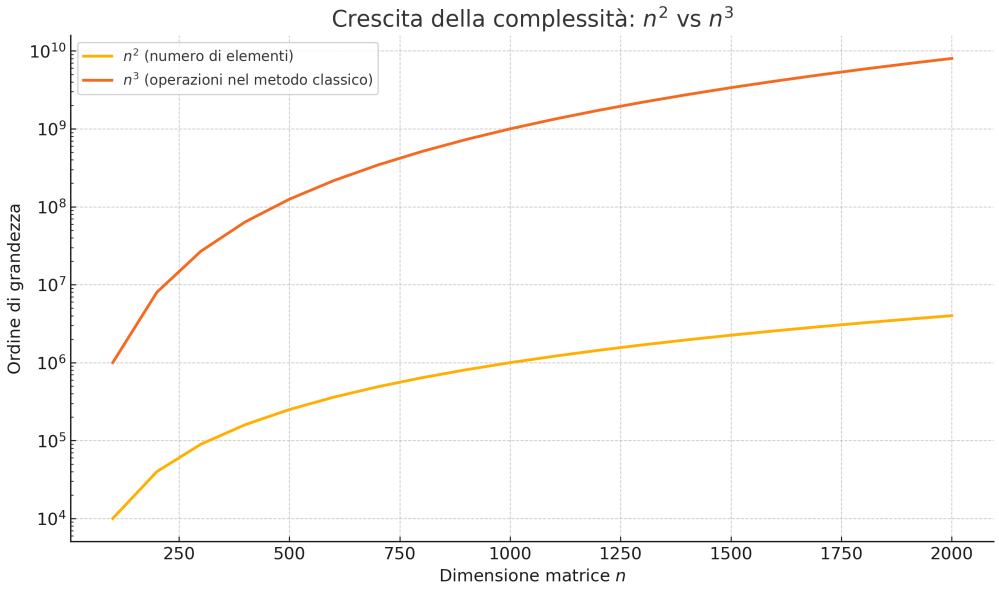


Figura 1.3: *Confronto tra la crescita quadratica n^2 (numero di elementi) e quella cubica n^3 (numero di operazioni nel metodo classico) in funzione della dimensione n della matrice. La scala logaritmica sull'asse y mette in evidenza la divergenza tra i due comportamenti al crescere di n.*

Si consideri ad esempio una macchina in grado di eseguire circa 10^9 operazioni in virgola mobile al secondo (1 GFLOPS). Supponiamo di dover moltiplicare due matrici quadrate di ordine $n = 10^4$. Il numero di moltiplicazioni richieste dal metodo classico è dell'ordine di $n^3 = 10^{12}$. In questo caso, anche trascurando le addizioni e ipotizzando un'esecuzione sufficientemente ottimizzata, l'operazione di moltiplicazione richiederebbe almeno

$$\frac{10^{12}}{10^9} = 10^3 \text{ secondi} \approx 17 \text{ minuti}.$$

Per matrici ancora più grandi o in presenza di diverse configurazioni hardware della macchina, i tempi possono aumentare drasticamente. Questo evidenzia come la moltiplicazione matriciale classica possa diventare rapidamente un'operazione non sostenibile

su scala industriale o scientifica e motiva la ricerca di algoritmi alternativi al metodo classico, argomento che analizzerò nelle sezioni successive.

In Figura 1.3 ho riportato un grafico utile alla visualizzazione del notevole salto presente tra la crescita quadratica e quella cubica.

1.3 La rivoluzione di Volker Strassen

Nel 1969, il matematico tedesco Volker Strassen pubblicò all'interno del volume *Numerische Mathematik* l'articolo *Gaussian Elimination is not Optimal*, in cui dimostrò che la moltiplicazione tra matrici poteva essere effettuata in meno di $\mathcal{O}(n^3)$ operazioni. I contenuti di questa sezione seguono in larga parte quanto esposto in [23], a cui si rimanda per una trattazione più estesa.

In particolare, Strassen mostrò che due matrici quadrate $n \times n$ possono essere moltiplicate con una complessità computazionale pari a

$$\mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.81}),$$

rompendo la barriera del metodo classico. L'algoritmo da lui proposto si basa su una tecnica ricorsiva che, nel caso base di matrici 2×2, permette di eseguire la moltiplicazione utilizzando soltanto 7 prodotti invece degli 8 richiesti dal metodo convenzionale.

Notazione: Tutti i logaritmi all'interno di questa sezione sono intesi in base 2, salvo diverse indicazioni.

Proposizione 1.3.1 (Il metodo Strassen)**.** Definiamo una famiglia di algoritmi $\alpha_{m,k}$ per la moltiplicazione di matrici di ordine $m \cdot 2^k$, costruita per induzione su k. L'algoritmo del passo base $\alpha_{m,0}$ coincide con l'algoritmo classico, che richiede m^3 moltiplicazioni e $m^2(m-1)$ addizioni. Supponendo noto $\alpha_{m,k}$, definiamo $\alpha_{m,k+1}$ come segue.

Siano date due matrici quadrate A, B di ordine $m \cdot 2^{k+1}$

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad e \quad AB = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

dove i blocchi A_{ij}, B_{ij}, C_{ij}, sono matrici di ordine $m \cdot 2^k$. Si calcola

$$\begin{aligned} I &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ II &= (A_{21} + A_{22})B_{11} \\ III &= A_{11}(B_{12} - B_{22}) \\ IV &= A_{22}(-B_{11} + B_{21}) \\ V &= (A_{11} + A_{12})B_{22} \\ VI &= (-A_{11} + A_{21})(B_{11} + B_{12}) \\ VII &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

Il prodotto $C = AB$ è poi ottenuto da:

$$\begin{aligned} C_{11} &= I + IV - V + VII \\ C_{12} &= III + V \\ C_{21} &= II + IV \\ C_{22} &= I + III - II + VI \end{aligned}$$

Utilizzando $\alpha_{m,k}$ per la moltiplicazione e l'algoritmo classico per l'addizione e la sottrazione di matrici di ordine $m \cdot 2^k$, si ottiene, per induzione su k, quanto segue:

Fatto 1.3.2. L'algoritmo $\alpha_{m,k}$ calcola il prodotto di due matrici di ordine $m \cdot 2^k$ effettuando $m^3 \cdot 7^k$ moltiplicazioni e $(5 + m)m^2 \cdot 7^k - 6(m \cdot 2^k)^2$ addizioni e sottrazioni.

Ne consegue che due matrici di ordine 2^k possono essere moltiplicate con 7^k moltiplicazioni e meno di $6 \cdot 7^k$ addizioni e sottrazioni.

Fatto 1.3.3. Il prodotto di due matrici quadrate di ordine n può essere calcolato con meno di

$$4.7 \cdot n^{\log 7}$$

operazioni aritmetiche.

Dimostrazione. Definiamo

$$k = \lfloor \log n - 4 \rfloor, \quad m = \lfloor n 2^{-k} \rfloor + 1;$$

allora $n \leq m \cdot 2^k$. Incorporando matrici di ordine n in matrici di ordine $m \cdot 2^k$, il numero di operazioni richiesto da $\alpha_{m,k}$ (Fatto 1.3.2) risulta:

$$\begin{aligned}
&(5+2m)m^2 7^k - 6(m2^k)^2 \\
&< (5+2(n2^{-k}+1))(n2^{-k}+1)^2 7^k \\
&< 2n^3 \left(\frac{7}{8}\right)^k + 12.03 \cdot n^2 \cdot \left(\frac{7}{4}\right)^k
\end{aligned}$$

(qui si è usato il fatto che $16 \cdot 2^k \leq n$)

$$\begin{aligned}
&= \left(2\left(\frac{8}{7}\right)^{\log n - k} + 12.03\left(\frac{4}{7}\right)^{\log n - k}\right) \cdot n^{\log 7} \\
&\leq \max_{4 \leq t \leq 5} \left(2\left(\frac{8}{7}\right)^t + 12.03\left(\frac{4}{7}\right)^t\right) \cdot n^{\log 7} \\
&\leq 4.7 \cdot n^{\log_2 7}.
\end{aligned}$$

L'ultima disuguaglianza è data dalla natura di $f(t) = \left(2\left(\frac{8}{7}\right)^t + 12.03\left(\frac{4}{7}\right)^t\right)$ che, in quanto somma di convesse, è convessa, e dunque all'interno di un intervallo chiuso $[a,b]$, raggiunge il massimo agli estremi. Scelto $[a,b] = [4,5]$, si ha

$$f(t) \leq \max_{4 \leq t \leq 5} f(t) = \max\left\{(2 \cdot \left(\frac{8}{7}\right)^4 + 12.03 \cdot \left(\frac{4}{7}\right)^4,\ 2 \cdot \left(\frac{8}{7}\right)^5 + 12.03 \cdot \left(\frac{4}{7}\right)^5\right\} \approx 4.7.$$

da cui la tesi. □

Osservazione 1.3.4. É utile chiarire la scelta di porre $k = \lfloor \log n - 4 \rfloor$ e $m = \lfloor n2^{-k} \rfloor + 1$. Poiché l'algoritmo di Strassen è applicabile a matrici quadrate che hanno come ordine una qualche potenza di 2, è necessario, nel caso in cui si voglia applicare l'algoritmo a matrici di ordine n, estendere la dimensione di suddette matrici a $m \cdot 2^k \times m \cdot 2^k$ per qualche intero m. Scegliendo k e m come sopra, la matrice di ordine n viene incapsulata in una matrice di ordine $m \cdot 2^k$, compatibile con il meccanismo ricorsivo di Strassen. La presenza del termine sottrattivo -4 nella definizione di k risponde a un'esigenza tecnica: si vuole garantire che 2^k sia sufficientemente più piccolo di n, così che m, il numero di blocchi per lato, risulti abbastanza grande da sfruttare effettivamente la ricorsione a blocchi dell'algoritmo. In questo modo, la matrice estesa (*padded*) risulta più vicina in termini

di dimensioni alla matrice originale, mantenendo basso l'*overhead* computazionale[1]. Per ulteriori approfondimenti si consiglia la consultazione di [6].

Segue un esempio di come il termine sottrattivo nella definizione di k migliori l'efficienza computazionale dell'algoritmo.

Esempio 1.3.5. Supponiamo che la matrice a cui vogliamo applicare l'algoritmo di Strassen abbia ordine $n = 130$.

- Scegliendo $k = \lfloor \log_2 n \rfloor$ otteniamo

$$k = \lfloor \log_2 130 \rfloor = 7, \quad 2^k = 128, \quad m = \lfloor 130/128 \rfloor + 1 = 2.$$

  Il risultato è una matrice estesa di ordine $m \cdot 2^k = 2 \cdot 128 = 256$, quasi il doppio dell'originale.

- Scegliendo invece $k = \lfloor \log_2 n - 4 \rfloor$ come nella dimostrazione data da Strassen, abbiamo

$$k = \lfloor \log_2 130 - 4 \rfloor = 3, \quad 2^k = 8, \quad e \quad m = \lfloor 130/8 \rfloor + 1 = 17,$$

  ottenendo una matrice estesa di ordine $m \cdot 2^k = 17 \cdot 8 = 136$, molto più vicino a n. Così, evitando di aggiungere numerose righe vuote alla matrice di partenza, abbiamo ottimizzato l'efficienza del metodo.

N.B: Nel caso in cui $n < 16$, si ha $k = \lfloor \log_2 n - 4 \rfloor < 0$. Questo risultato tuttavia non crea problemi, poiché, come vedremo più avanti, nel caso in cui n sia particolarmente piccolo, si continua a preferire la moltiplicazione classica.

Vediamo ora un esempio di moltiplicazione matriciale prima effettuata tramite il metodo classico e in seguito con il metodo di Strassen.

Esempio 1.3.6. Siano $A, B \in \mathbb{R}^{2\times 2}$

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}.$$

La matrice prodotto $C = AB$ può essere ottenuta tramite:

[1]Per *overhead* computazionale si intende il costo aggiuntivo — in termini di operazioni, memoria o tempo di esecuzione — introdotto da strutture ausiliarie o trasformazioni necessarie al corretto funzionamento dell'algoritmo. Nel caso specifico di Strassen, il *padding* consiste nell'estendere una matrice $n \times n$ a una matrice $m \cdot 2^k \times m \cdot 2^k$, riempiendo le celle aggiuntive con zeri, così da ottenere le dimensioni richieste dall'algoritmo.

- **il metodo classico**:

$$
\begin{aligned}
&I = a_{11}b_{11} = 1 \cdot 1 = 1\\
&II = a_{12}b_{21} = 2 \cdot 0 = 0\\
&III = a_{11}b_{12} = 1 \cdot (-1) = -1\\
&IV = a_{12}b_{22} = 2 \cdot 1 = 2\\
&V = a_{21}b_{11} = 3 \cdot 1 = 3\\
&VI = a_{22}b_{21} = 4 \cdot 0 = 0\\
&VII = a_{21}b_{12} = 3 \cdot (-1) = -3\\
&VIII = a_{22}b_{22} = 4 \cdot 1 = 4
\end{aligned}
$$

$$
\begin{aligned}
C_{11} &= I + II = 1 + 0 = 1\\
C_{12} &= III + IV = -1 + 2 = 1\\
C_{21} &= V + VI = 3 + 0 = 3\\
C_{22} &= VII + VIII = -3 + 4 = 1
\end{aligned}
$$

allora

$$
C = \begin{pmatrix} 1 & 1 \\ 3 & 1 \end{pmatrix},
$$

effettuando 8 moltiplicazioni e 4 somme.

- **il metodo Strassen**:

$$
\begin{aligned}
&I = (1 + 4)(1 + 1) = 10\\
&II = (3 + 4) \cdot 1 = 7\\
&III = 1 \cdot (-1 - 1) = -2\\
&IV = 4 \cdot (-1 + 0) = -4\\
&V = (1 + 2) \cdot 1 = 3\\
&VI = (-1 + 3)(1 - 1) = 0\\
&VII = (2 - 4)(0 + 1) = -2
\end{aligned}
$$

$$
\begin{aligned}
C_{11} &= 10 - 4 - 3 - 2 = 1\\
C_{12} &= -2 + 3 = 1\\
C_{21} &= 7 - 4 = 3\\
C_{22} &= 10 - 2 - 7 + 0 = 1
\end{aligned}
$$

allora

$$C = \begin{pmatrix} 1 & 1 \\ 3 & 1 \end{pmatrix},$$

effettuando 7 moltiplicazioni e 18 somme.

1.3.1 Limiti del metodo di Strassen

Sebbene l'algoritmo di Volker Strassen abbia rappresentato una svolta epocale nella teoria del calcolo matriciale, aprendo la strada agli algoritmi subcubici per la moltiplicazione matriciale e dimostrando che il limite inferiore di $\mathcal{O}(n^3)$ può essere superato, l'applicazione pratica del metodo presenta una serie di limiti che ne hanno ridimensionato l'adozione in contesti reali.

Dal punto di vista dell'implementazione dell'algoritmo, Strassen non si limita a effettuare moltiplicazioni e addizioni ma suddivide le matrici in sottoblocchi, effettua operazioni ricorsive e infine ricompone il risultato. Questa procedura richiede una gestione accurata degli indici e delle dimensioni dei blocchi, oltre a controlli per i casi base come dimensioni dispari o molto piccole delle matrici. Tutto ciò porta inevitabilmente a dover scrivere più codice per implementare Strassen e dunque, nelle applicazioni, il metodo risulta più soggetto a errori.

Un altro limite fondamentale è rappresentato dalla stabilità numerica del metodo: Strassen non è numericamente stabile in aritmetica in virgola mobile. Le numerose operazioni di somma e sottrazione tra blocchi tendono ad amplificare gli errori di arrotondamento e la perdita di cifre significative, specialmente in presenza di cancellazioni numeriche [18]. Questo rende l'algoritmo meno affidabile per applicazioni che richiedono alta precisione, come la risoluzione di sistemi lineari mal condizionati o il calcolo di autovalori.

Noti i numerosi vantaggi dati dalla ridotta complessità asintotica e i limiti appena elencati, numerose librerie numeriche ottimizzate integrano l'algoritmo di Strassen in modo ibrido: esso viene attivato solo quando l'ordine delle matrici supera una certa soglia, mentre per dimensioni inferiori si preferisce l'algoritmo classico, più stabile, prevedibile e veloce nei casi reali.

In sintesi, Strassen è stato determinante nel mostrare che la barriera cubica poteva essere infranta, ma il suo impatto concreto rimane oggi limitato a specifici contesti, in particolare su matrici molto grandi o in ambienti in cui l'efficienza asintotica prevale sulle altre considerazioni.

1.4 Algoritmi successivi: oltre Strassen

Dopo il lavoro di Strassen, numerosi studiosi si sono interrogati sulla possibilità di ridurre ulteriormente l'esponente della complessità, noto come ω, ossia il limite inferiore per cui esiste un algoritmo che moltiplica due matrici quadrate in $\mathcal{O}(n^\omega)$ operazioni.

Nel 1981 il matematico e informatico Arnold Schönhage implementò un algoritmo più veloce di quello di Strassen e introdusse all'interno dello studio del problema nuove tecniche di decomposizione tensoriale per ridurre il rango del tensore della moltiplicazione. Il concetto di tensore sarà di fondamentale importanza per la trattazione di AlphaTensor, argomento centrale della tesi, e verrà introdotto nel prossimo capitolo nella Sezione 2.1. Grazie a queste novità, Schönhage riuscì a ridurre ω a 2.552.

Un ulteriore passo avanti fu compiuto con l'algoritmo di Coppersmith e Winograd (1987), che portò ω a circa 2.376 tramite l'utilizzo di tensori trilineari. I miglioramenti successivi si basarono sull'ottimizzazione di quest'ultimo metodo; tra i più significativi ricordiamo i risultati del matematico francese Jean-François Le Gall che, nel 2014, ottenne un ulteriore riduzione di ω a 2.3728639 ottimizzando l'algoritmo di Coppersmith-Winograd senza alternarne la struttura di base.

Nell'ottobre 2020 Josh Alman e Virginia Vassilevska Williams del Massachusetts Institute of Technology proposero una tecnica altamente innovativa basata su un modello detto *Laser Method*, evitando la dipendenza dai precedenti metodi implementati che, seppur migliori rispetto all'algoritmo standard a livello teorico, rimanevano fortemente penalizzati dall'instabilità numerica e da tutti quei limiti che caratterizzavano anche l'algoritmo di Strassen (Sezione 1.3.1). Ottennero una nuova riduzione di complessità computazionale ($\omega < 2.37286$) rinnovando la speranza di riuscire ad arrivare all'ambito traguardo di $\mathcal{O}(n^2)$.

Tuttavia, nonostante i notevoli miglioramenti, anche questo metodo risulta inapplicabile in contesti utili a causa della sua natura fortemente esistenziale: Alman e Vassilevska non forniscono un algoritmo, ossia una sequenza esplicita di operazioni da eseguire per moltiplicare due matrici, ma presentano unicamente un risultato teorico che non si traduce in un vantaggio computazionale concreto. Questo dipende dal fatto che il risultato del loro lavoro è un miglioramento asintotico, ossia per $n \to +\infty$, dunque non rilevante per piccole-medie dimensioni e quindi per contesti applicativi.

In conclusione, la moltiplicazione matriciale rimane, ancora oggi, un'operazione costosa per le applicazioni reali e un problema aperto sul piano della complessità asintotica. Gli algoritmi successivi a Strassen hanno ridotto il valore dell'esponente ω, ma il loro utilizzo è rimasto confinato all'ambito teorico o a occasionali utilizzi pratici in contesti

specifici: le strutture su cui sono costruiti risultano eccessivamente onerose in termini di dimensioni dell'algoritmo stesso o prettamente teoriche e intraducibili in implementazioni efficienti e stabili.

In risposta a tale difficoltà, è nata e si è affermata negli ultimi anni una nuova linea di pensiero: abbandonare la progettazione algoritmica tradizionale e affidarsi a strumenti di ricerca automatizzata. In questo contesto si inserisce AlphaTensor, una grande innovazione dal team di Google DeepMind che combina algebra multilineare e intelligenza artificiale per scoprire nuovi algoritmi efficienti per la moltiplicazione tra matrici. Il capitolo successivo sarà interamente dedicato all'analisi di questo sistema di intelligenza artificiale e dei risultati da esso ottenuti.

Capitolo 2

AlphaTensor: l'IA alla Scoperta di Nuovi Algoritmi

2.1 Nozioni preliminari

Per comprendere al meglio il funzionamento di AlphaTensor, è necessario introdurre il concetto di *tensore*, centrale nella matematica alla base della formalizzazione del problema della moltiplicazione matriciale.

In seguito verranno riportate le definizioni di *prodotto tensoriale*, *tensore* e *decomposizione a rango R*. Per semplicità di notazione, la definizione di *prodotto tensoriale* verrà data nel caso bilineare, ossia nel caso in cui essa sia costruita a partire da due spazi vettoriali. La generalizzazione al caso multilineare, ossia a più spazi vettoriali, è immediata dalla 2.1.1.

Definizione 2.1.1. Siano V e W spazi vettoriali su un campo $\mathbb{K}$. Il *prodotto tensoriale* $V \otimes W$ è uno spazio vettoriale dotato di una mappa bilineare

$$\otimes : V \times W \to V \otimes W,$$

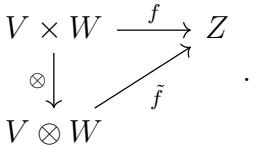
che soddisfa la seguente *proprietà universale*: per ogni spazio vettoriale Z e ogni mappa bilineare $f : V \times W \to Z$, esiste un'unica applicazione lineare

$$\tilde{f} : V \otimes W \to Z$$

tale che

$$\tilde{f}(v \otimes w) = f(v, w) \quad \text{per ogni} \quad (v, w) \in V \times W.$$

In diagramma:



Gli elementi dello spazio prodotto tensoriale sono detti *tensori*.

Generalizzando al caso multilineare, un *tensore* di *ordine* n su $V_1, \ldots, V_n$, $n \in \mathbb{N}$, spazi vettoriali, è un elemento del prodotto tensoriale $V_1 \otimes V_2 \otimes \cdots \otimes V_n$.

Un tensore $\mathcal{T} \in V_1 \otimes V_2 \otimes \cdots \otimes V_n$ della forma

$$v_1 \otimes v_2 \otimes \cdots \otimes v_n \text{ con } v_i \in V_i \quad \forall\, i = 1, \ldots, n,$$

viene detto *tensore semplice*.

Un tensore può inoltre essere visto come una generalizzazione dei concetti di scalare, vettore e matrice a strutture con un numero arbitrario di indici: per esempio uno scalare è un tensore di ordine 0, un vettore $v \in V$ è un tensore di ordine 1 e una matrice può essere descritta come un tensore di ordine 2. Come vedremo nella Sezione 2.2.1, la moltiplicazione tra matrici viene convenientemente rappresentata da un tensore *cubico* di ordine 3.

Definizione 2.1.2. Sia $\mathcal{T} \in V_1 \otimes V_2 \otimes \cdots \otimes V_n$ un tensore di ordine n. Diciamo che $\mathcal{T}$ ammette una *decomposizione a rango* R se può essere scritto come combinazione lineare di R tensori semplici, ossia

$$\mathcal{T} = \sum_{r=1}^{R} v_r^{(1)} \otimes v_r^{(2)} \otimes \cdots \otimes v_r^{(n)},$$

dove per ogni r si ha $v_r^{(i)} \in V_i$ per $i = 1, \ldots, n$. Il più piccolo intero R per cui esiste tale rappresentazione è detto *rango* del tensore $\mathcal{T}$ e si scrive $\mathrm{Rank}(\mathcal{T})$.

Una **decomposizione tensoriale** esprime dunque un tensore di ordine n come somma di tensori semplici, ciascuno ottenuto come prodotto tensoriale di vettori. Tale rappresentazione è particolarmente utile perché consente di ridurre un tensore a una combinazione di elementi più facilmente interpretabili, semplificando l'analisi delle sue proprietà strutturali.

2.2 AlphaTensor: la risposta di Google DeepMind

Durante l'ultimo decennio, l'azienda Google DeepMind, compagnia britannica del gruppo Alphabet, si è affermata tra le principali protagoniste nel panorama mondiale in

merito alla ricerca e allo sviluppo dell'intelligenza artificiale. Ciò è stato possibile grazie all'unione delle competenze di due laboratori di ricerca d'eccellenza: Google Brain e DeepMind. Questa fusione, guidata da Demis Hassabis, ha prodotto alcune delle più importanti innovazioni che hanno contribuito a plasmare l'attuale industria dell'IA.

DeepMind è stata fondata nel 2010 da Demis Hassabis, Shane Legg e Mustafa Suleyman. Il laboratorio di ricerca ha contribuito in modo significativo al progresso in ambiti quali il machine learning, le neuroscienze, l'ingegneria, la matematica e la robotica, introducendo parallelamente modalità innovative di organizzazione dell'attività scientifica. Il laboratorio ottenne i primi successi contribuendo in modo pionieristico alla nascita della teoria del *deep reinforcement learning* (DRL) 2.2.2, una combinazione delle tecniche di apprendimento profondo (*deep learning*) e apprendimento per rinforzo (*reinforcement learning*), e utilizzando i giochi come ambiente di test per i propri sistemi. Una delle sue prime innovazioni significative fu DQN, un programma in grado di imparare a giocare a 49 diversi videogiochi Atari partendo completamente da zero, semplicemente osservando i pixel sullo schermo e ricevendo come unico obiettivo quello di massimizzare il punteggio. Nel 2014 DeepMind è stata acquisita da Google. Diventando Google DeepMind l'azienda è cresciuta notevolmente, aumentando di più di dieci volte l'organico di ricercatori. Ma il vero punto di svolta mediatico e scientifico è avvenuto nel 2016 con AlphaGo, il primo programma capace di sconfiggere l'allora campione mondiale nel gioco del Go, Lee Sedol. I successori di AlphaGo, come AlphaZero e MuZero, hanno esteso questo paradigma a una gamma crescente di problemi, tra cui la compressione video e la scoperta automatica di nuovi algoritmi. Ed è proprio sull'orma di AlphaZero che nel 2022 viene presentato AlphaTensor, l'IA che si pone come obiettivo la scoperta di nuovi algoritmi per la moltiplicazione fra matrici. In particolare, DeepMind trasforma il problema della scoperta di algoritmi efficienti per la moltiplicazione matriciale in un gioco per giocatore singolo (*single-player game*) chiamato **TensorGame**, il cui tabellone di gioco è un tensore tridimensionale e il giocatore è un *agente IA* di nome AlphaTensor. Attraverso un insieme di mosse consentite, corrispondenti alle istruzioni dell'algoritmo, il giocatore tenta di modificare il tensore e azzerarne i valori. Quando il giocatore ci riesce, il risultato è un algoritmo di moltiplicazione di matrici dimostrabilmente corretto per qualsiasi coppia di matrici, e la sua efficienza è misurata dal numero di passaggi necessari per azzerare il tensore.

Nella prossima sezione verranno analizzati l'ambiente di gioco e le regole principali del TensorGame.

Tutti i contenuti trattati d'ora in avanti su AlphaTensor sono basati, salvo diversa indicazione, sull'articolo pubblicato da DeepMind sulla rivista *Nature* (2022) e sul blog

tecnico divulgativo di DeepMind, rispettivamente [8] e [7] in bibliografia.

2.2.1 Algoritmi come decomposizioni tensoriali: il TensorGame

La ricerca automatica di algoritmi in AlphaTensor si basa sulla riformulazione del problema della moltiplicazione matriciale come problema di decomposizione tensoriale. La moltiplicazione tra due matrici di ordine n a coefficienti in una campo $\mathbb{K}$ è infatti un'applicazione bilineare $M_n(\mathbb{K}) \times M_n(\mathbb{K}) \longrightarrow M_n(\mathbb{K})$ tale che $(A, B) \mapsto A \cdot B$; dunque corrisponde (fattorizzando grazie alla proprietà universale del prodotto tensoriale) a un'applicazione lineare $M_n(\mathbb{K}) \otimes M_n(\mathbb{K}) \longrightarrow M_n(\mathbb{K})$ che a sua volta può essere interpretata come un elemento del prodotto tensoriale $M_n(\mathbb{K})^* \otimes M_n(\mathbb{K})^* \otimes M_n(\mathbb{K})$, utilizzando il fatto che $\mathrm{Hom}(V, W) \cong V^* \otimes W$, dove il simbolo $*$ a indice di uno spazio vettoriale rappresenta lo spazio *duale* corrispondente[1]. Con la base canonica di $M_n(\mathbb{K})$, tale spazio è isomorfo allo spazio prodotto tensoriale $M_n(\mathbb{K}) \otimes M_n(\mathbb{K}) \otimes M_n(\mathbb{K})$. Dunque la moltiplicazione tra due matrici di ordine n può essere interpretata come un tensore $\mathcal{T}_n \in M_n(\mathbb{K}) \otimes M_n(\mathbb{K}) \otimes M_n(\mathbb{K})$ di ordine 3 e di dimensione $n^2 \times n^2 \times n^2$, con $n^2 = \dim(M_n(\mathbb{K}))$.

Vediamo un esempio di rappresentazione tensoriale della moltiplicazione tra due matrici entrambe di dimensione 2×2.

Esempio 2.2.1. Siano $A, B \in M_2(\mathbb{R})$ e sia $C = AB$ la matrice risultante dal loro prodotto. Allora il processo di moltiplicazione tra A e B può essere descritto come in Figura 2.1 dal tensore tridimensionale $\mathcal{T}_2$, dove ogni dimensione del cubo rappresenta una delle tre matrici coinvolte nell'operazione.

Come visto nella Sezione 1.2, la matrice prodotto C è costruita combinando gli elementi delle matrici A e B come mostrato in Figura 2.2. Gli elementi del tensore $\mathcal{T}_2$ assumono come valori possibili unicamente **0** (in grigio in Figura 2.2) e **1** (in arancione in Figura 2.2).

Le entrate del tensore uguali a uno rappresentano quali moltiplicazioni tra elementi di A e B sono necessarie per calcolare il rispettivo elemento di C. Per esempio, per calcolare l'elemento C_1 della matrice prodotto, l'algoritmo standard richiede di effettuare il calcolo $A_1B_1 + A_2B_3$. Dunque in Figura 2.2 si vedono, nella facciata frontale del cubo (sezione del tensore corrispondente all'elemento C_1 della matrice prodotto), colorati di arancione gli elementi di intersezione fra la colonna corrispondente all'elemento A_1 con la riga di B_1

[1] Sia V uno spazio vettoriale su un campo $\mathbb{K}$. Si definisce spazio *duale* di V, e si indica con V^*, l'insieme di tutti gli omomorfismi da V a $\mathbb{K}$, ossia $V^* = \mathrm{Hom}(V, \mathbb{K})$. Gli elementi di V^* sono detti *funzionali lineari*.

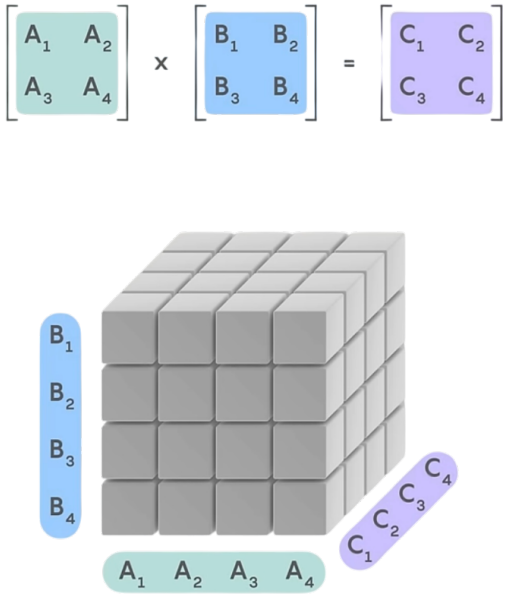


Figura 2.1: *Rappresentazione tensoriale della moltiplicazione tra due matrici di ordine $n = 2$. Immagine tratta da [4].*

(primo prodotto), e la colonna corrispondente a A_2 con la riga di B_3 (secondo prodotto). Si procede in egual maniera per ogni elemento della matrice C.

Dopo aver rappresentato la moltiplicazione di matrici con il tensore tridimensionale $\mathcal{T}_2$, possiamo applicare la decomposizione tensoriale per destrutturare il tensore in elementi di rango uno, ossia possiamo iniziare a giocare al TensorGame. Un metodo naturale basato sull'algoritmo standard di moltiplicazione matriciale per scomporre il tensore $\mathcal{T}_2$ consiste nello scrivere il tensore come somma di tensori di rango uno, ciascuno dei quali descrive una delle moltiplicazioni necessarie per calcolare gli elementi della matrice prodotto. Se per esempio si vuole rappresentare con un tensore M_1 di rango uno lo step moltiplicativo A_1B_1, necessario per calcolare C_1, si effettua il prodotto tensoriale tra i vettori rappresentanti gli elementi considerati. Dunque si avrà

$$M_1 = \underbrace{\begin{pmatrix}1\\0\\0\\0\end{pmatrix}}_{A} \otimes \underbrace{\begin{pmatrix}1\\0\\0\\0\end{pmatrix}}_{B} \otimes \underbrace{\begin{pmatrix}1\\0\\0\\0\end{pmatrix}}_{C}.$$

Procedendo in modo analogo per ogni moltiplicazione presente nell'algoritmo standard, ossia due moltiplicazioni per ognuno dei quattro elementi di C per un totale di otto moltiplicazioni, si ottiene una decomposizione tensoriale di $\mathcal{T}_2$ in otto tensori di rango

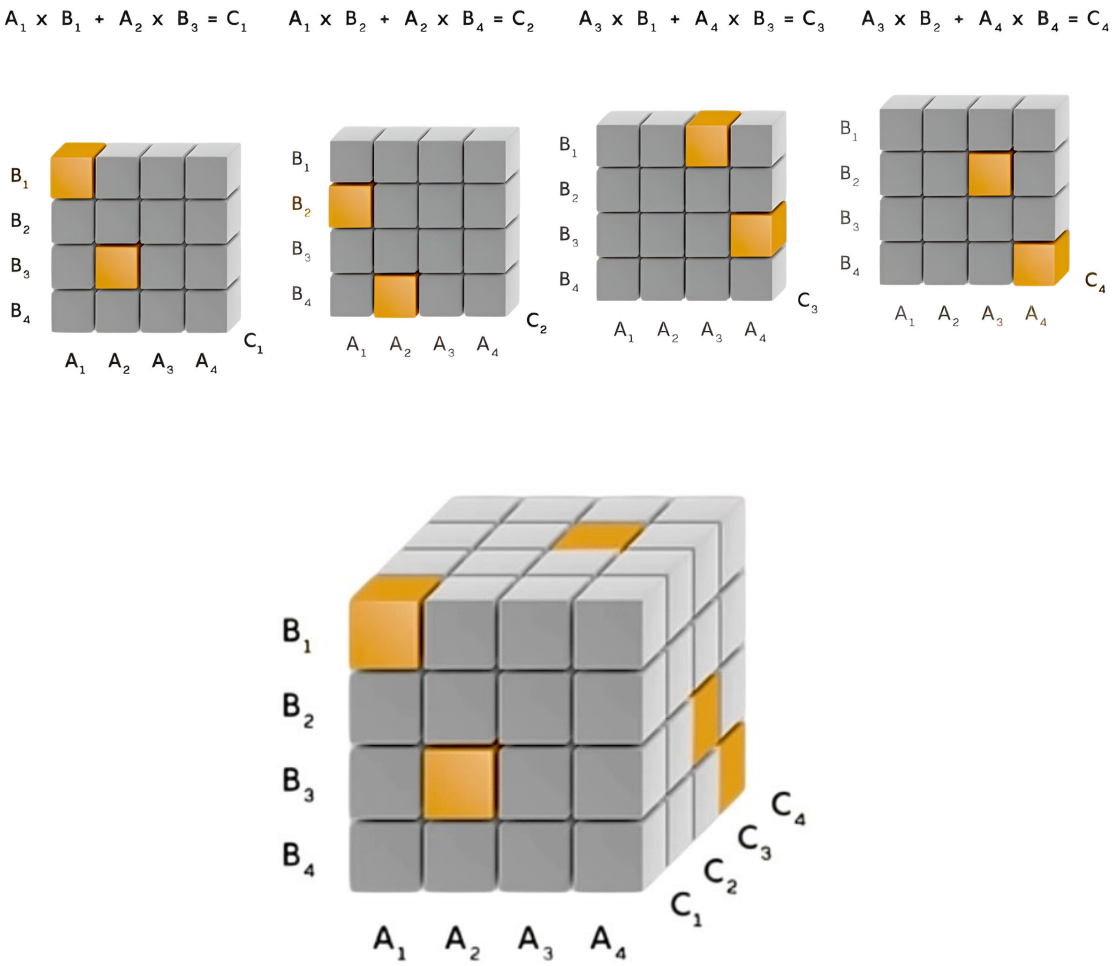


Figura 2.2: *Rappresentazione del tensore $\mathcal{T}_2$. Immagini tratte da [4].*

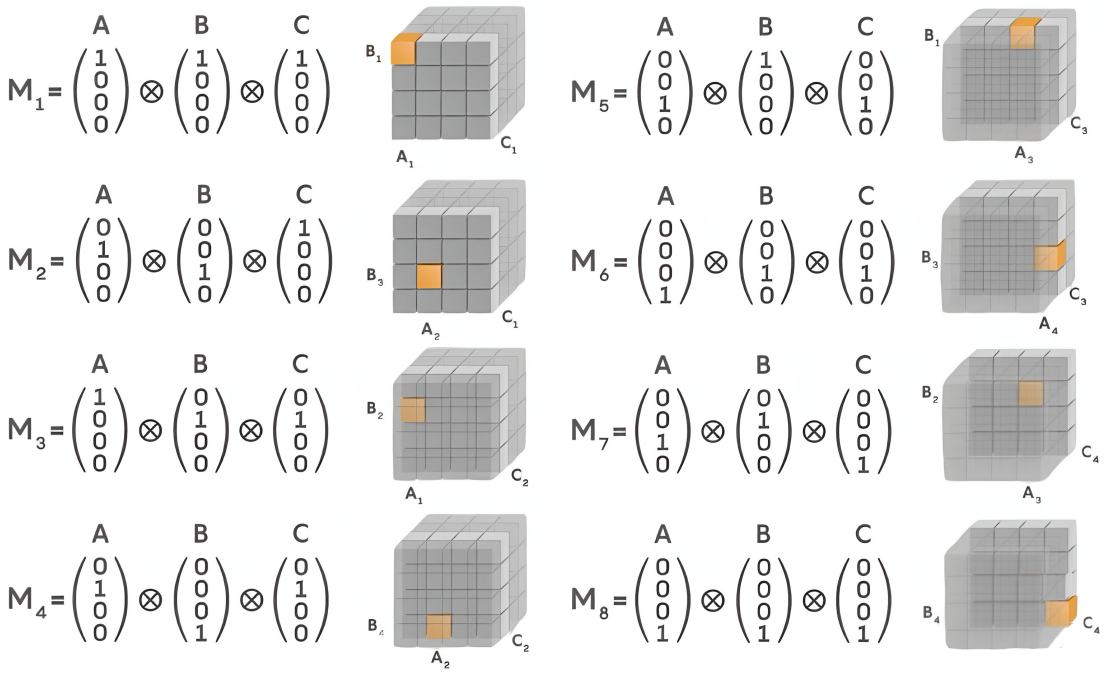


Figura 2.3: *Decomposizione tensoriale standard di $\mathcal{T}_2$. Immagine tratta da [4].*

uno. Dunque $\mathcal{T}_2 = M_1 + M_2 + ... + M_8 = \sum_{i=1}^{8} u_i^{(1)} \otimes v_i^{(1)} \otimes w_i^{(i)}$ dove $u_i, v_i, w_i \in \mathbb{R}^4$ e ogni tripla di vettori (u_i, v_i, w_i) rappresenta una delle otto moltiplicazioni scalari necessarie per costruire C, come mostrato in Figura 2.3.

Se una tale decomposizione esiste, allora possiamo costruire un algoritmo di moltiplicazione tra matrici che utilizza esattamente R moltiplicazioni scalari. Pertanto, **trovare decomposizioni di rango minimo equivale a cercare algoritmi più efficienti** per la moltiplicazione matriciale, dove l'efficienza è misurata dal numero di moltiplicazioni necessarie. Più in generale, si può dimostrare che il rango del tensore della moltiplicazione tra matrici di dimensione $n \times n$ fornisce il limite inferiore al numero di moltiplicazioni necessarie e dunque un obiettivo teorico per la complessità computazionale dell'operazione.

Come visto nella Sezione 1.3, nel caso delle matrici di ordine 2, Volker Strassen ha trovato un algoritmo in grado di calcolare la matrice prodotto con sette moltiplicazioni scalari al posto delle usuali otto del metodo standard. Si può dunque affermare che abbia scoperto una decomposizione tensoriale di $\mathcal{T}_2$ (tensore dell'esempio 2.2.1) in soli sette termini di rango uno, dimostrando $Rank(\mathcal{T}_2) \leq 7$. Di seguito sono riportati l'algoritmo di Strassen (Figura 2.4) e la corrispondente decomposizione tensoriale (Figura 2.5).

Come dichiarato da Alhussein Fawzi, ricercatore del team di Google DeepMind e

$$
\begin{aligned}
m_1 &= (a_1 + a_4)(b_1 + b_4) \\
m_2 &= (a_3 + a_4)\, b_1 \\
m_3 &= a_1\, (b_2 - b_4) \\
m_4 &= a_4\, (b_3 - b_1) \\
m_5 &= (a_1 + a_2)\, b_4 \\
m_6 &= (a_3 - a_1)(b_1 + b_2) \\
m_7 &= (a_2 - a_4)(b_3 + b_4) \\
c_1 &= m_1 + m_4 - m_5 + m_7 \\
c_2 &= m_3 + m_5 \\
c_3 &= m_2 + m_4 \\
c_4 &= m_1 - m_2 + m_3 + m_6
\end{aligned}
$$

Figura 2.4: *Algoritmo di Strassen.*

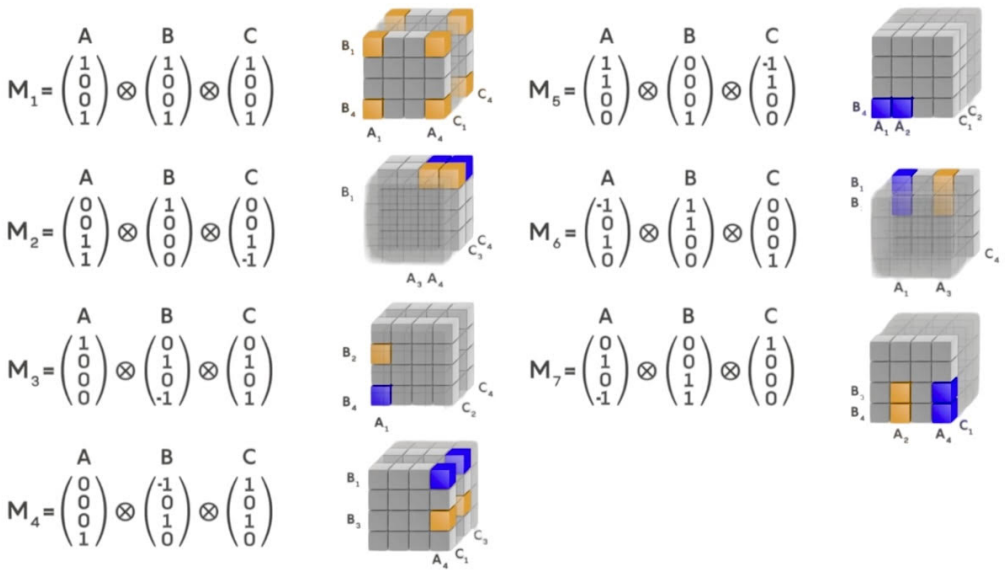


Figura 2.5: *Decomposizione di $\mathcal{T}_2$ secondo Strassen. In blu sono raffigurate le entrate di valore -1. Immagine tratta da [4].*

sviluppatore di AlphaTensor, all'interno di una video-intervista rilasciata a QuantaMagazine[2], l'esigenza di trasferire il problema dalla moltiplicazione tra matrici alla decomposizione di tensori deriva dalla necessità di ridurre e chiarire l'ambiente di ricerca per facilitare il compito all'intelligenza artificiale. La richiesta "*trova un algoritmo per questa moltiplicazione tra matrici che necessiti il minor numero di moltiplicazioni possibili*" non è infatti adatta per un algoritmo di *machine learning*: il richiedente non esplicita che tipo di struttura dovrebbe avere l'algoritmo, gli strumenti di ricerca adatti per affrontare il quesito e fornisce come unico riferimento matematico le matrici da moltiplicare. Al contrario, se la richiesta diventa "*scomponi il seguente tensore tridimensionale nel minor numero possibile di tensori di rango uno*", la struttura dell'algoritmo diventa chiara (una decomposizione tensoriale di $\mathcal{T}$), l'obiettivo preciso (minimizzare R) e l'ambiente di ricerca definito (le combinazioni possibili di decomposizioni). Inoltre in questo caso, le azioni dell'intelligenza artificiale saranno limitate alla scelta di aggiungere o sottrarre tensori di rango uno, e ogni azione genererà una ricompensa o una penalità garantendo costanti feedback alla macchina che dunque tenterà di risolvere un compito preciso e quantificabile. Questi aspetti legati al funzionamento dell'IA verranno approfonditi nella prossima sezione.

Riformulando il problema della moltiplicazione matriciale come un problema di decomposizione tensoriale, è lecito riformulare anche la domanda motivante l'intera trattazione: *esistono decomposizioni tensoriali migliori di quelle finora scoperte per $\mathcal{T}_2$? E in generale per $\mathcal{T}_n$?*

Per tentare di rispondere a queste domande, gli sviluppatori di DeepMind hanno creato un agente IA che potesse sconfiggere Strassen al TensorGame: AlphaTensor.

2.2.2 Fondamenti teorici: DNN, DRL e MCTS

AlphaTensor si fonda sull'integrazione tra una rete neurale profonda (*deep neural network*, DNN) e una procedura di ricerca ad albero chiamata *Monte Carlo Tree Search* (MCTS), all'interno di un quadro di *deep reinforcement learning* (DRL), concepito per esplorare in modo efficiente lo spazio delle possibili mosse nel *TensorGame*. Nei seguenti paragrafi verranno presentati tali concetti legati al *machine learning* come descritti in [1] e [27], al fine di poter, nell'ultima sezione del presente capitolo, studiare le specifiche dell'agente IA AlphaTensor.

[2]Link all'articolo contenente il video *How AI discovered a faster matrix multiplication algorithm* dove interviene Alhussein Fawzi: `https://www.quantamagazine.org/ai-reveals-new-possibilities-in-matrix-multiplication-20221123/`

Reti neurali profonde

Le **reti neurali profonde** (DNN) sono modelli computazionali ispirati alla struttura e al funzionamento del cervello umano, composti da molteplici strati (*layers*) di unità elementari dette neuroni artificiali.

Ogni livello della rete riceve in ingresso (*input*) il vettore delle attivazioni prodotte dal livello precedente e restituisce in uscita (*output*) una nuova rappresentazione, ottenuta applicando una trasformazione non lineare. Dunque ogni neurone è connesso ad ogni altro neurone dello strato successivo, il che significa che il suo valore di output diventa l'input per i prossimi neuroni.

Formalmente, una DNN implementa una funzione

$$F_\theta : \mathbb{R}^n \longrightarrow \mathbb{R}^m,$$

parametrizzata da un insieme θ di pesi e bias, che determina il comportamento della rete.

Il **peso** (*weight*) è un coefficiente reale che modula l'influenza di un ingresso nella determinazione dell'attivazione di un neurone: valori elevati in modulo implicano un'influenza significativa, valori prossimi allo zero riducono o annullano il contributo dell'ingresso. Il peso della connessione influenza dunque quanto input viene trasmesso da un neurone all'altro. Il **bias**, invece, è un termine scalare aggiunto alla somma pesata degli ingressi, che permette di traslare la funzione di attivazione lungo l'asse orizzontale. In tal modo, il bias modifica la soglia di attivazione del neurone e consente di rappresentare funzioni che non siano vincolate a passare per l'origine, aumentando la flessibilità del modello. Una rete neurale è in grado di generalizzare e modellare un problema proprio grazie al costante aggiornamento di pesi e bias, che modulano l'input e l'output di ogni singolo neurone. Il calcolo dell'uscita di un neurone j nello strato ℓ avviene secondo la relazione

$$z_j^{(\ell)} = \sigma\left(\sum_i w_{ij}^{(\ell)}\, x_i^{(\ell-1)} + b_j^{(\ell)}\right),$$

dove $x_i^{(\ell-1)}$ è l'uscita del neurone i nello strato precedente, $w_{ij}^{(\ell)}$ il peso della connessione e $b_j^{(\ell)}$ il bias. L'output di un neurone è dunque espresso da una formula del tipo $output = \sigma(inputs \times weights + bias)$ come mostrato in Figura 2.6. La funzione σ, detta *funzione di attivazione*, è una mappa non lineare $\sigma : \mathbb{R} \longrightarrow \mathbb{R}$, applicata all'uscita di ciascun neurone, fondamentale per dare potenza espressiva alla rete neurale. Senza una funzione di attivazione non lineare, infatti, anche una rete con molti strati si ridurrebbe a una singola trasformazione lineare dell'input, perdendo la capacità di modellare fenomeni complessi. Tra le funzioni di attivazione più utilizzate si trovano la funzione ReLU

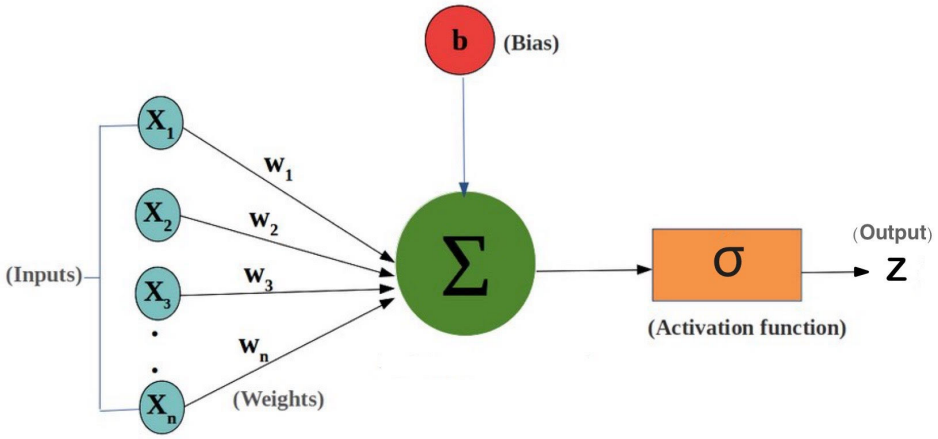


Figura 2.6: *Attivazione di un neurone.*

(definita come $\sigma(z) = \max(0, z)$), la sigmoide ($\sigma(z) = \frac{1}{1+e^{-z}}$) e la tangente iperbolica ($\sigma(z) = \tanh(z)$).

L'aggettivo *profonda*, riferito a una rete neurale, indica la presenza di più strati nascosti (*hidden layers*) tra l'input e l'output finale della rete. Questa architettura consente di apprendere rappresentazioni gerarchiche dei dati: gli strati iniziali estraggono caratteristiche di base mentre quelli successivi combinano tali caratteristiche in concetti di ordine superiore, fino ad arrivare all'output finale. Questo progressivo affinamento della rappresentazione prende il nome di *astrazione gerarchica* e costituisce una delle caratteristiche fondamentali che rendono le DNN adatte ad affrontare compiti complessi in domini quali, ad esempio, il riconoscimento visivo, la comprensione di segnali audio e la ricerca di algoritmi matematici.

Il processo di addestramento di una DNN si articola in una sequenza iterativa di passaggi volti a ridurre l'errore di previsione del modello, ossia la discrepanza tra il valore desiderato (*target*) e l'output reale della rete. La prima fase è la propagazione in avanti (*forward pass*), durante la quale il vettore input attraversa tutti gli strati della rete descritti in precedenza. L'output dell'ultimo strato rappresenta la predizione della rete per il dato in ingresso. Segue il calcolo della funzione di perdita (*loss function*) $\mathcal{L}$, che misura la discrepanza tra la predizione della rete e il target. La seconda fase - la più importante nel processo di addestramento di una DDN - è la retropropagazione dell'errore (*backpropagation*). Tale tecnica di addestramento è stata introdotta intorno al 1960, ma la sua importanza è stata compresa solo successivamente. Nel 1986 gli scienziati David Rumelhart, Geoffrey Hinton e Ronald Williams pubblicarono un influente lavoro intitolato *Learning Representations by Back-Propagating Errors* [21]. Questo studio diede nuova vita al campo delle reti neurali, poiché dimostrava come la retropropagazione dell'errore

potesse rendere efficiente l'addestramento di reti complesse. Fino ad allora, uno dei problemi principali delle reti neurali era l'incapacità di apprendere in modo efficace a causa dell'enorme numero di parametri da regolare. Grazie a tale scoperta questa sfida è stata portata con successo a termine, diventando uno dei fondamenti dell'addestramento delle reti neurali. Il termine *backpropagation* indica un algoritmo matematico il cui obiettivo è determinare come pesi e bias del modello debbano essere regolati per minimizzare l'errore misurato tramite la *loss function*. A livello matematico, mira a calcolare il gradiente della funzione di perdita rispetto a ciascuno dei singoli parametri della rete neurale, utilizzando la regola della catena per calcolare la velocità con cui la perdita cambia in risposta a qualsiasi modifica di un peso (o bias) specifico nella rete. Questo passaggio consente di determinare in che misura ogni parametro abbia contribuito all'errore complessivo. Il cervello umano può maturare formando nuove connessioni e modificando quelle esistenti attraverso la plasticità sinaptica, le reti neurali artificiali imitano questo processo regolando i pesi durante l'addestramento proprio grazie alla fase di *backpropagation*. Infine, nella fase di aggiornamento dei parametri, si modificano pesi e bias nella direzione opposta al gradiente per ridurre l'errore. Questo aggiornamento avviene tramite algoritmi di ottimizzazione, tra i quali la *stochastic gradient descent* (SGD) è il più semplice e diffuso. Varianti più avanzate, come Adam e RMSprop, introducono meccanismi di adattamento del tasso di apprendimento e di accumulo di stime dei gradienti, migliorando la velocità di convergenza e la stabilità dell'addestramento. Questo processo – *forward pass*, calcolo della perdita, *backpropagation* e aggiornamento dei parametri – viene ripetuto per molte epoche fino a quando la rete raggiunge prestazioni soddisfacenti sul set di validazione.

Le DNN sono spesso impiegate in contesti di apprendimento supervisionato, dove l'addestramento avviene su coppie input-output note. Tuttavia, possono essere adattate anche ad approcci non supervisionati o di apprendimento per rinforzo, come nel caso di AlphaTensor, nei quali la rete ha il compito di stimare funzioni più complesse come politiche decisionali, funzioni valore o modelli di transizione, aprendo la strada alla loro integrazione in sistemi di decisione autonomi.

Deep Reinforcement Learning

Il **Deep Reinforcement Learning** (DRL) è un ambito dell'apprendimento automatico che integra i principi del *reinforcement learning* (RL) con la capacità delle reti neurali profonde di approssimare funzioni complesse e rappresentare strutture ad alta dimensionalità. Questa sinergia consente di affrontare problemi decisionali caratterizzati da spazi di stato e di azione estremamente vasti, nei quali i metodi tabellari del RL tradizionale risultano impraticabili.

Con il termine *reinforcement* si vuole indicare un tipo di apprendimento basato su ricompense, capace di risolvere problemi decisionali in cui, attraverso un meccanismo di tentativi ed errori, l'agente compie interazioni autonome con l'ambiente in cui opera. Dunque a differenza degli approcci supervisionati e non, il RL si basa su un processo di apprendimento iterativo, dove l'agente impara attraverso un sistema di ricompense e punizioni.

Nell'apprendimento per rinforzo, l'interazione tra agente e ambiente è modellata come un *processo decisionale di Markov* (*Markov Decision Process*, MDP), definito dalla quintupla:

$$M = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$$

dove:

- $\mathcal{S}$ è lo spazio degli stati, ossia l'insieme delle possibili configurazioni in cui l'ambiente può trovarsi. In Figura 2.7 un esempio di un robot inserito in un tabellone a griglia. Lo spazio degli stati è l'insieme delle possibili posizioni in cui si può trovare il robot ad ogni step temporale t;

- $\mathcal{A}$ è lo spazio delle azioni disponibili all'agente. Nel nostro esempio rappresenta l'insieme di tutti i movimenti che cambiano lo stato attuale del robot in un altro stato $s \in \mathcal{S}$: *gira a destra, vai avanti, rimani fermo*, ecc.;

- $\mathcal{P}(s' \mid s, a)$ rappresenta la probabilità di transizione dallo stato s allo stato s' in seguito all'esecuzione dell'azione a. Può infatti succedere che non si conosca con esattezza il tipo di movimento del nostro robot, ad esempio c'è una minima probabilità che il passo non sia sufficientemente lungo per spostarsi da una casella all'altra con un solo comando *in avanti*, dunque in seguito alla scelta dell'azione rimanga nel medesimo stato o comunque non raggiunga lo stato desiderato.

- $\mathcal{R}(s, a)$ è la funzione di ricompensa immediata, che associa un valore numerico alla coppia stato–azione. Il robot riceve una ricompensa $r(s, a)$ se compie un azione a da uno stato s. Se la ricompensa è grande, ciò indica che l'azione è stata utile per raggiungere l'obiettivo (ad esempio si sposta in una casella più vicina alla casa verde), se la ricompensa è piccola, significa che l'azione a dallo stato s è stata meno utile al raggiungimento del task. Le ricompense vengono tarate dallo sviluppatore dell'algoritmo di apprendimento per rinforzo, ossia da colui che conosce l'obiettivo del compito dato all'agente;

- $\gamma \in [0, 1]$ è il fattore di sconto che regola l'importanza relativa delle ricompense future rispetto a quelle immediate. Supponiamo che il nostro robot cominci in

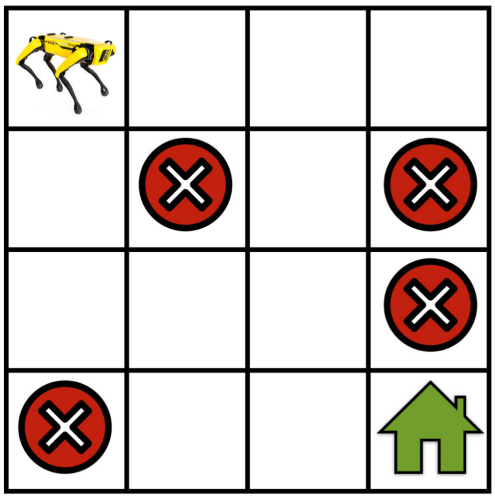


Figura 2.7: *Un esempio di compito: un robot in un tabellone a griglia deve trovare la strada fino alla posizione obiettivo (la casa verde) evitando le posizioni trappola (le croci rosse).*

una particolare casella (stato $s_0 \in S$) e compia diverse azioni fino a compiere una traiettoria $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \cdots)$. Il *ritorno* della traiettoria è la somma totale di tutte le ricompense ottenute ad ogni step t,

$$R(\tau) = r_0 + r_1 + r_2 + \cdots .$$

L'obiettivo è trovare la traiettoria che ottenga il ritorno R maggiore. Pensiamo tuttavia al caso in cui il nostro robot viaggi infinitamente a lungo all'interno della griglia senza mai raggiungere la casa; allora il ritorno risulterebbe infinito. Per evitare tale casistica si introduce il fattore di sconto $\gamma \in [0, 1]$ e si formula il ritorno della traiettoria come

$$R(\tau) = r_0 + \gamma r_1 + \gamma^2 r_2 + ... = \sum_t \gamma^t r_t.$$

Si noti che per γ molto vicino a 0 e t molto grande, la ricompensa ottenuta è fortemente scontata da γ^t. Questo incoraggerà il robot a compiere traiettorie corte che giungano all'obiettivo dopo un numero ragionevolmente basso di azioni. Se poniamo γ molto vicino a 1, il robot sarà portato a esplorare maggiormente per trovare la miglior traiettoria per giungere alla casa.

Dunque al tempo discreto t, l'agente osserva $s_t \in \mathcal{S}$, seleziona un'azione $a_t \in A$, riceve una ricompensa immediata $r_t = R(s_t, a_t)$ e transita in s_{t+1}. Come può il robot scegliere la migliore azione da compiere in ogni posizione per massimizzare il ritorno della traiettoria?

La strategia di un agente dipende dalla politica (*policy*) stocastica, ossia da una famiglia di distribuzioni $\pi(a \mid s)$ che, dato uno stato s, specifica la probabilità di scegliere ciascuna azione a. Può essere stazionaria (dipendente solo dallo stato corrente, non dal tempo), oppure dipendente dalla storia quando lo stato non è direttamente osservabile. Una politica deterministica è un caso speciale nel quale viene assegnata probabilità non nulla a una sola particolare azione e nulla a tutte le altre disponibili. Riprendendo l'esempio del robot, se $\mathcal{A} = \{\text{gira a sinistra, gira a destra, vai verso il basso, vai verso l'alto}\}$ è lo spazio delle azioni disponibili, la politica in una determinata posizione s per questo insieme è una distribuzione le cui probabilità per le quattro azioni potrebbero essere rispettivamente $[0.4, 0.2, 0.1, 0.3]$. Se la politica fosse deterministica potremmo avere $\pi(a \mid s) = [1, 0, 0, 0]$.

Immaginiamo ora che il robot sia posizionato in partenza nella casella s_0 e che ad ogni istante t scelga un'azione $a_t \sim \pi(a_t \mid s_t)$ secondo la politica π e che transiti nel successivo stato s_{t+1}. La traiettoria risultante può variare considerevolmente a seconda delle azioni scelte negli step temporali intermedi. Definiamo quindi il *ritorno medio* come

$$\mathcal{V}^{\pi}(s_0) = \mathbb{E}_{a_t \sim \pi(s_t)}\left[R(\tau)\right] = \mathbb{E}_{a_t \sim \pi(s_t)}\left[\sum_t \gamma^t r_t\right].$$

$\mathcal{V}_{\pi}(s_0)$ viene anche chiamata *funzione valore* per la politica π, ed esprime il ritorno atteso (scontato da γ), ottenuto dal robot iniziando la sua traiettoria dallo stato s_0 e scegliendo istante dopo istante le azioni dettate dalla strategia π.

È spesso utile nelle implementazioni poter calcolare il ritorno atteso di una traiettoria che ha inizio allo stato s_0 e di cui è già stata fissata la prima azione a_0. Questa quantità chiamata *funzione valore stato–azione* (*action–value function*), è definita come

$$\mathcal{Q}^{\pi}(s_0, a_0) = r_0 + \mathbb{E}_{a_t \sim \pi(s_t)}\left[\sum_{t=1}^{\infty} \gamma^t r_t\right].$$

Entrambe le funzioni valore e valore stato–azione dipendono fortemente dalla politica scelta, viene dunque naturale chiedersi quale sia la *politica ottimale* che aiuti l'agente ad ottenere il massimo ritorno medio possibile. Definiamo dunque

$$\pi^* = \arg\max_{\pi} \mathcal{V}^{\pi}(s_0),$$

ossia la miglior politica utile al raggiungimento dell'obiettivo.

L'esempio del robot che deve raggiungere la casa è un esempio di compito che ha un inizio e una fine prestabilite, ossia è un compito definito come *episodico*, dove l'agente compie una sequenza di azioni volta a terminare in uno stato finale s_n. Nelle applicazioni succede spesso di dover implementare algoritmi di addestramento per rinforzo profondo

per compiti *continui*, cioè processi che non hanno una fine definita, ma interagiscono costantemente con l'ambiente senza raggiungere uno stato finale. Un esempio è un sistema di controllo di un impianto industriale, il cui obiettivo è ottimizzare continuamente il processo produttivo. La distinzione tra compiti episodici e continui è importante perché influenza il modo in cui gli algoritmi di apprendimento sono progettati.

I principali sono:

- **Value-based**: la rete neurale approssima la funzione $Q^\pi(s, a)$ e la politica viene derivata scegliendo le azioni con valore massimo (es. *Deep Q-Network*, DQN);

- **Policy-based**: la rete neurale approssima direttamente una politica parametrizzata $\pi(a|s)$ massimizzando il ritorno atteso;

- **Actor–Critic**: combina i due metodi precedenti, ossia un *actor* (politica π) propone le azioni, un *critic* stima una funzione valore (ossia valuta gli stati o le coppie stato–azione).

Nei problemi episodici risulta spesso più agevole adottare metodi value-based o policy-based con obiettivi definibili sull'intero episodio. Nei compiti continui, invece, la necessità di contenere varianza e instabilità degli aggiornamenti motiva l'impiego di schemi actor–critic, che, combinando una stima del valore con una policy parametrica, sfruttano il *bootstrapping*, ossia l'utilizzo di stime ottenute usando ricompense immediate e una stima corrente del valore futuro, per aggiornamenti più frequenti e stabili.

Un altro aspetto cruciale del DRL è il bilanciamento tra *esplorazione* e *sfruttamento*. L'esplorazione consente all'agente di provare strategie nuove e potenzialmente più redditizie, mentre lo sfruttamento gli permette di utilizzare le conoscenze già acquisite per massimizzare la ricompensa. Il compromesso tra questi due aspetti sta nell'equilibrio che l'agente deve mantenere. Se l'agente esplora eccessivamente, rischia di non ottenere ricompense immediate, ma potrebbe scoprire strategie più efficaci utili in futuro. Se, invece, si concentra sullo sfruttamento tralasciando la fase di esplorazione, potrebbe perdere l'opportunità di imparare nuove strategie più redditizie. Questo equilibrio è gestito mediante tecniche come $\epsilon -$ greedy, *entropy regularization* o strategie di esplorazione guidata dall'incertezza.

Grazie a queste caratteristiche, il DRL ha raggiunto risultati notevoli in una vasta gamma di applicazioni, tra cui giochi complessi, controllo di sistemi fisici, pianificazione in ambienti ad alta dimensionalità e problemi di ottimizzazione combinatoria.

Monte Carlo Tree Search

Il **Monte Carlo Tree Search** (MCTS) è un algoritmo ideato per affrontare problemi caratterizzati da spazi decisionali estremamente ampi. Invece di esplorare esaustivamente tutte le mosse disponibili, il MCTS costruisce progressivamente un albero di ricerca eseguendo simulazioni casuali (*rollouts*) per orientare le proprie scelte. L'algoritmo bilancia la fase di esplorazione di nuove possibilità con lo sfruttamento dei percorsi già identificati come promettenti, concentrando così lo sforzo computazionale nelle aree più rilevanti dello spazio di ricerca e risultando estremamente efficiente in compiti decisionali non banali. È stato originariamente sviluppato per affrontare giochi combinatori a elevata complessità, come il Go, ma ha trovato applicazione in numerosi ambiti, dalla pianificazione robotica all'ottimizzazione combinatoria, fino all'integrazione con metodi di *reinforcement learning*.

L'idea di base consiste nell'esplorare lo spazio delle decisioni costruendo progressivamente un albero di ricerca in cui i nodi rappresentano stati dell'ambiente e gli archi corrispondono alle azioni disponibili. A differenza delle tecniche di ricerca esaustiva, il MCTS non richiede di esplorare completamente l'albero: esso privilegia le porzioni più promettenti, bilanciando l'esplorazione di mosse poco provate e lo sfruttamento di quelle che, in base alle simulazioni, hanno prodotto risultati favorevoli.

L'algoritmo MCTS si articola in quattro fasi principali, ripetute iterativamente, come mostrato in Figura 2.8.

1. **Selection**: a partire dalla radice, si seleziona un percorso discendendo lungo l'albero secondo una regola di scelta che bilancia esplorazione e sfruttamento. Una delle strategie più utilizzate è l'algoritmo UCT (*Upper Confidence Bound applied to Trees*), che seleziona l'azione a nello stato s massimizzando:

$$\mathrm{UCT}(s,a) = \bar{\mathcal{Q}}(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}}$$

   dove ($N(s)$ è il numero di visite al nodo s, $N(s,a)$ è il numero di volte che l'azione a è stata scelta nello stato s, $C > 0$ è un parametro che regola il bilanciamento tra esplorazione e sfruttamento e $\bar{\mathcal{Q}}(s,a)$ è un'approssimazione della funzione valore stato–azione presentata in precedenza;

2. **Expansion**: quando la fase di *selection* raggiunge un nodo non ancora completamente espanso (cioè con azioni non ancora esplorate), l'algoritmo espande l'albero aggiungendo uno o più nodi figli che rappresentano le possibili azioni eseguibili da quello stato;

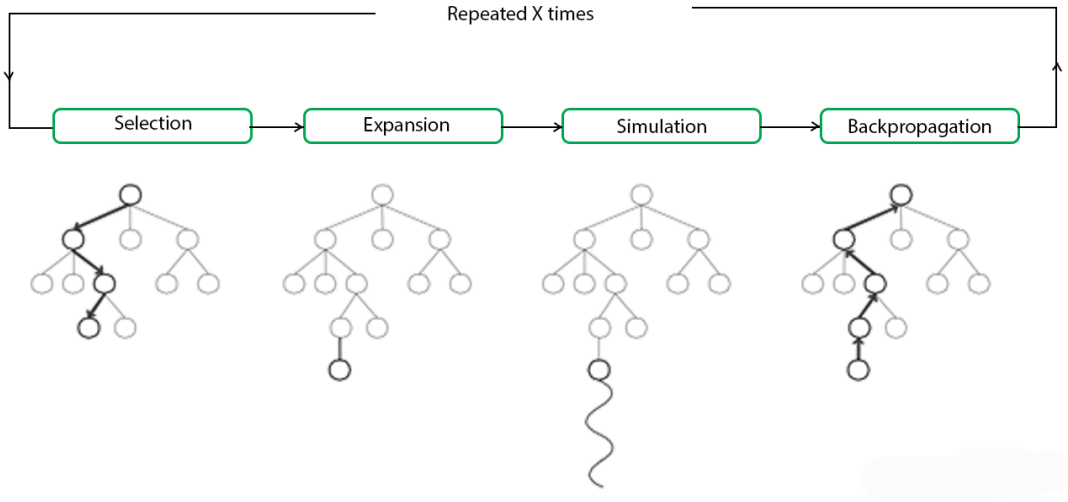


Figura 2.8: *Fasi del metodo di ricerca ad albero Monte Carlo Tree Search. Immagine tratta da [11].*

3. **Simulation**: a partire dal nuovo nodo, si esegue una simulazione (*playout*) fino a uno stato terminale o fino a un orizzonte prefissato, stimando l'esito mediante una politica casuale o euristica, così da mantenere basso il costo computazionale. Questa fase fornisce una stima grezza del valore del nodo appena aggiunto;

4. **Backup**: il risultato della simulazione viene propagato all'indietro lungo il percorso selezionato, aggiornando le statistiche $(\mathcal{Q}(s,a), N(s), N(s,a))$ di tutti i nodi visitati.

Ripetendo questo ciclo un numero sufficiente di volte, l'albero di ricerca si arricchisce di informazioni e le scelte alla radice diventano sempre più affidabili. Dunque anche per il MCTS il compromesso esplorazione–sfruttamento è di fondamentale importanza. In particolare, l'algoritmo UCT fornisce un meccanismo matematico per regolare questo bilanciamento, assicurando che tutte le azioni vengano esplorate ma con frequenza proporzionale al loro valore stimato.

Il MCTS presenta diversi vantaggi: non richiede una funzione di valutazione perfetta (potendo operare con simulazioni approssimate), è *anytime*, cioè può essere interrotto in qualunque momento fornendo la migliore stima disponibile fino a quel punto, e si adatta bene a problemi con spazi di stati di grandi dimensioni e ramificazione variabile.

Tuttavia, presenta anche limitazioni: l'efficienza dipende fortemente dalla qualità delle simulazioni e, in problemi con *branching factor* molto elevato, può risultare oneroso in termini computazionali.

Negli approcci moderni di DRL, il MCTS viene spesso potenziato integrando valutazioni derivate da reti neurali profonde. In questo contesto, la rete fornisce una stima della funzione valore per ridurre la necessità di simulazioni complete e una politica iniziale per guidare l'espansione, concentrando le risorse computazionali sulle linee di gioco più promettenti. Questa sinergia consente al MCTS di mantenere il proprio comportamento esplorativo, beneficiando al contempo della capacità predittiva e generalizzante delle DNN.

2.2.3 AlphaTensor

Come anticipato nella Sezione 2.2.1, il team di DeepMind riformula il problema di moltiplicazione matriciale come un problema di apprendimento per rinforzo, modellando l'ambiente come un gioco per giocatore singolo, il *TensorGame*.

La posizione di partenza $\mathcal{S}_0$ del gioco corrisponde al tensore $\mathcal{T}$ che rappresenta l'operazione bilineare di interesse, espressa in una qualsiasi base (si veda 2.2.1). A ogni passo t del gioco, il giocatore seleziona una terna di vettori $(\mathbf{u}^{(t)}, \mathbf{v}^{(t)}, \mathbf{w}^{(t)})$ e il tensore $\mathcal{S}_t$ viene aggiornato sottraendo il tensore di rango uno risultante:

$$\mathcal{S}_t \leftarrow \mathcal{S}_{t-1} - \mathbf{u}^{(t)} \otimes \mathbf{v}^{(t)} \otimes \mathbf{w}^{(t)}$$

L'obiettivo del giocatore è raggiungere il tensore $\mathcal{S}_t = 0$ con il minor numero di mosse. Quando il giocatore raggiunge il tensore nullo, i fattori trovati durante il gioco vanno a formare una fattorizzazione del tensore iniziale

$$\mathcal{S}_0 = \sum_{t=1}^{R} \mathbf{u}^{(t)} \otimes \mathbf{v}^{(t)} \otimes \mathbf{w}^{(t)},$$

dove R indica il numero di mosse effettuate. Per evitare di giocare partite inutilmente lunghe, viene limitato il numero di passaggi a un valore massimo, R_{limite}.

Per ogni passo compiuto, viene fornita una penalità di -1 per incoraggiare la ricerca del percorso più breve verso il tensore nullo. Se dopo R_{limite} passi il gioco termina con un tensore diverso da zero, l'agente riceve una penalità finale aggiuntiva pari a $-\gamma(\mathcal{S}_{R_{\text{limite}}})$, dove $\gamma(\mathcal{S}_{R_{\text{limite}}})$ è un limite superiore per il rango del tensore finale. Se il tensore residuo ha rango alto, si avrà una penalità maggiore; se ha rango basso, verrà assegnata una penalità meno severa. Sebbene questa ricompensa ottimizzi per il rango e quindi per la complessità dell'algoritmo risultante, altri schemi di ricompensa possono essere utilizzati per ottimizzare altre proprietà, come per esempio il *runtime*, ossia il tempo di esecuzione dell'algoritmo di moltiplicazione matriciale.

Architettura della rete neurale

Superare le sfide poste dal TensorGame, ovvero un enorme spazio d'azione e stati di gioco descritti da tensori che rappresentano un'operazione bilineare, ha richiesto grandi riflessioni sul tipo di architettura neurale da utilizzare. Per AlphaTensor, DeepMind ha sviluppato un'architettura basata sul *Transformer* [25], che incorpora bias induttivi per gli input tensoriali.

Fino a non molti anni fa, i migliori modelli per compiti *sequence-to-sequence* (come per esempio la traduzione tra lingue) erano principalmente reti neurali ricorrenti o reti neurali convoluzionali. Le reti neurali ricorrenti, sostanzialmente, leggono la sequenza in input un elemento alla volta, mantenendo una memoria del passato che influenza l'elaborazione del presente, rendendola adatta a traduzioni di testo e trascrizioni audio. Proprio per questa caratteristica di lettura ordinata elemento dopo elemento, le reti ricorrenti non sono adatte a modellare spazi di dati con dipendenze particolarmente lunghe o che necessitano di un'analisi parallela. Le reti convoluzionali, al contrario, sono architetture in cui lo strato principale applica filtri convoluzionali (*kernel*) che estraggono caratteristiche locali da un input strutturato (tipicamente immagini o sequenze). Il limite di tali reti è che una convoluzione vede solo una finestra locale di input (definita dalla dimensione del *kernel*), dunque per catturare dipendenze a lungo raggio bisogna combinare molti *kernel* o aumentare la dimensione dei filtri, con conseguente crescita di parametri e complessità.

Nel 2017, l'informatico Ashish Vaswani insieme a un team di ricercatori di Google Brain e Google Research, con il lavoro *Attention Is All You Need* [25], presenta un'innovativa architettura chiamata *Transformer*. L'idea centrale è utilizzare meccanismi di *self-attention* per modellare le relazioni tra gli elementi in input, eliminando convoluzioni e ricorrenze. Il modello è composto da due componenti principali, encoder e decoder, ciascuno organizzato in più strati. L'encoder riceve in ingresso la sequenza (ad esempio una frase o un tensore) e la trasforma in una rappresentazione interna arricchita di contesto: ogni elemento della sequenza non viene più descritto solo dalle proprie caratteristiche locali, ma viene riposizionato in uno spazio latente che tiene conto anche delle relazioni con tutti gli altri elementi. In questo modo l'encoder produce una sequenza di vettori che incorpora sia l'informazione originaria sia la struttura di dipendenze tra le parti dell'input.

Il decoder utilizza questa rappresentazione prodotta dall'encoder per generare la sequenza di output in maniera autoregressiva. Ad ogni step, il decoder combina due fonti di informazione: da un lato la rappresentazione contestualizzata dell'encoder, che fornisce una mappa globale dell'input, dall'altro la parte di output già generata, che assicura

coerenza e continuità nella sequenza. Grazie a questo meccanismo, il decoder è in grado di costruire l'output finale rispettando sia la struttura dei dati in ingresso sia la logica sequenziale propria dell'output.

Il cuore del Transformer è il meccanismo di **self-attention**, che consente all'encoder di stabilire, per ogni elemento della sequenza in ingresso, quali altri elementi siano rilevanti per la sua rappresentazione. Per ciascun input vengono costruiti tre vettori distinti:

- **query (Q)**: rappresenta l'informazione ricercata dall'elemento;
- **key (K)**: descrive quale tipo di informazione l'elemento offre;
- **value (V)**: contiene l'informazione effettiva da trasmettere.

L'*attention* viene calcolata confrontando le varie query degli elementi in ingresso con tutte le key della sequenza: più una key è simile a una query, maggiore sarà il peso associato al value corrispondente. Formalmente, il meccanismo è definito come:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V,$$

dove $Q \in \mathbb{R}^{n \times d_k}$ è la matrice delle query, $K \in \mathbb{R}^{n \times d_k}$ è la matrice delle key, $V \in \mathbb{R}^{n \times d_v}$ è la matrice dei value, QK^T è una matrice di punteggi che rappresenta quanto la query di un elemento si allinea con le key degli altri, d_k è la dimensione dei vettori delle key e delle query, utilizzata come fattore di normalizzazione per ridurre la varianza dei prodotti scalari, la funzione softmax agisce riga per riga sulla matrice dei punteggi trasformandola in una distribuzione di probabilità e la moltiplicazione finale con V produce, per ciascun elemento, una combinazione lineare pesata dei value, in cui i pesi riflettono la compatibilità tra query e key.

In questo modo, ogni rappresentazione in uscita incorpora informazioni provenienti da tutti gli altri elementi della sequenza, modulata dall'intensità della loro rilevanza calcolata attraverso il meccanismo di *self–attention*.

Esempio 2.2.2. Si consideri la frase

Il gatto siede sul tavolo

Quando il modello elabora la parola *siede*, la sua query viene confrontata con tutte le key delle altre parole. La similarità più alta si ottiene con la key della parola *gatto*, poiché il verbo *siede* ricerca un soggetto a cui riferirsi. Di conseguenza, il value associato a *gatto* contribuisce in modo rilevante alla rappresentazione dell'elemento *siede*.

Questo meccanismo permette al modello di catturare la dipendenza semantica tra soggetto e verbo senza bisogno di ricorrenza: la rappresentazione di *siede* non dipende solo dalla sua posizione locale, ma viene arricchita dal contesto fornito da *gatto*.

Ripetuto in più strati di encoder e decoder, questo processo consente all'architettura *Transformer* di catturare relazioni globali tra gli elementi della sequenza in maniera diretta e parallelizzabile, superando i limiti strutturali delle reti ricorrenti e convoluzionali.

Nel caso di AlphaTensor, l'input non è una sequenza testuale, ma uno stato del gioco rappresentato da un tensore che codifica un'operazione bilineare. L'input alla rete contiene tutte le informazioni rilevanti sullo stato attuale ed è composto da una lista di tensori e di scalari. L'informazione più importante è il tensore attuale $\mathcal{S}_t$ di dimensione $S \times S \times S$ (per semplicità assumiamo che tutte e tre le dimensioni del tensore siano uguali; la generalizzazione a dimensioni diverse è immediata). Inoltre, il modello riceve le ultime h azioni (con h iperparametro, tipicamente $h = 7$), rappresentate come h tensori di rango 1 e concatenate all'input. La lista degli scalari include l'indice temporale t dell'azione corrente ($0 \leq t < R_{\text{limite}}$).

Il *torso* è il corpo centrale della rete neurale ed è responsabile della mappatura congiunta di scalari e tensori in una rappresentazione utile. In altre parole, il torso trasforma lo stato del gioco in un insieme di caratteristiche che riassumono in modo coerente sia la struttura del tensore sia le informazioni scalari aggiuntive. Questa rappresentazione unica viene poi passata a due moduli diversi: la *policy head*, che la utilizza per decidere l'azione successiva, e la *value head*, per stimare il ritorno atteso. La sua architettura si basa su una modifica dell'architettura *Transformer* e opera su tre griglie $S \times S$ proiettate dai tensori di input $S \times S \times S$. Definendo le modalità del tensore come U, V, W, le righe e le colonne della prima griglia sono associate a $U \times V$, quelle della seconda a $W \times U$, e quelle della terza a $V \times W$. Ogni elemento di ogni griglia è un vettore di caratteristiche; il suo valore iniziale è dato dagli elementi dei tensori di input lungo la modalità mancante della griglia. Questi vettori vengono poi arricchiti concatenando una proiezione lineare $S \times S \times 1$ dagli scalari, seguita da un ulteriore livello lineare che proietta in uno spazio a 512 dimensioni. Il resto del torso è una sequenza di blocchi basati sull'*attention* con l'obiettivo di propagare informazione tra le tre griglie. Ciascun blocco ha tre stadi, uno per ogni coppia di griglie: (UV, WU), (WU, VW) e (VW, UV). In ogni stadio, le due griglie coinvolte vengono concatenate lungo l'asse condiviso e si applica l'*axial attention* (generalizzata) sulle colonne: per ciascuno degli S valori dell'asse condiviso, si eseguono in parallelo S operazioni di *self—attention* su sequenze di lunghezza $2S$. Questo schema riduce la complessità da $\mathcal{O}(S^4)$ (*self—attention* $2D$ piena) a $\mathcal{O}(S^3)$. La rappresentazione

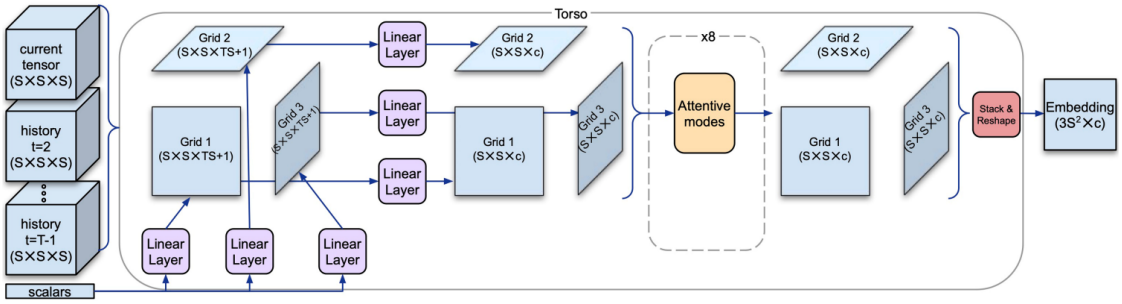


Figura 2.9: *Diagramma del torso della rete neurale di AlphaTensor. Immagine tratta da [8].*

inviata alla *policy head* corrisponde ai $3S$ vettori prodotti dall'ultimo strato del torso. Una descrizione dettagliata della struttura del torso è riportata in Figura 2.9.

La rappresentazione finale prodotta dal torso viene poi condivisa da due moduli distinti: la *policy head* e la *value head*. Entrambe si basano sulle stesse *feature*, ma perseguono obiettivi diversi e complementari, in analogia con gli algoritmi di *reinforcement learning* utilizzati in AlphaZero.

La **policy head** è implementata come un modello autoregressivo basato sull'architettura *Transformer*. L'azione da generare in AlphaTensor corrisponde a una tripletta di vettori $(\mathbf{u}, \mathbf{v}, \mathbf{w})$, ognuno di dimensione S e con coefficienti discreti scelti da un insieme finito $F = \{-2, -1, 0, 1, 2\}$. Tali vettori vengono rappresentati come una sequenza di k *token* di dimensione d, tali che $k \cdot d = 3S$. In questo contesto, un *token* è l'unità elementare della sequenza che codifica un coefficiente dei vettori $(\mathbf{u}, \mathbf{v}, \mathbf{w})$; per esempio, per $S = 4$, un'azione può essere scomposta in 12 token, ciascuno corrispondente a un elemento di un vettore della tripletta.

Il metodo autoregressivo genera questa sequenza passo dopo passo: ogni token a_t è predetto grazie all'osservazione dei precedenti $a_{<t}$ e delle informazioni sullo stato prodotte dal torso, tramite un meccanismo di *cross-attention*. In tal modo, la generazione di ciascun coefficiente della tripletta tiene conto sia della struttura già generata (coerenza interna) sia delle informazioni contestuali sullo stato del gioco (coerenza esterna).

Durante l'addestramento si utilizza la tecnica del *teacher forcing*: le azioni target (derivate dal miglioramento tramite MCTS) vengono scomposte in token e fornite come input al Transformer. Grazie al *masking* causale, la predizione di un token dipende esclusivamente dai token precedenti, riproducendo fedelmente la logica autoregressiva. In fase di inferenza, invece, la *policy head* genera nuove azioni campionando K sequenze complete di token dalla distribuzione $\pi_\theta(\cdot|s)$, utilizzate poi per espandere l'albero MCTS.

Un aspetto importante è che la rappresentazione delle feature prima dell'ultimo stra-

to lineare al primo passo viene riutilizzata come input per la *value head*, creando un collegamento diretto tra le due componenti della rete. In questo modo, policy e value condividono una base comune di rappresentazioni, ma vengono specializzate a compiti distinti: generazione di azioni e valutazione dello stato.

La **value head**, al contrario, non genera azioni ma valuta lo stato corrente del gioco. È implementata come un percettrone multistrato (MLP) che riceve in input la stessa rappresentazione del torso e stima la distribuzione dei rendimenti associata allo stato. In particolare, AlphaTensor utilizza una regressione sui quantili (vedi 2.2.3) per approssimare la distribuzione dei ritorni futuri, fornendo una misura non solo del valore atteso ma anche dell'incertezza sul risultato. Tale stima è fondamentale per il bilanciamento esplorazione–sfruttamento durante la ricerca MCTS, poiché consente di selezionare mosse non soltanto promettenti in media, ma anche robuste rispetto alla variabilità dei ritorni.

Sia la policy head sia la value head fanno largo uso della funzione di attivazione **ReLU** (*Rectified Linear Unit*), definita come:

$$\mathrm{ReLU}(x) = \max(0, x).$$

Se $x > 0$, l'uscita coincide con x; se $x < 0$, l'uscita è 0. Questa semplice trasformazione ha diversi vantaggi:

- garantisce efficienza computazionale, essendo poco costosa da calcolare;
- riduce il problema del *vanishing gradient*, facilitando l'addestramento di reti neurali profonde;
- promuove scarsità nelle rappresentazioni intermedie, poiché i valori negativi vengono azzerati, rendendo più chiara la separazione tra feature rilevanti e non.

Nello schema di AlphaTensor, ogni strato lineare delle due head è seguito da una ReLU, che permette di raffinare progressivamente le rappresentazioni prima di produrre la distribuzione sulle azioni (policy) o quella dei rendimenti (value).

In sintesi, la *policy head* e la *value head* costituiscono i due principali responsabili del processo decisionale: la prima orientata alla scelta dell'azione, la seconda alla valutazione dello stato. Entrambe si fondano sulla stessa rappresentazione latente fornita dal torso, ma la sfruttano in maniera diversa e complementare. L'integrazione della funzione di attivazione ReLU nei loro strati interni consente di ottenere modelli più espressivi e stabili, rendendo AlphaTensor in grado di unire capacità di generazione e di valutazione in un unico framework potenziato dalla ricerca MCTS. In Figura 2.10 un riassunto dell'architettura della rete neurale di AlphaTensor.

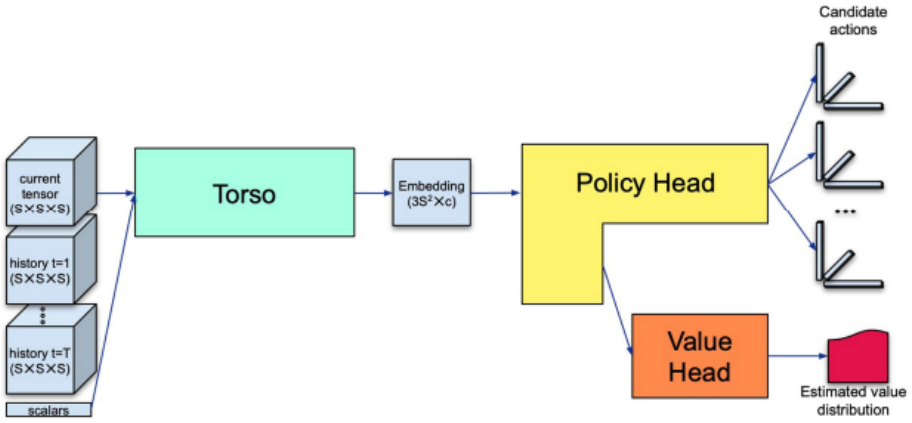


Figura 2.10: *Architettura della rete neurale di AlphaTensor. Immagine tratta da [8].*

Sampled MCTS

La procedura di ricerca adottata in AlphaTensor si ispira al Monte Carlo Tree Search classico, ma introduce alcune modifiche fondamentali per adattarsi alla natura del problema tensoriale. A ogni stato s_t, viene costruito un albero i cui nodi rappresentano gli stati del gioco e i cui archi corrispondono alle azioni possibili, ovvero le triplette di vettori $(\mathbf{u}, \mathbf{v}, \mathbf{w})$.

Ogni coppia stato–azione (s, a) conserva tre statistiche aggiornate dinamicamente:

- $N(s, a)$ il numero di volte in cui l'azione a è stata esplorata a partire dallo stato s (conteggio delle visite);
- $Q(s, a)$ il valore stimato associato a quell'azione;
- $\hat{\pi}(s, a)$ la probabilità empirica della politica, cioè la frequenza relativa con cui l'azione è stata selezionata.

La selezione delle azioni all'interno dell'albero si basa su un metodo di tipo UCT (si veda 2.2.2) rivisitato:

$$a^\star \in \arg\max_a \left[Q(s,a) + c(s) \cdot \hat{\pi}(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \right],$$

dove $c(s)$ è un fattore di esplorazione che regola l'influenza della politica empirica $\hat{\pi}(s, a)$ rispetto ai valori $Q(s, a)$. Esso è definito in forma dinamica, seguendo l'impostazione introdotta in AlphaZero:

$$c(s) = c_{\text{init}} + \log\left(\frac{1 + \sum_b N(s,b) + c_{\text{base}}}{c_{\text{base}}} \right),$$

dove $\sum_b N(s,b)$ rappresenta il numero totale di visite ai figli del nodo s, mentre c_{init} e c_{base} sono iperparametri che controllano la scala e la crescita dell'esplorazione. In questo modo, l'esplorazione è incentivata nelle fasi iniziali e progressivamente ridotta man mano che il nodo viene visitato più volte.

Una volta raggiunta una foglia s_t, la rete neurale fornisce due informazioni: un insieme di K azioni candidate campionate dalla distribuzione di policy $\pi(\cdot \mid s_t)$, da cui si ottiene la distribuzione empirica $\hat{\pi}$, e una stima di valore $\nu(s_L)$ prodotta dalla *value head*. La sostanziale differenza tra l'MCTS classico e l'MCTS *sampled* utilizzato in AlphaTensor consiste nel fatto che, nel metodo tradizionale, la *policy head* esplora l'intero spazio delle mosse disponibili, o comunque tutte quelle con probabilità non nulla, mentre nella versione *sampled* ciò non avviene. All'interno del TensorGame, lo spazio delle azioni è molto grande, *more than* 10^{12} *actions for most interesting cases*, come riportato in [8], sarebbe dunque impossibile riuscire ad esplorarle tutte (o quasi) ad ogni partita giocata mantenendo costi (computazionali e di *runtime*) ragionevoli. Per questo motivo viene utilizzata una versione modificata del metodo nella quale vengono campionate K azioni candidate da esplorare, riducendo drasticamente la complessità. Inoltre viene introdotto il valore $\nu(s_t)$, che non coincide con la media della distribuzione dei ritorni futuri come nei tradizionali algoritmi di RL, ma sfrutta i quantili per stimare traiettorie valide, privilegiando soluzioni che abbiano alta probabilità di portare al tensore nullo.

Definizione 2.2.3 (Quantile)**.** Data una variabile aleatoria X, il quantile di ordine $p \in (0,1)$ è il valore q_p tale che

$$P(X \leq q_p) = p.$$

In altre parole, un quantile è il valore soglia che divide la distribuzione in due parti, lasciando a sinistra una data frazione della probabilità totale.

I valori vengono infine propagati all'indietro (*back-up*) lungo i nodi visitati, aggiornando le statistiche. Inoltre, per gestire il fatto che diverse sequenze di azioni possano condurre allo stesso tensore residuo a causa della commutatività dell'operazione, viene utilizzata una tabella di trasposizione che ricombina percorsi equivalenti ed evita ridondanze.

Queste modifiche differenziano il *sampled* MCTS di AlphaTensor dalla versione standard utilizzata in AlphaZero: la procedura non mira a massimizzare una probabilità di vittoria, ma a trovare algoritmi di moltiplicazione matriciale efficienti, minimizzando il rango residuo e la lunghezza della traiettoria.

Cambi di Base e *Data Augmentation*

Un aspetto cruciale della formulazione del TensorGame riguarda la possibilità di esprimere la stessa operazione bilineare in basi differenti: un cambiamento di base non altera il rango del tensore, ma produce una rappresentazione alternativa che può essere sfruttata per aumentare la diversità delle traiettorie esplorate dall'agente. Questo approccio presenta tre vantaggi principali:

1. il cambiamento di base fornisce un meccanismo naturale di esplorazione, poiché giocare in basi differenti introduce automaticamente diversità nelle traiettorie affrontate dall'agente;

2. l'agente non deve necessariamente ottenere successo in tutte le basi: è sufficiente individuare una decomposizione a basso rango in almeno una di esse;

3. una decomposizione con elementi appartenenti a un insieme finito $F = \{-2, -1, 0, 1, 2\}$ trovata in una base arbitraria non deve necessariamente mantenere la stessa struttura di coefficienti quando viene ricondotta alla base canonica, ampliando così lo spettro degli algoritmi individuabili.

In generale, un cambiamento di base per un tensore tridimensionale di dimensione $S \times S \times S$ corrisponde a un cambiamento di base indipendente in ciascuno dei tre spazi vettoriali che compongono il suo prodotto tensoriale. Formalmente, se il tensore $\mathcal{T}$ appartiene allo spazio $V \otimes W \otimes U$, con $\dim V = \dim W = \dim U = S$, allora un cambiamento di base è determinato dalla scelta di tre matrici invertibili $A \in \mathrm{GL}(V)$, $B \in \mathrm{GL}(W)$ e $C \in \mathrm{GL}(U)$. L'azione del cambiamento di base sul tensore è data da

$$\mathcal{T}' = (A \otimes B \otimes C)\,\mathcal{T},$$

cioè si applica la trasformazione lineare rappresentata dalla matrice A alle coordinate del tensore relative allo spazio vettoriale V, B a quelle relative a W e C a quelle relative a U. Nella procedura adottata in AlphaTensor le basi vengono campionate in modo casuale imponendo due restrizioni specifiche:

1. $A = B = C$, scelta motivata dal fatto che questa configurazione ha mostrato prestazioni migliori nei primi esperimenti;

2. $\det(A) \in \{-1, +1\}$, condizione che assicura che, una volta ricondotta una fattorizzazione alla base canonica, essa contenga ancora esclusivamente elementi interi. Questa proprietà risulta utile sia per la semplicità di rappresentazione sia per la stabilità numerica dell'algoritmo risultante.

Oltre a giocare in basi differenti, AlphaTensor adotta anche una strategia di *data augmentation* ogni volta che la rete neurale viene interrogata in un nuovo nodo del MCTS. In fase di valutazione, il tensore di input viene trasformato mediante un cambiamento di base definito da una permutazione casuale con segno, ovvero una matrice di permutazione $P \in \mathrm{GL}(S)$ i cui elementi non nulli sono ± 1. Tali matrici rappresentano trasformazioni lineari che riorganizzano le coordinate degli spazi vettoriali di base e, opzionalmente, ne invertono il segno. La trasformazione corrisponde all'applicazione simultanea della stessa permutazione con segno ai tre spazi vettoriali che compongono lo spazio prodotto tensoriale a cui appartiene il tensore tridimensionale $\mathcal{T}$.

Sebbene, in linea di principio, il procedimento possa essere esteso a qualsiasi matrice di cambiamento di base, in AlphaTensor si impiegano specificamente permutazioni con segno, per motivi di efficienza computazionale. Durante l'addestramento, infatti, ogni tripletta costituita da *input*, target di *policy* e target di valore viene modificata applicando una permutazione con segno casuale sia all'input sia ai target e la rete viene addestrata sulla versione trasformata.

Ottimizzazione con AdamW

Per quanto riguarda l'addestramento della rete neurale, l'aggiornamento dei parametri θ avviene tramite l'ottimizzatore Adam (*Adaptive Moment Estimation*), nella variante con decadimento del peso disaccoppiato, nota come AdamW. Adam è uno degli algoritmi più utilizzati perché combina i vantaggi di due metodi classici: Momentum, che accumula una media dei gradienti per seguire direzioni stabili, e RMSProp, che adatta il passo di aggiornamento al variare della magnitudine dei gradienti per ciascun parametro. Formalmente, Adam mantiene due stime mobili:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$$

dove g_t è il gradiente al passo t, m_t la stima del primo momento (media) e v_t la stima del secondo momento (varianza).

Dopo una correzione dei bias iniziali, i parametri vengono aggiornati secondo:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon},$$

dove η è il *learning rate* e ϵ un termine di stabilizzazione numerica.

Una componente fondamentale per evitare overfitting nei modelli di grandi dimensioni è il *weight decay*, ossia una penalizzazione che tende a ridurre i valori dei pesi durante l'addestramento. Nella formulazione classica, il decadimento viene aggiunto direttamente

al gradiente:

$$\theta \leftarrow \theta - \eta\big(\nabla_\theta L + \lambda\theta\big),$$

dove λ è il coefficiente di regolarizzazione. Tuttavia, in Adam questa strategia può risultare problematica, perché la regolarizzazione viene mescolata con la stima adattiva dei momenti, modificando l'effetto desiderato.

Per risolvere questo problema, nel 2019 Ilya Loshchilov e Frank Hutter hanno proposto AdamW tramite il lavoro *Decoupled Weight Decay Regularization* [15], pubblicato in occasione di una conferenza sul tema. La novità principale introdotta da AdamW è il disaccoppiamento del *weight decay* dall'aggiornamento adattivo. In AdamW la penalizzazione non viene sommata al gradiente, ma applicata separatamente ai pesi:

$$\theta \leftarrow \theta - \eta\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} - \eta\lambda\theta.$$

In questo modo, AdamW conserva i benefici dell'ottimizzazione adattiva, mentre il termine $\eta\lambda\theta$ agisce come una contrazione uniforme dei parametri, evitando che essi crescano in modo incontrollato. Questo approccio migliora la stabilità numerica e la capacità di generalizzazione, rendendo il modello meno incline all'overfitting.

Nel contesto di AlphaTensor, che richiede reti profonde e addestramenti molto lunghi su milioni di esempi generati tramite self-play, AdamW offre un equilibrio ideale tra velocità di convergenza e regolarizzazione efficace, permettendo di sfruttare a pieno la capacità espressiva della rete neurale senza perdere stabilità.

Capitolo 3

Risultati e Conclusioni

Un primo insieme di risultati ottenuti da AlphaTensor riguarda la moltiplicazione di matrici di dimensione $n \times m$ e $m \times p$ con $n, m, p \leq 5$. Le matrici di piccola dimensione costituiscono infatti un contesto di analisi ideale: da un lato permettono di confrontare le prestazioni dell'agente con gli algoritmi già noti e dall'altro evidenziano la capacità di AlphaTensor di individuare nuove decomposizioni tensoriali più efficienti anche in scenari già ampiamente studiati. In particolare, AlphaTensor è riuscito a individuare algoritmi che corrispondono o migliorano i più efficienti algoritmi conosciuti prima della sua invenzione. In Figura 3.1 sono riassunti i risultati di AlphaTensor per varie configurazioni di matrici entro le dimensioni sopracitate, messi a confronto con i metodi precedenti più efficienti. Per ogni terna (n, m, p), è indicato il rango minimo della decomposizione tensoriale già documentata in letteratura e, accanto, quello identificato dall'agente, sia nel caso di operazioni in aritmetica modulare (colonna *modular*) sia in aritmetica standard (colonna *standard*). L'agente è stato infatti utilizzato per moltiplicare matrici con coefficienti a valori sia nell'anello quoziente $\mathbb{Z}_{/2\mathbb{Z}} = \{[0], [1]\}$ (dove $[0]$ è la classe dei numeri pari e $[1]$ quella dei numeri dispari), sia in $\mathbb{R}$, operando così rispettivamente in aritmetica modulare e in aritmetica standard. Questa duplice impostazione consente di valutare l'efficacia degli algoritmi scoperti in scenari differenti e di evidenziare come la scelta dell'aritmetica possa influire sul rango delle decomposizioni e sulle prestazioni complessive.

Tra i risultati più importanti vediamo il superamento dell'algoritmo di Strassen per matrici 4×4 a coefficienti in $\mathbb{Z}_{/2\mathbb{Z}}$. Come visto nella Sezione 1.3, il metodo Strassen richiede 7 moltiplicazioni per il prodotto tra due matrici di dimensione 2×2, quindi nel caso di matrici di dimensione 4×4, grazie al metodo ricorsivo, le moltiplicazioni necessarie diventano $7^2 = 49$. AlphaTensor ha scoperto un algoritmo capace di moltiplicare due matrici di dimensione 4×4 a coefficienti in $\mathbb{Z}_{/2\mathbb{Z}}$ con solo 47 moltiplicazioni, migliorando

Size (n, m, p)	Best method known	Best rank known	AlphaTensor rank Modular	AlphaTensor rank Standard
$(2, 2, 2)$	(Strassen, 1969)[2]	7	7	7
$(3, 3, 3)$	(Laderman, 1976)[15]	23	23	23
$(4, 4, 4)$	(Strassen, 1969)[2] $(2, 2, 2) \otimes (2, 2, 2)$	49	47	49
$(5, 5, 5)$	$(3, 5, 5) + (2, 5, 5)$	98	96	98
$(2, 2, 3)$	$(2, 2, 2) + (2, 2, 1)$	11	11	11
$(2, 2, 4)$	$(2, 2, 2) + (2, 2, 2)$	14	14	14
$(2, 2, 5)$	$(2, 2, 2) + (2, 2, 3)$	18	18	18
$(2, 3, 3)$	(Hopcroft and Kerr, 1971)[16]	15	15	15
$(2, 3, 4)$	(Hopcroft and Kerr, 1971)[16]	20	20	20
$(2, 3, 5)$	(Hopcroft and Kerr, 1971)[16]	25	25	25
$(2, 4, 4)$	(Hopcroft and Kerr, 1971)[16]	26	26	26
$(2, 4, 5)$	(Hopcroft and Kerr, 1971)[16]	33	33	33
$(2, 5, 5)$	(Hopcroft and Kerr, 1971)[16]	40	40	40
$(3, 3, 4)$	(Smirnov, 2013)[18]	29	29	29
$(3, 3, 5)$	(Smirnov, 2013)[18]	36	36	36
$(3, 4, 4)$	(Smirnov, 2013)[18]	38	38	38
$(3, 4, 5)$	(Smirnov, 2013)[18]	48	47	47
$(3, 5, 5)$	(Sedoglavic and Smirnov, 2021)[19]	58	58	58
$(4, 4, 5)$	$(4, 4, 2) + (4, 4, 3)$	64	63	63
$(4, 5, 5)$	$(2, 5, 5) \otimes (2, 1, 1)$	80	76	76

Figura 3.1: *Da sinistra: la colonna (n, m, p) individua il tensore di moltiplicazione $\mathcal{T}_{n,m,p}$ da decomporre, la complessità è misurata dal numero di termini nella decomposizione del tensore, Best rank known (miglior rango noto) si riferisce al limite superiore noto del rango del tensore, mentre AlphaTensor rank riporta i limiti superiori del rango ottenuti dall'agente nei due casi corrispondenti all'aritmetica utilizzata. Tabella tratta da [8].*

così la complessità da $\mathcal{O}(n^{2.81})$ a $\mathcal{O}(n^{2.778})$.

Per quanto riguarda l'aritmetica standard, un risultato rilevante è il caso del prodotto tra due matrici di dimensioni rispettivamente 4×5 e 5×5 (tensore di moltiplicazione $\mathcal{T}_{4,4,5}$), contesto in cui AlphaTensor trova un algoritmo richiedente 76 moltiplicazioni invece delle 80 necessarie con l'ultimo metodo scoperto dai matematici.

I miglioramenti scoperti da AlphaTensor nei casi di matrici di piccole dimensioni hanno un impatto diretto anche sugli algoritmi per matrici più grandi, grazie all'utilizzo del meccanismo di ricorsione sulle decomposizioni tensoriali. In Figura 3.2 è riportata la differenza tra il rango minimo già noto e quello trovato da AlphaTensor. Si osservi che i miglioramenti non si limitano a pochi casi isolati, ma si estendono anche a configurazioni di dimensione elevata: per esempio, per la terna $(11, 12, 12)$ AlphaTensor riduce il rango di oltre 30 rispetto al miglior algoritmo precedentemente noto. Si può notare inoltre che i miglioramenti più grandi si ottengono proprio nei casi in cui i ranghi noti erano

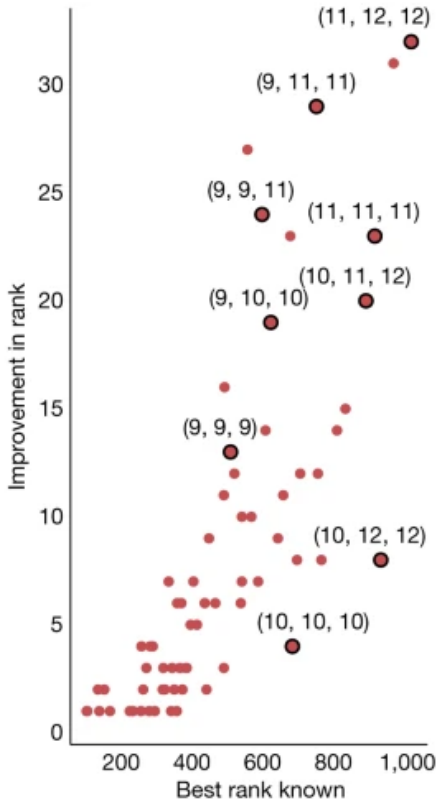


Figura 3.2: *Risultati di AlphaTensor per matrici di dimensione superiore. L'asse orizzontale rappresenta il rango minimo già documentato per un tensore di moltiplicazione $T_{n,m,p}$, mentre l'asse verticale indica la riduzione ottenuta dall'agente. Ogni punto corrisponde a una specifica terna (n, m, p), con etichette per i casi più significativi. Più il punto rosso è in alto, più AlphaTensor ha ridotto il numero di moltiplicazioni rispetto all'algoritmo precedente. Immagine tratta da [8].*

molto elevati (a destra nel grafico in Figura 3.2): questo evidenzia che AlphaTensor è particolarmente utile quando lo spazio delle possibili decomposizioni è complesso e difficile da esplorare manualmente. Utilizzando dunque questo metodo ricorsivo sulle matrici di dimensioni ridotte, AlphaTensor ha migliorato gli algoritmi di moltiplicazione per oltre 70 differenti configurazioni di matrici con $n, m, p \leq 12$.

Un aspetto particolarmente interessante emerso dall'impiego di AlphaTensor è la ricchezza e varietà degli algoritmi che esso è in grado di individuare. Dal punto di vista matematico, questi risultati suggeriscono che lo spazio degli algoritmi di moltiplicazione matriciale è molto più ampio di quanto si pensasse in precedenza. Per esempio, prima della pubblicazione di questo lavoro, era nota un'unica fattorizzazione di rango 49 del

tensore $\mathcal{T}_4$, ottenuta come prodotto tensoriale di due copie del tensore $\mathcal{T}_2$:

$$\mathcal{T}_4 = \mathcal{T}_2 \otimes \mathcal{T}_2,$$

cioè l'algoritmo derivante dall'applicazione del metodo Strassen due volte (di conseguenza detto *quadrato di Strassen*). AlphaTensor, invece, ha scoperto più di 14.000 fattorizzazioni di rango 49 che non sono equivalenti a questa, ossia non ottenibili tramite semplici trasformazioni di simmetria. Queste nuove soluzioni presentano proprietà differenti, ad esempio in termini di sparsità delle matrici coinvolte o di distribuzione dei coefficienti. Con il termine *non equivalenti* intendiamo precisamente che non esiste una trasformazione di simmetria (ad esempio, un riordinamento dei vettori o una permutazione delle matrici nei prodotti tensoriali) che permetta di passare da una fattorizzazione all'altra. Lo studio delle simmetrie e delle classi di equivalenza degli algoritmi può quindi fornire informazioni preziose non solo dal punto di vista pratico (scegliere fattorizzazioni più sparse o con proprietà numeriche favorevoli), ma anche dal punto di vista teorico, contribuendo a chiarire i limiti e le potenzialità dei metodi per ridurre l'esponente ω della moltiplicazione.

In prospettiva, esplorare questo spazio ricco e diversificato di algoritmi potrà generare nuovi risultati, sia con applicazioni immediate, sia come spunto per avanzamenti teorici futuri nella ricerca matematica.

Limiti e prospettive future

Nonostante i risultati di grande rilievo, AlphaTensor presenta alcune limitazioni strutturali che ne riducono l'immediata applicabilità e aprono la strada a sviluppi futuri. In primo luogo, l'agente opera su un insieme predefinito di coefficienti $F = \{-2, -1, 0, 1, 2\}$: tale scelta rende discreto lo spazio di ricerca ma comporta anche il rischio di escludere algoritmi potenzialmente più efficienti al di fuori dell'insieme considerato. Inoltre, il problema della scalabilità rimane cruciale: lo spazio delle possibili mosse è enorme, supera le 10^{12} azioni per molte configurazioni di interesse. Questo implica un fabbisogno computazionale estremamente elevato, come dimostra il training condotto su TPU/GPU di ultima generazione con migliaia di attori in parallelo. Gli algoritmi scoperti risultano infatti fortemente dipendenti dall'hardware di riferimento: quelli ottimizzati per una specifica architettura (ad esempio GPU Nvidia V100 o TPU v2) mostrano prestazioni significativamente inferiori se trasferiti su dispositivi diversi, segnalando una limitata generalizzabilità dei risultati. Un ulteriore limite riguarda le metriche di ottimizzazione adottate: AlphaTensor è stato progettato per ridurre il numero di moltiplicazioni scalari

o, in alternativa, per ottimizzare il tempo di esecuzione su hardware specifici. Tuttavia, non sono stati ancora considerati aspetti fondamentali nelle applicazioni in contesti produttivi, quali la stabilità numerica o l'efficienza energetica, che gli stessi autori individuano come possibili direzioni per ricerche future. In sintesi, sebbene AlphaTensor rappresenti un importante avanzamento nell'automazione della scoperta algoritmica, i suoi limiti attuali ne delineano un potenziale più esplorativo che immediatamente applicativo, ponendo le basi per futuri miglioramenti sia sul piano teorico che computazionale.

Un miglioramento concreto in questa direzione è arrivato nel 2023 con lo sviluppo da parte di Google DeepMind di un nuovo agente di intelligenza artificiale specializzato nella risoluzione di problemi di natura matematica chiamato AlphaEvolve. Questo innovativo agente ha affrontato alcune delle criticità di AlphaTensor, in particolare riguardo alla stabilità numerica, all'efficienza energetica e alla portabilità tra hardware differenti. AlphaEvolve è inoltre riuscito a migliorare l'algoritmo di Strassen (da 49 a 48 moltiplicazioni necessarie) per matrici 4×4 a coefficienti complessi, risultato che AlphaTensor, agente specializzato nella ricerca di algoritmi per la moltiplicazione matriciale, non era riuscito a ottenere. I progressi compiuti da AlphaEvolve rendono il metodo più adatto a un'adozione pratica, confermando come le limitazioni iniziali di AlphaTensor possano e siano destinate a essere progressivamente superate grazie ad affinamenti metodologici.

In conclusione, risulta evidente come i progressi ottenuti da sistemi come AlphaTensor e AlphaEvolve non sostituiscano, bensì valorizzino la ricerca matematica, delineando un futuro in cui la sinergia tra ingegno umano e intelligenza artificiale potrà accelerare in maniera decisiva la scoperta di nuovi algoritmi potenzialmente rivoluzionari per il progresso scientifico e tecnologico.

Bibliografia

[1] Bishop, C.M. *Pattern Recognition and Machine Learning*. Springer, 2006.

[2] Bower, R.M., Wang, C.C. *Introduction to vectors and tensors*. Capitolo 7. 2009, Dover Publications.

[3] Boyer, C. B., Merzbach, U. C. *A History of Mathematics*, 2a Edizione. Wiley, 1991.

[4] Brubaker, B. *AI Reveals New Possibilities in Matrix Multiplication*. QuantaMagazine, 2022. `https://www.quantamagazine.org/ai-reveals-new-possibilities-in-matrix-multiplication-20221123/`.

[5] Casi, R. and Pizzarelli, C. *Didattica della matematica. Dalla ricerca alle pratiche d'aula*, vol 7. 2020, pp. 97–122.

[6] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, 3a Edizione. Capitolo 4, sezione 2. MIT Press, 2009.

[7] Fawzi A., Balog, M., Romera-Paredes, B., Hassabis, D., Kohli, P. *Discovering Novel Algorithms with AlphaTensor*. DeepMind Blog, 2022. `https://deepmind.google/discover/blog/discovering-novel-algorithms-with-alphatensor/`

[8] Fawzi A., Balog, M., Romera-Paredes, B., Hassabis, D., Kohli, P. *Discovering faster matrix multiplication algorithms with reinforcement learning*. Nature 610, 47–53, 2022. `https://doi.org/10.1038/s41586-022-05172-4`

[9] Gonzalez, R. C., Woods, R. E. *Digital Image Processing*, 3a Edizione. Prentice Hall, 2007.

[10] Google DeepMind. *Build AI responsibly to benefit humanity*. `https://deepmind.google/about/`.

[11] Anonimo. *Monte Carlo Tree Search (MCTS) in Machine Learning*. GeeksforGeeks, 2023.

https://www.geeksforgeeks.org/machine-learning/ml-monte-carlo-tree-search-mcts/.

[12] Horn, R. A., Johnson, C. R. *Matrix Analysis*, 2a Edizione. Capitolo 7, sezione 4. Cambridge University Press, 2013.

[13] Anonimo. *Jiuzhang Suanshu (The Nine Chapters on the Mathematical Art)*. Capitolo 8, Cina antica.

[14] LeCun, Y., Bengio, Y., Hinton, G. *Deep learning*. Nature, vol. 521, pp. 436–444, 2015. https://doi.org/10.1038/nature14539

[15] Loshchilov, I, Hutter, F. *Decoupled Weight Decay Regularization*. Conference paper, 2019. https://arxiv.org/pdf/1711.05101.

[16] Needham, J. *Science and Civilisation in China*, vol. 3: *Mathematics and the Sciences of the Heavens and the Earth*. Cambridge University Press, 1959.

[17] O'Connor, J. J., Robertson, E. F. *Brahmagupta*. MacTutor History of Mathematics Archive, University of St Andrews, 1996.

[18] Quarteroni, A., Sacco, R., Saleri, F., Gervasio, P. *Matematica Numerica*, 4a Edizione. Capitolo 2. Springer, 2014.

[19] Rashed, R., Morelon, R. *Encyclopaedia of the History of Arabic Science*. Routledge, 1996.

[20] Roman, S. *Advanced Linear Algebra*, 3a Edizione. Capitolo 14. Springer, 2008.

[21] Rumelhart, D., Hinton, G., Williams, R. *Learning Representations by Back-Propagating Errors*. Nature, 1986. https://www.nature.com/articles/323533a0

[22] Strang, G. *Linear Algebra and Its Applications*, 4a Edizione. Capitolo 6, pp. 367–369. Brooks/Cole, 2006.

[23] Strassen, V. *Gaussian Elimination is not Optimal. Numerische Mathematik*, vol. 13, 1969, pp. 354–356.

[24] Sutton, R.S., Barto, A.G. *Reinforcement Learning: An Introduction*, 2a Edizione. MIT Press, 2018.

[25] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., Kaiser, L., Polosukhin, I. *Attention Is All You Need*. `https://papers.nips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html`

[26] Wann-Sheng, H. *HPM Newsletter 4:2b*, Taiwan edition. International Study Group on History and Pedagogy of Mathematics, 2004.

[27] Zhang, A., Lipton, Z., Li, M., Smola, A.J. *Dive into Deep Learning*. Cambridge University Press, 2023.

Ringraziamenti

Desidero esprimere la mia più sincera gratitudine a tutte le persone che mi hanno accompagnata in questo percorso e che hanno reso possibile il raggiungimento di questo traguardo.

Un ringraziamento speciale va al Professore Giovanni Paolini, per i preziosi consigli, la pazienza e la disponibilità con cui ha curato questo elaborato. È stato un piacere poter lavorare con Lei.

Grazie a mia mamma Luisa per essere da sempre la mia più grande sostenitrice e per aver creduto in me in ogni istante di questo percorso. Mi hai insegnato ad avere passione e tenacia, e per questo ti sarò sempre grata.

Grazie ad Alberto per essere stato al mio fianco sempre, per avermi spronato a dare il meglio di me e per tutto ciò che fai per me ogni giorno per rendere la mia vita meravigliosa. Questo traguardo è anche tuo.

Grazie a Rosa e ad Alessandro per avermi sostenuto, accompagnato e consigliato a ogni passo e a ogni incertezza, siete famiglia.

Grazie a tutta la mia famiglia, perché la gioia di un successo non è gioia senza poterla condividere con voi. Grazie per esserci sempre e per dare valore a ognuno di noi.

Grazie a Veronica per ogni nottata di studio, per ogni incoraggiamento e per avermi tenuto la mano in questo lungo cammino, senza di te sarebbe stato diverso.

Infine un grazie speciale va a Pietro, Juri, Matilde, a tutti i miei amici e agli amici di famiglia, lontani e vicini. Grazie per avermi consolata nei momenti più bui, per avermi ispirata e per aver gioito con me a ogni traguardo. Rendete la mia vita più luminosa, vi voglio bene.