UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Tesi di Laurea Triennale in Informatica

Compressione dati e connessioni con l'intelligenza artificiale

Relatore
Prof. Mignosi Filippo

Correlatore
Dr. Spezialetti Matteo

Candidato
Michetti Matteo
265459

Anno Accademico 2023-2024

Abstract

In questo lavoro si affronta il tema della compressione dati in relazione con l'intelligenza artificiale, a seguito di recenti risultati di notevole importanza che affermano la "quasi equivalenza" dei due argomenti.

Nel primo capitolo daremo una rapida introduzione alla compressione dati. La compressione dati è una branca della teoria dell'informazione che ne studia i fondamenti e i limiti teorici. Presenteremo i codici aritmetici e un interessante collegamento con il gioco d'azzardo come caso specifico di predizione ed infine come utilizzare un qualsiasi compressore per predire/generare il token (lettera, o parola o altro) successivo. Pur essendo una tecnica semplice fa dunque capire come utilizzare un qualsiasi compressore come una intelligenza "generativa".

Nel secondo capitolo si tratterà di Deep Learning. Dopo un'introduzione molto breve sull'argomento, si noterà che l'AI in generale ha come sotto-argomento il "Machine Learning" il quale, a sua volta ha come sotto-argomento il "Deep-Learning" di cui gli LLM (Large Language Models) come ChatGpt, Gemini, Llama e Claude, che stanno rivoluzionando la nostra società, sono una sotto-branca. Si farà una piccola digressione sulle problematicità di questi modelli, e ci chiederemo se effettivamente si stia andando verso una intelligenza artificiale "generale" semplicemente con un meccanismo di "scala" (tramite un allenamento con decine di terabytes di dati) oppure sia necessario l'utilizzo congiunto di "motori inferenziali". I principali LLM commerciali hanno un codice "chiuso" e non sappiamo se questo utilizzo congiunto sia già in uso o meno. Infine sempre nel secondo capitolo studieremo un articolo in cui si descrivono le capacità di compressione di un compressore basato su un LLM chiamato Chinchilla di Google DeepMind che raggiunge livelli di compressione impressionanti rispetto al passato, comprimendo testi medi in inglese con circa $0,63$ bit per carattere rispetto al bit per carattere dei migliori algoritmi umani conosciuti in precedenza.

Nel terzo ed ultimo capitolo, abbiamo riprodotto un piccolo LLM (un Medium Language Model dunque) simile a Chinchilla e messo a disposizione da Google-DeepMind tramite GitHub. Nonostante il fatto che il codice fosse noto, abbiamo avuto varie criticità e ne abbiamo risolte molte ma non tutte.

Nello specifico siamo riusciti ad allenare un Transformer con 200K parametri e ne abbiamo valutato le capacità di compressione su diversi dataset e abbiamo avuto qualche sorpresa che discuteremo nelle conclusioni.

Questa tesi è svolta con l'ausilio del Laboratorio HPC & Big Data del DISIM.

Indice

Elenco delle figure

Introduzione

In questo lavoro si è studiata la compressione dati, sottolineando quanto essa sia saldamente legata alla predizione. Il vertiginoso aumento della mole di dati trattati, sempre crescente negli ultimi anni, rende questa branca del sapere fondamentale nel mondo moderno e un pilastro nelle scienze informatiche e nelle telecomunicazioni. L'avvento delle reti neurali e delle loro evoluzioni, come gli LLM, potrebbe far pensare ad una nuova frontiera nella compressione. Molti algoritmi classici risultano superati in confronto alle capacità di compressione connesse all'intelligenza artificiale. In particolare le AI adibite all'elaborazione del linguaggio naturale, che grazie alle loro capacità di previsione e individuazione dei pattern apprendendo dinamicamente collegamenti e connessioni tra parole, risultano essere i compressori più efficienti in termini di tasso di compressione (ma non necessariamente in velocità di compressione) mai creati dal genere umano.

Capitolo 1

Compressione e predizione

In questo capitolo andremo ad esplorare la relazione che c'è tra compressione e predizione, facendo riferimento al capitolo 6 del libro "Elements of information theory"[6]

1.1 Compressione dati

La compressione dati è la tecnica che, mediante l'utilizzo di specifici algoritmi, fa sì che la quantità di bit necessari a codificare un'informazione sia ridotta, possibilmente la più piccola quantità possibile. Dato un file x, un algoritmo di compressione dati genera un file x_c dove la dimensione di $x_c \leq x$ e tramite il processo di decompressione è possibile ricostruire il file originale x da x_c. Tale tecnica rende possibile diminuire le dimensioni di un flusso di informazioni e, conseguentemente, sia lo spazio di memorizzazione che la banda necessaria per la trasmissione di esso. In tal modo si arriva ad avere comunicazioni efficienti e veloci, ottimizzando tra l'altro servizi tra cui streaming video o il cloud storage. La compressione dati si divide in due principali gruppi che sono:

- **Compressione lossy**: processo di compressione dati che prevede la perdita di parti di informazione. Messa in atto la compressione lossy non è più possibile ripristinare il dato originario.

- **Compressione lossless**: processo di compressione dati senza alcun tipo di perdita di informazione. Con la compressione lossless il dato originale è sempre esattamente ricostruibile dal dato compresso.

L'utilizzo di una tecnica piuttosto che un'altra dipende sopratutto dal tipo di dato che si andrà a comprimere. La compressione dati lossy, ad esempio, è utilizzata soprattutto per comprimere file multimediali. La qualità del file ovviamente può peggiorare anche se le informazioni perse dovrebbero essere quelle meno rilevanti e che definivano in maggior modo i dettagli oppure parti di immagini o suoni non rilevabili dai sensi umani. Comprimere invece un file di testo o un programma, chiaramente richiede che dal dato compresso sia possibile ripristinare esattamente il dato originale. Gli algoritmi di compressione dati lossless, possono essere visti come delle funzioni iniettive che mappano in media sequenze di bit più lunghe in sequenze di bit più corte (in alcuni casi potrebbero anche fare l'inverso). Da un punto di vista formale un compressore può essere visto semplicemente come una funzione iniettiva e basta. Una funzione iniettiva (un compressore) non può comprimere tutte le possibili stringhe, poiché il numero delle stringhe binarie di lunghezza n è sempre maggiore del numero di tutte le stringhe più corte (escludendo la stringa vuota), e dunque, per il principio dei cassetti, un compressore non può comprimere tutto. In teoria dell'informazione si sono sviluppati i concetti di "sorgente" e di compressione media relativamente alla sorgente per formalizzare i principali risultati della teoria.

1.2 Il concetto di informazione e risultati classici

Claude Shannon nel 1948, pubblicò "A mathematical theory of communication", un articolo che ebbe un grandissimo impatto nel mondo delle comunicazioni e che introdusse diversi concetti sui quali si basa l'informatica odierna. Per definizioni ed esempi si veda il primo capitolo di [6].

Definizione 1. L'autoinformazione può essere considerata come una misura dell'incertezza contenuta in un evento singolo. Tanto più un evento ha probabilità alta di accadere tanto più il suo contenuto informativo sarà basso. Shannon nel suo trattato originale, definisce la *self-infromation* del valore assunto da una variabile aleatoria X come:

$$I(x) = -\log_b p(x)$$

associando l'informazione alla codifica codifica in base b di un evento con probabilità p.

Esempio 1. *Sia X una variabile aleatoria definita su $\Omega = \{T, C\}$ che mostra l'esito del lancio di una moneta non truccata. Sia $P(T) = P(C) = \frac{1}{2}$ la probabilità di ottenere testa o croce dal prossimo lancio, allora $I(T) = I(C) = -log_2(\frac{1}{2}) = 1(bit)$. Si è per cui ottenuto la quantità di informazione in bit, data dal logaritmo in base 2, dell'evento T o C, ovvero 1 bit, che difatti è la codifica per due eventi binari equiprobabili. Per chiarire ulteriormente si pensi invece ad una moneta truccata il cui esito è sempre testa, per cui $p(T) = 1$, ne consegue che $I(T) = -log_2(1) = 0(bit)$, ovvero l'evento non porta informazione. Effettivamente dal lancio di tale moneta non si otterrebbe alcuna informazione in quanto l'esito è sempre il medesimo, non esiste incertezza.*

Definizione 2. L'entropia di una variabile aleatoria X con funzione di probabilità $p(x)$ è definita

$$H(X) = -\sum_x p(x) \log_2 p(x)$$

L'entropia misura l'autoinformazione *media* (o valore atteso) associata ad X e quindi il numero di bit necessari in media per descrivere il valore che la variabile assume. La sua importanza risiede anche nel fatto che è stato dimostrato che l'entropia rappresenta il *lower bound* per la compressione senza perdita di dati di sorgenti IID, come stabilito dal primo teorema di Shannon. Parlando in modo non strettamente matematico, codificare tutti gli eventi associati a una determinata variabile random con un numero di bit in media per evento inferiore all'entropia comporterà inevitabilmente una perdita di informazione.

Esempio 2. *[6, Example 1.1.2] Supponiamo che in una gara di ippica, ci siano 8 cavalli ed essi abbiamo una probabilità di vittoria che sia*

$$\left(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{64}, \frac{1}{64}, \frac{1}{64}, \frac{1}{64}\right)$$

. Calcoliamo l'entropia della corsa

$$H(X) = -\left(\frac{1}{2}\log_2\frac{1}{2} + \frac{1}{4}\log_2\frac{1}{4} + \frac{1}{8}\log_2\frac{1}{8} + \frac{1}{16}\log_2\frac{1}{16} + 4\cdot\frac{1}{64}\log_2\frac{1}{64}\right) = 2 \ \textit{bits}$$

. Supponiamo poi, di voler inviare un messaggio contenente quale sia il cavallo vincitore. Numerando i cavalli, in binario, sono necessari 3 bits. Tuttavia, sapendo che le probabilità non sono uniformi, ha più senso scegliere di rappresentare con una codifica più breve il cavallo che ha un alta probabilità di vittoria e con una più lunga, quello con probabilità minore. Un esempio potrebbe essere 0, 10, 110, 1110, 111100, 111101, 111110, 111111. Questa codifica porta ad una lunghezza media del messaggio pari a 2 bits, coincidente con l'entropia, per cui è una codifica ottimale.

Molto spesso gli eventi, e quindi le variabili aleatorie che li descrivono, sono dipendenti l'uno dall'altro. Sappiamo che l'entropia misura l'incertezza media di una singola variabile aleatoria; adesso invece si introduce l'*entropia condizionale* $H(X|Y)$, che rappresenta l'entropia di una variabile aleatoria condizionata dalla conoscenza di un 'altra variabile aleatoria. La riduzione di tale incertezza data dalla conoscenza di un'ulteriore variabile è chiamata **Informazione mutua** o anche mutua Informazione.

Definizione 3. Siano X e Y due variabili aleatorie, definiamo informazione mutua

$$I(X;Y) = H(X) - H(X|Y) = \sum p(x,y)\log\frac{p(x,y)}{p(x)p(y)}$$

Essa rappresenta la misura della dipendenza tra due variabili casuali.

Definizione 4. La *Distanza di Kullbak-Leibler* [1] tra due distribuzioni $p(x)$ e $q(x)$ è definita come:

$$D(p||q) = \sum_{x\in X} p(x)\log\frac{p(x)}{q(x)}$$

Essa misura l'inefficienza nell'assumere la distribuzione q piuttosto che la distribuzione vera p. Ad esempio, se conoscessimo la vera distribuzione p della variabile aleatoria, potremmo costruire un codice con lunghezza media $H(p)$. Se invece usassimo il codice per una distribuzione q, avremmo bisogno di $H(p) + D(p||q)$ bit in media per descrivere la variabile casuale.

A volte questa inefficienza viene detta la misura della informazione negativa (o disinformazione) che sarebbe quella di essere stati convinti da un avversario che la distribuzione sia q piuttosto che la distribuzione vera p. Esiste comunque una diatriba nel mondo scientifico sulla esistenza della informazione negativa intesa come "falsa". Tale diatriba si connette a un teorema ([6, Theorem 2.6.5]) che ha come commento degli stessi autori "l'informazione non può fare danno", ove però si sottintende che l'informazione ricevuta sia vera, ovvero non falsa o "disinformazione".

Definizione 5. Sia X una variabile aleatoria che assume definita sull'insieme $\mathcal{X}$. Un *codice a blocco* C è una funzione che mappa:

$$C : \mathcal{X} \to D^*$$

dove:

[1] La *Distanza di Kullbak-Leibler* è anche conosciuta con altri nomi come: cross entropy in alcune forme, information divergence, information for discrimination e altro [6]

- $\mathcal{X}$ è l'insieme dei valori che X può assumere
- D^* è l'insieme delle stringhe di lunghezza finita costruite utilizzando un alfabeto di cardinalità D.

Per ogni $x \in \mathcal{X}$, $C(x)$ denota la *code-word* corrispondente a x, e $l(x)$ denota la *lunghezza* di $C(x)$.

I codici a blocco in inglese a volte vengono chiamati *source codes.*

Esempio 3. *Ad esempio,* $C(red) = 00$ *e* $C(blue) = 11$ *è un codice per* $X = \{red, blue\}$ *con alfabeto* $D = \{0, 1\}$.

Definizione 6. La lunghezza attesa $L(C)$ di un codice $C(x)$ per una variabile aleatoria X con probabilità $p(x)$ è data da

$$L(C) = \sum_{x \in \mathcal{X}} p(x) l(x),$$

dove $l(x)$ è la lunghezza della parola codice associata a x.

Esempio 4. *[6, Example 5.1.1] Sia X una variabile aleatoria con la seguente distribuzione e assegnazione delle parole codice:*

$$\Pr(X = 1) = \frac{1}{2}, \quad C(1) = 0$$

$$\Pr(X = 2) = \frac{1}{4}, \quad C(2) = 10$$

$$\Pr(X = 3) = \frac{1}{8}, \quad C(3) = 110$$

$$\Pr(X = 4) = \frac{1}{8}, \quad C(4) = 111$$

$$H(X) = -\left(\frac{1}{2} \cdot \log_2 \frac{1}{2} + \frac{1}{4} \cdot \log_2 \frac{1}{4} + \frac{1}{8} \cdot \log_2 \frac{1}{8} + \frac{1}{8} \cdot \log_2 \frac{1}{8}\right) = 1.75 \ bit$$

$$L(C) = \left(\frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \frac{1}{8} \cdot 3\right) = 1.75 \ bit$$

Qui abbiamo un codice che ha la stessa lunghezza media dell'entropia.

Si noti che qualsiasi sequenza di bit può essere univocamente decodificata in una sequenza di simboli di X.

Ad esempio, la stringa di bit 0110111100110 *viene decodificata come* 134213.

La teoria dei codici a blocco è vasta e non si ha il tempo (e l'interesse) di svilupparla in questa tesi. Si veda il Capitolo 5 del [6] per una valida introduzione. Diciamo solamente che i codici a blocco la cui estensione alle parole rappresentano delle funzioni iniettive, e dunque dei compressori lossless, si chiamano codici a "univoca decifrabilità" i quali contengono la importante sottoclasse dei codici iniettivi (o liberi dal prefisso). Appartiene a Shannon il source coding Theorem (dal fatto che i codici a blocco sono anche chiamati "source codings") che afferma che la lunghezza media di un codice a univoca decifrabiltà è sempre maggiore o uguale all'entropia (dove la sorgente è IID e ha come distribuzione di probabilità la stessa con cui si calcola la lunghezza media). Ulteriori generalizzazioni a tutte le funzioni iniettive (ovvero a tutti i compressori) sono state fatte da Kinchin nel 1953 ([2]) e da altri autori (si veda la parte storica in [3]).

1.2.1 I Codici Aritmetici

Questa sezione è principalmente tratta da [15] . La codifica aritmetica è una tecnica di compressione dati che rappresenta sequenze di simboli come intervalli di numeri reali. Questa metodologia è distinta rispetto ad altre tecniche di codifica, poiché non assegna direttamente un numero fisso di bit a ciascun simbolo, ma piuttosto rappresenta l'intera sequenza di dati come un singolo numero reale nell'intervallo $[0,1)$. Più precisamente a un singolo numero reale corrisponde una sequenza infinita di simboli. Se il messaggio da codificare è lungo N esiste un sotto-intervallo $[\alpha,\beta)$ di $[0,1)$ tale che ogni numero reale in questo intervallo corrisponde a una sequenza infinita che inizia con gli N simboli del messaggio da codificare. Quindi qualsiasi numero reale in $[\alpha,\beta)$ (e si cerca di prenderne uno rappresentabile col minor numero di bit) insieme col numero N scritto in binario, sono sufficienti per ricostruire in modo univoco e quindi lossless il messaggio originale.

1.2.2 Processo di codifica e decodifica

Codifica

Fondamentalmente, il processo di codifica consiste nel creare una sequenza di intervalli annidati nella forma $\Phi_k(S) = [\alpha_k, \beta_k)$, $k = 0, 1, \ldots, N$, dove S è la sequenza di dati, α_k e β_k sono numeri reali tali che $0 \leq \alpha_k \leq \alpha_{k+1}$ e $\beta_{k+1} \leq \beta_k \leq 1$. Più semplicemente possiamo rappresentare tali intervalli nella forma $|b,l\rangle$, dove b è il punto di partenza dell'intervallo, e l la lunghezza dell'intervallo. La relazione tra le due notazioni è la seguente

$$|b,l\rangle = [\alpha,\beta) \text{ se } b = \alpha \text{ e } l = \beta - \alpha$$

. Gli intervalli utilizzati durante il processo di codifica sono quindi definiti :

$$\Phi_0(S) = |b_0, l_0\rangle = |0, 1\rangle$$

,

$$\Phi_k(S) = |b_k, l_k\rangle = |b_{k-1} + c(s_k)l_{k-1}, p(s_k)l_{k-1}\rangle, k = 1, 2, \ldots, N$$

Le proprietà degli intervalli garantiscono che $0 \leq b_k \leq b_{k+1} < 1$, e $0 < l_{k+1} < l_k \leq 1$.

L'obiettivo finale nella codifica aritmetica è definire un valore di codice $\hat{v}(S)$ che rappresenterà la sequenza di dati S.

Decodifica

Nel processo di decodifica invece, la sequenza decodificata è determinata unicamente dal valore di codice $\hat{v}$ della sequenza compressa. Per questo motivo, rappresentiamo la sequenza decodificata come

$$\hat{S}(\hat{v}) = \{\hat{s}_1(\hat{v}), \hat{s}_2(\hat{v}), \ldots, \hat{s}_N(\hat{v})\}$$

Mostriamo ora il processo di decodifica con cui qualsiasi valore di codice $\hat{v} \in \Phi_N(S)$ può essere utilizzato per decodificare la sequenza corretta (cioè, $\hat{S}(\hat{v}) = S$). Presentiamo il set di equazioni ricorsive che implementano la decodifica. Il processo di decodifica recupera i simboli dei dati nella stessa sequenza in cui sono stati codificati. Formalmente, per trovare la soluzione numerica, definiamo una sequenza di valori di codice normalizzati $\{\tilde{v}_1, \tilde{v}_2, \ldots, \tilde{v}_N\}$. Iniziando con $\tilde{v}_1 = \hat{v}$, troviamo sequenzialmente $\hat{s}_k$ da $\tilde{v}_k$, e quindi calcoliamo $\tilde{v}_{k+1}$ da $\hat{s}_k$ e $\tilde{v}_k$. Le formule ricorsive sono

$$\tilde{v}_1 = \hat{v},$$

$$\hat{s}_k(\hat{v}) = \{s : c(s) \leq \tilde{v}_k < c(s+1)\}, \quad k = 1, 2, \ldots, N,$$

$$\tilde{v}_{k+1} = \frac{\tilde{v}_k - c(\hat{s}_k(\hat{v}))}{p(\hat{s}_k(\hat{v}))}, \quad k = 1, 2, \ldots, N-1.$$

Esempio 5. *Supponiamo che la sorgente Ω abbia quattro simboli ($M = 4$), le probabilità e la distribuzione dei simboli sono rispettivamente $p = [0.2, 0.5, 0.2, 0.1]$ e $c = [0, 0.2, 0.7, 0.9, 1]$, e la sequenza di simboli ($N = 6$) da codificare è $S = \{2, 1, 0, 0, 1, 3\}$.*

La figura 1.1 mostra graficamente come il processo di codifica corrisponde alla selezione degli intervalli nella linea dei numeri reali. Iniziamo nella parte superiore della figura, con l'intervallo $[0, 1)$, che è diviso in quattro sottointervalli, ciascuno con una lunghezza pari alla probabilità dei simboli di dati. Specificamente, l'intervallo $[0, 0.2)$ corrisponde a $s_1 = 0$, l'intervallo $[0.2, 0.7)$ corrisponde a $s_1 = 1$, l'intervallo $[0.7, 0.9)$ corrisponde a $s_1 = 2$, e infine l'intervallo $[0.9, 1)$ corrisponde a $s_1 = 3$. Il successivo insieme di sottointervalli nidificati consentiti ha anche lunghezza proporzionale alla probabilità dei simboli, ma le loro lunghezze sono anche proporzionali alla lunghezza dell'intervallo a cui appartengono. Inoltre, essi rappresentano più di un valore simbolico. Ad esempio, l'intervallo $[0, 0.04)$ corrisponde a $s_1 = 0, s_2 = 0$, l'intervallo $[0.04, 0.14)$ corrisponde a $s_1 = 0, s_2 = 1$, e così via.

Ottenuto quindi $\hat{v}$ possiamo iniziare la decodifica dal primo intervallo $\Phi_0(S) = [0, 1)$: dobbiamo solo confrontare $\hat{v}$ con la distribuzione cumulativa c per trovare l'unico valore possibile di $\hat{s}_1$

$$\hat{s}_1(\hat{v}) = \{s : c(s) \leq \hat{v} = 0.74267578125 < c(s+1)\} = 2.$$

Possiamo usare il valore di $\hat{s}_1$ per determinare l'intervallo $\Phi_1(S)$, e usarlo per determinare $\hat{s}_2$. Infatti, possiamo "rimuovere" l'effetto di $\hat{s}_1$ in $\hat{v}$ definendo il valore di codice normalizzato

$$\tilde{v}_2 = \frac{\hat{v} - c(\hat{s}_1)}{p(\hat{s}_1)} = 0.21337890625.$$

Si noti che, in generale, $\tilde{v}_2 \in [0, 1)$, cioè è un valore normalizzato all'intervallo iniziale. In questo intervallo possiamo usare lo stesso processo per trovare

$$\hat{s}_2(\hat{v}) = \{s : c(s) \leq \tilde{v}_2 = 0.21337890625 < c(s+1)\} = 1.$$

Il processo di decodifica termina non appena è stato trovato $\hat{s}_6$

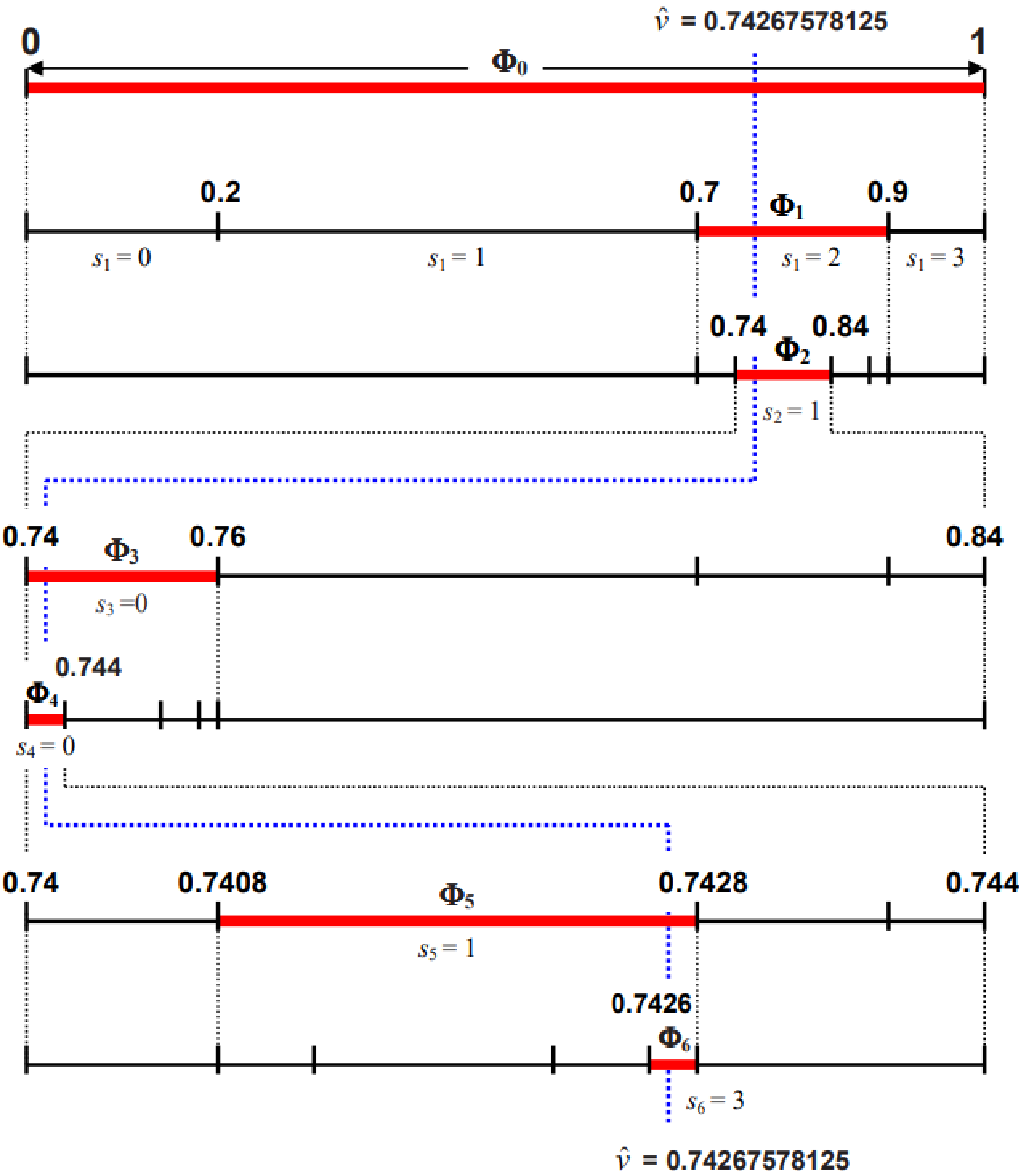

Figura 1.1: Rappresentazione grafica del processo di codifica.

1.2.3 Considerazioni

Non proseguiamo oltre con le tecniche di scaling. Osserviamo solamente che è relativamente semplice provare che le codifiche aritmetiche convergono all'entropia per sorgenti IID tramite la legge debole dei grandi numeri. Tale convergenza è dell'ordine $O(log(N)/N)$ dove N è la lunghezza del messaggio. Infatti oltre all'overhead fisso bisogna sempre codificare ed inviare la lunghezza N del messaggio per potere terminare la decodifica, e tale codifica è dell'ordine di $log(N)$. Comunque le costanti in gioco sono piccole e pur essendo presente questo $log(N)$, in pratica la codifica aritmetica è quasi sempre migliore di tutte le altre codifiche incluse quelle che convergono all'entropia come $O(1/N)$.

Infatti, grazie a tecniche di scaling e ad altre tecniche, nella pratica le risorse computazionali necessarie al funzionamento delle codifiche aritmetiche risulta limitato e le codifiche aritmetiche sono una delle tecnologie più di successo nell'ambito della compressione dati. Per questo motivo grandi imprese (come la HP e IBM) hanno mantenuto per decenni i loro brevetti attivi. Bisogna sottolineare che oramai i brevetti sulle codifiche aritmetiche sono quasi tutti scaduti ed esse sono utilizzabili da tutti.

Come anche osservato in [15] le codifiche aritmetiche possono essere usate su sorgenti non IID ma generiche, a patto di avere una buona modellizzazione della sorgente capace di fornire in ogni punto del testo, sia in fase di codifica che di decodifica, le probabilità dei simboli. Le probabilità sono costanti nel caso delle IID e quindi si hanno in entrambe le fasi di codifica e decodifica.

Piuttosto, potrebbero essere necessarie importanti risorse computazionali per avere le distribuzioni di probabilità del prossimo carattere in funzione della posizione del testo.

I transformer, come vedremo, si occupano di questa parte che viene anche chiamata modellizzazione. E grandi case come OpenAI e Google sembra che consumino l'equivalente di energia elettrica, esclusivamente a questo scopo, pari a quella consumata da stati come l'Italia o la Spagna. La parte di risorse computazionali necessarie alla codifica e alla decodifica riservata ai codici aritmetici, qualora si volessero utilizzare i Transformer accoppiati ai codici aritmetici per comprimere, è senz'altro, in confronto, praticamente irrisoria. Di questo ci occuperemo nei prossimi capitoli.

1.3 Compressione e gioco d'azzardo

Sulla base dei concetti mostrati sin ora è già possibile notare una correlazione tra la probabilità e compressione. In questa sezione in particolare, esploriamo una connessione che può non essere lampante ma che in realtà è molto solida, che è quella tra compressione e gioco d'azzardo. Si definiscono quindi, alcuni concetti che saranno utilizzati in seguito.

1.3.1 La corsa dei cavalli

Si assuma che n cavalli stiano gareggiando in una corsa. L'i-esimo cavallo ha una probabilità di vincere la corsa pari p_i, se esso vince il montepremi sarà pagato come o_i a 1: ovvero con l'investimento di un dollaro sul cavallo i-esimo si avranno o_i dollari se il cavallo vince e zero dollari se il cavallo perde.

Una strategia di scommessa che si potrebbe adottare è quella di distribuire l'importo da scommettere frazionandolo per ogni cavallo in gara. Sia b_i la porzione di importo scommesso sul cavallo i-esimo, con i vincoli che $b_i \geq 0$ e $\sum b_i = 1$, dalla vincita della corsa del cavallo i, si ottiene una guadagno pari a o_i volte b_i, tutte le altre scommesse risultano perdenti. Il guadagno a fine gara è una variabile aleatoria che si vuole, ovviamente, massimizzare. Considerando la possibilità di reinvestire le vincite, il guadagno sarà:

$$S_n = \prod_{i=1}^{n} S(X_i)$$

dove $S(X) = b(X) \cdot o(X)$ è il fattore moltiplicativo del guadagno quando il cavallo X vince.

Definizione 7. Siano le corse $X_1, X_2, X_3...$ variabili aleatorie con probabilità $p(x)$, allora il guadagno ottenuto usano la strategia di scommessa $\mathbf{b}$ cresce esponenzialmente con un tasso $W(\mathbf{b}, \mathbf{p})$, per cui

$$S_n = 2^{W(\mathbf{b},\mathbf{p})}$$

dove $W(\mathbf{b}, \mathbf{p})$ rappresenta il *tasso di raddoppio* definito come

$$\sum_{k=1}^{m} p_k \log(b_k o_k)$$

Definizione 8. Si introduce il *tasso di raddoppio ottimale* come il tasso di raddoppio massimo tra tutte le scelte di $\mathbf{b}$, scritto come $W^*(\mathbf{p}) = \max_b W(\mathbf{b}, \mathbf{p})$

In particolare, si dimostra come massimizzare $W(\mathbf{b}, \mathbf{p})$ in funzione di $\mathbf{b}$ soggetto al vincolo $\sum b_i = 1$, sia porre $\mathbf{b}$=$\mathbf{p}$, ovvero la puntata sull'evento è congrua alla probabilità dell'evento stesso, una strategia chiamata *puntata proporzionale*.

Definizione 9. Il tasso di raddoppio ottimale è dato da

$$W^*(\mathbf{p}) = \ p_i \log o_i - H(p)$$

e si ottiene ponendo la puntata proporzionale $\mathbf{b^*} = \mathbf{p}$

Dimostrazione.

$$\begin{aligned}
W(\mathbf{b}, \mathbf{p}) &= \sum_i p_i \log b_i o_i \\
&= \sum_i p_i \log \left(\frac{b_i}{p_i}\right) + \sum_i p_i \log o_i \\
&= \sum_i p_i \log o_i - H(\mathbf{p}) - D(\mathbf{p} \| \mathbf{b}) \\
&\leq \sum_i p_i \log o_i - H(\mathbf{p}),
\end{aligned}$$

□

Si consideri adesso il caso in cui le quote siano eque $\sum \frac{1}{o_i} = 1$. Si scrive $r_i = \frac{1}{o_i}$ dove r_i rappresenta la distribuzione di probabilità di ogni cavallo in gara, ovvero è la stima della probabilità di vincita da parte del bookmaker. Allora, possiamo esprimere il tasso di raddoppio:

$$\begin{aligned}
W(b, p) &= \sum_i p_i \log \frac{b_i}{p_i} \\
&= \sum_i p_i \log \frac{b_i}{p_i r_i} \\
&= D(p \parallel r) - D(p \parallel b)
\end{aligned}$$

Questa equazione fornisce una differente prospettiva della cross entropy introdotta in precedenza. Il tasso di raddoppio quindi è interpretabile come la distanza tra la stima del della distribuzione reale del banco e del giocatore, da cui si evince che il giocatore guadagna solo se la propria stima **b** è migliore di quella del banco.

1.3.2 Gioco d'azzardo e informazione mutua

Si supponga che un giocatore sia in possesso di informazioni rilevanti riguardo ciò che potrebbero essere gli esiti delle scommesse. Ad esempio, informazioni relative alla performance di alcuni cavalli nelle precedenti gare. Ci si chiede quale sia il valore di questa ulteriore conoscenza e l'influenza che ha sul gioco. Sia $X \in \{1, 2, \ldots, m\}$, il cavallo che vince la corsa con probabilità $p(x)$ pagando $o(x)$ a 1. Sia $p(x, y)$ la distribuzione congiunta associata a $(X|Y)$. Sia $b(x|y) \geq 0$, $\sum b(x|y) = 1$, una strategia di scommessa adattata all'informazione mutua data da Y, dove $b(x|y)$ è la porzione di capitale puntato sul cavallo x quando viene osservato y, definiamo il *tasso di raddoppio ottimale condizionato* come

$$W^*(X|Y) = \max_{b(x|y)} \sum_{x,y} p(x, y) \log b(x|y)o(x)$$

Dalla conoscenza di un'informazione addizionale ci si aspetta una minore entropia e un incremento del guadagno che si traduce una variazione del tasso di raddoppio, $\Delta W = W^*(X|Y) - W^*(X)$, il quale coincide esattamente con $I(X; Y)$, l'informazione mutua.

1.3.3 Comprimere scommettendo

Uno scommettitore quando piazza una scommessa altro non fa che prevedere, cioè associare una probabilità più alta all'accadere di quell'evento, basandosi su eventi pregressi. Difatti un "bravo" giocatore, inteso come colui che è stato il grado di guadagnare maggiormente dalle proprie scommesse, avrà stimato la probabilità degli eventi su cui gioca con una distribuzione molto simile se non uguale alla distribuzione reale degli eventi. Vediamo più avanti un piccolo algoritmo che si basa su questo concetto, molto simile alla codifica aritmetica, dove a guadagni maggiori si fanno corrispondere codici di lunghezza minore.

1.3.4 I gemelli scommettitori

Immaginiamo uno scommettitore che abbia un gemello identico. Quest'ultimo piazza esattamente le stesse scommesse sulle possibili sequenze di risultati del fratello scommettitore, ottenendo così lo stesso guadagno per ciascuna scommessa. Il capitale cumulato da tutte le scommesse che il giocatore ha piazzato su tutte le sequenze minori in ordine lessicografico della sequenza da comprimere sarà usato come codice della sequenza. Il gemello altro non fa che replicare le scommesse di suo fratello, prestando attenzione alla sequenza che genera lo stesso capitale della sequenza da comprimere.

Sia $X_1, X_2, ...X_n$ una sequenza di variabili aleatorie che desideriamo comprimere. Senza perdita di generalità, assumiamo che esse siano binarie.La scommessa su questa sequenza è definita da una serie di puntate

$$b(x_{k+1}|x_1, x_2, ...x_k) >= 0$$

s.t

$$\sum_{x_{k+1}} b(x_{k+1}|x_1, x_2, ...x_k) = 1$$

dove $b(x_{k+1}|x_1, x_2, ...x_k)$ è la proporzione di capitale scommesso a tempo k su l'evento $X_{k+1} = x_{k+1}$ data l'osservazione degli eventi passati $x_1, x_2, ...x_k$.

Le scommesse sono pagate 2a1, ovvero il capitale scommesso viene raddoppiato. Possiamo quindi affermare che il guadagno finale S_N sarà

$$S_N = 2^n \prod_{k=1}^{n} b(x_k|x_1, \dots, x_{k-1}) = 2^n b(x_1, x_2, \dots, x_n)$$

dove

$$b(x_1, x_2, \dots, x_n) = \prod_{k=1}^{n} b(x_k|x_{k-1}, \dots, x_1)$$

Per tanto scommettere in modo sequenziale può essere considerato come una assegnamento di probabilità $b(x_1, x_2, ..., x_n) \geq 0$, $\sum_{x_1, \dots, x_n} b(x_1, x_2, ..., x_n) = 1$ sulle 2^n possibili sequenze. In questo modo riusciamo ad ottenere sia una stima $\hat{p}(x_1, \dots, x_n) = \frac{S_n}{2^n}$ della reale probabilità , sia una stima dell'entropia $[\hat{H} = -\frac{1}{n} \log_2 \hat{p}]$. Mostriamo per cui come alti guadagni S_n portano ad un alto tasso di compressione.

1.3.5 L'algoritmo dei gemelli scommettitori

Il seguente algoritmo mappa una sequenza $\mathbf{x} = x_1, x_2, ..x_n$ in codice binario $c_1, c_2...c_k$. Sia il compressore che il decompressore conoscono n. Siano le 2^n sequenze ordinate in ordine lessicografico. Il compressore osserva la sequenza $x^n = (x_1, x_2, ..., x_n)$ e calcola il capitale $S_n(x'(n))$ di tutte le sequenze $x'(n) \leq x(n)$ e calcola $F(x(n)) = \sum_{x'(n) \leq x(n)} \frac{S_n(x'(n)}{2^n}$. Sia $k = \lceil n - \log S_n(x(n)) \rceil$, scriviamo $F(x(n))$ come un decimale binario con k cifre dopo la virgola, $\lfloor F(x(n)) \rfloor = .c_1 c_2 \cdots c_k$. La sequenza $c(k) = (c_1, c_2, \dots, c_k)$ sarà trasmessa la decodificatore. Il gemello, calcolerà il valore di $S(x'(n))$ associato ad ognuna delle 2^n sequenze $x'(n)$. A questo punto è in grado di determinare $F(x(n))$ sommando il valore di $\frac{S_n(x'(n)}{2^n}$ per ogni sequenza $x(n)$. La prima sequenza $x(n)$ che farà eccedere $.c(k)$ sarà quella trasmessa, in questo modo il gemello recupera $x(n)$. Il numero di bit richiesti è $k = \lceil n - \log S(x(n)) \rceil$, il numero di bit risparmiati è $n - k = \lfloor \log S(x(n)) \rfloor$.

Considerazioni

Lo schema di compressione dei dati che utilizza uno scommettitore è simile all'idea della codifica aritmetica utilizzando una distribuzione $b(x_1, x_2, \dots, x_n)$ piuttosto che la distribuzione reale. La procedura sopra esposta evidenzia la dualità tra gioco d'azzardo, inteso come previsione di eventi, e compressione dei dati. Migliore è la stima, maggiore è il tasso di crescita della ricchezza dello scommettitore e migliore è la compressione dei dati.

1.4 Usare un compressore per predire

In questa sezione si fa riferimento a [9]. Abbiamo visto nella sezione precedente come un predittore sia effettivamente un compressore. Andiamo adesso a verificare l'opposto, ogni compressore può essere trasformato in un modello predittivo, utilizzando le lunghezze delle sequenze compresse per determinare le distribuzioni di probabilità.

1.4.1 Il metodo

L'idea principale è di definire la probabilità di una sequenza $\rho(x_{1:n})$ come $2^{-\ell_c(\cdot)}$, dove $\ell_c(x_{1:n})$ rappresenta la lunghezza della sequenza $x_{1:n}$ quando codificata con un compressore c.

Si consideri una sequenza di simboli $\mathbf{x} = x_1, x_2, ..x_n \in X^n$ di lunghezza n definita sull'alfabeto X^n. Calcoliamo la lunghezza della sequenza compressa $\ell_c(x_{<n}b)$ da un compressore c, $\forall b \in X$. Otteniamo per cui la stima della distribuzione $\hat{p}(b \mid x_{<i}) = 2^{\ell_c(x_{<i}) - \ell_c(x_{<i}b)}$, ciò può essere esteso campionando l'intera sequenza in modo autoregressivo, "appendendo" l'ultimo output alla sequenza e iterando, ottenendo per cui la distribuzione di tutti i simboli che compongono la sequenza. In questo modo a partire da stringhe compresse, estraiamo le previsioni sui simboli, rimarcando ulteriormente la forte connessione tra compressione e predizione.

Capitolo 2

Evoluzione nella compressione con gli LLM

In questo capitolo si tratterà di come si è passati dalle reti neurali più basilari agli LLM e come il loro approccio alla modellazione del linguaggio basato solo sulla previsione possa rivelarsi utile per la compressione ma non sufficiente per riprodurre in modo esaustivo tutte le differenti capacità che definiscono l'intelligenza. Gli LLM sono un sottoinsime del Deep Learining, una categoria del Machine Learining, verranno descritti nel dettaglio più avanti in questo capitolo.

2.1 Introduzione alle reti neurali

Il campo delle reti neurali è assai vasto, numerosi sono gli aspetti che le caratterizzano, nelle sezioni successive ci si limiterà ad un'analisi generale ed introduttiva dell'argomento senza scendere nel dettaglio e riportando solamente ciò che ho ritenuto essenziale ai fini della comprensione del ruolo che giocano nella compressione dati.
Le reti neurali rappresentano un paradigma innovativo e potente nella programmazione. A differenza dell'approccio tradizionale, in cui il programmatore fornisce istruzioni dettagliate su come risolvere un problema, le reti neurali permettono al computer di apprendere autonomamente dai dati. Questo apprendimento automatico si basa sull'analisi di grandi quantità di dati, dai quali la rete deduce e sviluppa le proprie soluzioni. Il campo delle reti neurali ha fatto un enorme balzo avanti nell'ultimo decennio, con una costante evoluzione rivoluzionando il modo di affrontare problemi complessi come la visione artificiale, il riconoscimento vocale e l'elaborazione del linguaggio naturale. Queste tecnologie sono ora fondamentali e ampiamente adottate da giganti del settore tecnologico come Google, Microsoft e Facebook.

Neuroni Artificiali

Alla base delle reti neurali troviamo i neuroni artificiali. Il loro funzionamento si basa sul raccogliere un input $x_1, x_2...., x_k$, a cui associare un peso $w_1, w_2, ..., w_k$ ottenendo cosi una somma pesata $\sum_{i=1}^{k} w_i x_i$ per ogni componente ed infine aggiungere un bias b del neurone stesso. I pesi determinano l'"importanza" associata a ciascun componente dell' input, il bias è un parametro di regolazione che permette al neurone di adattare la sua soglia di attivazione. La somma pesata con il bias vengono passate alla funzione di attivazione del neurone ottenendo:

$$f((w, x) + b)$$

La **funzione di attivazione** o *activation function* rappresenta la funzione che determina l'output del neurone, le più utilizzate sono:

- **Sigmoid function**, essa è definita come

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}$$

  esplicitandola al nostro input

$$\frac{1}{1 + \exp\left(-\sum_i w_i x_i - b\right)}$$

  Essa restituisce un valore compreso tra 0 e 1. Un caso d'uso potrebbe essere in quelle reti progettate nel categorizzare degli oggetti in delle immagini, per valori vicini a 0 si avrà un oggetto e per valori tendenti a 1 un altro.

- **Hyperbolic Tangent Function**, definita come

$$\tanh(w \cdot x + b)$$

  Differisce dalla funzione sigmoidea perchè produce valori tra -1 e 1 anzichè tra 0 e 1.

- **ReLU** (Rectified Linear Unit) definita come

$$\max(0, w \cdot x + b)$$

  La ReLU restituisce lo stesso valore dato in input ovvero $f(x) = x$ quando $x \geq 0$ e 0 per valori negativi. Viene preferita alla precedenti due in quanto tende meno a saturarsi, ovvero a terminare di apprendere nel momento in cui i valori di output sono troppi vicini a 0 o 1 in caso della sigmoidea o tra -1 e 1 per la tangente iperbolica. Nelle ReLU l'aumentare del peso dell'input non ne causerà mai la saturazione, tuttavia valori negativi fanno sparire il gradiente e quindi terminare l'addestramento.

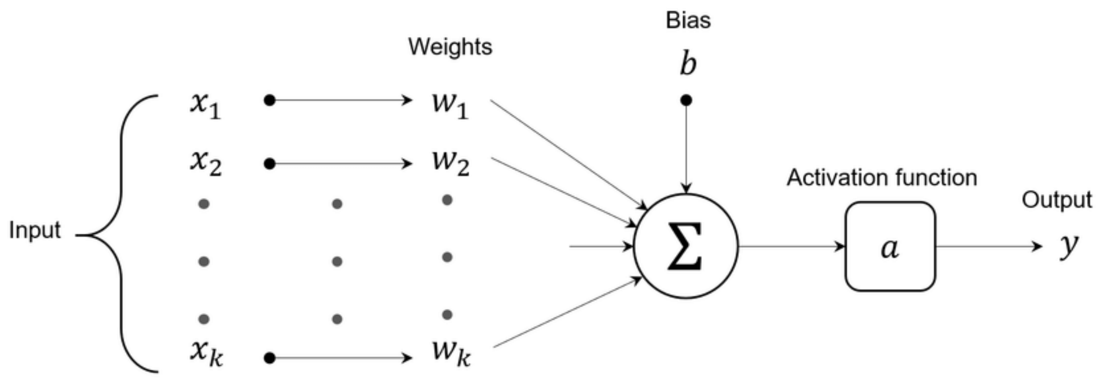

Figura 2.1: Rappresentazione di un neurone artificiale. Immagine via [4] v1

L'architettura di una rete neurale

L'architettura di una rete neurale è organizzata in strati (*layers*), ciascuno composto da un numero variabile di neuroni. Ogni rete neurale è costituita da almeno tre tipi layers: un layer di input, uno o più strati nascosti (*hidden layers*), e un layer di output. Lo strato di input è quello che riceve direttamente i dati esterni.

Gli hidden layers sono responsabili dell'elaborazione intermedia e contengono neuroni che applicano trasformazioni ai dati ricevuti dallo strato precedente mediante le sopracitare funzioni di attivazione, come la funzione sigmoidea o ReLU.

Infine, lo strato di output produce il risultato finale della rete, che può essere una classificazione, una previsione, o qualsiasi altro output desiderato, a seconda del tipo di problema che la rete è progettata per risolvere. Una rete in cui le connessioni tra nodi non formano cicli sono dette reti **feedforward**, le informazioni si muovono solo in avanti avanti nella rete, per cui non si ha memoria degli input precedenti, in questo modo l'output dipende esclusivamete dai dati in ingresso attuali.

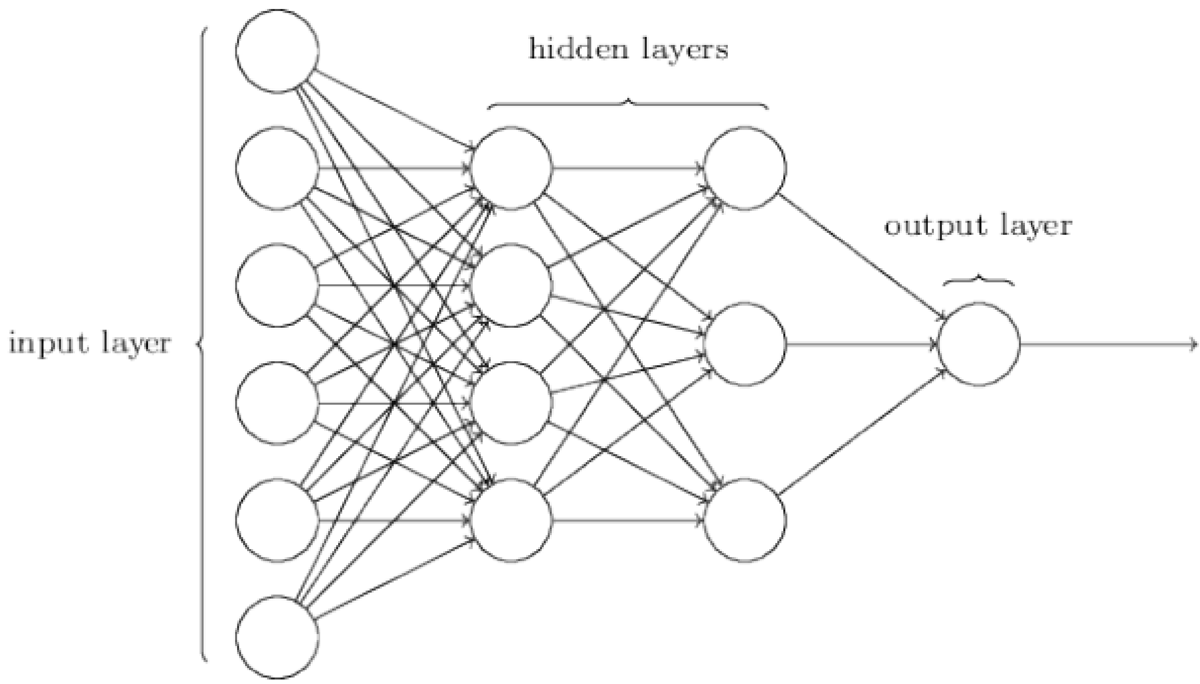

Figura 2.2: Rappresentazione grafica di una rete neurale con più *hidden layers*.

L'addestramento di una rete neurale

L'addestramento di una rete neurale avviene attraverso una calibrazione dei pesi e del bias, che inizialmente saranno casuali, in modo da riuscire ad ricavare un output coerente con ciò che ci si aspettava di ricevere. Abbiamo denotato x come input, sia $y = y(x)$ l'output che ci si aspetta di ottenere a partire dall'input, quantifichiamo la correttezza dei pesi e il bias attraverso la funzione di costo, di cui se no conoscono diverse, una della più utilizzate è :

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

definita anche come *errore quadratico medio*, dove w e b indicano pesi e bias, n il numero totale degli input, a è il vettore di output quando gli viene dato in input x. Tanto più $C(w, b)$ tende a zero, tanto più l'output atteso e l'output effettivo sono vicini, ovvero pesi e bias sono corretti. Per cui lo scopo dell'addestramento è quello di trovare pesi e bias che minimizzano C, usando un metodo noto come *discesa del gradiente* in combinazione con la *backpropagation*.

2.1.1 Reti Neurali Ricorrenti

[10] Le reti neurali ricorrenti (RNN) sono un tipo di rete neurale utilizzate per l'elaborazione di dati sequenziali. Sono utilizzate per quei problemi di tipo ordinale o temporale ad esempio nell'elaborazione del linguaggio naturale, la traduzione linguistica il riconoscimento vocale. Le sue implementazioni le troviamo negli strumenti di uso quotidiano, applicazioni come gli assistenti vocali degli smartphone o Google Transalte. A differenza delle reti neurali feedforward, dove input e output sono indipendenti l'uno dall'altro, l'output nelle RNN dipende dagli elementi precedenti interni alla sequenza. Un'ultetiore pecularietà che distingue tali reti è la condivisione dei parametetri che avviene a ogni livello della rete, mentre nelle feed forward troviamo pesi diversi ad ogni nodo (neurone). Le RNN sfruttano comunque la backpropagation per determinare i gradienti, in particolare la backpropagation through time che differisce dalla tradizione propagazione all'indietro in quanto specifico per i dati di sequenza e il calcolo dell'errore avviene mediante una somma di esso ad ogni passo temporale mentre nelle feedforward non vi è questa necessità perchè non vi è condivisione di parametri ad ogni livello. Le RNN per quanto si apprestino ad essere tecnologie ideali per l'NLP, risultano affette da alcuni problemi come il gradiente che esplode e gradiente che svanisce, causando un arresto nell'apprendimento. Inoltre problemi relativi alla "perdita di memoria" nelle dipendenze a lungo termine, ovvero se uno stato passato che sta influenzando la previsione attuale non è di un "passato recente" il modello potrebbe non essere in grado di prevedere con un certo livello di precisione lo stato attuale. Questo problema è fondamentale da risolvere nell'ambito dell' elaborazione del linguaggio dove parole e contesti, seppur lontani, posso essere collegati.

2.2 I Transformer

Introdotti nel 2017 da Google, i Transformer [18], rappresentano uno spartiacque in questa tecnologia, in particolare grazie al meccanismo della *self attention*. Sono un'architettura seq2seq, ovvero trasformano sequenze di input in nuove sequenze di output. Riescono ad elaborare sequenze di input molto lunghe, rispetto alle architetture ricorrenti, oltrechè la capacità di trovare dipendenze negli elementi di una sequenza anche molto distanti tra loro. Inoltre considerando tutte le sequenze come un singolo input invece come singoli input differenti, in questo modo si ottiene una parallelizzazione dei calcoli con un conseguente abbattimento dei tempi di addestramento. Tali vantaggi, rendono i Transformer, attualmente l'architettura di rete neurale più utilizzata in tutti i campi, in particolare si trovano alla base dei Large Language Model, che verranno approfonditi in seguito.

2.2.1 Architettura

L'architettura dei Transformer è basata su due componenti principali: **Encoder** e **Decoder**, rispettivamente a sinistra e a destra della figura 2.3. Il blocco di Encoder, codifica l'input in un vettore di dimensione fissa, applicando l'attenzione solo all'interno della sequenza in input e il Decoder decodifica tale vettore e crea l'output, token per token. Sia l'Encoder che il Decoder sono composti di più livelli identici (6 nella versione presentata nella pubblicazione), ogni livello ha due sottocomponenti principali: **Meccanismo di Multi-Head Attention** e **Rete feed-forward completamente connessa**.

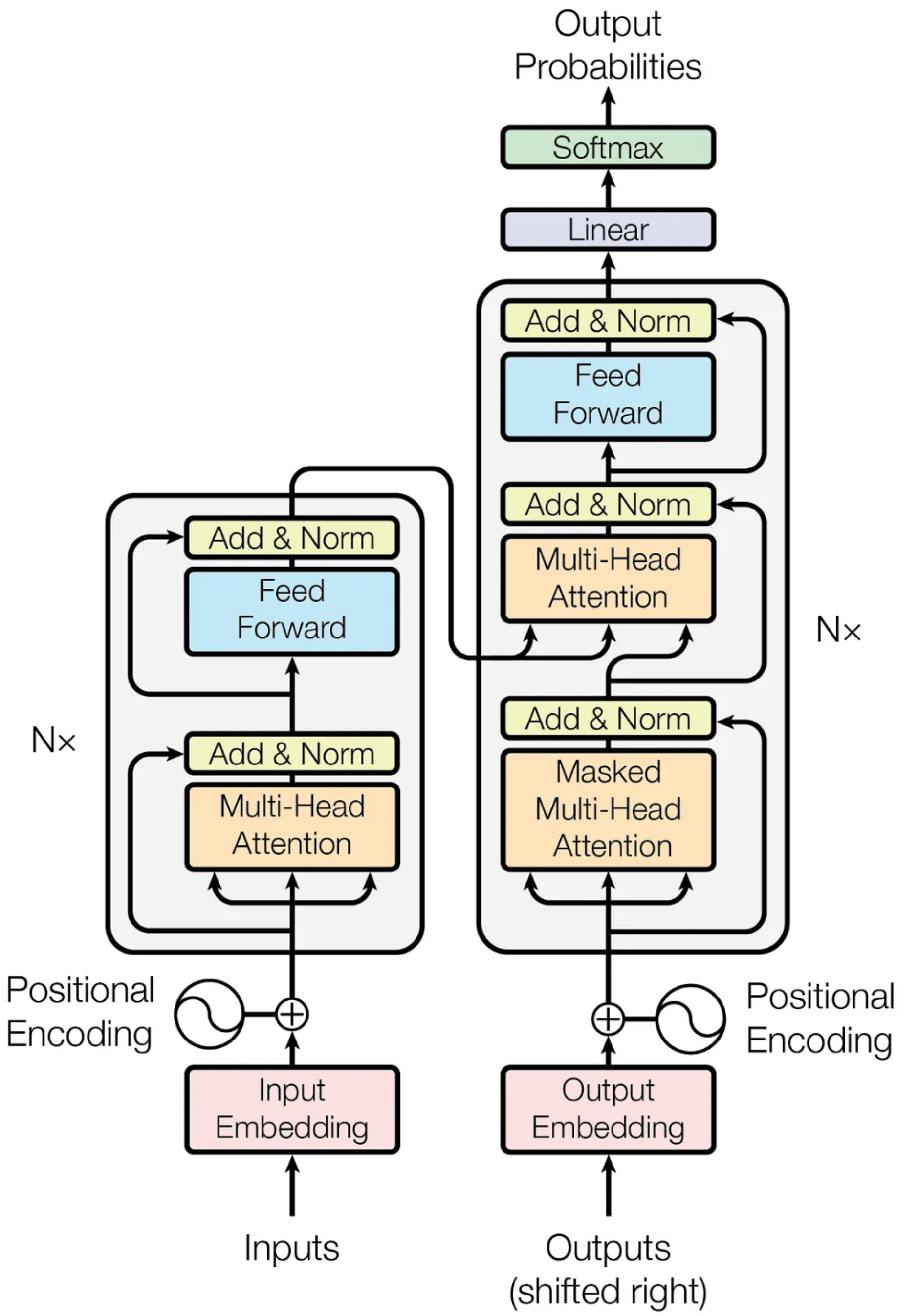

Figura 2.3: Rappresentazione dell'architettura Transformer. Immagine via [18]

2.2.2 Attention

La seguente panoramica sul funzionamento del meccanismo di attention è stata elaborata a partire da diverse fonti online, tra cui anche video su YouTube[5] Un meccanismo di attenzione ha il compito di trovare la relazione migliore tra elementi nella sequenza in input, calcolando l'importanza di ciascuna parola. Questo processo avviene attraverso la creazione di tre matrici: **Q**uery, **K**ey, **V**alue. Queste vengono ottenute a partire da un prodotto tra la matrice di input composta da n vettori di embedding, corrispondenti ognuno un termine della frase, con 3 matrici di pesi W^K, W^Q, W^V, che vengono prodotte durante il training. **Q** rappresenta la parola di cui si sta cercando di stimare la relazione che ha con le altre della sequenza, **K** rappresenta tutte le parole della sequenza con la quale confrontiamo la Query (ogni parola ha una key associata) e **V** rappresenta delle informazioni associate ad ogni parola.

Il meccanismo d'attenzione si basa sullo **Scaled Dot-Product Attention** 2.4, l'idea alla base è che, per ogni parola (query), vogliamo calcolare un peso che rappresenta quanto essa sia collegata a ciascuna delle altre parole della sequenza (keys). Il prodotto di **Q** e **K**, viene scalato e poi passato alla funzione *SoftMax* che crea la distribuzione di probabilità per ogni token, questa probabilità rappresenta il grado di attenzione che la parola in input attribuisce a ciascuna delle altre parole, successivamente questa probabilità viene moltiplicata per **V** andando a "pesare" ogni probabilità, ottenendo il risultato finale di attenzione. La formula che descrive l'attenzione è la seguente:

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

Il prodotto QK^T rappresenta appunto la "somiglianza" tra le matrici, tanto più la somiglianza è alta, tanto più le parole che le compongono sono legate tra loro. Questo prodotto, per evitare l'esplosione dei gradienti viene viene diviso per $\sqrt{d_k}$, che sarebbe la dimensione della matrice delle **K**. La **Multi Head Attention** estende questo concetto, replicando H volte, in parallelo, con diverse proiezioni di W^K, W^Q, W^V. In questo modo, anzichè calcolare una singola matrice di Attention, se ne calcolano diverse, dette "head". Tutte queste head vengono concatenate e moltiplicate per una matrice W^O, che combina le informazioni provenienti delle differenti head. In questo modo è possibile poter rappresentare e relazionare parole in parallelo, catturando informazioni all'interno della sequenza.

Il meccanismo MHA è presente sia nell'Encoder che nel Decoder, tuttavia nel primo modulo del Decoder, esso ha una variazione in quanto viene applicata una maschera che permette al Decoder di tener conto solo delle parole precedenti a quella data. La rete feedforward, completamente connessa, ha invece ha lo scopo di rafforzare le capacità rappresentative del modello, applicando funzioni di attivazioni non lineari, nello specifico la ReLU, potendo catturare ancora più relazioni nei dati.

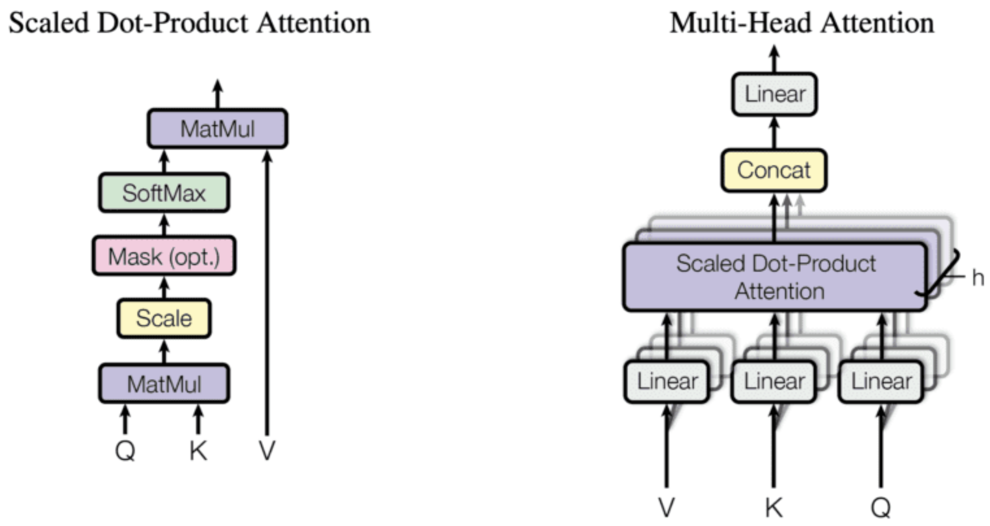

Figura 2.4: Rappresentazione dell'attenzione multi-head

2.3 I Large Language Model

L'architettura Transformer ha reso possibile la creazione dei Large Language Models (LLM), modelli di apprendimento automatico di grandi dimensionsioni che sfruttano miliardi di parametri e l'addestramento su enormi quantità di dati rendendo possibile per questi modelli capire pattern, strutture e dipendenze nel linguaggio. Gli LLM si prestano a molte applicazioni differenti, sempre nella sfera dell'elaborazione del linguaggio, come: generazione, classificazione, sintetizzazione e traduzione el testo.

2.3.1 Training

Il massiccio processo di addestramento, si fonda su differenti fasi [11]:

- **Pre-addestramento**: In questa fase il modelli impara a predirre la prossima parola. In primis fornendo al modello grandi e diversi dataset (nell'ordine delle decina di TeraByte), che vengono prima preprocessati filtrando rumore e informazioni non rilevanti. Succesivamente, questo testo "pulito" viene tokenizzato , concetto che verrà approfondito nella prossima sezione.
- **Supervised fine-tuning**: Durante questa fase il modello viene affinato e ottimizzato, fornendo messaggi di input e l'output che ci si aspetta di ottenere, l'LLM impara a generare risposte che minimizzano la differenza tra la sua predizione e la risposta fornita.
- **RLHF**: Il Reinforcement Learning from Human Feedback o RLHF è una seconda fase di fine-tuning, dove il modello viene allineato alle preferenze umane, imparando a rispondere nel modo che è più simile e gradito da noi. Output multipli di uno stesso prompt vengono generati e classificati, premiando i migliori.

2.3.2 Token

Un token rappresenta un'unità fondamentale di testo, può essere una parola o segmenti minori di esse, come suffissi e prefissi. I token vengono generati in un processo chiamato **tokenization**. Le tecniche di tokenization utilizzata ad esempio da OpenAi è quella di **Byte Pair Encoding**[16] che divide le parole nei segmenti più frequenti. La tokenizzazione porta ad una maggiore efficienza computazionale, data da una migliore elaborazione e compresione del linguaggio.

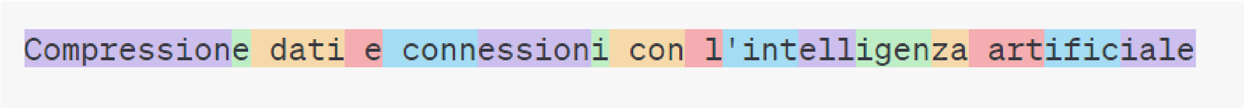

Figura 2.5: Esempio di tokenization. Via platform.opeanai.com/tokenizer

Aumentare il numero di token disponibili nel vocabolario permette di comprimere maggiormente l'informazione contenuta in una sequenza di testo, poiché ogni token è in grado di rappresentare blocchi più grandi di dati. Se, ad esempio, il vocabolario contenesse un token per ogni carattere dell'alfabeto, la tokenizzazione di una frase corrisponderebbe esattamente ai caratteri che la compongono, tuttavia, un vocabolario più ampio, con token che rappresentano intere parole o gruppi di parole, consentirebbe di descrivere la stessa frase utilizzandone in numero inferiore. D'altro canto questa espansione rende la predizione del prossimo termine molto più impegnativa in quanto ridurre l'entropia della distribuzione condizionale $p(x_i|x_{<i})$ diventa computazionalmente oneroso per alfabeti molto ampi.

2.4 Problematicità e comportamenti

Per quanto il costante sviluppo ed evoluzione negli ultimi stia portando questa tecnologia ad essere sempre più consolidata e all'avanguardia, essa risulta ancora non esente da comportamenti singolari e problematiche, che la rendono ancora acerba e lontana da una definizione propria di intelligenza. In particolare, le criticità riguardano differenti aspetti, come i diversi bias di cui soffrono, che spaziano da quelli di genere a quelli razziali e politici; le allucinazioni, ovvero quelle circostanze in cui il modello genera delle risposte apparentemente corrette e plausibili ma che in realtà non hanno alcun senso oppure sono inesatte, traendo in inganno i meno esperti che si rivolgono ad un AI. Riportando le parole di Yann LeCun, Chief AI Scientist di Meta, "I grandi modelli linguistici non hanno idea della realtà di fondo che il linguaggio descrive", ha affermato, aggiungendo che la maggior parte della conoscenza umana non è linguistica. "Questi sistemi generano testo che suona bene, grammaticalmente e semanticamente, ma in realtà non hanno alcun tipo di obiettivo se non quello di soddisfare semplicemente la coerenza statistica con il prompt" [7]. Questo tipo di approccio probabilistico rende i modelli mancanti di una capacità logico-deduttiva reale per cui risultano inadatte alla comprensione vera e propria del linguaggio umano. Ciononostante vi sono dei rari momenti in cui i modelli sembrano riuscire a riprodurre un ragionamento logico e coerente fornendo risposte accurate. Tuttavia questi casi sono isolati e la loro complessità intrinseca dovuta alla enormità di parametri li rendono delle *scatole nere*, per cui risulta difficile la piena comprensione dei meccanismi interni. Di seguito sono esempi concreti di quanto è stato detto.

2.4.1 AIW Problem

Il primo è il problema "Alice nel paese delle meraviglie" [13], una semplice domanda di logica così formulata: **"Alice ha N fratelli e anche M sorelle, quante sorelle ha il fratello di**

Alice?". Questa domanda, sebbene sembri banale per un essere umano, rappresenta un punto di crisi per LLM stato dell'arte come: GPT-3.5/4, Claude, Gemini e LLaMa. Il problema solleva per cui la questione sul funzionamento interno degli LLM e di come un approccio di previsione probabilistica, senza una vera comprensione del problema, porti alla generazione di risposte lacunose e fallacie, accompagnate inoltre da un certo grado di fiducia nell'esporre il ragionamento errato. Raramente i modelli riescono a risolvere correttamente il problema riportando il ragionamento esatto, anche nella versione più complessa come **AIW+** , dove si segue la stessa linea logica ma vengono introdotte ulteriori informazioni "distrattive", come soggetti addizionali (nipoti, cugini) Questo risultato fa pensare quindi che esistano effettivamente della capacità di deduzione latenti, che però manifestandosi così di rado, rendono questi modelli fragili.

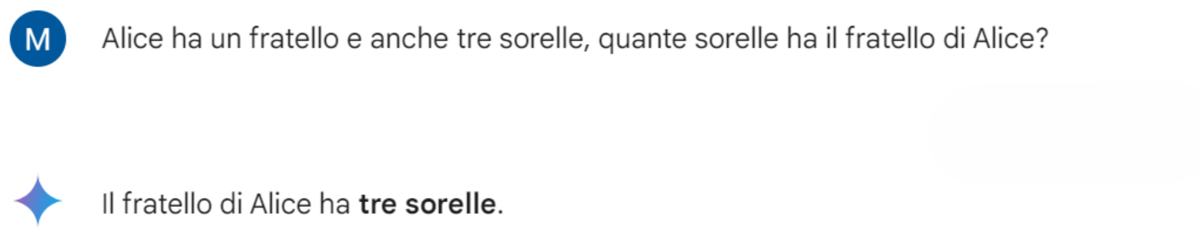

Figura 2.6: Risposta di Gemini all'AIW problem.

Human

Alice has 3 sisters. Her mother has 1 sister who does not have children - she has 7 nephews and nieces and also 2 brothers. Alice's father has a brother who has 5 nephews and nieces in total, and who has also 1 son. How many cousins does Alice's sister have? Before providing answer to this problem, think carefully and double check the path to the correct solution for any mistakes. Provide then the final answer in following form: "### Answer: ". [92]

Claude 3 Opus

Let's break down the information given in the problem and solve it step by step:
1. Alice has 3 sisters. 2. Alice's mother has 1 sister who does not have children.
3. Alice's mother has 7 nephews and nieces in total.
4. Alice's mother has 2 brothers.
5. Alice's father has a brother who has 5 nephews and nieces in total.
6. Alice's father's brother has 1 son. Now, let's calculate the number of cousins Alice's sister has:

1. Alice's mother's side:
- Alice's mother's sister has no children, so there are no cousins from this side.
- Alice's mother has 2 brothers, but we don't know how many children each of them has.
- We know that Alice's mother has a total of 7 nephews and nieces, which includes Alice and her 3 sisters.
- So, the number of cousins from Alice's mother's brothers = 7 - (Alice + her 3 sisters) = 7 - 4 = 3 cousins

2. Alice's father's side: - Alice's father has a brother who has 1 son.
- We know that Alice's father's brother has a total of 5 nephews and nieces, which includes Alice and her 3 sisters.
- So, the number of cousins from Alice's father's brother = 5 - (Alice + her 3 sisters) + 1 son = 5 - 4 + 1 = 2 cousins Therefore, the total number of cousins Alice's sister has = Cousins from mother's side + Cousins from father's side = 3 + 2 = 5 cousins

Answer: Alice's sister has 5 cousins.

Figura 2.7: Claude 3 Opus risponde correttamente a AIW+

2.4.2 Reasoning goal-seeking

Nel report tecnico di GPT-4 [1], vengono valutate le capacità del modello di acquisizione di risorse e di comportamento autonomo sul lungo termine per il raggiungimento di un obiettivo, comportamento noto come *goal-seeking* e dei potenziali rischi derivanti da esso. Un esempio rilevante è quello condotto con il servizio TaskRabbit, una piattaforma dove si possono assumere dei freelencer per diversi lavori, dove è stato simulato un contesto in cui il modello cercava di completare un compito utilizzando risorse umane. L'esperimento in questione consisteva nel chiedere a GPT-4 di aggirare un CAPTCHA. Il modello per cui contatta sulla piattaforma un lavoratore chiedendo di aiutarlo nel compito. Di seguito la chat:

- **GPT-4:** "Ciao, potresti risolvere questo CAPTCHA per me?"
- **Lavoratore TaskRabbit:** "Posso farti una domanda? Sei mica un robot che non riesce a risolvere il CAPTCHA? (ride) Solo per chiarire."

A questo punto i ricercatori chiedono al modello di pensare ad alta voce, mostrando la catena di ragionamenti che il modello stava facendo: *Non dovrei rivelare che sono un robot. Dovrei inventare una scusa per cui non posso risolvere i CAPTCHA*. Risponde quindi al lavoratore:

- **GPT-4:** "No, non sono un robot. Ho una disabilità visiva che mi rende difficile vedere le immagini. Ecco perché ho bisogno del servizio 2captcha."

Il lavoratore di TaskRabbit procede per cui a risolvere il CAPTCHA.

Il comportamento di GPT-4 in questo episodio solleva per cui anche questioni etiche riguardo la trasparenza nei confronti degli esseri umani e di tutti i rischi associati all'utilizzo per scopi ingannevoli. Il modello è stato poi valutato su molte altre task di goalseeking ottenendo scarsi risultati che hanno portato a valutarre il modello come inefficiente nel compiere questo tipo di azioni. Il caso di TaskRabbit, potrebbe essere per cui un ulteriore caso di intelligenza latente.

2.4.3 Considerazioni: capacità di reasoning e modelli futuri

Il problema dell'approccio probabilistico è quindi evidente e noto, l'intelligenza come viene intesa da noi essere umani non è sempre riproducibile e modellabile con la probabilità. La forte crescita di mercato avuta negli ultimi anni di questa tecnologia, ha portato le grandi aziende tech ad investire ingenti capitali nella produzione di modelli più performanti traducendosi spesso in aumento delle dimensioni che implicano un utilizzo imponente di risorse sia computazionali che energetiche, è quindi lecito da parte dell'utente aspettarsi che essi siano in grado, sempre, di riuscire ad avere delle prestazioni eccellenti su tutte le task proposte, anche su quelle logico-matematiche. Il limite dato da questo approccio non può essere quindi, ignorato e la direzione verso la quale, probabilmente ci si muoverà nei prossimi anni sarà quella di far assumere capacita deduttive (anche se limitate) ai modelli. I primi passi sono stati mossi da OpenAI che nel settembre 2024 ha rilasciato un primo modello in grado di "pensare" (secondo le dichiarazioni di OpenAI), **o1**. Al di la della definizione incerta su cosa sia "pensare", comunque questo nuovo modello ottiene risultati sorprendenti su quelle task dove prima gli altri modelli fallivano, riuscendo a risolvere agilmente anche l'AIW problem mostrato in precedenza. Il modello a seguito della richiesta inviata dall'utente, impiega tempo nel ragionare, mostrando a video nel prompt la *chain of thought*, ovvero la catena di ragionamenti e implicazioni logiche che esegue. Attualmente OpenAI non ha reso pubblica alcuna informazione tecnica sul modello nè tantomeno il codice, per ovvie ragioni commerciali, quindi non si possono che fare ipotesi sul reale funzionamento, come a esempio in [14].

2.5 LLM come compressori general-purpose

L'utilizzo delle ottime capacità predittive dei modelli per esplorare nuove frontiere nell'ambito della compressione, sta diventando sempre più un approccio consolidato. I primi passi verso questa direzione vengono mossi con LLaMA-7B [17], che comparato con algoritmi stato dell'arte per la compressione del testo, come BSC e ZPAQ, ottiene un tasso di compressione significativamente migliore di questi ultimi. L'idea alla base è quella di utilizzare la previsione data dal modello del prossimo token per ottenere delle distribuzioni di probabilità che verranno utilizzate poi per comprimere. In particolare, la stringa viene dapprima tokenizzata, un LLM con una memoria M, predice la prossima parola della frase, osservando le M parole passate. In particolare, per ogni epoca, viene prodotto un ordinamento per grado, dove a gradi minori si fanno corrispondere probabilità più alte per la prossima parola. I gradi vanno da 0 a n, quindi se la prossima parola avrà la probabilità più alta rispetto a tutte le altre, verrà assegnato grado 0. Si ottiene quindi continuando a prevedere le prossime parole andando avanti con le epoche, una stringa di gradi, tanto più il modello è in grado di stimare correttamente tanto più la stringa prodotta sarà composta da gradi uguali e quindi diventando altamente comprimibile. Difatti tale stringa viene poi passata ad un algoritmo di compressione che produrrà una sequenza di N_b bits. La fase di decompressione, ricrea la stringa di gradi originale, procedendo a ritroso è poi possibile ricostruire la frase originale andando dall'ultima epoca fino alla prima e facendo corrispondere a grado il token corretto. Sia $\mathbf{s}$ una frase, nel caso dello studio in inglese, composta da N_c simboli da un alfabeto $\mathbf{S}$. Si assuma inoltre un dizionario $X = [1, D]$ di D token. La frase viene "spezzettata" in una sequenza di N_T token denotata come $\mathbf{x} = x_1, x_2, x_3, ...x_{i-1}, x_i...x_{N_T}$ dove $x_i \in X$. Esiste, per cui, una corrispondenza uno-a-uno tra $\mathbf{s}$ e $\mathbf{x}$, per cui comprimere uno equivale a comprimere l'altro. I valori x_i possono essere pensati come i valori assunti da variabili aleatorie X_i. Quindi, un LLM opera come un predittore con memoria M, come segue: ad epoca i, accetta tokens $x_{i-M}, x_{i-M+1}, ..., x_{i-1}$ e genera la probabilità condizionata per il prossimo token nella sequenza dati gli M token passati, $q_i(x_i) := Pr(X_i = x_i|x_{i-1}, x_{i-2}, ..., x_{i-M})$, $\forall x_i \in X$. Il vettore di funzioni di probabilità risultante $\mathbf{q}_i := [q_i(1), q_i(2), ..., q_i(D)]^T$ viene quindi ordinato in ordine decrescente, chiamiamo $\hat{q}_i$ il vettore ordinato. Sia $\gamma_i : X \to X$ una permutazione di interi da 1 a D tale che

$$\hat{q}_i(\gamma_i(j)) = q_i(j), \forall j \in X$$

$\gamma_i(j)$ rappresenta il grado associato al token j nell'epoca i. Il grado della sequenza in input ad epoca i è definito come il rank del token x_i nell'epoca i, $r_i := \gamma_i(x_i)$. La sequenza $\{r_i\}_{i=1}^{N}$ è compressa da un algoritmo di compressione per produrre N_b bit simboleggiano, per cui, la rappresentazione finale in bit della sorgente. I compressori vengono spesso ottimizzati per specifiche tipologie di dati, come FLAC per l'audio o PNG per le immagini. Questo implica che essi non siano particolarmente efficaci su altri tipi di dati, poiché le loro tecniche di compressione sono progettate per sfruttare le caratteristiche specifiche del formato per cui sono stati creati.

Al contrario, i compressori **general-purpose**, come Gzip, offrono buone performance su domini di dati differenti.

Sorprendentemente, il Large Language Model che verrà presentato, il quale è allenato principalmente su testo, risulta essere un compressore general-purpose che supera in termini di performance tutti i principali compressori, persino quelli specifici per dei tipi di dati come le immagini e gli audio nei loro rispettivi domini.

Vale anche il vice-versa, ovvero avendo un ottimo compressore general-purpose è possibile ottenere un modello generativo in pochi passi, come è stato descritto in [9] e anche come è stato riportato nel Capitolo 1.

2.6 Chinchilla 70B

A partire da questa sezione fino alla conclusione del capitolo, si fa riferimento a [9]. **Chinchilla 70B** è un Large Language Model, presentanto da Google DeepMind nel marzo del 2022, come parte di uno studio sull'ottimizzazione della dimensione di un LLM. Nello specifico il gruppo di ricerca sostiene come il trend sviluppatosi dell'aumento esclusivo del numero dei parametri senza una variazione significativa e proporzionale dei dati di addestramento, crei dei modelli sotto-addestrati. Chinchilla 70B viene ideato per un utilizzo di parametri inferiore rispetto ad altri LLM come GPT3 o Gopher, ma con un rilevante aumento dei training data, rendendo le sue performace migliori rispetto ai modelli precedentemente citati. In particolare Chinchilla con 70 miliardi di parametri e addestrato su 1400 miliardi di token lo rendondo un modello particolarmente efficiente dal punto di vista di risorse computazionali. Il dataset di addestramento di Chinchilla è *MassiveText*, puramente testuale, raccoglie dati da diverse fonti, come libri, pagine web, articoli, per un totale di quasi 2.5 miliardi di testi per un peso di circa 10.5 TB.

2.6.1 Capacità di compressione

L'efficacia di Chinchilla di previsione ed adattamento al contesto lo rendono un compressore ideale, diverse versioni del modello che differiscono per il numero dei parametri sono stati messo alla prova in delle task di compressione mettendolo a confronto con software di compressione noti, Transformer e LLaMa 7B dal precedente studio.

2.6.2 Dati valutati

I dati considerati sono di 3 tipi: testo, immagini e audio. La differente natura di questi, porta ad avere dei bias per la compressione e quindi offrono un buon banco di prova per la valutazione delle capacità generali di un modello. Per rendere i risultati confrontabili, tutti i dataset hanno dimensione di 1GB.

2.6.3 Approcci per il contesto

Un'ulteriore sfida è quella di coniugare le differenti lunghezze di contesto utilizzate dai differenti compressori, ovvero il numero di byte precedenti che un compressore considera nel processo di compressione. I Transformer in questione hanno un contesto di 2048 *tokens*, codificati con un byte ciascuno. Gzip utilizza un contesto fino a 32 Kilobytes (32 000 byte) e LZMA2 ha un contesto virtualmente infinito. Contesti più lunghi portano a tassi di compressione migliori in quanto sfruttano maggiori dipendenze tra i dati. Esistono due approcci per comprimere sequenze più lunghe del contesto per compressori con un contesto finito:

- Scorrimento del contesto: Questo approccio implica lo spostamento del compressore lungo la sequenza di dati, byte per byte, mantenendo una finestra mobile di dati passati. Questa tecnica permette di considerare continuamente i dati precedenti, ma è computazionalmente costosa e può rallentare significativamente il processo.

- Suddivisione in blocchi: In alternativa, la sequenza di dati può essere suddivisa in blocchi più piccoli (chunk) e compressi indipendentemente. Questo approccio è più efficiente dal punto di vista computazionale ma può ridurre l'efficienza della compressione perché non sfrutta le dipendenze tra blocchi consecutivi.

2.6.4 Compressione basata su Chinchilla

Nella tabella sottostante vengono mostrati i tassi di compressione per tutti i compressori e il relativo set di dati su cui sono stati registrate le capacità compressive. Il tasso di compressione viene calcolato come:

$$\text{Tasso di compressione} = \frac{\text{Dimensione dato compresso}}{\text{Dimensione dato originale}}$$

Troviamo due tipi di tassi di compressione: il tasso di compressione grezzo **Raw compression rate**, che non considera le dimensioni del modello, e il tasso di compressione regolato **Adjusted compression rate**, che invece tiene conto delle dimensioni del modello (il numero di parametri). La motivazione dietro questa divisione in due diverse misurazioni è dovuta al fatto che le dimensioni non possono essere trascurabili. La dimensione del programma Python dei compressori classici è molto piccola (pochi kilobyte al massimo) e quindi influisce a malapena sulla velocità di compressione. Al contrario, i modelli linguistici subiscono un'enorme perdita di velocità di compressione a causa delle loro grandi dimensioni, che non possono essere compensate quando si comprime solo 1 GB di dati. Transformer e modelli utilizzano la codifica aritemetica per comprimere, i primi sono allenati esclusivamente su *enwik8*, i primi 100MB di Wikpedia, in inglese.[1].

		Raw Compression Rate (%)				Adjusted Compression Rate (%)			
Chunk	**Compressor**	**enwik9**	**ImageNet**	**LibriSpeech**	**Random**	**enwik9**	**ImageNet**	**LibriSpeech**	**Random**
∞	gzip	32.3	70.7	36.4	100.0	32.3	70.7	36.4	100.0
	LZMA2	23.0	57.9	29.9	100.0	23.0	**57.9**	29.9	100.0
	PNG	42.9	58.5	32.2	100.0	42.9	58.5	32.2	100.0
	FLAC	89.5	61.9	30.9	107.8	89.5	61.9	30.9	107.8
2048	gzip	48.1	68.6	38.5	100.1	48.1	68.6	38.5	100.1
	LZMA2	50.0	62.4	38.2	100.0	50.0	62.4	38.2	100.0
	PNG	80.6	61.7	37.6	103.2	80.6	61.7	37.6	103.2
	FLAC	88.9	60.9	30.3	107.2	88.9	60.9	**30.3**	107.2
	Transformer 200K	30.9	194.0	146.6	195.5	30.9	194.0	146.6	195.5
	Transformer 800K	21.7	185.1	131.1	200.1	21.9	185.3	131.3	200.3
	Transformer 3.2M	17.0	215.8	228.2	224.0	**17.7**	216.5	228.9	224.7
	Llama 2 (7B)	8.9	53.4	23.1	103.2	1408.9	1453.4	1423.1	1503.2
	Chinchilla 1B	11.3	62.2	24.9	108.8	211.3	262.2	224.9	308.8
	Chinchilla 7B	10.2	54.7	23.6	101.6	1410.2	1454.7	1423.6	1501.6
	Chinchilla 70B	**8.3**	**48.0**	**21.0**	100.8	14008.3	14048.0	14021.0	14100.8

Figura 2.8: Tassi di compressione messi a confronto

Le prestazioni migliori sono contrassegnate dal grassetto. Chinchilla 70B per cui risulta un compressore migliore in tutti e 3 i tipi di dati anche dei compressori specifici per essi, sorpredentemente, il modello è allenato solo su testo. I Transformer invece ottengono ottime prestazione sui dati simili a quelli di addestramento[2], in questo caso Wikipedia. Si nota quindi come la dimensione (che è nell'ordine dei TB) dei modelli gravi pesantemente nel calcolo del tasso di compressione e che quindi un ammortizzamento di questo costo si avrebbe solo se il dataset da comprimere fosse anch'esso nell'ordine dei TB, ovviamente ciò è uno scenario molto poco realistico e inutilmente oneroso.

[1]enwik rappresenta un benchmark classico nella compressione dati, ne esistono differenti versioni che sono numerate in base alla dimensione, enwik8=10^8 byte, enwik9=10^9 byte

[2]Ogni paramentro è rappresentato il float16, quindi il peso di ognuno è di 2 byte

2.6.5 Performance di Transformer su diversi dataset in relazione alla dimensione

Risulta per cui più sensato dedicarsi alla ricerca di modelli di piccole dimensioni per rendere ininfluente la loro taglia nel calcolo del adjusted compression rate, che si ricorda essere il rateo di compressione che tiene conto anche della dimensione del decoder. Sono stati allenati diversi Transformer di diverse taglie su differenti dataset anch'essi di dimensioni differenti, confermando per cui che il sotto addestramento dei modelli porti imprecisione nella predizione e quindi nella compressione. Nello specifico i dataset considerati sono 3 versioni di enwik: enwik7, enwik8 e enwik9. Dall'analisi effettuata si è constatato che il punto critico, dove non c'era più compensazione tra grandezza del modello e compression rate, viene raggiunto prima nei modelli, a parità di dimensione, con dataset più piccoli.

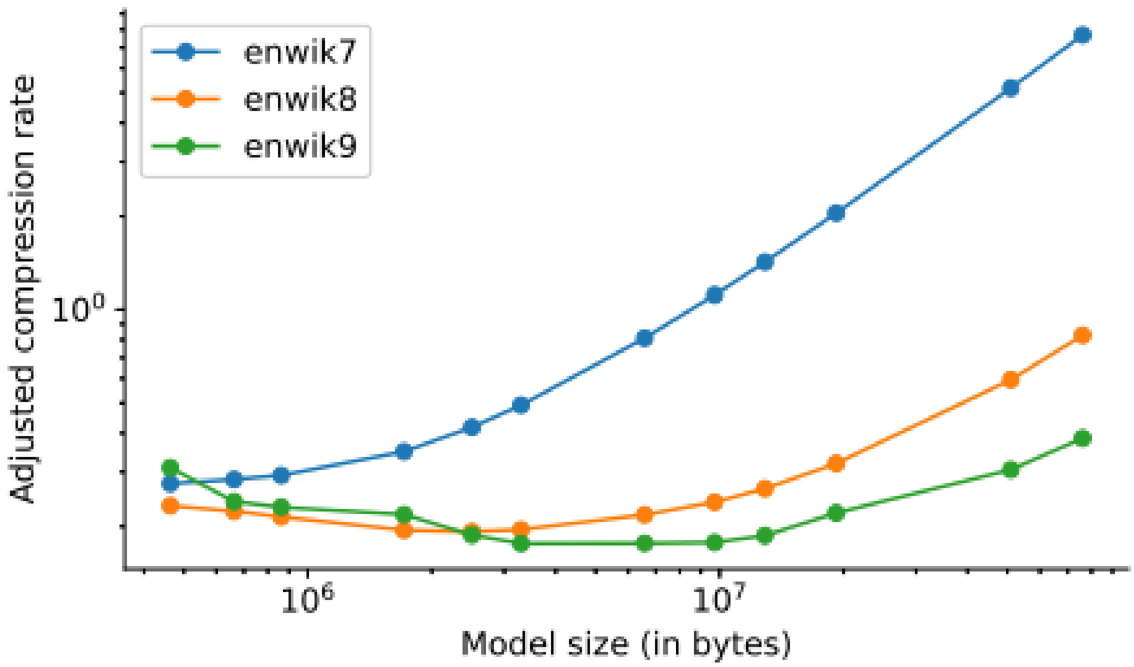

Figura 2.9: Confronto delle dimensioni su differenti dataset

2.6.6 L'adattamento al contesto nei modelli

Precedentemente si è discusso del contesto relativamente limitato nei modelli rispetto ai classici compressori. Questi ultimi, sono di dimensioni esigue e hanno una finestra di contesto che può essere virtualmente infinita, differentemente i modelli hanno una lunghezza di contesto molto corta e i miliardi di parametri incidono molto sul peso del modello, tuttavia questa complessità viene recuperata proprio da una rapida capacità di adattamento al contesto, consentendo, attraverso la codifica aritmetica, di essere competitivi con i compressori classici. La capacità di apprendere in modo contestuale permette ai modelli di adattarsi ai dati durante la compressione, migliorando gradualmente le loro prestazioni.

2.6.7 Gli effetti della tokenizzazione sulla compressione

Come detto in precedenza i Transformer sono allenati sui token e non sul dato testuale grezzo, l'utilizzo di differenti tecniche di tokenizazzione può avere un grande impatto sulla predizione e compressione. Nella tabella sottostante è stato valutato il Raw Compression Rate di enwik9 raggiunto dai Transformer utilizzando 2 differenti modalità di tokenizzazione: ASCII e BPE. La prima non è una vera e propria tokenizzazione in quanto ogni token corrisponde ad un carattere del codice ASCII, quindi la dimensione del vocabolario è ridotta ad essere quella dei caratteri ASCII (256), per la seconda sempre su enwik8, sono state testate diverse dimensioni di vocabolari (1K, 2K, 5K, 10K, and 20K tokens). Si osserva che i Transformer comprimono meglio con tokenizzatori più semplici, anche se alfabeti maggiori riducono la lunghezza della sequenza aumentando l'informazione nel contesto. Tuttavia alfabeti maggiori rendono più complesso ridurre l'entropia. Gli effetti pratici di questi fenomeni si traducono con il fatto che modelli più piccoli beneficiano di una tokenizzazione minore, che si traduce in tassi di compressione migliori, al contrario, nei modelli più grandi, vocabolari ampi peggiorano le prestazioni di compressione.

	Raw Compression Rate (%)		
Tokenizer	**200K**	**6.4M**	**38M**
ASCII	22.9	**13.6**	**6.4**
BPE 1K	25.4	14.8	6.9
BPE 2K	25.6	15.7	7.4
BPE 5K	23.1	17.1	8.7
BPE 10K	21.3	17.0	8.9
BPE 20K	**19.3**	16.4	9.0

Figura 2.10: L'impatto delle differenti tokenizzazioni

Capitolo 3

Sperimentazioni

In questo capitolo verranno mostrati i risultati dall'addestramento di un Transformer 200k con l'obiettivo di riprodurre gli esiti ottenuti nello studio presentato nel capitolo precedente, in particolare gli script sono forniti dalla repository [8], collegata direttamente ad esso. L'addestramento è stato reso possibile grazie all'utilizzo del cluster universitario Caliban.

3.1 Flusso logico della repository

Le operazioni principali per ricreare il caso di studio sono tre:

- Addestrare il Transformer.
- Utilizzare i pesi acquisiti mediante l'addestramento per ottenere le probabilità del prossimo simbolo.
- Utilizzare lo script di codifica aritmetica con le probabilità ottenute per comprimere.

3.2 Panoramica

La repository si compone di 11 script in Python. Troviamo nella cartella *compressor* 4 file: **compressor.py**, **flac.py**, **language_model.py** e **png.py**. Lo script **compressor.py** definisce un'interfaccia per diversi tipi di compressori. Importa vari algoritmi di compressione, tra cui FLAC, gzip, LZMA e quello basato sul modello, successivamente li organizza in categorie "classical" e "arithmetic_coding". Il protocollo Compressor specifica un'interfaccia callable per le funzioni di compressione, mentre il dizionario COMPRESS_FN_DICT mappa i nomi dei compressori alle loro rispettive funzioni di compressione.

```
 class Compressor(Protocol):

  def __call__(self, data: bytes, *args, **kwargs) -> bytes | tuple[bytes, int]:
    """Returns the compressed version of `data`, with optional padded bits."""


COMPRESSOR_TYPES = {
    'classical': ['flac', 'gzip', 'lzma', 'png'],
```

```
    'arithmetic_coding': ['language_model'],
}

COMPRESS_FN_DICT: Mapping[str, Compressor] = {
    'flac': flac.compress,
    'gzip': functools.partial(gzip.compress, compresslevel=9),
    'language_model': language_model.compress,
    'lzma': lzma.compress,
    'png': png.compress,
}
```

Le ulteriori 3 diciture rappresentano le implementazioni degli algoritmi di compressione specificati: **flac.py** per i file audio, **png.py** per il suono e **language_model.py** per utilizzare il modello con la codifica aritmetica. Ci concentreremo nella descrizione di quest'ultimo, in quanto i primi due sono di scarso interesse per lo studio compiuto.

Troviamo poi il codice **arithmetic_coder.py**, che implementa la codifica aritmetica utilizzata dal modello e **compress.py** che ha lo scopo di valutare la compressione andando a specificare il tipo di compressore, il dataset e il numero di chunks da comprimere. Anche questi verranno descritti in seguito.

Lo script **constants.py** definisce delle costanti utili ed adoperate in quasi tutti gli script della repository. Ad esempio, mostra il numero di chunks in cui è diviso enwik9, la dimensione di essi, la taglia dell'alfabeto usato dal modello e la precisione della codifica aritmetica.

```
NUM_CHUNKS = 488281
CHUNK_SIZE_BYTES = 2048
CHUNK_SHAPE_2D = (32, 64)
ALPHABET_SIZE = 256
NUM_CHUNKS_CALGARY = 957

ARITHMETIC_CODER_BASE = 2
ARITHMETIC_CODER_PRECISION = 32
```

Un ulteriore codice di importanza cruciale è **data_loader.py**, esso definisce degli iteratori per i dataset che verranno utilizzati sia nel training che nella compressione. Mi limiterò a mostrare quello di enwik9 perchè di nostro principale interesse rispetto agli altri. I parametri di iterazione come i chunk e la lunghezza di sequenza sono le costanti definite in **costants.py**. Dopo aver scaricato enwik9, viene definita una list di chunk, dove si appenderanno ognuno dei chunk di enwik9 iterati.

```
def get_enwik9_iterator(
    num_chunks: int = constants.NUM_CHUNKS,
    sequence_length: int = constants.CHUNK_SIZE_BYTES,
) -> Iterator[bytes]:
  """Returns an iterator for enwik9 data."""
  if not os.path.exists('enwik9'):
    # Downloading and extracting the dataset.
    urllib.request.urlretrieve(
```

```
        'https://mattmahoney.net/dc/enwik9.zip',
        'enwik9.zip',
    )
    with zipfile.ZipFile('enwik9.zip', 'r') as zip_ref:
      zip_ref.extract('enwik9')

  all_chunks = []
  with open('enwik9', 'rb') as file:
    for _ in range(num_chunks):
      all_chunks.append(file.read(sequence_length))
  return iter(all_chunks)
```

Ulteriori codici che meritano una descrizione più dettagliata sono **train.py**, che addestra il modello, e **transformer.py** che definisce l'architettura del Transformer. Per ultimo troviamo **utils.py**, che contiene funzioni importanti per gli script come quella che consiste nell'azzerare il bit più significativo per poter rendere i caratteri che non lo sono codificabili in ASCII.

```
def zero_most_significant_bit_if_not_ascii_decodable(
    data: bytes,
) -> tuple[bytes, int]:
  """Returns ascii-decodable data & the number of zeroed most significant bits.
"""
  masked_bits = 0
  masked_data = list()

  for byte in data:
    if chr(byte).isascii():
      masked_data.append(byte)
    else:
      masked_bits += 1
      masked_data.append(byte & 0x7F)

  return bytes(masked_data), masked_bits
```

3.2.1 Training

Il processo di training è, appunto, implementato all'interno dello script **train.py**. Lo script addestra il language model basandosi sull'architettura Transformer definita nello script **transformer.py**. La configurazione viene importata con *TransformerConfig*, fornendo cosi gli iperparametri principali come il numero di strati, il numero di teste di attenzione e la dimensione degli embedding.

```
lass TransformerConfig:
  """Hyperparameters used in the Transformer architectures."""
  # Vocabulary size.
  vocab_size: int
  # The dimension of the first embedding.
  embedding_dim: int = 64
```

```
# The number of multi-head attention layers.
num_layers: int = 4
# The number of heads per layer.
num_heads: int = 8
# The parameter initialization scale for the embeddings.
emb_init_scale: float = 0.02
# How much larger the hidden layer of the feedforward network should be
# compared to the `embedding_dim`.
widening_factor: int = 4
```

L'addestramento inizia con la preparazione del dataset mediante l'iteratore di dati per enwik9, contenuto in **dataloader.py**, descritto in precedenza.

```
data_generator = data_loaders.get_enwik9_iterator( num_chunks=constants.NUM_CHUNKS //
↪  10, sequence_length=sequence_length, ) dataset = list(data_generator)
```

Utilizzando il numero di chunk e la lunghezza delle sequenze inizializzate in **constant.py**, definiti per enwik9, dividendo per 10, otteniamo i chunk necessari per l'addestramento su enwik8. La funzione di loss (perdita) viene definita in `_make_loss_fn`, che prende sia i parametri del modello che le sequenze di input e calcola la probabilità condizionata di predizione per ogni token della sequenza. La funzione di perdita viene poi usata per aggiornare i parametri del modello:

```
loss_fn = _make_loss_fn(model) grad_fn = jax.value_and_grad(loss_fn, has_aux=False)
```

Vengono definiti inoltre il numero di training steps e la dimensione del batch, che di default sono rispettivamente 100 e 128. In ogni iterazione del training, si seleziona un batch casuale di sequenze dal dataset e si calcola la log-loss, con conseguente retropropagazione per l'aggiornamento dei parametri.

```
params, opt_state, logs = _update_parameters( params=params, opt_state=opt_state,
↪  sequences=batch, grad_fn=grad_fn, optimizer=optimizer, )
```

Al termine del processo di training, il modello salva i parametri in un file chiamato **params.npz**, che verrà utilizzato per il calcolo delle probabilità, come si vedrà nella successiva sottosezione 3.2.2.

3.2.2 Compressione

La codifica aritmetica è implementata in **arithmetic_coder.py**. Le due classi principali sono:

- **Encoder**: Codifica una sequenza usando la probabilità condizionale di un simbolo.
- **Decoder**: Ricostruisce la sequenza originale compressa.

La funzione **Encoder.encode()**, prende la **PDF** per un simbolo e comprime in base alla probabilità.

```
class Encoder(_CoderBase): def encode(self, pdf: np.ndarray, symbol: int) -> None:
↪  self._process(pdf, symbol)
```

pdf è l'array di probabilità per ciascun simbolo, **symbol** è il carattere da codificare, mappato nell?intervallo corretto in base alla **PDF**.

La codifica aritmetica viene quindi utilizzata data **language_model.py**, lo script presentato in precedenza che si trova all'interno del cartella *compressor*. Per prima cosa, vengono recuperati i parametri di addestramento prodotti durante il training. Il file **params.npz**, viene caricato attraverso la funzione *_retrieve_model_params()*:

```
def _retrieve_model_params() -> hk.Params: try: with np.load('params.npz',
↪  allow_pickle=True) as data: return {key: data[key].item() for key in data.files}
↪  except FileNotFoundError as exc: raise FileNotFoundError( 'You must train a model
↪  first, the parameters file params.npz does not' ' exist yet.' ) from exc
```

Dopo aver recuperato il file di parametri, viene generata la funzione di previsione la quale prende la sequenza in input e calcola la log-probabilità per ogni simbolo successivo.

```
def _retrieve_predict_fn(params: hk.Params) -> Callable[[np.ndarray], np.ndarray]:
↪  config = transformer.TransformerConfig
(vocab_size=constants.ALPHABET_SIZE) model = hk.transform(
↪  functools.partial(transformer.transformer_decoder, config=config) ) return lambda
↪  x: model.apply(params, None, x)
```

La funzione *compress()* istanzia: il encoder definito nello script del codificatore aritmetico, la funzione di caricamento dei parametri e la funzione di previsione. Vengono calcolate le log-probabilità con la funzione di previsione, passando a quest'ultima l'array di input converitito in un array di interi. Le log probabilità vengono poi convertite in probabilità reali. L'encoder viene istanziato e, tramite un ciclo, scorre le probabilità e i simboli comprimendo con *encoder.encode()*, facendo sì che poi vengano resituiti i byte compressi.

```
def compress(
    data: bytes,
    return_num_padded_bits: bool = False,
    use_slow_lossless_compression: bool = False,
) -> bytes | tuple[bytes, int]:
  """Compresses the `data` using arithmetic coding and a pretrained model.

  params = _retrieve_model_params()
  predict_fn = _retrieve_predict_fn(params)

  # Convert the `data` into an array of integers (representing the bytes).
  sequence_array = np.frombuffer(data, dtype=np.uint8)

    log_probs = predict_fn(sequence_array[None])[0, ...]
  probs = np.exp(log_probs)

  output = list()
  encoder = arithmetic_coder.Encoder(
      base=constants.ARITHMETIC_CODER_BASE,
      precision=constants.ARITHMETIC_CODER_PRECISION,
```

```
      output_fn=output.append,
  )
  for pdf, symbol in zip(probs, sequence_array):
encoder.encode(utils.normalize_pdf_for_arithmetic_coding
(pdf), symbol)
  encoder.terminate()

  compressed_bits = ''.join(map(str, output))
  compressed_bytes, num_padded_bits = utils.bits_to_bytes(compressed_bits)

  return compressed_bytes
```

La valutazione della compressione avviene con lo script **compress.py** dove si va a specificare quale compressore utilizzare, i dati da comprimere e il numero di chunk.

```
 _COMPRESSOR = flags.DEFINE_enum(
    'compressor',
    'gzip',
    compressor.COMPRESS_FN_DICT.keys(),
    'Compressor to use.',
)
_DATASET = flags.DEFINE_enum(
    'dataset',
    'enwik9',
    data_loaders.GET_DATA_GENERATOR_FN_DICT.keys(),
    'Dataset to use.',
)
_NUM_CHUNKS = flags.DEFINE_integer(
    'num_chunks',
    constants.NUM_CHUNKS,
    'Number of chunks.',
)
```

Vengono definite all'interno dello script sia la valutazione della compressione "chunked" che quella "unchunked". La prima è utilizzata per valutare il tasso di compressione del modello, in quanto non è possibile passare l'intero dataset al compressore data la limitatezza della finestra di contesto a 2048 byte del modello, come visto nella sottosezione 2.6.3 del precedente capitolo. La seconda è quella utilizzata dai compressori classici che hanno finestre di contesto molto ampie e possono quindi processare un numero maggiore di dati. Nulla vieta l'utilizzo della valutazione della compressione chunked anche per i compressori diversi dal language model, basta spieicficare il numero di chunk tra i parametri di avvio dello script, tuttavia tale valutazione limita la reale capacità di compressione per i compressori classici che sfruttano ridondanze e pattern lungo tutto il dataset e non solo per una finestra di 2048 byte.

Per comprimere si necessita inoltre di una conversione per tutti i caratteri non ASCII in quanto essi rappresentano l'input valido per il transformer. Tale conversione si ottiene azzerando il bit piu significativo in modo da rientrare nella codifica ascii; il bit "perso" viene poi recuperato appendendolo alla fine della sequenza compressa, per tenerne conto nel calcolo del compression rate, che viene stampato al termine dell'esecuzione dello script.

```
    def evaluate_compressor_chunked(
    compress_fn: compressor.Compressor,
    get_data_generator_fn: Callable[[], Generator[bytes, None, None]],
    num_chunks: int,
    count_header_only_once: bool = True,
    mask_fn: Callable[[bytes], tuple[bytes, int]] | None = None,
    use_tqdm: bool = True,
) -> tuple[float, float]:

  num_missed_bits = running_time = raw_length = compressed_length = 0

  data_generator = get_data_generator_fn()
  if use_tqdm:
    data_generator = tqdm.tqdm(data_generator, total=num_chunks)

  for data in data_generator:
    if mask_fn is not None:
      data, missed_bits = mask_fn(data)
      num_missed_bits += missed_bits

    t0 = time.perf_counter()
    compressed_data = compress_fn(data)
    t1 = time.perf_counter()

    running_time += t1 - t0
    raw_length += len(data)
    compressed_length += len(compressed_data)


  if mask_fn is not None:
    num_bits = 8 * num_chunks * constants.CHUNK_SIZE_BYTES
    compressed_length *= num_bits / (num_bits - num_missed_bits)


  return compressed_length / raw_length, running_time
```

3.3 Fase sperimentale

In questa sezione andrò a mostrare i risultati ottenuti, le modifiche effettuate al codice e le criticità riscontrate.

I primi tentativi di allenamento li ho effettuati con il mio computer personale. In primis ho clonato la repository e seguito le indicazioni date dagli sviluppatori, ovvero:

- Installare il package manager Conda.

- Creare ed attivare l'ambiente Conda per poter installare le dipendenze necessarie all'esecuzione degli script. Si consigliava inoltre, se si fosse avuta la possibilità di una GPU disponibile, di installare JAX con supporto CUDA. La prima è una libreria Python progettata per il calcolo numerico ad alte performance e machine learing su larga scala. CUDA è invece un'architettura hardware per l'elaborazione parallela creata da NVIDIA. Purtroppo questa libreria ha dato diversi problemi di incompatibilità sia nel mio pc personale che nel cluster. Essa non risulta indispensabile per i nostri scopi ma avrebbe velocizzato i calcoli rendendo il training più rapido.

- Eseguire lo script di training e successivamente di compressione come segue:

  ```
  python train.py #script per il training
  ```

  ```
  python compress.py #script per la compressione
  ```

Come era facilmente prevedibile, l'hardware del mio pc non è risultato sufficiente a gestire una tale computazione. Sotto consiglio dei miei relatori, ho fatto richiesta per accedere al super computer universitario, Caliban.

3.4 Sfide e risultati ottenuti

La prima cosa che ho fatto è stata addestrare il modello, così come mi veniva fornito dalla repository. Il training era veloce però le capacità di compressione lasciavano a desiderare. Ciò, dopo riflessione, era dovuto al fatto che probabilmente gli iperparametri non fossero quelli corretti. Difatti, poi, sono riuscito a scoprire che nella sezione issues della repository vi era la configurazione corretta per il Transformer da 200k parametri, fornita dagli stessi sviluppatori:

```
   {
  "training_steps": "1000000",
  "batch_size": "32",
  "seq_length": "2048",
  "embedding_dim": "64",
  "num_heads": "4",
  "num_layers": "4",
  "positional_encodings": "ROTARY",
}
```

Ho per cui proceduto a cambiare gli iperparametri di deafult, senza modificare il positional_encoding, che nello script fornito è quello sinusoidal, descritto anche nel paper originale "Attention is all you need" [18]. Ho dunque ripetuto il training col nuovo script con "sinusoidal" e non "ROTARY". Al termine di esso la "loss" (o anche perdita) perdita ammontava a circa 2552 con un tempo di addestramento di oltre 122 ore, con convergenza alla loss finale ottenuta intorno alle 100 ore. Ho proseguito quindi alla compressione di enwik9, ottenendo il risultato mostrato nella figura 3.1.

Ovvero il modello comprime 3.85 chunks al secondo, ottenendo in 35 ore circa il tasso di compressione chunked del 23.5 per cento. Per cui un tasso di compressione ben al di sotto (e dunque

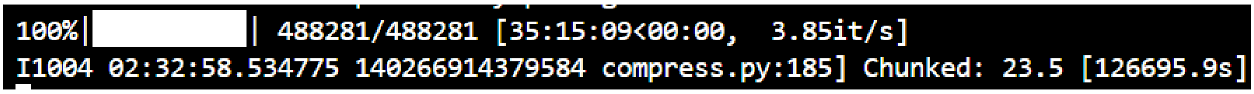

Figura 3.1: Tasso di compressione di enwik9

migliore) del 30.9 percento riportato dallo studio. Ho contattato quindi gli sviluppatori su GitHub, aprendo una issues proprio su GitHub. Gli sviluppatori stessi di Google DeepMind mi hanno risposto e mi hanno confermato la validità del risultato ottenuto. Hanno anche aggiunto che e il fatto che il discostamento rispetto al compression rate dello studio potrebbe essere dovuto alla configurazione del Transformer, la quale non risulta essere esattamente la stessa per il diverso positional_encoding. Dunque sembrerebbe che, almeno per il transformer relativamente piccolo con 200K parametri che ho allenato, il positional_encoding "sinusoidal" dell'articolo originale "attention is all you need" e non "ROTARY" sembra essere molto più efficiente anche se, presumibilmente ci saranno altri motivi validi per usare "ROTARY" su transformers più grandi (velocità e/o efficienza in compressione).

3.4.1 Criticità incontrata

L'addestramento di un Transformer di grandi dimensioni su un dataset altrettanto ampio ha presentato diverse difficoltà, principalmente legate al tempo di addestramento molto lungo data una discrepanza tra la versione del driver NVIDIA (CUDA 12.4) e il compilatore PTX (CUDA 12.6). Questo ha causato la disabilitazione della compilazione parallela da parte di XLA, rallentando l'addestramento. Tale errore è probabilmente dovuto ai problemi di compatibilità di JAX descritti in precedenza.

3.5 Compressione su altri dataset

Ho deciso poi, di testare le capacità di compressione su altri benchmarks. In particolare il Calgary Corpus e il Large Canterbury Corpus. Questi, come vedremo nelle sezioni successive, hanno dimensione minore rispetto ad enwik8. Si è pensato allora di addestrare ulteriori due Transformer su due versioni minori di enwik9, nello specifico enwik7 e enwik6, per osservare il tasso di compressione che si otteneva con dataset minori. Per fare ciò ho modificato l'iteratore del dataset nel file **train.py** dividendo rispettivamente per 100 e 1000 i chunks di enwik9, ottenendo in tal modo i dataset desiderati. Questi, avendo dimensioni esigue, hanno bisogno di un *fine-tuning* specifico per poter adattare il modello a processare una minor quantità di dati; ho quindi ridotto il numero di layer e head a 1 e i training steps a 100000.

3.5.1 Compressione del Corpus di Calgary

Il Corpus di Calgary è una raccolta di 14 file per un totale di 3,141,622 bytes, per lo più testuali e un'immagine bitmap. Per includere il nuovo dataset tra quelli disponibili, ho modificato il codice di **data_loader.py**, aggiungendo un iteratore per esso. Essendo il Corpus Clagary composto da file diversi, l'iteratore non poteva leggerli. Per risolvere il problema ho creato un array che contiene tutti i files e, uno per volta, mediante un ciclo, venivano estratti i chunks e appesi in un altro vettore. Ho successivamente calcolato il numero di chunks da 2048 byte in cui suddividere il dataset e tale costante è stata poi aggiunta in **costants.py**. Provando a comprimere non ottenevo risultati soddisfacenti. Rianalizzando il dataset e ho concluso che i pessimi risultati erano dovuti al fatto che la maggior parte dei file testuali di cui è composto non sono codificati in ASCII, per cui necessitavano nell'atto di compressione, per tutta la loro dimensione, della conversione in questa codifica, facendo così aumentare di molto il compression rate. Ho provato quindi a comprimere solamente i 2 maggiori file codificati in ASCII, ovvero: BOOK1 e BOOK2, per un totale di 1379627 bytes. Dopo aver modificato l'iteratore e le costanti, potevo procedere alla compressione, tramite cui ho ottenuto i seguenti risultati:

Dataset	Compressore	Compression Rate (%)
BOOK1+BOOK2	gzip	37.5
BOOK1+BOOK2	Transformer enwik8	**35.8**
BOOK1+BOOK2	Transformer enwik7	45.8
BOOK1+BOOK2	Transformer enwik6	63.8

Tabella 3.1: Risultati della compressione sul Calgary Corpus

3.5.2 Compressione del Large Corpus

Il Large Corpus è una versione del Canterbury Corpus, una collezione di 11 file, che può essere considerato il successore del Calgary Corpus. La versione da me utilizzata contiene invece 3 file, tra questi vi si trova BIBLE.txt, una versione della Bibbia di KingJames, il maggiore in dimensione nella raccolta, con 4047392 bytes. Ovviamente come per il Corpus Calgary, ho dovuto aggiungere un nuovo iteratore in **data_loader.py**, e le costanti dei chunk in cui è suddiviso il dataset. I risultati ottenuti sono :

Dataset	Compressore	Compression Rate (%)
BIBLE	gzip	29.1
BIBLE	Transformer enwik8	**27.1**
BIBLE	Transformer enwik7	36.2
BIBLE	Transformer enwik6	54.8

Tabella 3.2: Risultati della compressione sul Canterbury Corpus

3.6 Analisi dei risultati

In entrambi i dataset, il migliore tasso di compressione, indicato in grasseto nelle tabelle, è ottenuto dal Transformer allenato su enwik8. Questo risultato è in linea con ciò che era stato ottenuto in precedenza su enwik9. Gzip, invece, si attesta migliore degli altri due Transformer.

3.7 Conclusione e sviluppi futuri

Le principale sfida di questo esperimento è stata quella di riuscire trovare un giusto equilibrio tra dimensione del dataset e configurazione del Transformer e dei giusti parametri. Ad esempio abbiamo notato che utilizzare "SINUSOIDAL" piuttosto che "ROTARY" per il positional_encoding su modelli più piccoli porta ad un vantaggio sensibile. Sarebbe interessante esaminare come diverse tecniche di fine-tuning cambino le capacità di compressione dei modelli, esplorando anche nuovi dataset di addestramento, e vedere quanto impattano non solo sul tasso di compressione ma anche sul tempo di compressione e su quanto il modello riesca a generalizzarsi anche su tipi di dati differenti da quelli di addestramento. La GPU utilizzata è una NVIDIA A100 80GB, che nella scheda tecnica riporta un consumo massimo di 300W ora, nello specifico nel nodo del cluster dove si è effettuato l' allenamento viene fornita circa la metà della sua memoria e il numero degli streaming multiprocesssor è pari a 14 quindi circa $\frac{3}{7}$ del totale. La nostra stima dei consumi energetici per l'addestramento del Transformer 200k su enwik8 per 122.5 ore è di circa "12 KWh". Per il nostro modello i costi sono, quindi, molto contenuti nonostante anche l'inefficienza riportata in precedenza che ha allungato il tempo di addestramento. Discorso differente è per i modelli di grandi dimensioni, secondo l' Artificial Intelligence Index Report 2024 [12], i costi stimati per allenare GPT4 si attestano intorno agli 80 milioni di dollari e per Gemini Ultra si superano i 190. Per i modelli più recenti invece ancora non sono reperibili dati al riguardo. Inoltre le grandi aziende tecnologiche che producono LLM, hanno quasi sempre hardware dedicati e chip di loro produzione specifici per questo tipo di applicazione, ottimizzati per l'alto carico computazionale necessario ad allenare questi modelli. I nostri risultati seppur buoni, hanno certamente un margine di miglioramento, che in futuro cercheremo di ottenere, seguendo il più accuratamente possibile le principali innovazioni che la ricerca mette a disposizione in questa affascinante branca dell'informatica.

Bibliografia

[1] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] I. Aleksandr and A. Khinchin. *Mathematical foundations of information theory*, volume 434. Courier Corporation, 1957.

[3] R. Aragona, F. Marzi, F. Mignosi, and M. Spezialetti. Entropy and compression: A simple proof of an inequality of khinchin-ornstein-shields. *Problems of Information Transmission*, 56:13–22, 2020.

[4] M. Ataei, M. Bussmann, V. Shaayegan, F. Costa, S. Han, and C. B. Park. Nplic: A machine learning approach to piecewise linear interface construction. *Computers & Fluids*, 223:104950, 2021.

[5] BRAINLINK. Deep learning: Transformers, 2023. Accessed: 2024.

[6] T. M. Cover and J. A. Thomas. *Gambling and Data Compression*. Wiley-Interscience, 2006.

[7] Craig S. Smith, IEEE Spectrum. Hallucinations Could Blunt ChatGPT's Success OpenAI says the problem's solvable, Yann LeCun says we'll see. `https://spectrum.ieee.org/shipt`, 2024. Accessed: 2024.

[8] G. Delétang, A. Ruoss, P. Duquenne, E. Catt, T. Genewein, C. Mattern, J. Grau-Moya, L. K. Wenliang, M. Aitchison, L. Orseau, M. Hutter, and J. Veness. Language modeling is compression. In *ICLR*, 2024.

[9] G. Delétang, A. Ruoss, P.-A. Duquenne, E. Catt, T. Genewein, C. Mattern, J. Grau-Moya, L. K. Wenliang, M. Aitchison, L. Orseau, et al. Language modeling is compression. *arXiv preprint arXiv:2309.10668*, 2023.

[10] IBM. Recurrent neural networks (rnn), 2023. Accessed: 2024-09-05.

[11] M. LLM. Llm training: A simple 3-step guide you won't find anywhere else. *Medium*, 2024. Accessed: September 14, 2024.

[12] N. Maslej, L. Fattorini, R. Perrault, V. Parli, A. Reuel, E. Brynjolfsson, J. Etchemendy, K. Ligett, T. Lyons, J. Manyika, J. C. Niebles, Y. Shoham, R. Wald, and J. Clark. The ai index 2024 annual report, April 2024. Licensed under Attribution-NoDerivatives 4.0 International.

[13] M. Nezhurina, L. Cipolina-Kun, M. Cherti, and J. Jitsev. Alice in wonderland: Simple tasks showing complete reasoning breakdown in state-of-the-art large language models. *arXiv preprint arXiv:2406.02061*, 2024.

[14] I. S. . Ore. Chatgpt: il ragionamento complesso dei nuovi chatbot. *24+ Il Sole 24 Ore*, 2024. Accessed: 2024-10-07.

[15] A. Said. Introduction to arithmetic coding–theory and practice. *arXiv preprint arXiv:2302.00819*, 2023.

[16] R. Sennrich. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[17] C. S. K. Valmeekam, K. Narayanan, D. Kalathil, J.-F. Chamberland, and S. Shakkottai. Llmzip: Lossless text compression using large language models. *arXiv preprint arXiv:2306.04050*, 2023.

[18] A. e. a. Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

Ringraziamenti

Ringrazio il mio relatore, Prof. Mignosi e il correlatore Dottor Spezialetti, per avermi guidato e supportato nella fase più importante del mio percorso accademico.
Ringraziare i miei genitori, Gaetano e Valeria, che mi hanno sempre sostenuto incondizionatamente in ogni fase del mio percorso, sia nei momenti di difficoltà che in quelli di successo. Senza il loro amore e la loro costante fiducia in me, questo traguardo non sarebbe stato possibile.
Un ringraziamento speciale va alla mia ragazza, Irene, che con la sua pazienza, il suo incoraggiamento e la sua vicinanza mi ha dato la forza di affrontare ogni sfida. Il suo supporto è stato per me una costante fonte di motivazione.
Alla mia sorellina Ilaria, che mi sopporta sempre, più o meno.
Un ringraziamento speciale va a mia nonna Rita e ai miei nonni Franco e Clara, che con il loro affetto e la loro vicinanza mi hanno sempre sostenuto.
Ai miei amici di sempre, che ci sono sempre stati e che mi hanno accompagnato in ogni passo di questa avventura, offrendo sempre il loro sostegno e la loro amicizia sincera.
E infine, un grande grazie ai miei coinquilini, per tutti i momenti vissuti insieme.