Large Language Models

Architettura, Addestramento e Impatti Applicativi

[Il tuo nome]

Anno Accademico 2024–2025

Indice

Capitolo 1

Introduzione

1.1 Introduzione

Il 30 Novembre 2022, con l'avvento di ChatGPT è stata segnata una data storica per l'umanità, non solo nel campo della tencologia ma anche nell'ambito filosofico in quanto per la prima volta l'uomo ha iniziato a parlare con qualcosa di altro da sè che sembra dar prova che il lunguaggio umano, veicolo di signficato, non sia escluivo dell'uomo ma possa essere appreso e riprodotto da una macchina.

Capitolo 2

Autoencoders

2.1 Introduzione

In questo capitolo vengono introdotti gli *autoencoders*, una classe di modelli di apprendimento non supervisionato ampiamente utilizzata per l'apprendimento di rappresentazioni latenti dei dati. Dopo averne presentato la formulazione generale e i principi di funzionamento, verranno discussi i principali limiti degli autoencoders classici, in particolare in termini di capacità di apprendere rappresentazioni interpretabili.

Successivamente, il capitolo introduce gli *Sparse Autoencoders*, una estensione degli autoencoders tradizionali che impone vincoli di sparsità sullo spazio latente, favorendo il disentanglement delle feature e l'interpretabilità delle rappresentazioni apprese. Questi modelli costituiscono il fondamento teorico delle metodologie utilizzate nel resto del lavoro di tesi.

2.2 Autoencoders: definizione e formulazione generale

2.2.1 Apprendimento non supervisionato

Gli autoencoders sono modelli di apprendimento non supervisionato, in quanto non richiedono etichette associate ai dati di input durante la fase di addestramento. Si consideri un dataset di addestramento S_T costituito da M

osservazioni non etichettate $\mathbf{x}_i$, con $i = 1, \ldots, M$:

$$S_T = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M\}. \tag{2.1}$$

In generale, ciascuna osservazione appartiene allo spazio $\mathbb{R}^n$, ovvero $\mathbf{x}_i \in \mathbb{R}^n$. L'obiettivo di un autoencoder è apprendere una rappresentazione dei dati tale da permettere la ricostruzione dell'input nel modo più accurato possibile, minimizzando una misura dell'errore di ricostruzione. L'interesse verso questo tipo di modelli risiede nel fatto che la rappresentazione latente appresa può essere utilizzata in numerose applicazioni, come la riduzione della dimensionalità, l'estrazione di caratteristiche, il denoising e l'anomaly detection. Una definizione formale di autoencoder è la seguente.

Autoencoder *Un autoencoder è un tipo di algoritmo il cui scopo principale è apprendere una rappresentazione dei dati, utilizzabile per diverse applicazioni, imparando a ricostruire in modo sufficientemente accurato un insieme di osservazioni di input* [1].

2.2.2 Encoder, decoder e spazio latente

Un autoencoder è composto da due blocchi principali: un **encoder** e un **decoder**. La struttura generale del modello è illustrata in Figura 2.1.

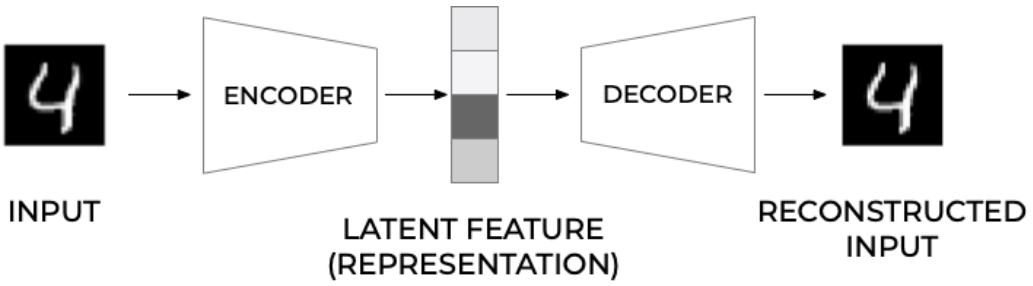

Figura 2.1: Schema di funzionamento di un autoencoder [2]

Nella maggior parte dei casi, l'encoder e il decoder sono implementati come reti neurali. Seguendo l'impostazione descritta in [2], l'encoder può essere rappresentato come una funzione g, dipendente da un insieme di parametri apprendibili, che associa a ciascun dato di input una rappresentazione nello spazio latente:

$$\mathbf{h}_i = g(\mathbf{x}_i). \tag{2.2}$$

Qui $\mathbf{h}_i \in \mathbb{R}^q$ rappresenta il vettore delle *features latenti* ed è l'output del blocco di encoder quando la funzione g viene valutata sull'input $\mathbf{x}_i$. Ne consegue che l'encoder realizza una mappatura del tipo

$$g : \mathbb{R}^n \to \mathbb{R}^q. \tag{2.3}$$

Il decoder ha il compito di ricostruire il dato originale a partire dalla rappresentazione latente. L'output della rete, indicato con $\hat{\mathbf{x}}_i$, può essere espresso tramite una seconda funzione generica f:

$$\hat{\mathbf{x}}_i = f(\mathbf{h}_i) = f(g(\mathbf{x}_i)), \tag{2.4}$$

dove $\hat{\mathbf{x}}_i \in \mathbb{R}^n$ rappresenta la ricostruzione dell'input $\mathbf{x}_i$.

2.2.3 Funzione obiettivo e errore di ricostruzione

L'addestramento di un autoencoder consiste nel determinare le funzioni $g(\cdot)$ e $f(\cdot)$ tali da minimizzare una misura della discrepanza tra i dati di input e le rispettive ricostruzioni. Formalmente, il problema di ottimizzazione può essere espresso come

$$\arg\min_{f,g} \langle \Delta(\mathbf{x}_i, f(g(\mathbf{x}_i))) \rangle \,, \tag{2.5}$$

dove Δ indica una funzione di perdita che quantifica la differenza tra l'input e l'output dell'autoencoder, mentre $\langle \cdot \rangle$ denota la media su tutte le osservazioni del dataset di addestramento.

2.3 Il problema dell'identità e la necessità di vincoli

In assenza di vincoli sull'architettura o sulla funzione obiettivo, un autoencoder dotato di capacità sufficiente può apprendere una semplice funzione identità, ottenendo una ricostruzione perfetta ma priva di utilità pratica. Per evitare questo comportamento degenerato, è comune introdurre specifiche strategie di regolarizzazione, come la presenza di una strozzatura dimensionale nello spazio latente oppure l'aggiunta di termini di regolarizzazione alla funzione di costo.

Nota. Un autoencoder efficace deve bilanciare due obiettivi contrastanti: da un lato una ricostruzione sufficientemente accurata dell'input, dall'altro l'apprendimento di una rappresentazione latente che catturi le caratteristiche essenziali dei dati, evitando soluzioni banali come l'identità.

2.3.1 Bottleneck e riduzione della dimensionalità

Al fine di evitare che l'autoencoder apprenda una banale funzione identità e di favorire l'apprendimento di rappresentazioni astratte e informative dei dati, una strategia comunemente adottata consiste nell'imporre una riduzione della dimensionalità tra lo spazio di input e lo spazio latente. Tale configurazione architetturale prende il nome di **bottleneck** o **strozzatura**. In un'architettura con bottleneck, la dimensione dello spazio latente q è strettamente inferiore alla dimensione dell'input n ($q < n$). In queste condizioni, l'encoder realizza una mappatura che comprime l'informazione contenuta nei dati di ingresso:

$$g : \mathbb{R}^n \to \mathbb{R}^q, \quad q < n, \tag{2.6}$$

costringendo il modello a selezionare e preservare esclusivamente le componenti più rilevanti dell'input ai fini della ricostruzione. La presenza della strozzatura impedisce quindi una copia diretta dei dati e spinge l'autoencoder a catturare strutture, correlazioni e regolarità latenti presenti nel dataset. Al termine dell'addestramento, lo spazio latente costituisce una rappresentazione compatta e astratta dei dati, che può essere interpretata come una codifica delle caratteristiche essenziali dell'input e utilizzata per compiti successivi quali riduzione della dimensionalità, visualizzazione o analisi delle feature.

2.3.2 Introduzione di vincoli

Oltre alla strozzatura architetturale, un ulteriore approccio per evitare che l'autoencoder apprenda una semplice funzione identità consiste nell'introduzione di vincoli aggiuntivi nella funzione obiettivo, tipicamente sotto forma di termini di regolarizzazione. Tali vincoli agiscono limitando la capacità espressiva del modello o penalizzando soluzioni considerate indesiderabili, favorendo l'apprendimento di rappresentazioni latenti più strutturate e informative.

In questo contesto, la funzione di costo dell'autoencoder non si limita più a misurare esclusivamente l'errore di ricostruzione, ma include uno o più termini addizionali che impongono specifiche proprietà alla rappresentazione latente o ai parametri del modello. In forma generale, il problema di ottimizzazione può essere scritto come

$$\arg\min_{f,g} \langle \Delta(\mathbf{x}_i, f(g(\mathbf{x}_i))) \rangle + \lambda\, \Omega(g, f), \tag{2.7}$$

dove $\Omega(g, f)$ rappresenta un termine di regolarizzazione e $\lambda > 0$ ne controlla l'importanza relativa rispetto all'errore di ricostruzione. A seconda della scelta del termine di regolarizzazione, è possibile indurre diverse proprietà nel modello. Ad esempio, la penalizzazione della norma dei pesi limita la complessità della rete e migliora la capacità di generalizzazione, mentre vincoli applicati direttamente allo spazio latente possono favorire caratteristiche quali la **sparsità**, la robustezza al rumore o la separazione delle feature. In particolare, l'introduzione di vincoli di sparsità sulle attivazioni latenti costituisce il principio alla base degli *Sparse Autoencoders*, che verranno discussi nel seguito.

Un esempio comune di regolarizzazione consiste nell'introdurre un vincolo direttamente sulle attivazioni dello spazio latente. In questo caso, la funzione obiettivo dell'autoencoder assume la forma

$$\arg\min_{f,g} \langle \Delta(\mathbf{x}_i, f(g(\mathbf{x}_i))) \rangle + \lambda \|\mathbf{h}_i\|_2^2, \tag{2.8}$$

dove $\mathbf{h}_i = g(\mathbf{x}_i)$ denota il vettore delle attivazioni latenti associate all'osservazione $\mathbf{x}_i$. Tale penalizzazione di tipo ℓ_2 scoraggia rappresentazioni latenti di grande norma, favorendo codifiche più compatte e contribuendo alla stabilità del modello.

Un'alternativa è rappresentata dalla regolarizzazione di tipo ℓ_1 applicata allo spazio latente:

$$\arg\min_{f,g} \langle \Delta(\mathbf{x}_i, f(g(\mathbf{x}_i))) \rangle + \lambda \|\mathbf{h}_i\|_1. \tag{2.9}$$

A differenza della norma ℓ_2, la regolarizzazione ℓ_1 tende a produrre rappresentazioni sparse, in cui solo un numero limitato di componenti del vettore latente risulta attivo per ciascun input. Questo comportamento favorisce una decomposizione più interpretabile delle feature e costituisce il principio alla base degli *Sparse Autoencoders*, che verranno analizzati nel seguito.

L'aggiunta di vincoli nella funzione obiettivo consente quindi di superare i limiti degli autoencoders classici, guidando l'apprendimento verso soluzioni non banali e semanticamente più significative, anche in assenza di una riduzione esplicita della dimensionalità dello spazio latente.

2.3.3 Relazioni con la PCA

Dal momento che gli autoencoders possono essere utilizzati per la riduzione della dimensionalità dei dati, è di interesse evidenziare la loro relazione con il metodo delle *Principal Component Analysis* (PCA). La PCA è una tecnica di analisi statistica che consente di ridurre la dimensionalità di un dataset preservando la maggior parte della varianza presente nei dati originali, mediante una trasformazione lineare delle variabili.

Sia dato un dataset di M osservazioni centrate

$$\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M \in \mathbb{R}^n, \qquad \frac{1}{M}\sum_{i=1}^{M} \mathbf{x}_i = 0.$$

Definendo la matrice dei dati

$$X = \begin{bmatrix} \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \\ \vdots \\ \mathbf{x}_M^\top \end{bmatrix} \in \mathbb{R}^{M \times n},$$

la matrice di covarianza empirica è data da

$$C = \frac{1}{M} X^\top X \in \mathbb{R}^{n \times n}.$$

L'obiettivo della PCA consiste nell'individuare una direzione unitaria $\mathbf{w}_1 \in \mathbb{R}^n$ lungo la quale la proiezione dei dati presenti la massima varianza. Indicando con $y_i = \mathbf{w}_1^\top \mathbf{x}_i$ la proiezione dell'osservazione $\mathbf{x}_i$ lungo tale direzione, la varianza dei dati proiettati può essere espressa come

$$\mathrm{Var}(X\mathbf{w}_1) = \frac{1}{M}\sum_{i=1}^{M} \left(\mathbf{w}_1^\top \mathbf{x}_i\right)^2,$$

dove si è utilizzato il fatto che i dati sono centrati, e quindi la media delle proiezioni risulta nulla. Riscrivendo la precedente espressione in forma matriciale si ottiene

$$\mathrm{Var}(X\mathbf{w}_1) = \mathbf{w}_1^\top \left(\frac{1}{M}\sum_{i=1}^{M} \mathbf{x}_i \mathbf{x}_i^\top\right) \mathbf{w}_1 = \mathbf{w}_1^\top C \mathbf{w}_1.$$

Il problema della ricerca della direzione di massima varianza può quindi essere formulato come il seguente problema di ottimizzazione vincolata:

$$\max_{\|\mathbf{w}_1\|_2 = 1} \mathbf{w}_1^\top C \mathbf{w}_1. \tag{2.10}$$

Figura 2.2: L'immagine mostra Davide Mononcello in uno stato di sonno

Tale problema può essere risolto mediante il metodo dei moltiplicatori di Lagrange, introducendo la lagrangiana

$$L(\mathbf{w}, \lambda) = \mathbf{w}^\top C\mathbf{w} - \lambda(\mathbf{w}^\top \mathbf{w} - 1).$$

Imponendo la condizione di stazionarietà rispetto a $\mathbf{w}$ si ottiene

$$\nabla_{\mathbf{w}} L(\mathbf{w}, \lambda) = 2C\mathbf{w} - 2\lambda\mathbf{w} = 0,$$

da cui segue il problema agli autovalori

$$C\mathbf{w} = \lambda\mathbf{w}. \tag{2.11}$$

Le soluzioni ammissibili sono pertanto gli autovettori di C, mentre i moltiplicatori di Lagrange coincidono con i corrispondenti autovalori. La derivata della lagrangiana rispetto a λ restituisce inoltre il vincolo di normalizzazione

$$\mathbf{w}^\top \mathbf{w} = 1.$$

Sia $\mathbf{v}_k$ un autovettore unitario di C associato all'autovalore λ_k. Per tali vettori vale

$$\mathrm{Var}(X\mathbf{v}_k) = \mathbf{v}_k^\top C\mathbf{v}_k = \lambda_k,$$

ossia ciascun autovalore rappresenta la varianza dei dati lungo la corrispondente direzione $\mathbf{v}_k$.

Ordinando gli autovalori in ordine decrescente

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0,$$

le direzioni associate agli autovalori maggiori individuano le componenti principali del dataset. In particolare, la prima componente principale $\mathbf{v}_1$ è la direzione di massima varianza, mentre le componenti successive massimizzano la varianza residua sotto il vincolo di ortogonalità rispetto alle precedenti.

Caso di un autoencoder lineare

Si consideri un autoencoder costituito da un encoder e un decoder entrambi lineari, addestrato su dati centrati. L'encoder realizza una mappatura del tipo

$$\mathbf{h} = W\mathbf{x},$$

dove $W \in \mathbb{R}^{q \times n}$ e $q < n$ è la dimensione dello spazio latente. Il decoder ricostruisce l'input mediante

$$\hat{\mathbf{x}} = W^\top \mathbf{h} = W^\top W\mathbf{x},$$

dove, senza perdita di generalità, si è assunto che i pesi del decoder siano vincolati a essere la trasposta di quelli dell'encoder.

L'addestramento dell'autoencoder consiste nel minimizzare l'errore quadratico medio di ricostruzione:

$$\mathcal{L}(W) = \frac{1}{M} \sum_{i=1}^{M} \left\| \mathbf{x}_i - W^\top W \mathbf{x}_i \right\|_2^2 . \tag{2.12}$$

Osservando che $W^\top W$ è una matrice simmetrica di rango al più q, tale termine può essere interpretato come una proiezione lineare sul sottospazio generato dalle righe di W. L'errore di ricostruzione misura quindi la distanza tra ciascun dato e la sua proiezione su tale sottospazio.

Sfruttando l'ipotesi di dati centrati, la funzione di costo può essere riscritta come

$$\mathcal{L}(W) = \frac{1}{M} \sum_{i=1}^{M} \|\mathbf{x}_i\|_2^2 - \frac{1}{M} \sum_{i=1}^{M} \|W\mathbf{x}_i\|_2^2, \tag{2.13}$$

dove il primo termine è indipendente da W. Ne consegue che minimizzare l'errore di ricostruzione equivale a massimizzare la quantità

$$\frac{1}{M} \sum_{i=1}^{M} \|W\mathbf{x}_i\|_2^2. \tag{2.14}$$

Indicando con $\mathbf{w}_1, \ldots, \mathbf{w}_q$ le righe di W, si ottiene

$$\frac{1}{M}\sum_{i=1}^{M}\|W\mathbf{x}_i\|_2^2 = \sum_{j=1}^{q}\frac{1}{M}\sum_{i=1}^{M}(\mathbf{w}_j^\top \mathbf{x}_i)^2 = \sum_{j=1}^{q}\mathrm{Var}(X\mathbf{w}_j), \tag{2.15}$$

ossia la somma delle varianze dei dati proiettati lungo le direzioni $\mathbf{w}_j$.

Pertanto, il problema di addestramento dell'autoencoder lineare equivale alla ricerca di q direzioni ortonormali che massimizzino la varianza totale dei dati proiettati. Questo coincide esattamente con il problema risolto dalla PCA, la cui soluzione è fornita dagli autovettori della matrice di covarianza associati ai q maggiori autovalori.

In particolare, il minimo della funzione di costo è ottenuto quando le righe di W coincidono (a meno di una trasformazione ortogonale) con gli autovettori $\mathbf{v}_1, \ldots, \mathbf{v}_q$ associati agli autovalori $\lambda_1 \geq \cdots \geq \lambda_q$. In tal caso vale

$$W^\top W = P_q,$$

dove P_q denota il proiettore ortogonale sul sottospazio generato dalle prime q componenti principali.

Ne consegue che un autoencoder lineare, addestrato mediante minimizzazione dell'errore quadratico medio, apprende lo stesso sottospazio individuato dalla PCA. Le coordinate latenti possono differire da quelle ottenute tramite PCA per una trasformazione ortogonale, ma lo spazio latente appreso coincide con lo span delle prime q componenti principali, mostrando come la PCA possa essere interpretata come un caso particolare di autoencoder lineare.

Caso di un autoencoder non lineare

Si consideri ora un autoencoder in cui almeno uno tra encoder e decoder è una funzione non lineare. In particolare, si assuma un encoder del tipo

$$\mathbf{h} = g(\mathbf{x}) = \sigma(W\mathbf{x} + \mathbf{b}),$$

dove $\sigma(\cdot)$ è una funzione di attivazione non lineare applicata elemento per elemento, mentre il decoder ricostruisce l'input mediante una funzione generica

$$\hat{\mathbf{x}} = f(\mathbf{h}).$$

L'addestramento dell'autoencoder consiste ancora nella minimizzazione dell'errore quadratico medio di ricostruzione:

$$\mathcal{L}(f, g) = \frac{1}{M} \sum_{i=1}^{M} \|\mathbf{x}_i - f(g(\mathbf{x}_i))\|_2^2 . \tag{2.16}$$

A differenza del caso lineare, la mappatura complessiva $\mathbf{x} \mapsto \hat{\mathbf{x}}$ non è più una proiezione lineare su un sottospazio di dimensione ridotta. Di conseguenza, la funzione di costo non può essere riscritta in termini di varianza proiettata, né ricondotta a un problema agli autovalori della matrice di covarianza. In particolare, non è più possibile esprimere l'errore di ricostruzione come differenza tra una quantità costante e la varianza dei dati proiettati lungo un insieme di direzioni fisse.

L'autoencoder non lineare è quindi in grado di catturare strutture complesse e non lineari presenti nel dataset, che non possono essere rappresentate in modo efficace mediante una combinazione lineare di componenti principali rendendo possibile l'apprendimento di rappresentazioni latenti più flessibili e adatte a dati che giacciono approssimativamente su varietà non lineari.

2.4 Interpretabilità delle feature latenti

Uno degli obiettivi centrali nell'apprendimento di rappresentazioni è ottenere codifiche latenti che non siano solamente utili per la ricostruzione dei dati, ma anche interpretabili dal punto di vista umano. Nel contesto degli autoencoders, tale interpretabilità è strettamente legata alla capacità del modello di catturare e separare i fattori di variazione che governano la generazione dei dati osservati.

Definizione (Fattori di variazione). Si definiscono *fattori di variazione* le variabili latenti, generalmente non osservabili, che parametrizzano il processo generativo dei dati e ne determinano le principali modalità di cambiamento. Ciascun fattore di variazione corrisponde a una dimensione semantica distinta secondo cui le osservazioni possono variare, come ad esempio la forma, la posizione, l'orientamento, il colore o la presenza di specifici oggetti. [3]

L'introduzione di una strozzatura nello spazio latente o di vincoli di regolarizzazione nella funzione obiettivo costringe l'autoencoder a comprimere l'infor-

mazione contenuta nei dati di input, preservando principalmente gli aspetti rilevanti ai fini della ricostruzione. In linea di principio, questo processo può favorire l'apprendimento di rappresentazioni latenti che riflettono i fattori di variazione sottostanti ai dati, anziché limitarsi a una memorizzazione non strutturata delle osservazioni.

In uno scenario ideale, le componenti dello spazio latente risultano semanticamente interpretabili: la variazione di una singola variabile latente corrisponde a una modifica controllata e riconoscibile di un attributo specifico dell'osservazione ricostruita. In tal caso, i valori quantitativi assunti dalle feature latenti possono essere ricondotti a descrizioni qualitative comprensibili, rendendo lo spazio latente non solo compatto, ma anche concettualmente significativo.

Tuttavia, nella pratica, l'interpretabilità delle feature latenti non è garantita. Gli autoencoders standard sono addestrati esclusivamente per minimizzare l'errore di ricostruzione e tendono pertanto a organizzare lo spazio latente in modo funzionale a tale obiettivo, senza alcuna esplicita pressione a separare o strutturare semanticamente l'informazione. Di conseguenza, le rappresentazioni apprese risultano spesso difficili da interpretare e caratterizzate da una forte mescolanza dei fattori di variazione.

2.4.1 Rappresentazioni latenti disentangled

Una rappresentazione latente si dice *disentangled* quando i diversi fattori di variazione che descrivono i dati sono codificati in componenti latenti distinte e, idealmente, statisticamente indipendenti. In una tale rappresentazione, ciascuna variabile latente controlla un singolo fattore di variazione, mentre risulta invariata rispetto agli altri.

In presenza di una rappresentazione disentangled, la manipolazione di una singola dimensione dello spazio latente produce una variazione interpretabile e localizzata nell'output ricostruito, senza influenzare gli altri attributi dell'osservazione. Questa proprietà rende le rappresentazioni disentangled particolarmente desiderabili in applicazioni quali l'analisi esplorativa dei dati, il controllo generativo, la robustezza a variazioni spurie e il trasferimento di conoscenza tra domini.

Nonostante il loro interesse teorico e pratico, le rappresentazioni disentangled non emergono spontaneamente nell'addestramento di autoencoders classici. La sola presenza di una strozzatura dimensionale non è sufficiente a garantire la separazione dei fattori di variazione, e in molti casi il modello apprende

combinazioni complesse e non interpretabili di tali fattori, dando luogo a rappresentazioni *entangled*.

Per favorire l'apprendimento di rappresentazioni disentangled è quindi necessario introdurre vincoli aggiuntivi o specifiche scelte architetturali e di regolarizzazione. Tra queste rientrano l'imposizione di sparsità nello spazio latente, la promozione dell'indipendenza statistica tra le feature, o l'introduzione di termini di penalizzazione che incoraggino una separazione esplicita dei fattori di variazione. Tali strategie costituiscono la base di numerosi modelli avanzati, tra cui gli *Sparse Autoencoders*, che verranno analizzati nel seguito.

2.5 Sparse Autoencoders

Una possibile strategia per favorire l'apprendimento di rappresentazioni latenti astratte, disentangled e interpretabili consiste nell'introdurre esplicitamente vincoli di sparsità sulle attivazioni dello spazio latente. I modelli che adottano questa impostazione prendono il nome di **Sparse Autoencoders**.

A differenza degli autoencoder classici con strozzatura (bottleneck), nei quali la capacità di rappresentazione è limitata riducendo la dimensionalità dello spazio latente, negli Sparse Autoencoders si abbandona tale vincolo architetturale a favore di un vincolo di sparsità sulle attivazioni. In questo caso, lo spazio latente può avere dimensione pari o superiore a quella dell'input, e l'encoder realizza una mappatura del tipo

$$g : \mathbb{R}^n \to \mathbb{R}^q, \quad q \geq n, \tag{2.17}$$

richiedendo tuttavia che, per ciascun input, solo una frazione limitata delle unità latenti risulti significativamente attiva.

L'idea centrale è che, pur disponendo di uno spazio latente ad alta dimensionalità, il modello sia costretto a rappresentare ogni osservazione utilizzando un numero ridotto di componenti. Questo comportamento induce una codifica selettiva, nella quale le singole unità latenti tendono a rispondere a pattern o attributi specifici dei dati, favorendo rappresentazioni più strutturate e potenzialmente interpretabili.

Formalmente, dati un encoder g_θ e un decoder f_ϕ, la funzione obiettivo di uno Sparse Autoencoder può essere espressa come

$$\mathcal{L}(f_\phi, g_\theta) = \langle \Delta(\mathbf{x}_i, f_\phi(g_\theta(\mathbf{x}_i))) \rangle + \lambda \, \mathcal{R}_{\text{sparse}}\big(g_\theta(\mathbf{x}_i)\big), \tag{2.18}$$

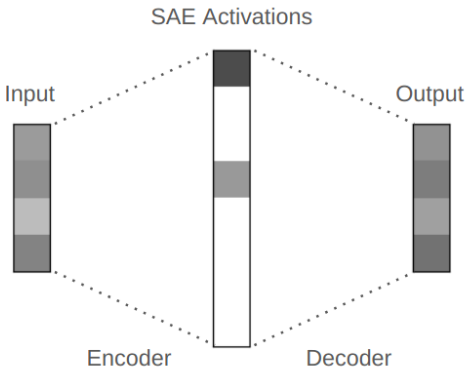

Figura 2.3: La figura mostra l'architettura di uno Sparse Autoencoder nel quale la dimensione dello stato latente è maggiore di quella di input.

dove $\mathcal{R}_{\text{sparse}}(\cdot)$ è un termine di regolarizzazione che impone vincoli di sparsità sulle attivazioni latenti.

Una delle scelte più comuni consiste nell'applicare una penalizzazione di tipo ℓ_1 alle attivazioni latenti,

$$\mathcal{R}_{\text{sparse}}(\mathbf{h}_i) = \|\mathbf{h}_i\|_1, \tag{2.19}$$

che incoraggia soluzioni in cui molte componenti del vettore latente sono nulle o prossime allo zero. In alternativa, è possibile imporre vincoli di tipo *top-k* (o k–sparsity), nei quali, per ciascun input, solo le k attivazioni di maggiore ampiezza vengono mantenute, mentre tutte le altre sono forzate a zero. Questo approccio impone una sparsità esplicita e controllata, indipendente dalla scala delle attivazioni.

Sebbene la sparsità non garantisca in senso rigoroso una completa separazione statistica dei fattori di variazione, essa introduce una forte pressione strutturale sulla rappresentazione latente, riducendo la codifica diffusa dell'informazione e favorendo l'emergere di feature più selettive e spesso *monosemantiche*. Per questo motivo, gli Sparse Autoencoders costituiscono uno strumento particolarmente efficace per l'analisi e l'interpretazione di rappresentazioni dense apprese da modelli complessi, oltre a rappresentare una base concettuale naturale per lo studio del disentanglement.

Capitolo 3

Embeddings

> Nets are for fish; once you get the fish you can forget the net. Words are for meaning; once you get the meaning you can forget the words.
>
> Zhuangzi

3.1 Introduzione

Quando leggiamo un testo, noi esseri umani siamo dotati della capacità di coglierne un aspetto di significato. Questo implica che dietro alle parole che leggiamo si cela una rappresentazione semantica che vorremmo potenzialmente far apprendere anche alle macchine. Dal momento che le macchine parlano con la lingua dei numeri e, non come noi, con quella delle parole è stato necessario introdurre degli strumenti che vorrebbero in linea di principio assegnare ad ogni parola un numero rappresentatitivo di un singificato. Tale strumenti sono chiamati *embeddings* e in questo capitolo, basandoci sul libro Speech and Language Processing di Stanford [4], si vedrà come vengono costruiti ed implementati per processare il testo.

3.2 L'ipotesi distribuzionale

Supponiamo di non conoscere il significato della parola *ongchoi*, ma di incontrarla nei seguenti contesti:

1. *L'ongchoi è deliziosa saltata con aglio.*
2. *L'ongchoi è ottima servita con riso.*
3. *...foglie di ongchoi con salse salate...*

Ora immaginiamo di aver già visto molte di queste parole-contesto in altri esempi, come:

1. *...gli spinaci saltati con aglio serviti sul riso...*
2. *...le coste, con i loro gambi e foglie, sono molto gustose...*
3. *...il cavolo riccio e altre verdure a foglia dal sapore salato...*

Il fatto che *ongchoi* compaia insieme a parole come *riso*, *aglio*, *deliziosa* e *salata*, proprio come *spinaci*, *coste* o *cavolo riccio*, suggerisce che l'ongchoi sia una **verdura a foglia** simile a queste altre verdure. Questo è il principio dell'ipotesi distribuzionale per il quale la parola *doctor-eye* o *oculist* è probabile che la troviamo nello stesso contesto.

Ipotesi Distribuzionale Si definisce ipotesi distribuzionale quella ipotesi per la quale parole simili compaiono in contesti simili.

Tale ipotesi suggerisce che il signficato delle parole venga appreso sulla base del contesto di dove queste appaiono. Se questa intuizione viene seguita allora può divenire possibile trovare una soluzione per assegnare dei numeri a delle parole sulla base della loro occorrenza dentro contesti. Prima di arrivare però a capire come costurire gli emebddings è necessario introdurre una ulteriore intuizione attibuita ad Osgood nel 1957.

3.3 Ipotesi di Osgood

Un contributo fondamentale alla rappresentazione del significato proviene dal lavoro di Osgood et al. (1957), che studiarono la componente affettiva delle parole. Osgood mostrò che i giudizi emotivi associati a una parola possono essere descritti lungo tre dimensioni principali:

1. **Valenza**: quanto la parola è percepita come positiva o negativa.
2. **Arousal**: quanto la parola induce attivazione emotiva.
3. **Dominanza**: quanto la parola implica controllo o sottomissione.

Ogni parola può quindi essere rappresentata come una tripla di valori numerici che ne definiscono la posizione in questo spazio tridimensionale. Ad esempio:

$$\textit{heartbreak} \rightarrow [2.5,\ 5.7,\ 3.6]$$

L'intuizione rivoluzionaria di Osgood è la seguente:

Ipotesi di Osgood
Il significato di una parola può essere rappresentato come un vettore in uno spazio semantico.

Questa idea è stata la prima ad anticipare direttamente i moderni modelli di *word embeddings*, in cui ogni parola è descritta come un punto in uno spazio multidimensionale corrispondente ad un singificato.

3.4 Embeddings

L'unione dell'ipotesi distribuzionale e dell'ipotesi di Osgood ha aperto la strada agli embeddings come modello fondamentale per la rappresentazione computazionale del significato. Da un lato, l'ipotesi distribuzionale fornisce il principio secondo cui il significato delle parole può essere inferito dai contesti in cui esse compaiono; dall'altro, l'ipotesi di Osgood suggerisce che tale significato possa essere rappresentato come un vettore numerico in uno spazio semantico. In questa sezione introduciamo i primi modelli di embedding basati su conteggi, che costituiscono il punto di partenza storico e concettuale delle moderne rappresentazioni distribuzionali. Per orientare il lettore, è utile chiarire fin da subito le principali tipologie di embeddings che verranno introdotte nel seguito. In base alla natura della rappresentazione prodotta, è possibile distinguere due grandi famiglie: embeddings **statici** ed embeddings **dinamici**.

Embeddings statici Si definisce statico un embedding in cui ogni parola del vocabolario è associata a un unico vettore pre-computato. Tale rappresentazione rimane invariata a prescindere dal contesto specifico in cui la parola appare. (Esempi: Matrici di co-occorrenza, Word2Vec, GloVe).

Negli embeddings statici, a ciascun tipo di parola del vocabolario è associato un unico vettore, indipendente dal contesto in cui la parola appare. Questa categoria include sia gli embeddings distribuzionali basati su conteggi, come le matrici termine–documento e termine–termine eventualmente ridotte tramite SVD, sia gli embeddings predittivi appresi mediante modelli neurali, come Word2Vec. Gli embeddings dinamici, o contestuali, producono invece una rappresentazione dipendente dal contesto: la stessa parola può essere associata a vettori diversi a seconda della frase in cui compare. Tali rappresentazioni sono generate da modelli di linguaggio neurali profondi, a partire da architetture ricorrenti fino ai moderni modelli Transformer, come BERT.

Embeddings dinamici (Contextual) Si definisce dinamico un embedding in cui la rappresentazione vettoriale di una parola viene generata "al volo" in funzione dell'intera sequenza di input. La stessa parola riceve quindi vettori diversi a seconda del contesto semantico e sintattico circostante. (Esempi: ELMo, BERT, GPT).

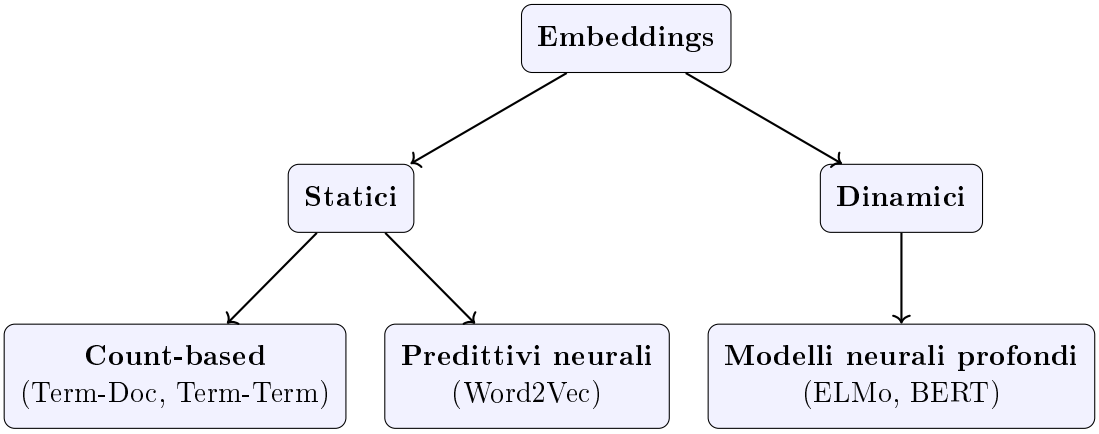

Figura 3.1: Classificazione delle principali tipologie di embeddings trattate nel capitolo.

3.4.1 Embeddings count-based

Il modo più semplice per costruire embeddings vettoriali delle parole è basato sulla **matrice di co-occorrenza**, una struttura che codifica quante volte determinati elementi linguistici compaiono insieme all'interno di un corpus. Esistono diverse varianti di matrici di co-occorrenza; in questa sezione ne introduciamo due fondamentali: la *term-document matrix* e la *term-term matrix*. Iniziamo dal caso più semplice.

Matrice termine-documento

In una matrice termine-documento ogni riga rappresenta una parola del vocabolario e ogni colonna rappresenta un documento appartenente a una collezione di testi. Ogni cella della matrice contiene il numero di volte in cui la parola associata alla riga compare nel documento associato alla colonna. Un esempio di term-document matrix è riportato nella Tabella 3.1, che mostra le occorrenze di quattro parole in quattro opere di Shakespeare.

Questa matrice può essere interpretata in due modi distinti ma complementari. Se si considerano le **colonne** della matrice, ciascun documento è rappresentato come un vettore in uno spazio di dimensione $|V|$, dove $|V|$ è la dimensione del vocabolario. In questo spazio, ogni asse corrisponde a una parola e il valore lungo ciascuna dimensione indica la frequenza della parola nel documento. Tale rappresentazione costituisce il fondamento del *vector space model* per il recupero dell'informazione, in cui documenti simili sono

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Tabella 3.1: Term-document matrix per quattro parole in quattro opere di Shakespeare. Ogni cella contiene il numero di occorrenze della parola (riga) nel documento (colonna). [3]

associati a vettori geometricamente vicini. Alternativamente, se si considerano le **righe** della matrice, ogni parola può essere interpretata come un vettore in uno spazio di dimensione pari al numero di documenti. In questo caso, le dimensioni del vettore non corrispondono più a parole, ma ai documenti del corpus, e il valore lungo ciascuna dimensione indica quanto frequentemente la parola compare in ciascun documento. Questa seconda interpretazione è particolarmente rilevante dal punto di vista semantico.

Interpretazione delle righe della matrice termine-documento
Due parole risultano simili se presentano distribuzioni simili sui documenti, ovvero se tendono a comparire negli stessi testi con frequenze comparabili.

La term-document matrix fornisce quindi una prima, semplice forma di *embedding distribuzionale* delle parole, in cui il significato emerge dalla loro distribuzione nei documenti del corpus.

Matrice termine-termine

Un'alternativa alla matrice termine–documento per la rappresentazione distribuzionale delle parole è la matrice termine-termine, detta anche *word–word matrix* o *term–context matrix*. In questo caso, le colonne della matrice non sono più etichettate da documenti, bensì da parole del vocabolario. La matrice ha quindi dimensionalità $|V| \times |V|$, dove $|V|$ indica la dimensione del vocabolario. In una matrice termine-termine, ogni riga rappresenta una **parola target** e ogni colonna rappresenta una **parola di contesto**. Ciascuna cella contiene il numero di volte in cui la parola di contesto compare nel contesto della parola target all'interno di un corpus di addestramento. Formalmente, la cella $M_{i,j}$ registra il numero di co-occorrenze tra la parola w_i

(target) e la parola w_j (contesto). Il concetto di *contesto* può essere definito in diversi modi. Una possibilità consiste nel considerare l'intero documento come contesto; tuttavia, nella pratica è molto più comune utilizzare contesti locali, definiti tramite una **finestra scorrevole** attorno alla parola target. Ad esempio, fissata una finestra di ampiezza $\pm k$, una parola è considerata di contesto se compare entro k posizioni a sinistra o a destra della parola target nel testo. Considerando tutte le occorrenze di ciascuna parola nel corpus e contando le parole che compaiono nelle rispettive finestre di contesto, è possibile costruire una matrice di co-occorrenza parola–parola. La Tabella 3.2 riporta un estratto reale di una matrice termine–termine calcolata sul corpus Wikipedia, in cui sono mostrate quattro parole target e alcune parole di contesto selezionate a scopo illustrativo [3].

Parola	**aardvark**	**computer**	**data**	**result**	**pie**	**sugar**
cherry	0	2	8	9	442	25
strawberry	0	0	0	1	60	19
digital	0	1670	1683	85	5	4
information	0	3325	3982	378	5	13

Tabella 3.2: Estratto di una matrice termine–termine calcolata sul corpus Wikipedia. Ogni cella contiene il numero di co-occorrenze tra la parola target (riga) e la parola di contesto (colonna) all'interno di una finestra di contesto locale [3].

In questa rappresentazione, ogni parola è associata a un vettore in uno spazio di dimensione $|V|$, in cui ciascuna dimensione corrisponde a una parola di contesto. Parole semanticamente simili tendono ad avere vettori simili, poiché compaiono in contesti linguistici analoghi. Ad esempio, dalla Tabella 3.2 si osserva che *cherry* e *strawberry* co-occorrono frequentemente con parole come *pie* e *sugar*, suggerendo una forte affinità semantica, mentre *digital* e *information* presentano distribuzioni simili rispetto a contesti come *computer* e *data*.

Interpretazione della matrice termine–termine
Due parole risultano semanticamente simili se presentano vettori di co-occorrenza simili, ovvero se tendono a comparire negli stessi contesti linguistici, anche nel caso in cui non compaiano mai direttamente insieme.

A questo punto abbiamo ottenuto una matrice termine-temrine,le cui colonne sono le parole di contesto, e i valori le co-occorrenze. Data $|V|$ la dimensione

del vocabolario, tale matrice ha una dimensionalità

$$|V| \times |V|.$$

Si hanno tuttavia due problemi.

1. Dal momento che ogni parole co-occorrerà solo con pochissime altre, *la dimensionalità della matrice è enorme.*
2. La maggior parte delle celle è nulla, e quindi *i vettori sono estremamente sparsi.*

Per affrontare i problemi legati all'elevata dimensionalità e alla natura estremamente sparsa della matrice termine-termine, ci sono diverse possibilità. Una di queste è la singular value decomposition in seugto descritta, e un'altra è quella di cambiare approccio e verrà presentata un'altra tipologia di embeddings basati su un altro paradigma di generazione diverso dal count-based che saranno successivamete presentati.

3.4.2 Riduzione dimensionale tramite SVD

Un metodo per la riduzione della dimensionalità è la Singular Value Decomposition applicata alla word–context matrix. Sia $M \in \mathbb{R}^{|V| \times |V|}$ la word–context matrix, eventualmente pesata tramite tf-idf. La decomposizione ai valori singolari (Singular Value Decomposition, SVD) consente di fattorizzare M come prodotto di tre matrici:

$$M = U\Sigma V^\top$$

dove U e V sono matrici ortogonali e Σ è una matrice diagonale contenente i valori singolari ordinati in modo decrescente. Ogni valore singolare rappresenta l'importanza di una direzione latente nello spazio semantico. I valori singolari maggiori catturano le correlazioni più rilevanti tra parole e contesti, mentre quelli più piccoli tendono a modellare rumore o variazioni locali meno informative. Per ottenere una rappresentazione a dimensionalità ridotta, si considera una versione troncata della decomposizione, mantenendo solo i primi k valori singolari:

$$M \approx U_k \Sigma_k V_k^\top$$

con $k \ll |V|$. Le righe della matrice $U_k\Sigma_k$ costituiscono una rappresentazione densa delle parole target in uno spazio latente di dimensione k. In

questo nuovo spazio, ogni parola è descritta da un vettore a dimensionalità ridotta, in cui le correlazioni semantiche risultano più evidenti rispetto alla rappresentazione originale sparsa. È importante osservare che la riduzione dimensionale non elimina esplicitamente la sparsità della matrice originale, ma proietta le parole in uno spazio denso in cui le relazioni semantiche emergono in forma compressa e più robusta.

3.4.3 Cosine Similarity

Dal momento che i vettori di embeddings vivono in uno spazio vettoriale che è anche uno spazio semantico, è possibile calcolare l'affinità di significato che due vettori hanno tramine la cosine similarity. La **cosine similarity** è una misura di similarità tra vettori che valuta il coseno dell'angolo compreso tra essi nello spazio vettoriale. Data la sua indipendenza dalla lunghezza dei vettori, risulta particolarmente adatta a confrontare vettori di frequenze o di pesi, come quelli derivati da matrici parola-contesto. Dati due vettori u e v, la similarità coseno è definita come:

$$\text{cosine\_sim}(u, v) = \frac{u \cdot v}{\|u\| \, \|v\|} = \frac{\sum_i u_i v_i}{\sqrt{\sum_i u_i^2} \sqrt{\sum_i v_i^2}}. \tag{3.1}$$

Il valore risultante è compreso tra -1 e 1:

- 1 indica che i vettori puntano nella stessa direzione (massima similarità),
- 0 indica che sono ortogonali (nessuna similarità),
- valori negativi indicano direzioni opposte (molto raro nei contesti di NLP).

Nelle applicazioni di elaborazione del linguaggio naturale la cosine similarity è spesso preferita alla distanza Euclidea, perché ci interessa confrontare il *pattern* delle co-occorrenze piuttosto che le loro magnitudini assolute. Ad esempio, due parole che co-occorrono con gli stessi termini di contesto, anche se con frequenze diverse, risulteranno comunque simili. La cosine similarity è quindi il principale strumento per valutare la similarità tra vettori distribuzionali e rappresenta un passaggio fondamentale prima di introdurre i modelli predittivi come Word2Vec e discendenti.

3.4.4 Word2Vec: un approccio predittivo

Sebbene i metodi basati su conteggi e la riduzione dimensionale tramite SVD permettano di ottenere rappresentazioni semanticamente dense, essi presentano limiti strutturali non trascurabili. Il calcolo della decomposizione ai valori singolari su matrici di co-occorrenza è computazionalmente oneroso, con una complessità che cresce sensibilmente rispetto alla dimensione del vocabolario, rendendo difficile la scalabilità su corpora massicci. Per superare queste criticità, Mikolov et al. (2013) hanno introdotto *Word2Vec*, un framework basato su un paradigma radicalmente diverso: la **predizione**. Invece di riassumere statistiche globali, Word2Vec apprende gli embeddings processando il testo localmente. Lo spostamento di paradigma risiede nel fatto che, anziché contare le occorrenze totali, addestriamo un classificatore su un compito di **classificazione binaria**. Il modello deve rispondere alla domanda:

"Data la parola target w (es. albicocca), qual è la probabilità che la parola candidata c (es. marmellata) compaia nel suo contesto?"

In questo approccio, noto come **self-supervision**, il testo stesso fornisce le etichette: ogni parola c che appare effettivamente vicino a w nel corpus fornisce un esempio positivo (etichetta 1). Al contrario, per addestrare il classificatore, il modello genera artificialmente degli esempi negativi campionando parole casuali dal vocabolario che non compaiono nel contesto di w.

Il classificatore e la funzione sigmoide

L'intuizione alla base del classificatore è che due parole siano semanticamente vicine se i loro vettori di embedding sono simili. Per misurare questa affinità, utilizziamo il **prodotto scalare** tra il vettore della parola target $\mathbf{w}$ e il vettore della parola di contesto $\mathbf{c}$:

$$\text{Similarity}(w, c) \approx \mathbf{w} \cdot \mathbf{c}$$

Poiché il prodotto scalare può assumere qualsiasi valore reale, utilizziamo la funzione **sigmoide** $\sigma(x)$ per mappare il risultato in una probabilità compresa tra 0 e 1:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.2}$$

Il modello stima quindi la probabilità che la coppia (w, c) sia un esempio positivo $(+)$ come:

$$P(+ \mid w, c) = \sigma(\mathbf{w} \cdot \mathbf{c}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{c}}} \tag{3.3}$$

Nel caso generale, data una parola target w e un'intera finestra di L parole di contesto $c_{1:L}$, il modello assume che le parole nel contesto siano indipendenti tra loro. La probabilità complessiva è dunque data dal prodotto delle probabilità individuali:

$$\log P(+ \mid w, c_{1:L}) = \sum_{i=1}^{L} \log \sigma(\mathbf{w} \cdot \mathbf{c}_i) \tag{3.4}$$

Apprendimento e Negative Sampling

L'addestramento consiste nello spostare iterativamente i vettori nello spazio semantico affinché il prodotto scalare tra parole che compaiono realmente insieme sia massimizzato, mentre quello tra parole prive di relazione sia minimizzato. Per rendere questo processo efficiente, si utilizza il **Negative Sampling**. Per ogni esempio positivo (w, c_{pos}) osservato nel testo, il modello genera k esempi negativi selezionando parole casuali dal vocabolario. Queste parole di "rumore" vengono scelte secondo una distribuzione unigramma pesata $P_\alpha(w)$:

$$P_\alpha(w) = \frac{\text{count}(w)^\alpha}{\sum_{w' \in V} \text{count}(w')^\alpha} \tag{3.5}$$

dove $\text{count}(w)$ indica la frequenza assoluta della parola w nel corpus, il simbolo w' al denominatore, che serve a calcolare il valore totale di normalizzazione, rappresenta l'indice della sommatoria che scorre su tutte le parole del vocabolario V e α è un parametro di smoothing (solitamente 0.75). Elevando le frequenze a questa potenza, si riduce la probabilità di campionare troppo spesso parole estremamente comuni e si aumenta quella delle parole più rare.

Perché due matrici? Il ruolo di W e C

Una caratteristica distintiva di Word2Vec è il mantenimento di **due rappresentazioni distinte** per ogni parola, organizzate in due matrici di pesi

separate (Figura 3.2). Una matrice W relativa alle parole target che contiene i vettori utilizzati quando la parola è il centro della finestra, e una matrice C che contiene i vettori utilizzati quando la parola appare nel contesto di un'altra o viene estratta come esempio negativo.

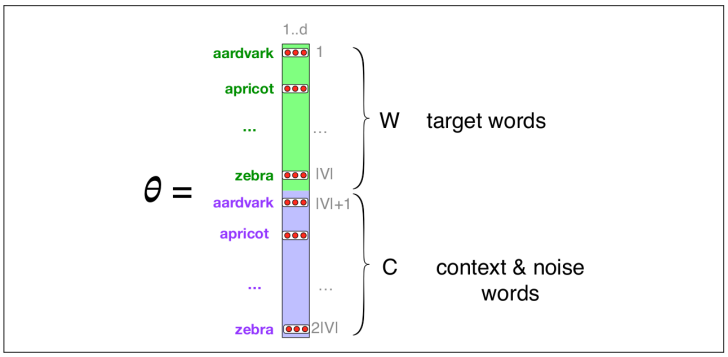

Figura 3.2: Lo skip-gram apprende in totale due insiemi di embedding, uno per i target (W) e uno per i contesti (C), per un totale di $2|V|$ vettori. L'addestramento mira a massimizzare la probabilità che parole vicine nel testo abbiano vettori simili.

Sdoppiando le matrici, il modello garantisce la **stabilità dell'ottimizzazione**. Se usassimo un unico vettore $\mathbf{v}$, il modello cercherebbe di massimizzare il prodotto scalare $\mathbf{v} \cdot \mathbf{v}$ (auto-similarità), portando i valori a crescere all'infinito. Con due matrici, il modello apprende relazioni distribuzionali senza questo vincolo. Al termine, si utilizzano solitamente i vettori di W o la media $W + C$.

3.4.5 Proprietà semantiche degli embeddings

L'apprendimento di questi vettori tramite il processo di ottimizzazione descritto non produce semplici sequenze numeriche prive di struttura, ma genera uno spazio geometrico capace di riflettere profonde relazioni linguistiche. La natura delle informazioni catturate da tali rappresentazioni dipende, in prima istanza, dalla configurazione della finestra di contesto, indicata come $L = 2m$, dove m rappresenta il raggio d'azione a destra e a sinistra della parola target. Una finestra ristretta (con m pari a 1 o 2) tende a privilegiare una similarità di tipo **sintattico**, raggruppando termini che condividono lo stesso ruolo grammaticale, come nel caso di verbi che occorrono in strutture frasali analoghe (ad esempio *scrive*, *dice* o *risponde*). Al contrario, l'adozione di una finestra più ampia (con m pari a 5 o più) sposta l'enfasi verso

una similarità di tipo **topico** o tematico, associando parole che appartengono allo stesso ambito semantico, come *ospedale*, *ambulanza* e *infermiere*, indipendentemente dalla loro funzione sintattica immediata. Questa capacità di astrazione permette agli embeddings densi di catturare efficacemente la similarità di secondo ordine.

Associazione paradigmatica (Similarità di secondo ordine)
Due parole risultano vicine nello spazio vettoriale non perché compaiono necessariamente insieme nel testo (associazione sintagmatica di primo ordine), ma perché sono circondate da contesti simili.

Mentre l'ipotesi distribuzionale fornisce la base metodologica per inferire il significato, l'associazione paradigmatica ne rappresenta il successo fenomenologico più rilevante negli embeddings densi: la capacità di mappare vicini due termini che, pur non incontrandosi mai direttamente, svolgono la medesima funzione semantica nel discorso. Una delle manifestazioni più celebri di questa proprietà è la facoltà di supportare il ragionamento analogico attraverso il cosiddetto modello del parallelogramma.

Modello del parallelogramma
Il modello del parallelogramma postula che le relazioni semantiche tra coppie di parole siano codificate come differenze vettoriali costanti nello spazio degli embeddings. Data un'analogia del tipo "a sta ad a^* come b sta ad b^*" (es. *uomo* : *donna* = *re* : *regina*), il termine incognito b^* può essere approssimato tramite l'operazione algebrica:

$$b^* \approx b - a + a^*$$

Geometricamente, ciò implica che il vettore che collega a ad a^* sia approssimativamente parallelo e di uguale lunghezza a quello che collega b a b^*, formando idealmente i lati di un parallelogramma nello spazio vettoriale.

Tale regolarità permette di catturare relazioni grammaticali e semantiche complesse, dai passaggi di genere, come nella nota equazione $\text{king} - \text{man} + \text{woman} \approx \text{queen}$, fino ai rapporti geopolitici come $\text{Paris} - \text{France} + \text{Italy} \approx \text{Rome}$. Da un punto di vista puramente geometrico, lo spazio degli embeddings è caratterizzato da proprietà di **parallelismo** e **ortogonalità**: mentre vettori paralleli indicano relazioni analoghe che si ripetono tra coppie diverse di parole, l'ortogonalità segnala l'indipendenza concettuale tra i termini.

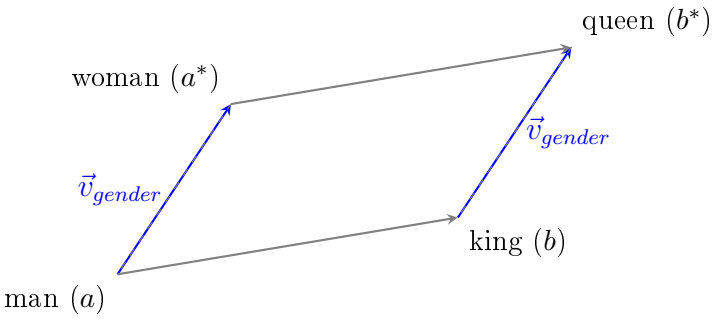

Figura 3.3: Rappresentazione geometrica del modello del parallelogramma applicato all'analogia di genere.

In sintesi, queste rappresentazioni apprese in modalità *self-supervised* incorporano automaticamente una vasta gamma di informazioni sintattiche e semantiche, rendendo la scelta della dimensione del vettore e dell'ampiezza della finestra i parametri critici per determinare la qualità del risultato finale. Tuttavia, nonostante la loro straordinaria efficacia, questi modelli presentano un limite intrinseco dovuto alla loro natura **statica**: ogni parola del vocabolario è associata a un unico vettore che deve condensare in sé tutti i possibili sensi del termine, rendendo difficile la gestione della polisemia. Per superare questa rigidità e permettere alla rappresentazione di adattarsi dinamicamente al contesto specifico di ogni singola frase, la ricerca si è evoluta verso lo sviluppo degli embeddings dinamici.

3.5 Embeddings dinamici

Nelle sezioni precedenti abbiamo analizzato gli embeddings statici, i quali associano a ogni termine del vocabolario un unico vettore numerico invariante. Come evidenziato dai modelli classici come Word2Vec e GloVe, tale approccio permette di catturare relazioni semantiche globali, ma presenta un limite intrinseco: l'incapacità di gestire la polisemia e l'omonimia. In una rappresentazione statica, parole come *spesso* (che può indicare una frequenza o uno spessore) o *capo* (che può riferirsi a un indumento o a un superiore) sono costrette in un unico punto dello spazio vettoriale, che risulta essere una "media" forzata dei diversi sensi della parola. Il passaggio verso gli **embeddings dinamici**, o contestuali, segna un cambiamento di paradigma fondamentale nel Natural Language Processing. In questo approccio, la rappresentazione di una parola non è più estratta da una tabella di ricerca (*lookup table*) fis-

sa, ma viene generata dinamicamente come funzione dell'intera sequenza di input. Ciò implica che la stessa parola riceverà vettori numerici differenti a seconda del contesto sintattico e semantico in cui appare.

Embeddings contestuali
Data una sequenza di token $x_1, \ldots, x_n$, un embedding contestuale è una rappresentazione vettoriale h_i per il token x_i tale che $h_i = f(x_i, C)$, dove C rappresenta l'intera sequenza circostante (contesto). A differenza degli embeddings statici, h_i rappresenta un'istanza specifica della parola (*word instance*) e non la sua categoria astratta nel vocabolario (*word type*).

Nella pratica, queste rappresentazioni emergono dagli stati interni di modelli di linguaggio neurali profondi. Mentre un embedding statico è un parametro del modello appreso durante l'addestramento e poi congelato, un embedding dinamico è il risultato computazionale dell'attivazione della rete durante la fase di inferenza. Per ottenere una rappresentazione robusta, è comune non limitarsi all'output dell'ultimo strato della rete neurale, ma combinare (ad esempio tramite media o concatenazione) le attivazioni provenienti da diversi livelli intermedi del modello, i quali catturano informazioni a diversi gradi di astrazione, dalla morfologia alla semantica complessa. Questa flessibilità permette di risolvere il problema della polisemia: il modello "osserva" le parole circostanti e adatta il vettore della parola target per rifletterne il senso specifico in quel momento. Per comprendere come sia possibile trasformare una sequenza di parole in una sequenza di vettori così precisi, è necessario analizzare l'evoluzione delle architetture neurali, partendo dalle strutture progettate per gestire il tempo e la sequenzialità: le reti neurali ricorrenti.

3.6 Reti Neurali Ricorrenti

Il linguaggio è un fenomeno intrinsecamente temporale: la comprensione di una parola o di una frase dipende dalla sequenza di eventi acustici o testuali che l'hanno preceduta. I modelli precedentemente analizzati, come le reti feedforward a finestra fissa o gli embeddings statici, trattano l'input in modo simultaneo o limitato localmente, fallendo nel catturare dipendenze a lungo raggio. Le Reti Neurali Ricorrenti (RNN), e in particolare le reti di Elman, introducono un meccanismo per gestire sequenze di lunghezza arbitraria tramite connessioni cicliche, permettendo al modello di mantenere

una "memoria" del passato.

Rete Neurale Ricorrente
Una RNN è una rete neurale in cui l'output di un neurone al tempo t viene reimmesso come input allo stesso neurone al tempo $t+1$. Questo ciclo permette di modellare sistemi dinamici in cui l'output corrente dipende dall'intera storia degli input precedenti.

Formalmente, data una sequenza di input rappresentata da vettori **one-hot** $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_T)$, dove ogni $\mathbf{x}_t \in \{0,1\}^{|V|}$ ha una dimensione pari alla cardinalità del vocabolario, il primo passaggio consiste nel recuperare l'embedding corrispondente. Sia $\mathbf{E} \in \mathbb{R}^{d \times |V|}$ la matrice di embedding, il vettore denso $\mathbf{e}_t$ al tempo t è ottenuto tramite il prodotto:

$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t \tag{3.6}$$

Lo stato nascosto $\mathbf{h}_t$ viene calcolato combinando l'embedding corrente $\mathbf{e}_t$ con lo stato nascosto precedente $\mathbf{h}_{t-1}$:

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t + \mathbf{b}) \tag{3.7}$$

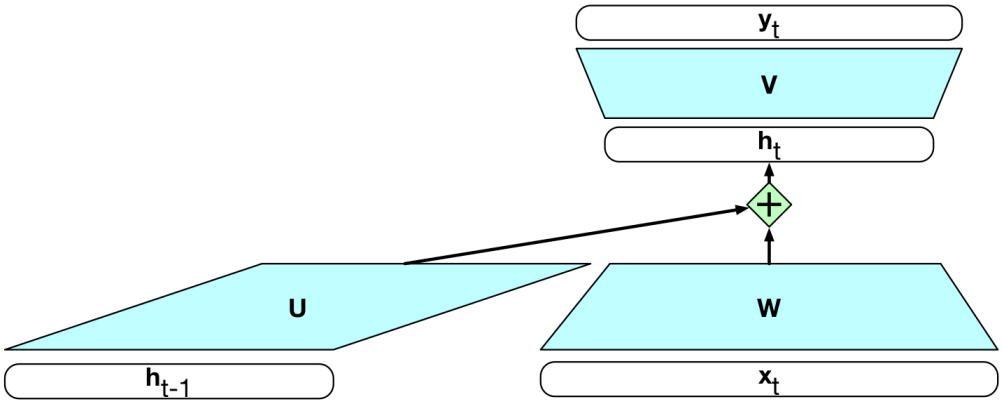

Figura 3.4: Illustrazione del funzionamento di una RNN. Ad ogni passo l'informazione contestuale viene consegnata a quello successivo.

Dove:

- $\mathbf{W} \in \mathbb{R}^{d \times d}$ è la matrice dei pesi per l'input (embedding).

- $\mathbf{U} \in \mathbb{R}^{d \times d}$ è la matrice dei pesi ricorrenti.
- $\mathbf{b} \in \mathbb{R}^{d}$ è il vettore di bias.
- g è una funzione di attivazione non lineare (solitamente tanh o ReLU).

L'output della rete al tempo t, denotato con $\hat{\mathbf{y}}_t$, è una distribuzione di probabilità calcolata tramite la funzione **softmax** applicata a una proiezione dello stato nascosto:

$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t + \mathbf{c}) \tag{3.8}$$

Dove $\mathbf{V} \in \mathbb{R}^{|V| \times d}$ proietta lo stato nascosto nuovamente nello spazio del vocabolario per generare i punteggi (logits) per ogni parola.

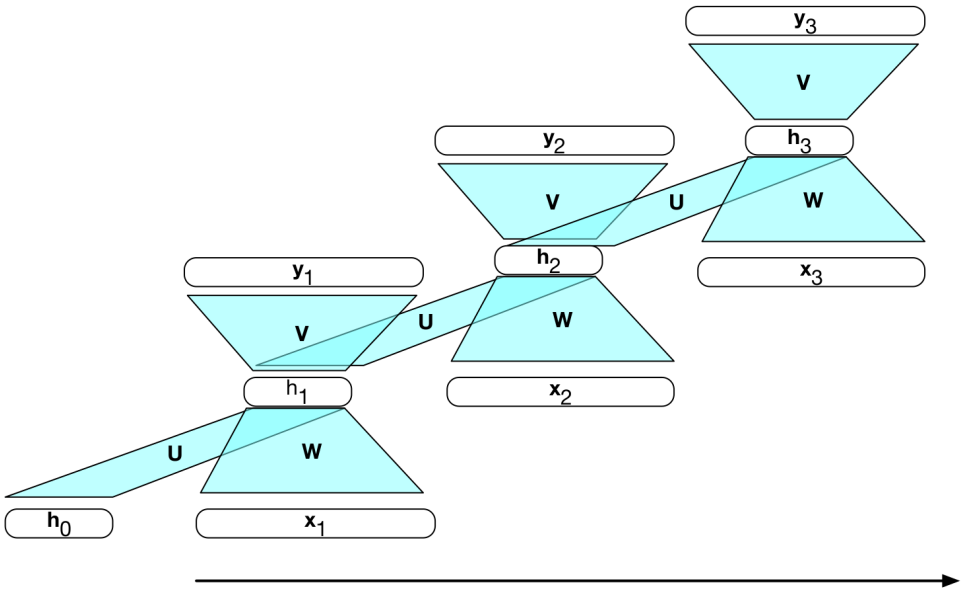

Figura 3.5: Rappresentazione srotolata (unrolled) di una RNN. Le frecce orizzontali mostrano il passaggio dello stato nascosto attraverso il tempo (pesi $\mathbf{U}$), mentre quelle verticali indicano l'elaborazione dell'input ($\mathbf{W}$) e la generazione dell'output ($\mathbf{V}$).

3.6.1 RNN come Language Models

I modelli di linguaggio hanno il compito di assegnare una probabilità a una sequenza di parole o, equivalentemente, di predire la parola successiva da-

to un contesto precedente. Per una sequenza di parole $w_{1:n}$, la probabilità congiunta può essere scomposta tramite la **regola della catena** (chain rule):

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i \mid w_{1:i-1}) \tag{3.9}$$

Utilizzando una RNN, modelliamo questa probabilità condizionata assumendo che la dimensione dell'embedding d_e e quella dello stato nascosto d_h siano le stesse (dimensione del modello d). La probabilità che la parola successiva al tempo $t+1$ sia l'elemento k del vocabolario è rappresentata dalla k-esima componente del vettore $\hat{\mathbf{y}}_t$:

$$P(w_{t+1} = k \mid w_1, \ldots, w_t) = \hat{\mathbf{y}}_t[k] \tag{3.10}$$

Di conseguenza, la probabilità dell'intera sequenza è definita come il prodotto delle probabilità dei termini corretti effettivamente osservati ad ogni passo temporale:

$$P(w_{1:n}) = \prod_{i=1}^{n} \hat{\mathbf{y}}_i[w_i] \tag{3.11}$$

3.6.2 Generaizone di Embeddings tramite RNN

Questa architettura si presta a generare rappresentazioni delle parole, gli emebddings, capaci di adattarsi al contesto nel quale compaiono.

Embedding Contestuale nelle RNN
In una RNN, il vettore $\mathbf{h}_t$ costituisce l'embedding contestuale della parola w_t. A differenza dell'embedding statico $\mathbf{e}_t$ (che è una semplice riga della matrice $\mathbf{E}$), il vettore $\mathbf{h}_t$ è dinamico: esso integra informazioni riguardanti l'intera storia della frase fino a quel momento.

Per estrarre questi embeddings da una RNN addestrata, si preleva lo stato $\mathbf{h}_t$ dal layer ricorrente. Questo vettore cattura sia la semantica intrinseca della parola corrente sia l'influenza sintattico-semantica del contesto precedente, risolvendo (in parte) i limiti degli embeddings statici nella gestione della polisemia.

3.6.3 RNN Bidirezionali (Bi-RNN)

Un limite intrinseco delle RNN analizzate finora è che lo stato nascosto $\mathbf{h}_t$ cattura esclusivamente le informazioni provenienti dal passato (contesto sinistro). Tuttavia, nel linguaggio naturale, il significato di una parola spesso dipende anche dalle parole che la seguono (contesto destro). Per ovviare a questo limite si utilizzano le RNN bidirezionali (Bi-RNN). L'idea alla base è quella di addestrare due reti distinte:

1. Una RNN forward ($\overrightarrow{RNN}$) che processa la sequenza da sinistra a destra, generando una sequenza di stati $\overrightarrow{\mathbf{h}}_1, \ldots, \overrightarrow{\mathbf{h}}_n$.
2. Una RNN backward ($\overleftarrow{RNN}$) che processa la sequenza da destra a sinistra (dalla fine all'inizio), generando una sequenza di stati $\overleftarrow{\mathbf{h}}_n, \ldots, \overleftarrow{\mathbf{h}}_1$.

Embedding Contestuale Bidirezionale
L'embedding contestuale finale per una parola w_t si ottiene concatenando lo stato nascosto della rete forward con quello della rete backward:

$$\mathbf{h}_t^{bi} = [\overrightarrow{\mathbf{h}}_t \oplus \overleftarrow{\mathbf{h}}_t] \tag{3.12}$$

In questo modo, il vettore risultante $\mathbf{h}_t^{bi}$ di dimensione $2d$ rappresenta la parola w_t tenendo conto dell'intera frase circostante.

Questa rappresentazione è estremamente potente perché permette al modello di "sapere" cosa succederà in futuro mentre interpreta il presente, rendendo l'embedding molto più ricco per compiti come la traduzione automatica o il riconoscimento di entità nominate (NER).

3.6.4 Il problema del Gradiente Svanente

Nonostante la loro eleganza teorica, le RNN presentano dei limiti pratici significativi quando devono gestire sequenze molto lunghe. Il problema principale è noto come Vanishing Gradient. Durante l'addestramento tramite l'algoritmo di *Backpropagation Through Time* (BPTT), i gradienti vengono propagati all'indietro attraverso ogni passo temporale. Poiché il calcolo coinvolge moltiplicazioni ripetute della stessa matrice di pesi $\mathbf{U}$, se i valori di questa matrice sono piccoli, il segnale del gradiente tende a ridursi esponenzialmente man mano che ci si allontana nel passato.

1. *Perdita di dipendenze a lungo raggio:* Se una parola all'inizio della frase è cruciale per predire una parola alla fine (es. l'accordo di genere tra un soggetto e un participio molto distanti), la RNN "dimentica" l'informazione iniziale perché il gradiente non riesce a trasportare l'errore così indietro nel tempo.

2. *Instabilità:* Al contrario, se i pesi sono molto grandi, il gradiente può crescere a dismisura (*Exploding Gradient*), rendendo l'addestramento instabile.

Per risolvere l'incapacità delle RNN classiche di mantenere una memoria a lungo termine stabile, sono state introdotte architetture più sofisticate basate sul concetto di meccanismi di gating, come le LSTM (Long Short-Term Memory) che analizzeremo nella prossima sezione.

3.7 LSTM: Long Short-Term Memory

Nonostante l'accesso all'intera sequenza precedente, le RNN classiche faticano a gestire informazioni distanti dal punto corrente di elaborazione. Come osservato in [4], in una frase come:

> *"The flights the airline was canceling were full."*

Assegnare una probabilità corretta a *were* è difficile perché il soggetto plurale *flights* è distante, mentre il singolare *airline* è più vicino nel contesto intermedio. Idealmente, una rete dovrebbe mantenere l'informazione del plurale fino a quando non è necessaria. Questa incapacità delle RNN deriva da due problemi principali. La prima è la sovrapposizione di compiti: lo strato nascosto deve simultaneamente fornire informazioni per la decisione corrente e aggiornare/trasportare informazioni per il futuro. Mentre la seconda è il già accennato problema del gradiente svanante (Vanishing Gradient): durante l'addestramento tramite *backpropagation through time*, i gradienti vengono sottoposti a moltiplicazioni ripetute. Spesso questo processo porta i gradienti a zero, rendendo impossibile apprendere dipendenze a lungo raggio. Le reti LSTM risolvono questi problemi dividendo la gestione del contesto in due compiti: rimuovere le informazioni non più necessarie e aggiungere quelle utili per il futuro. Lo fanno introducendo uno strato di contesto esplicito (c_t) e dei meccanismi di gating.

3.7.1 Meccanismi di Gating

I gate sono implementati tramite strati feedforward seguiti da una funzione sigmoide (σ) e da una moltiplicazione puntuale ($\odot$, o prodotto di Hadamard). La sigmoide spinge i valori verso 0 o 1, agendo come una maschera binaria: i valori vicino a 1 passano quasi invariati, quelli vicino a 0 vengono cancellati.

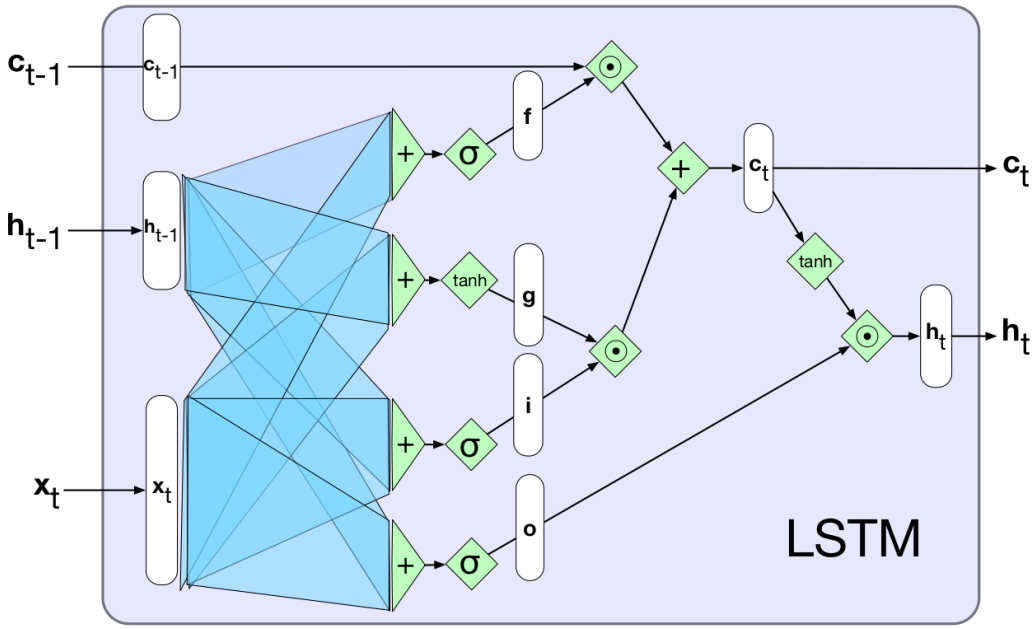

Figura 3.6: Rappresentazione di una singola unità LSTM come grafo computazionale. Gli input consistono nell'input attuale x_t, lo stato nascosto precedente h_{t-1} e il contesto precedente c_{t-1}. Gli output sono il nuovo stato nascosto h_t e il contesto aggiornato c_t [4].

3.7.2 Le equazioni del modello

Seguendo fedelmente la formulazione di Jurafsky & Martin [4], il calcolo per una singola unità LSTM all'istante t è definito dalle seguenti operazioni:

1. *Forget Gate*. Determina quali informazioni eliminare dal contesto precedente c_{t-1}:

$$f_t = \sigma(U_f h_{t-1} + W_f x_t) \tag{3.13}$$

$$k_t = c_{t-1} \odot f_t \tag{3.14}$$

2. *Add Gate.* Estrae le nuove informazioni e genera la maschera per decidere cosa aggiungere al contesto:

$$g_t = \tanh(U_g h_{t-1} + W_g x_t) \tag{3.15}$$

$$i_t = \sigma(U_i h_{t-1} + W_i x_t) \tag{3.16}$$

$$j_t = g_t \odot i_t \tag{3.17}$$

3. *Aggiornamento del contesto.* Il nuovo vettore di contesto c_t si ottiene sommando il contesto filtrato (k_t) e la nuova informazione selezionata (j_t):

$$c_t = j_t + k_t \tag{3.18}$$

4. *Output Gate.* Decide quali informazioni del contesto devono essere esposte nello stato nascosto h_t:

$$o_t = \sigma(U_o h_{t-1} + W_o x_t) \tag{3.19}$$

$$h_t = o_t \odot \tanh(c_t) \tag{3.20}$$

3.7.3 Modularità ed Embeddings

Lo stato nascosto h_t fornisce l'output della LSTM a ogni passo temporale. Questo vettore costituisce l'embedding contestuale della parola, capace di integrare memorie a lungo termine. La complessità dell'unità è incapsulata, mantenendo una modularità che permette di sostituire le RNN con le LSTM in qualsiasi architettura complessa.

3.8 Architettura Encoder-Decoder e limite del *bottleneck*

Le LSTM (e, più in generale, le RNN) risolvono in modo parziale il problema del *vanishing gradient* e consentono di modellare dipendenze più lunghe rispetto alle RNN "classiche". Tuttavia, quando il compito richiede di trasformare un'intera sequenza in un'altra sequenza, come accade nella traduzione automatica, nel riassunto, o nella risposta a domande in forma generativa, emerge un ulteriore vincolo strutturale. In questi scenari non basta più produrre uno stato nascosto per ogni parola: occorre un meccanismo che condensi l'informazione dell'input e la renda utilizzabile per generare l'output.

3.8.1 Il problema *sequence-to-sequence*

Consideriamo un input tokenizzato come

$$X = (x_1, x_2, \ldots, x_n),$$

e un output

$$Y = (y_1, y_2, \ldots, y_m),$$

con m e n in generale diversi. L'obiettivo è modellare la distribuzione condizionata $P(Y \mid X)$, ovvero la probabilità di generare una sequenza Y data la sequenza X. Come in ogni modello linguistico autoregressivo, si applica la regola della catena:

$$P(Y \mid X) = \prod_{t=1}^{m} P(y_t \mid y_{<t}, X), \tag{3.21}$$

dove $y_{<t} = (y_1, \ldots, y_{t-1})$. L'architettura Encoder–Decoder affronta il problema separando comprensione e generazione in due moduli distinti: un encoder che legge l'input e costruisce una rappresentazione interna, e un decoder che, condizionato da tale rappresentazione, genera l'output un token alla volta [4]. Come si vede in Figura **??**, questa archiettura si scompone in due sotto architetture:

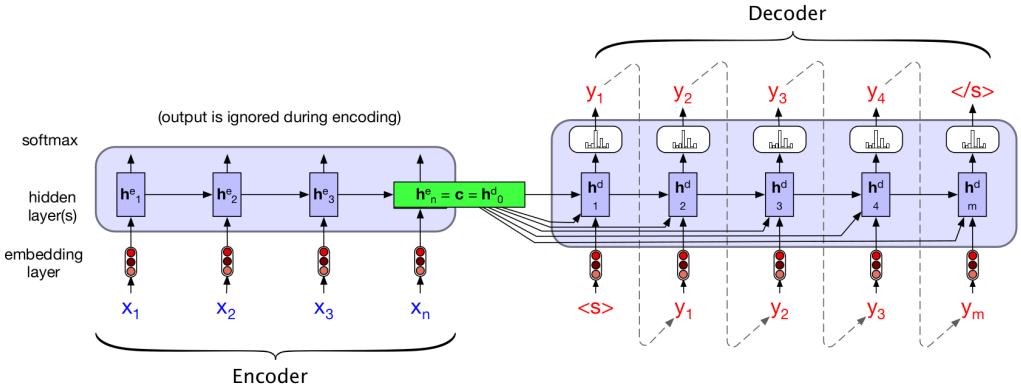

Figura 3.7: Schema formale dell'encoder–decoder ricorrente: l'encoder produce una sequenza di stati $h_1^{(e)}, \ldots, h_n^{(e)}$ e il suo stato finale $h_n^{(e)}$ viene identificato con il vettore di contesto c, usato per inizializzare il decoder ($h_0^{(d)}$) e, nella variante mostrata, reso disponibile a ogni passo di decodifica. Questa dipendenza da un unico vettore di contesto anticipa il problema del *bottleneck* discusso in seguito. Adattata da [4].

1. *Encoder.* E' tipicamente una RNN/LSTM che processa l'input in ordine:
$$h_t^{(e)} = f_{\text{enc}}(x_t, h_{t-1}^{(e)}), \qquad t = 1, \dots, n, \tag{3.22}$$
dove $h_t^{(e)} \in \mathbb{R}^d$ è lo stato nascosto (o, nel caso LSTM, una funzione di stato nascosto e memoria), e f_{enc} indica la dinamica ricorrente. Nel modello "classico" Encoder-Decoder, l'intera sequenza viene riassunta in un singolo vettore a dimensione fissa, spesso identificato con l'ultimo stato dell'encoder:
$$c \triangleq h_n^{(e)}. \tag{3.23}$$
Questo vettore c è chiamato *context vector* (o *sentence embedding*): dovrebbe contenere le informazioni necessarie per ricostruire, tradurre o generare coerentemente l'output.

2. *Decoder.* E' una seconda RNN/LSTM che definisce uno stato $h_t^{(d)}$ e produce, a ogni passo, una distribuzione sul vocabolario tramite una proiezione e `softmax`:
$$h_t^{(d)} = f_{\text{dec}}(y_{t-1}, h_{t-1}^{(d)}, c), \tag{3.24}$$
$$P(y_t \mid y_{<t}, X) = \text{softmax}(W_o h_t^{(d)} + b_o). \tag{3.25}$$
In pratica, c viene usato per inizializzare lo stato del decoder (o come input addizionale a ciascun passo). Una scelta comune è
$$h_0^{(d)} = \phi(c), \tag{3.26}$$
dove ϕ è una trasformazione appresa (ad esempio lineare + non linearità).

Encoder-Decoder
L'encoder trasforma una sequenza variabile X in un vettore fisso c; il decoder usa c come "condizione" per generare Y in modo autoregressivo, implementando la fattorizzazione in (3.21).

3.8.2 Il limite del *bottleneck* informativo

Il punto critico dell'architettura è evidente dalla definizione in (3.23): l'intera frase (potenzialmente lunga e ricca di dipendenze) viene compressa in un unico vettore $c \in \mathbb{R}^d$.

Questa scelta introduce un vero e proprio collo di bottiglia informativo:

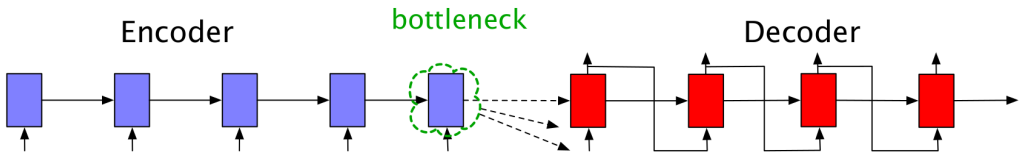

Figura 3.8: Schema Encoder–Decoder: quando il contesto c coincide con il solo stato nascosto finale dell'encoder, tutta l'informazione della sequenza sorgente deve attraversare un *collo di bottiglia* rappresentazionale prima di essere utilizzata dal decoder.

1. *Compressione forzata.* All'aumentare della lunghezza e complessità dell'input, cresce la quantità di informazione che deve essere "stipata" in una dimensione fissa d.

2. *Sensibilità alla distanza.* Anche con LSTM, la rappresentazione finale tende a privilegiare informazione recente nella sequenza; dettagli importanti all'inizio possono attenuarsi o venire sovrascritti durante la ricorrenza.

3. *Difficoltà di allineamento.* In traduzione (e in molti task seq2seq) serve sapere *quale parte* dell'input è rilevante per generare *quel particolare* token di output. Un vettore unico c non esplicita alcun allineamento fine-grained tra posizioni.

Come illustrato in Figura 3.8, imporre che il contesto c coincida con il solo stato finale dell'encoder costringe tutta l'informazione della frase sorgente a passare attraverso un'unica rappresentazione a dimensione fissa, introducendo un collo di bottiglia. Questo limite è concettualmente analogo a quello discusso nel tuo Capitolo 2 sugli autoencoders: anche lì un input complesso viene proiettato in uno spazio latente di dimensione fissa, e il decoder tenta una ricostruzione a partire da tale compressione. Nel caso seq2seq, però, la pressione sullo spazio latente è ancora più marcata, perché la ricostruzione non riguarda "lo stesso oggetto" ma una sequenza diversa (es. tradotta), che richiede sia contenuto semantico sia informazione strutturale [**jurafsky2024speech**, 4].

Limite del bottleneck
Nel modello Encoder-Decoder ricorrente, il contesto c gioca il ruolo di uno *spazio latente* a dimensione fissa. Se c non riesce a preservare le informazioni necessarie, il decoder è costretto a generare una sequenza basandosi su una rappresentazione incompleta: questo è il *bottleneck*.

3.8.3 Soluzione al bottleneck: Meccanismo dell'attenzione

Il bottleneck dell'encoder–decoder ricorrente deriva dal vincolo (3.23): l'unica informazione sull'input X disponibile al decoder è un singolo vettore c, spesso identificato con lo stato finale dell'encoder. Questo implica che c debba contenere tutta la semantica della frase sorgente e, per sequenze lunghe, le informazioni distanti (in particolare quelle all'inizio) possono risultare meno accessibili [4]. L'attenzione è una soluzione a questo collo di bottiglia:

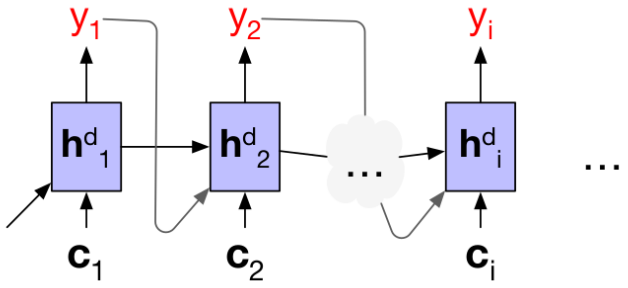

Figura 3.9: Nel meccanismo di attenzione, a ciascun passo di decodifica i il decoder utilizza un contesto *dinamico* c_i (diverso per ogni token generato), calcolato come funzione di tutti gli stati nascosti dell'encoder. Adattata da [4].

invece di obbligare il decoder a basarsi su un contesto statico, gli permette di ottenere informazione da tutti gli stati nascosti dell'encoder, selezionando dinamicamente la parte più rilevante dell'input a ogni passo di generazione (Figura 3.9 e Figura 3.10).

Meccanismo dell'attenzione
Il meccanismo di attenzione sostituisce il contesto statico c con un contesto dinamico c_i, calcolato come somma pesata degli stati dell'encoder. I pesi α_{ij} dipendono dallo stato del decoder e implementano un allineamento differenziabile tra posizioni dell'input e token generati in output, mitigando il *bottleneck* dell'encoder–decoder ricorrente.

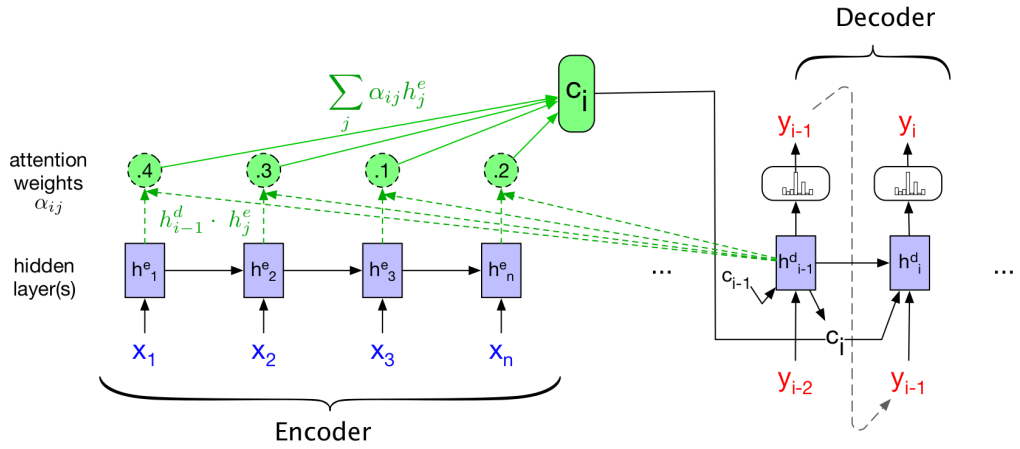

Figura 3.10: Schema encoder–decoder con attenzione, focalizzato sul calcolo di c_i. Per ogni stato precedente del decoder $h_{i-1}^{(d)}$ si calcola un punteggio di rilevanza rispetto a ciascuno stato dell'encoder $h_j^{(e)}$; i punteggi vengono normalizzati in pesi α_{ij} e usati per ottenere c_i come somma pesata degli stati dell'encoder. Adattata da [4].

Formalmente, sia $H = (h_1^{(e)}, \ldots, h_n^{(e)})$ la sequenza degli stati dell'encoder. L'attenzione introduce un vettore di contesto c_i che viene ricalcolato a ogni passo di decodifica i come funzione di tutti gli stati dell'encoder e dello stato precedente del decoder:

$$c_i = f\big(h_{1:n}^{(e)},\, h_{i-1}^{(d)}\big).$$

Il contesto c_i entra direttamente nel calcolo dello stato nascosto del decoder, che diventa dipendente sia dalla storia del decoder sia dall'informazione selezionata dalla sorgente:

$$h_i^{(d)} = g\big(y_{i-1},\, h_{i-1}^{(d)},\, c_i\big). \tag{3.27}$$

Il primo passo per costruire c_i consiste nel determinare quanto ciascuno stato dell'encoder $h_j^{(e)}$ sia rilevante per il passo corrente di decodifica. Per farlo

si definisce una funzione di compatibilità (o *alignment score*) tra lo stato precedente del decoder e lo stato j dell'encoder:

$$\text{score}\big(h_{i-1}^{(d)}, h_j^{(e)}\big). \tag{3.28}$$

La forma più semplice, detta dot-product attention, misura questa compatibilità come prodotto scalare:

$$\text{score}\big(h_{i-1}^{(d)}, h_j^{(e)}\big) = h_{i-1}^{(d)} \cdot h_j^{(e)}. \tag{3.29}$$

I punteggi così ottenuti vengono normalizzati con una softmax per produrre una distribuzione di pesi:

$$\alpha_{ij} = \frac{\exp\big(\text{score}(h_{i-1}^{(d)}, h_j^{(e)})\big)}{\sum_{k=1}^{n} \exp\big(\text{score}(h_{i-1}^{(d)}, h_k^{(e)})\big)}. \tag{3.30}$$

Per costruzione, $\alpha_{ij} \geq 0$ e $\sum_{j=1}^{n} \alpha_{ij} = 1$. Il contesto dinamico si ottiene quindi come media pesata degli stati dell'encoder:

$$c_i = \sum_{j=1}^{n} \alpha_{ij}\, h_j^{(e)}. \tag{3.31}$$

In questo modo, pur mantenendo un vettore $c_i \in \mathbb{R}^d$ a dimensionalità fissa, il modello evita la compressione in un unico riassunto globale: l'informazione viene invece recuperata in modo selettivo dall'intera memoria dell'encoder e aggiornata passo per passo secondo le necessità del decoder [4].

3.8.4 Verso i Transformer

Una volta introdotta l'attenzione come accesso diretto a una memoria di stati, il passo successivo (diventato centrale nella storia recente del NLP) è chiedersi se la ricorrenza sia ancora necessaria. Se un modello può collegare direttamente, tramite pesi di attenzione, posizioni lontane della sequenza, allora molte delle funzioni della memoria ricorrente possono essere replicate — e spesso superate — da meccanismi puramente basati su attenzione. Questa osservazione è il punto di partenza dei **Transformer**, in cui la dipendenza sequenziale della ricorrenza viene rimpiazzata da **self-attention** e computazione parallelizzabile [4].

In sintesi, la traiettoria logica è:

RNN/LSTM $\rightarrow$ Encoder–Decoder (bottleneck) $\rightarrow$ Attention (contesto dinamico) $\rightarrow$ Tran

Nella prossima sezione formalizziamo il meccanismo di attenzione e mostriamo come esso implementi, in modo differenziabile, un allineamento tra posizioni dell'input e token generati in output.

3.9 Il Transformer

"The true art of memory is the art of attention."

Samuel Johnson, *The Idler*

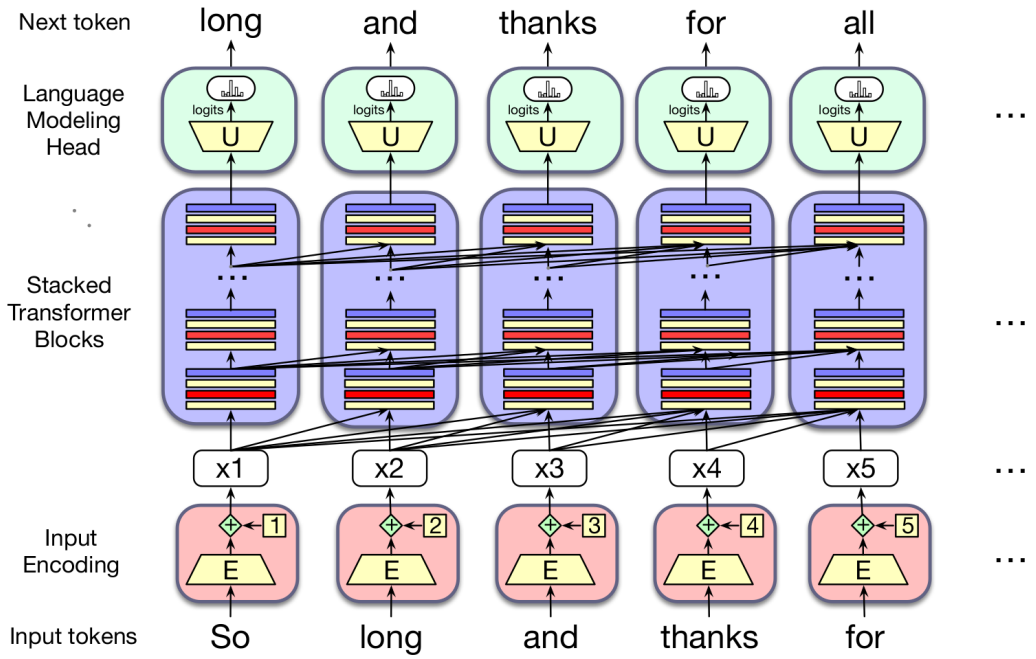

Figura 3.11: Schema di un Transformer causale (left-to-right) per language modeling. Ogni token in input viene codificato (embedding del token e della posizione), processato da una pila di blocchi Transformer, e infine proiettato tramite una testa di language modeling per predire il token successivo.

Il Transformer è l'architettura di riferimento per la costruzione dei moderni modelli di linguaggio neurali. A differenza delle reti ricorrenti, esso non elabora la sequenza in modo strettamente sequenziale: la dipendenza tra token viene modellata tramite un meccanismo di attenzione applicato in parallelo alle posizioni della sequenza. Questa scelta consente di costruire rappresentazioni contestuali profonde, integrando informazione proveniente da token anche molto distanti, e rende il calcolo altamente parallelizzabile [4]. In questa sezione consideriamo, come caso base, il Transformer per language modeling autoregressivo (left-to-right), in cui dato un prefisso di token $(w_1, \ldots, w_i)$ si stima una distribuzione di probabilità sul token successivo w_{i+1}. L'idea gene-

rale è che il modello produca, per ogni posizione i, un vettore contestuale che riassume ciò che è rilevante del contesto precedente per predire il prossimo token. La Figura 3.11 sintetizza la struttura, che può essere descritta tramite tre componenti principali.

1. La prima componente è l'input encoding, che trasforma ciascun token w_i in un vettore $x_i \in \mathbb{R}^d$ combinando un embedding lessicale con un'informazione di posizione. La sequenza di input viene così rappresentata come una matrice $X \in \mathbb{R}^{n \times d}$, dove n è la lunghezza della finestra di contesto e d è la dimensionalità del modello.

2. La seconda componente è una pila di blocchi Transformer (stacked transformer blocks). Ogni blocco implementa una trasformazione che mappa la sequenza di vettori in input $(x_1, \ldots, x_n)$ in una sequenza di vettori in output $(h_1, \ldots, h_n)$ della stessa lunghezza e stessa dimensionalità. Il cuore di ciascun blocco è la multi-head self-attention, che costruisce una rappresentazione contestuale per ogni posizione pesando e combinando le rappresentazioni delle altre posizioni nel contesto. A questa si affiancano una rete feedforward punto-a-punto, connessioni residue e normalizzazioni, che stabilizzano e potenziano l'apprendimento [4]. L'impilamento di molti blocchi consente al modello di costruire progressivamente rappresentazioni sempre più ricche, passando da segnali locali a dipendenze globali.

3. La terza componente è la language modeling head, che a partire dal vettore prodotto per ciascuna posizione (tipicamente all'ultimo blocco) calcola dei punteggi sul vocabolario e, tramite softmax, una distribuzione di probabilità sul token successivo. In questa fase compare spesso una matrice di proiezione verso il vocabolario, concettualmente interpretabile come un'operazione inversa rispetto all'embedding (unembedding) [4].

Nei paragrafi successivi entreremo nel dettaglio dei singoli elementi: prima formalizziamo la self-attention (Sezione **??**), poi descriviamo la struttura completa del blocco Transformer, la codifica posizionale e la testa di language modeling. Questa trattazione costituirà il passaggio naturale verso i modelli encoder bidirezionali, in particolare BERT, che utilizzano la stessa architettura di base ma con un regime di attenzione e un obiettivo di addestramento differenti [4].

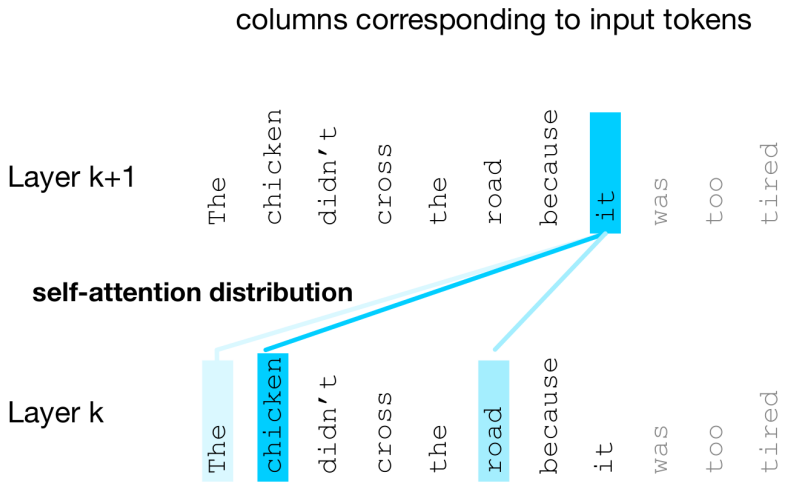

Figura 3.12: Esempio di distribuzione dei pesi di self-attention α: per costruire una rappresentazione contestuale del token *it* in uno strato superiore, il modello attribuisce peso maggiore ad alcuni token precedenti particolarmente informativi (ad es. *chicken* e *road*). In quel punto della sequenza, infatti, la coreferenza del pronome non è ancora disambiguata; è quindi plausibile che la rappresentazione di *it* debba incorporare evidenza proveniente da entrambe le possibili entità.

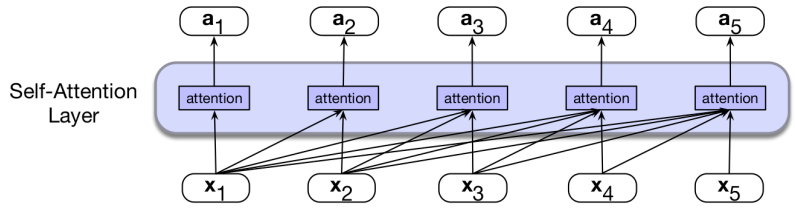

Figura 3.13: Self-attention causale (left-to-right): in posizione i il modello combina esclusivamente i token nelle posizioni $j \leq i$ e, tramite mascheramento, non può incorporare informazione proveniente da posizioni future.

3.9.1 Self-attention

Motivazione: dalle rappresentazioni statiche alle rappresentazioni contestuali

Negli *embeddings statici* (ad es. Word2Vec o GloVe) a ciascun tipo lessicale del vocabolario viene associato un vettore fissato una volta per tutte: la parola assume dunque la medesima rappresentazione indipendentemente dall'enunciato in cui compare. Questo assunto, pur utile per catturare regolarità semantiche globali, risulta limitante quando il significato di un token dipende in modo cruciale dal contesto linguistico. In particolare, numerosi fenomeni del linguaggio naturale richiedono di mettere in relazione elementi anche molto distanti nella sequenza: (i) *coreferenza* e risoluzione pronominale (un pronome può riferirsi a entità diverse), (ii) *accordo morfosintattico* (ad esempio tra soggetto e verbo), (iii) *disambiguazione del senso* (parole polisemiche che cambiano interpretazione in base al contesto), (iv) *dipendenze sintattiche a lungo raggio*.

L'obiettivo di un Transformer è costruire, strato dopo strato, una sequenza di *rappresentazioni contestuali* (o *contextual embeddings*), in cui il vettore associato a una posizione non descrive più soltanto il token in senso astratto, bensì la sua *istanza* nel contesto specifico in cui occorre. Formalmente, a partire da una sequenza di input $X = (x_1, \ldots, x_n)$, ogni strato produce una sequenza di output della stessa lunghezza, in cui ciascun elemento integra informazione selezionata dal contesto in modo differenziabile e appreso dai dati [4]. In questo quadro, la *self-attention* è il meccanismo centrale che determina *da quali posizioni* recuperare informazione e *con quale intensità*, producendo per ogni token una distribuzione di pesi sul contesto rilevante (Figura 3.12).

Idea chiave
La self-attention implementa un'operazione di *retrieval differenziabile* sul contesto: per ciascuna posizione i, il modello calcola coefficienti α_{ij} che misurano la rilevanza delle posizioni j (ammissibili) e costruisce una nuova rappresentazione come combinazione pesata delle informazioni recuperate. L'intero processo è appreso end-to-end durante l'addestramento [4].

Self-attention causale: dominio informativo e vincolo di autoregressione

Nel caso dei modelli autoregressivi (left-to-right), l'architettura deve essere coerente con la fattorizzazione causale della probabilità linguistica:

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i \mid w_{1:i-1}).$$

Di conseguenza, nel calcolo della rappresentazione della posizione i non è consentito utilizzare token in posizioni future $j > i$, poiché ciò introdurrebbe una dipendenza non ammissibile (leakage di informazione) e renderebbe il modello incoerente con il compito di predizione del token successivo. In una self-attention *causale*, pertanto, l'output in posizione i dipende unicamente dal prefisso $(x_1, \ldots, x_i)$:

$$a_i = f(x_1, \ldots, x_i),$$

e l'intero strato realizza una mappatura parallela tra sequenze

$$(x_1, \ldots, x_n) \mapsto (a_1, \ldots, a_n),$$

in cui però, per ogni i, l'insieme delle posizioni consultabili è vincolato a $j \leq i$ (Figura 3.13). Operativamente, tale vincolo viene implementato tramite una *maschera causale* (o *causal mask*) che annulla i contributi delle posizioni future prima della normalizzazione softmax.

Intuizione: self-attention come combinazione pesata del contesto

Per fissare l'idea, è utile introdurre una forma semplificata della self-attention: l'output a_i in posizione i può essere visto come una combinazione lineare delle rappresentazioni disponibili nel contesto ammissibile:

$$a_i = \sum_{j \leq i} \alpha_{ij} x_j, \tag{3.32}$$

dove:

- $\alpha_{ij} \in \mathbb{R}$ è un coefficiente scalare che misura il contributo della posizione j all'aggiornamento della posizione i;
- nel caso standard, i pesi sono non-negativi e normalizzati (*row-wise*) in modo da formare una distribuzione: $\alpha_{ij} \geq 0$ e $\sum_{j \leq i} \alpha_{ij} = 1$.

Questa formulazione mette in evidenza il punto essenziale: la rappresentazione contestuale di un token non è ottenuta "memorizzando" rigidamente il passato (come avviene nelle RNN), bensì *selezionando* e *aggregando* informazione da posizioni precedenti in modo adattivo. In altri termini, non tutti i token del prefisso sono ugualmente rilevanti: la self-attention apprende una strategia di selezione che enfatizza le parti del contesto più informative per il token corrente.

Scaled dot-product attention: ruoli di query, key e value

La formulazione effettivamente impiegata nei Transformer introduce tre proiezioni distinte della rappresentazione x_i (tipicamente l'output dello strato precedente). Per ogni posizione i si definiscono:

$$q_i = x_i W^Q, \qquad k_i = x_i W^K, \qquad v_i = x_i W^V, \tag{3.33}$$

dove W^Q, W^K, W^V sono matrici di parametri apprendibili. L'interpretazione standard è la seguente [4]:

- q_i (*query*) codifica ciò che la posizione i "richiede" al contesto, ossia quali caratteristiche cerca per aggiornare la propria rappresentazione;
- k_j (*key*) codifica ciò che la posizione j "offre" come criterio di confronto rispetto alle query;
- v_j (*value*) contiene l'informazione effettiva che verrà aggregata se la posizione j risulta rilevante.

Punteggi di compatibilità (scores). La rilevanza di una posizione j per l'aggiornamento della posizione i è stimata tramite un punteggio di compatibilità basato su prodotto scalare tra query e key:

$$\text{score}(i, j) = \frac{q_i \cdot k_j}{\sqrt{d_k}}, \tag{3.34}$$

dove d_k è la dimensionalità dei vettori q e k. Il fattore di scala $1/\sqrt{d_k}$ svolge un ruolo numerico fondamentale: senza scaling, all'aumentare di d_k i prodotti scalari tendono ad assumere magnitudini elevate, e la softmax può saturare (distribuzioni eccessivamente "piccate"), riducendo l'efficacia del gradiente durante l'addestramento [4].

Normalizzazione tramite softmax e vincolo causale. I punteggi $\text{score}(i,j)$ vengono convertiti in pesi di attenzione normalizzati tramite softmax sul contesto ammissibile:

$$\alpha_{ij} = \frac{\exp(\text{score}(i,j))}{\sum_{t \le i} \exp(\text{score}(i,t))} \qquad (j \le i). \tag{3.35}$$

Nel caso causale, la restrizione $j \le i$ si realizza introducendo una maschera M tale che $M_{ij} = 0$ se $j \le i$ e $M_{ij} = -\infty$ se $j > i$: i punteggi mascherati producono, dopo softmax, pesi nulli sulle posizioni future.

Aggregazione dei value e proiezione in output. L'output di una singola testa di attenzione (attention head) in posizione i è quindi una somma pesata dei value:

$$\text{head}_i = \sum_{j \le i} \alpha_{ij} v_j, \qquad a_i = \text{head}_i W^O, \tag{3.36}$$

dove W^O è una matrice di proiezione che riporta l'output alla dimensionalità del modello. In questa prospettiva, la self-attention può essere interpretata come un operatore che costruisce, per ciascuna posizione, una sintesi mirata del contesto, in cui la "selezione" è guidata dall'allineamento tra q_i e le k_j del prefisso.

Forma matriciale e dimensioni (utile per l'implementazione)

Sebbene la notazione per-indice chiarisca il significato, in un'implementazione la self-attention viene calcolata in forma matriciale. Indicando con $X \in \mathbb{R}^{n \times d}$ la matrice che raccoglie le rappresentazioni x_i (una riga per token), si definiscono:

$$Q = XW^Q, \qquad K = XW^K, \qquad V = XW^V.$$

La matrice dei punteggi è $S = \frac{QK^\top}{\sqrt{d_k}} \in \mathbb{R}^{n \times n}$; applicando (eventualmente) una maschera causale M si ottiene $S' = S + M$. La matrice dei pesi di attenzione si calcola applicando una softmax riga per riga:

$$A = \text{softmax}(S'),$$

e infine l'output complessivo dello strato di attenzione (prima della proiezione finale) è:

$$H = AV.$$

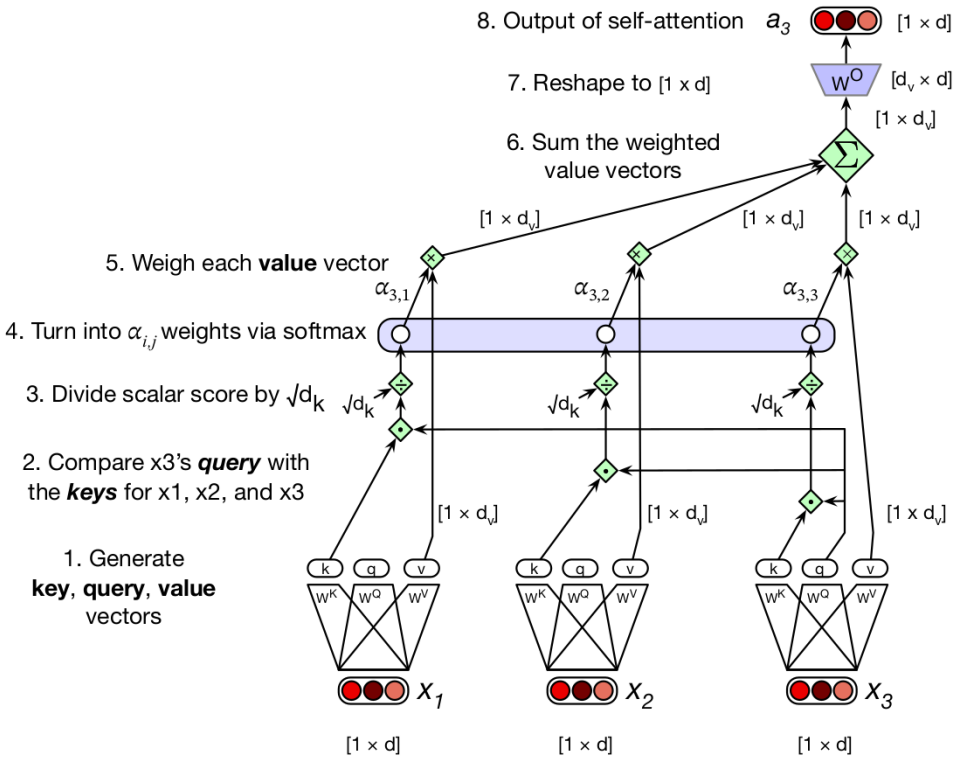

Figura 3.14: Calcolo di una singola testa di self-attention causale: (i) proiezioni in query/key/value; (ii) punteggi di compatibilità via dot-product scalato; (iii) normalizzazione in pesi α_{ij}; (iv) somma pesata dei value per ottenere head_i; (v) proiezione finale per ottenere a_i in dimensione del modello.

Questa forma rende esplicito un aspetto computazionale cruciale: la self-attention confronta ogni posizione con (potenzialmente) tutte le altre posizioni del contesto, producendo una matrice $n \times n$; ciò implica un costo in tempo e memoria tipicamente quadratico in n [4]. Tale considerazione diventa rilevante quando si discute la scalabilità del Transformer a finestre di contesto molto ampie.

Multi-head attention: pluralità di criteri di selezione

Una singola testa di attenzione produce una sola distribuzione $\alpha_{i\cdot}$ e, di conseguenza, una sola combinazione pesata del contesto per ciascuna posizione. Tuttavia, le relazioni linguistiche di interesse sono molteplici (sintattiche, semantiche, morfologiche) e non è in generale desiderabile costringere un unico meccanismo a catturarle simultaneamente. Per questo motivo, i Transformer impiegano la *multi-head attention*: si calcolano A teste in parallelo, ciascuna con parametri distinti, e si combinano i risultati [4]. Per ogni testa $c = 1, \ldots, A$:

$$q_i^{(c)} = x_i W_c^Q, \qquad k_i^{(c)} = x_i W_c^K, \qquad v_i^{(c)} = x_i W_c^V, \tag{3.37}$$

da cui si ottengono i corrispondenti output $\text{head}_i^{(c)}$. Gli output delle teste vengono quindi concatenati e proiettati per ripristinare la dimensionalità del modello:

$$a_i = \big(\text{head}_i^{(1)} \oplus \text{head}_i^{(2)} \oplus \cdots \oplus \text{head}_i^{(A)}\big) W^O. \tag{3.38}$$

In termini dimensionali, se ciascuna testa produce un vettore in $\mathbb{R}^{d_v}$, la concatenazione produce un vettore in $\mathbb{R}^{Ad_v}$; la matrice W^O ha quindi forma $(Ad_v) \times d$ e riporta l'output in $\mathbb{R}^d$, rendendo lo strato compatibile con connessioni residue e con l'impilamento di più blocchi. La conseguenza concettuale è che teste diverse possono apprendere criteri di selezione diversi e complementari (ad esempio, una testa può privilegiare dipendenze sintattiche, un'altra relazioni semantiche o segnali di coreferenza), aumentando la capacità espressiva del modello e la ricchezza delle rappresentazioni contestuali [4].

Osservazione conclusiva: self-attention e contestualizzazione progressiva

È opportuno sottolineare che la contestualizzazione non avviene in un singolo passaggio: in un Transformer profondo, la self-attention è applicata ripetutamente in una pila di blocchi. In ciascuno strato, i vettori x_i (che possono

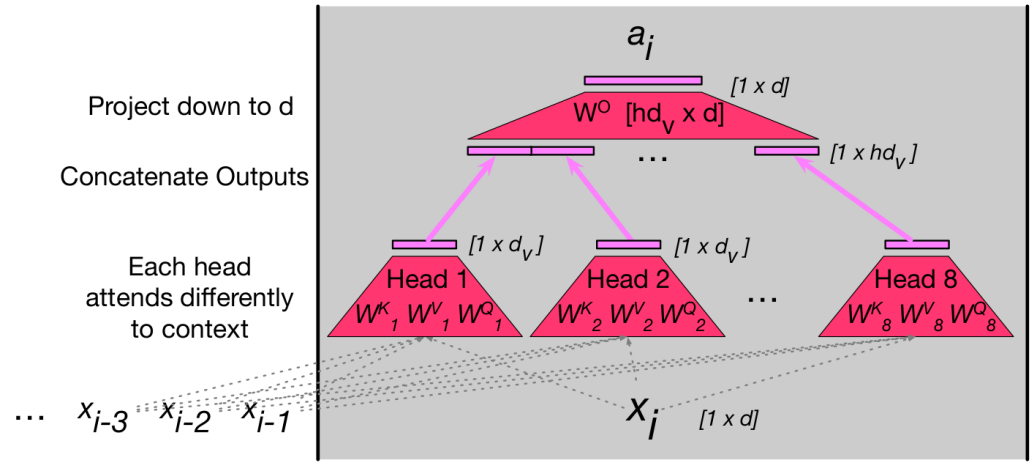

Figura 3.15: Multi-head attention: più teste calcolano in parallelo combinazioni pesate del contesto con parametri indipendenti; gli output vengono concatenati e proiettati per ottenere un vettore finale nella stessa dimensionalità dell'input.

essere visti come "stati" intermedi) vengono ricalibrati in funzione del contesto tramite pesi α_{ij} che dipendono a loro volta dalle rappresentazioni correnti. Ne risulta un processo di raffinamento progressivo, in cui gli strati inferiori tendono a catturare regolarità più locali, mentre quelli superiori integrano dipendenze più astratte e a più lungo raggio [4]. In questo senso, la self-attention costituisce il principale meccanismo con cui il Transformer trasforma embeddings inizialmente non contestuali in rappresentazioni contestuali idonee alla predizione autoregressiva (o, in altre varianti, alla comprensione bidirezionale).

3.9.2 Blocco Transformer

Nella sezione precedente abbiamo visto come la *self-attention* consenta a ciascun token di costruire una rappresentazione contestuale combinando informazione da altre posizioni della sequenza. Tuttavia, un Transformer reale non è composto da un singolo strato di attenzione: la **unità modulare** che viene impilata molte volte per ottenere rappresentazioni sempre più ricche è il *blocco Transformer* (*Transformer block*) [4]. Lo scopo del blocco è duplice: (i) permettere alla self-attention di *trasferire informazione* tra token, e (ii) arricchire *localmente* (token per token) la rappresentazione attraverso una trasformazione non lineare, mantenendo al tempo stesso stabilità numerica

e facilità di ottimizzazione quando si impilano decine o centinaia di strati.

Un blocco Transformer contiene quattro componenti principali [4]:

1. **Multi-Head Self-Attention** (integra informazione tra posizioni diverse);
2. **Feedforward Network** punto-a-punto (trasformazione non lineare applicata indipendentemente a ogni token);
3. **Connessioni residue** (somma che preserva e propaga informazione lungo la profondità);
4. **Layer Normalization** (normalizzazione che stabilizza l'addestramento).

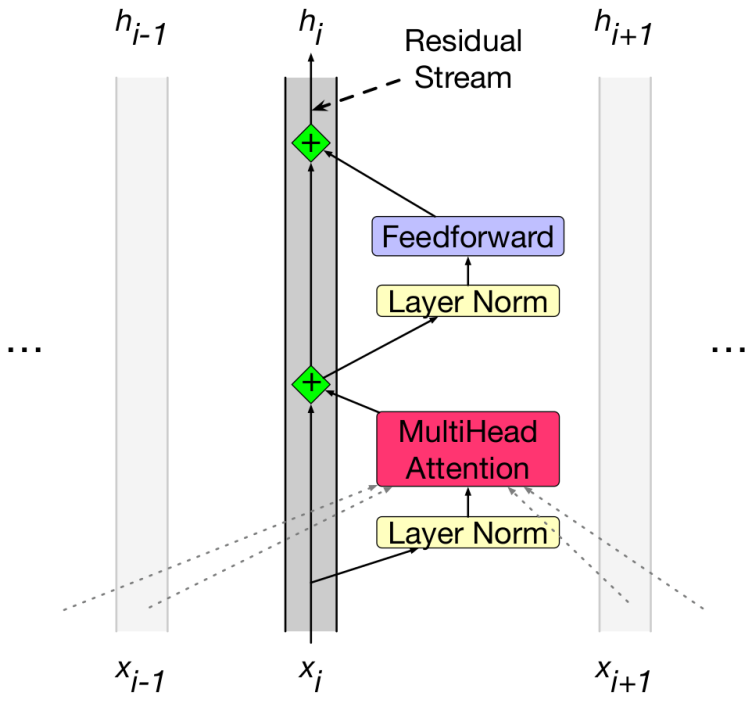

Figura 3.16: Architettura di un blocco Transformer nella variante *prenorm* e interpretazione tramite *residual stream*: ciascun modulo legge dallo stream del token e aggiunge il proprio output allo stesso stream tramite una somma residua [4].

Input del blocco e informazione di posizione

Sia $w_1, \ldots, w_N$ una sequenza di token. All'ingresso di un blocco Transformer, ogni posizione i è rappresentata da un vettore $x_i \in \mathbb{R}^d$. In generale, questo vettore non codifica soltanto l'identità lessicale del token, ma anche *la sua posizione* nella sequenza:

$$x_i = e(w_i) + p_i. \tag{3.39}$$

Qui $e(w_i)$ è l'embedding del token, mentre p_i è l'**embedding posizionale**. Quest'ultimo è necessario perché la self-attention, considerata da sola, è (in prima approssimazione) *invariante alle permutazioni*: senza un segnale di posizione, il modello non distinguerebbe una sequenza da una sua riorganizzazione. Gli embeddings posizionali (sinusoidali o apprendibili) iniettano quindi l'informazione d'ordine, rendendo possibile modellare fenomeni dipendenti dalla struttura sequenziale (ad esempio dipendenze sintattiche) [4].

Residual stream: il flusso informativo del token

Jurafsky & Martin propongono una metafora utile per leggere il blocco Transformer: il **residual stream** [4]. L'idea è considerare, per ciascun token i, un unico flusso di vettori in $\mathbb{R}^d$ che evolve attraversando i sottolivelli del blocco. Ogni sottolivello (attenzione o feedforward) non *sostituisce* la rappresentazione, ma calcola un aggiornamento che viene *aggiunto* allo stream tramite una connessione residua. Questo meccanismo ha due effetti importanti:

- preserva informazione utile anche quando il modello è molto profondo (il segnale può "scorrere" lungo le somme residue);
- facilita l'ottimizzazione, perché ogni sottolivello apprende una *correzione* (residuo) rispetto a una rappresentazione già presente.

Perché la LayerNorm (motivazione)

Nel Transformer, le somme residue e l'impilamento di molti blocchi possono far crescere o oscillare l'ampiezza delle attivazioni. La **layer normalization** serve a rendere l'addestramento più stabile mantenendo i valori del vettore in un range numericamente ben condizionato per la discesa del gradiente [4]. È importante notare che la layer norm è applicata a *un singolo token alla volta* (cioè a un vettore in $\mathbb{R}^d$), non a un intero batch o a tutta la sequenza.

Dato $x \in \mathbb{R}^d$, si calcolano media e deviazione standard sulle sue componenti:

$$\mu = \frac{1}{d}\sum_{k=1}^{d} x_k, \qquad \sigma = \sqrt{\frac{1}{d}\sum_{k=1}^{d}(x_k - \mu)^2}, \tag{3.40}$$

e poi si normalizza introducendo due parametri apprendibili di scala e traslazione:

$$\mathrm{LayerNorm}(x) = \gamma\frac{x-\mu}{\sigma} + \beta. \tag{3.41}$$

Feedforward network (ruolo)

Accanto all'attenzione, ogni blocco include una rete **feedforward** applicata indipendentemente a ciascun token. Mentre l'attenzione miscela informazione *tra posizioni*, la FFN agisce come una trasformazione non lineare *locale* che aumenta la capacità espressiva del modello sul singolo token. La forma standard è una rete a due strati:

$$\mathrm{FFN}(x) = \mathrm{ReLU}(xW_1 + b_1)\,W_2 + b_2, \tag{3.42}$$

con $W_1 \in \mathbb{R}^{d\times d_{ff}}$ e $W_2 \in \mathbb{R}^{d_{ff}\times d}$, e tipicamente $d_{ff} \gg d$ (ad esempio $d = 512$, $d_{ff} = 2048$ nel Transformer originale) [4].

Equazioni del blocco (variante *prenorm*)

Nella variante **prenorm** (Figura 3.16), la normalizzazione precede attenzione e feedforward. Per chiarire la notazione, introduciamo una sequenza di vettori intermedi $t_i^{(1)}, \ldots, t_i^{(5)} \in \mathbb{R}^d$ che rappresentano lo *stato del residual stream* del token i nei diversi passaggi interni al blocco. In altre parole, $t_i^{(k)}$ è semplicemente "ciò che scorre" nello stream dopo il k-esimo sotto-passaggio del blocco [4].

Il blocco opera come segue:

$$t_i^{(1)} = \mathrm{LayerNorm}(x_i), \tag{3.43}$$

$$t_i^{(2)} = \mathrm{MultiHeadAttention}\big(t_i^{(1)},\, t_1^{(1)}, \ldots, t_N^{(1)}\big), \tag{3.44}$$

$$t_i^{(3)} = x_i + t_i^{(2)}, \tag{3.45}$$

$$t_i^{(4)} = \mathrm{LayerNorm}(t_i^{(3)}), \tag{3.46}$$

$$t_i^{(5)} = \mathrm{FFN}(t_i^{(4)}), \tag{3.47}$$

$$h_i = t_i^{(3)} + t_i^{(5)}. \tag{3.48}$$

L'output $h_i \in \mathbb{R}^d$ ha la stessa dimensionalità dell'input x_i, il che rende possibile impilare molti blocchi consecutivi senza cambiare forma ai vettori.

L'attenzione come "movimento" di informazione tra stream

Un fatto cruciale è che, all'interno del blocco, **solo la multi-head attention** accede a informazione proveniente da altre posizioni della sequenza: FFN e LayerNorm operano localmente sul token i [4]. Per questo motivo, nella metafora del residual stream, una testa di attenzione può essere interpretata come un meccanismo che *trasferisce* informazione dallo stream di un token A a quello di un token B, aggiornando la rappresentazione di B con contenuto selezionato dal contesto.

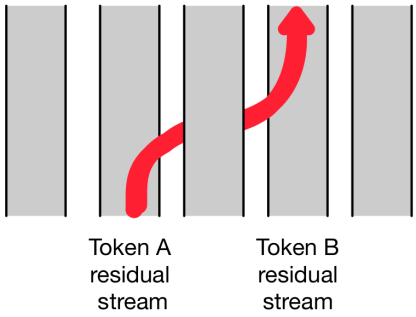

Figura 3.17: Interpretazione della self-attention: una testa può spostare informazione dal residual stream del token A a quello del token B, integrando in B contenuto recuperato da altre posizioni [4].

3.9.3 Parallelizzazione del calcolo con una singola matrice X

Nelle sezioni precedenti abbiamo descritto la self-attention e il blocco Transformer *dal punto di vista di un singolo token* in posizione i, seguendo l'idea del residual stream. Questa prospettiva è utile per capire *che cosa* fa il modello, ma nasconde un fatto computazionale cruciale: il calcolo della rappresentazione di ciascun token (attenzione, layer norm e feedforward) è *indipendente* da quello degli altri token, una volta fissata la finestra di contesto. Per questo motivo l'intera computazione può essere eseguita in parallelo con operazioni di algebra lineare, sfruttando routine ottimizzate di moltiplicazione tra matrici [4].

Impacchettare la sequenza in una matrice

Sia N la lunghezza della sequenza (window di contesto) e d la dimensione del modello. Raccogliamo i vettori di input $x_i \in \mathbb{R}^d$ in un'unica matrice:

$$X \in \mathbb{R}^{N \times d}, \qquad X = \begin{bmatrix} x_1^\top \\ \vdots \\ x_N^\top \end{bmatrix}.$$

Ogni riga di X rappresenta quindi un token (token embedding + positional embedding, come discusso nella sezione precedente), e lavorare in questa forma consente di calcolare in parallelo tutte le rappresentazioni contestuali prodotte da un attention head o da un intero blocco Transformer.

Self-attention in forma matriciale (una testa)

Consideriamo una singola testa di attenzione. Invece di calcolare q_i, k_i, v_i per ogni token separatamente, proiettiamo l'intera matrice X in tre matrici:

$$Q = XW^Q, \qquad K = XW^K, \qquad V = XW^V, \tag{3.49}$$

dove $W^Q, W^K \in \mathbb{R}^{d \times d_k}$ e $W^V \in \mathbb{R}^{d \times d_v}$. Ne segue che $Q, K \in \mathbb{R}^{N \times d_k}$ e $V \in \mathbb{R}^{N \times d_v}$.

Il passo chiave è osservare che *tutti* i confronti query–key $q_i \cdot k_j$ possono essere ottenuti in un'unica moltiplicazione:

$$S = QK^\top \in \mathbb{R}^{N \times N},$$

dove la cella S_{ij} contiene esattamente $q_i \cdot k_j$ (Figura 3.18).

Una volta ottenuti gli score, si applica lo scaling $1/\sqrt{d_k}$, la softmax riga-per-riga e si combina il risultato con V:

$$\text{head} = \text{softmax}\Big(\text{mask}\big(\tfrac{QK^\top}{\sqrt{d_k}}\big)\Big)\, V, \tag{3.50}$$

ottenendo una matrice head $\in \mathbb{R}^{N \times d_v}$: una rappresentazione vettoriale per ciascun token della sequenza, calcolata in parallelo [4].

Mascheramento causale: eliminare il futuro

La formulazione $QK^\top$ produce score tra ogni query q_i e ogni key k_j, inclusi i casi $j > i$. In un language model autoregressivo questo è inammissibile:

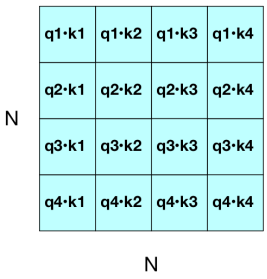

Figura 3.18: La matrice $N \times N$ $QK^\top$: ogni cella contiene un confronto $q_i \cdot k_j$, calcolato simultaneamente con una singola moltiplicazione tra matrici [4].

per predire il token successivo, il modello non deve accedere a token futuri. Per imporre il vincolo causale si applica una maschera triangolare superiore, impostando gli elementi con $j > i$ a $-\infty$; la softmax li trasforma quindi in probabilità nulle [4]:

$$S' = S + M, \quad \text{con} \quad M_{ij} = \begin{cases} 0 & \text{se } j \leq i, \\ -\infty & \text{se } j > i. \end{cases}$$

Il risultato è illustrato in Figura 3.19.

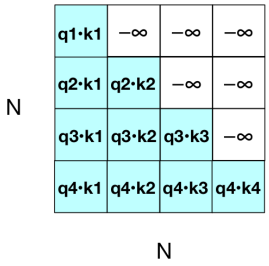

Figura 3.19: Mascheramento causale di $QK^\top$: la parte triangolare superiore (con $j > i$) è impostata a $-\infty$, così la softmax annulla i contributi dei token futuri [4].

Schema completo per una testa (in parallelo)

La Figura 3.20 riassume l'intero flusso computazionale in forma matriciale: dalla costruzione di Q, K, V, al prodotto $QK^\top$, al mascheramento e alla combinazione pesata con V.

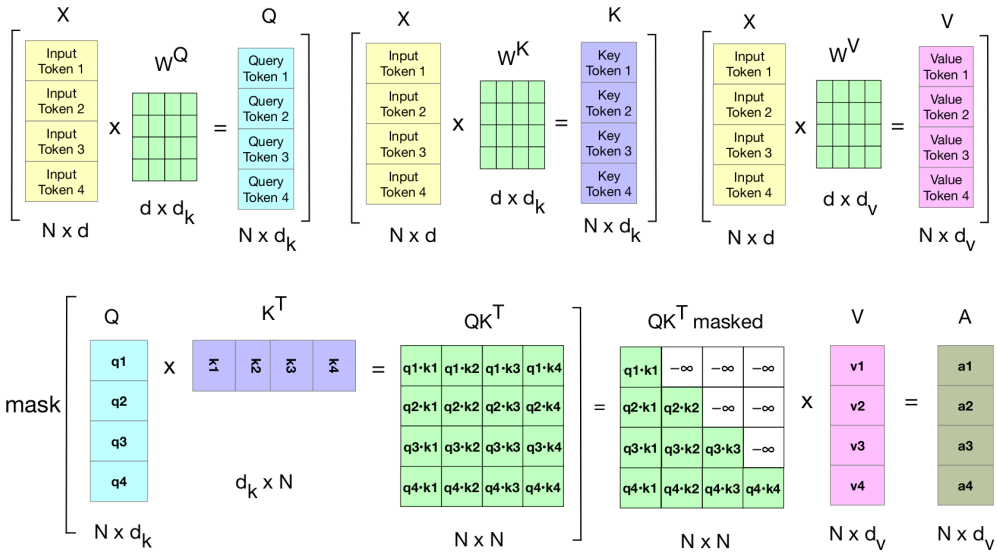

Figura 3.20: Computazione parallela di una singola testa di self-attention: (i) proiezioni Q, K, V da X; (ii) calcolo di $QK^\top$; (iii) maschera causale; (iv) (softmax non mostrata nello schema); (v) somma pesata tramite V per ottenere l'output della testa [4].

Costo computazionale e dipendenza quadratica

Le Figure 3.18 e 3.19 rendono evidente un limite strutturale: ad ogni strato il modello deve calcolare confronti tra *ogni coppia* di token nel contesto, producendo una matrice $N \times N$. Ne segue che tempo e memoria della self-attention standard crescono tipicamente come $\mathcal{O}(N^2)$ rispetto alla lunghezza del contesto [4].

Multi-head attention in parallelo

Nel caso multi-head, per ciascuna testa $c = 1, \ldots, A$ si usano matrici distinte W_c^Q, W_c^K, W_c^V, ottenendo:

$$Q_c = XW_c^Q, \qquad K_c = XW_c^K, \qquad V_c = XW_c^V, \tag{3.51}$$

e la testa:

$$\text{head}_c = \text{SelfAttention}(Q_c, K_c, V_c).$$

Le A uscite $\text{head}_1, \ldots, \text{head}_A \in \mathbb{R}^{N \times d_v}$ vengono concatenate lungo la dimensione delle feature e poi proiettate per ritornare alla dimensione del

modello:

$$\text{MultiHeadAttention}(X) = \big(\text{head}_1 \oplus \cdots \oplus \text{head}_A\big)W^O, \tag{3.52}$$

dove $W^O \in \mathbb{R}^{(Ad_v)\times d}$ e l'output complessivo ha forma $N \times d$ [4].

Il blocco Transformer in forma parallela

Infine, anche l'intero blocco Transformer può essere scritto in modo compatto usando matrici. Nella variante *prenorm* [4]:

$$O = X + \text{MultiHeadAttention}(\text{LayerNorm}(X)), \tag{3.53}$$

$$H = O + \text{FFN}(\text{LayerNorm}(O)). \tag{3.54}$$

Qui X è l'input del blocco (per il primo strato è la matrice di embedding iniziale, per gli strati successivi è l'output H dello strato precedente), mentre $H \in \mathbb{R}^{N\times d}$ è l'output del blocco. Notazione come $\text{FFN}(O)$ indica che *la stessa* rete feedforward viene applicata in parallelo a ciascuna riga (cioè a ciascun token) della matrice [4].

3.9.4 L'input del Transformer: embeddings di token e di posizione

Nella sezione precedente abbiamo visto come l'intera computazione di self-attention possa essere espressa e parallelizzata tramite una singola matrice di input $X \in \mathbb{R}^{N\times d}$, dove N è la lunghezza del contesto e d la dimensione del modello. Resta ora da chiarire **da dove provenga** questa matrice X, ovvero come una sequenza discreta di token venga trasformata in una sequenza di vettori densi adatti a essere processati dai blocchi Transformer [4].

L'idea fondamentale è che l'input venga ottenuto combinando due sorgenti di informazione:

1. un **embedding lessicale** (token embedding), che codifica l'identità del token;
2. un **embedding posizionale** (positional embedding), che codifica la posizione del token nella sequenza.

La loro somma produce una rappresentazione composita che costituisce l'input effettivo del Transformer.

Token embeddings e matrice di embedding

Sia V il vocabolario dei token (tipicamente ottenuto tramite BPE o SentencePiece). Una volta tokenizzato l'input, ogni token viene rappresentato da un indice intero $w_i \in \{1, \ldots, |V|\}$. Gli embeddings iniziali dei token sono memorizzati in una matrice:

$$E \in \mathbb{R}^{|V| \times d},$$

dove la riga $E[w_i]$ (di dimensione d) rappresenta l'embedding del token con indice w_i [4]. Dunque, dato un token in posizione i, il suo embedding lessicale è:

$$e(w_i) = E[w_i] \in \mathbb{R}^d.$$

Selezione via one-hot (interpretazione equivalente). Un modo alternativo (concettuale) per vedere la selezione dell'embedding consiste nel rappresentare il token con un vettore one-hot $\mathbf{o}(w_i) \in \{0,1\}^{|V|}$, con un unico 1 nella posizione w_i. Moltiplicando $\mathbf{o}(w_i)$ per la matrice E si "estrae" la riga corrispondente:

$$\mathbf{o}(w_i)^\top E = E[w_i].$$

La Figura 3.21 illustra graficamente questa interpretazione [4].

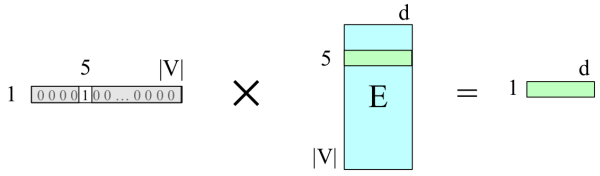

Figura 3.21: Selezione dell'embedding del token w_i a partire dalla matrice E mediante un vettore one-hot: il prodotto $\mathbf{o}(w_i)^\top E$ restituisce la riga $E[w_i]$ [4].

Dalla sequenza alla matrice. Estendendo l'idea, l'intera sequenza di N token può essere rappresentata come una matrice di one-hot $O \in \{0,1\}^{N \times |V|}$ (una riga per posizione). Il prodotto:

$$OE \in \mathbb{R}^{N \times d}$$

restituisce una matrice che contiene, riga per riga, gli embeddings lessicali della sequenza (Figura 3.22) [4]. Nella pratica si usa indicizzazione diretta sulle righe di E (più efficiente), ma l'interpretazione one-hot chiarisce perché l'operazione sia lineare e parallelizzabile.

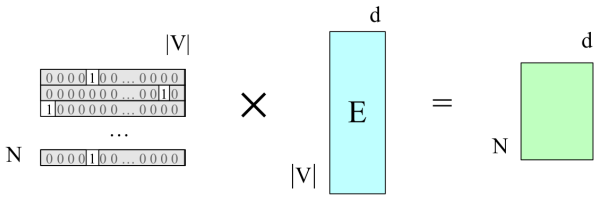

Figura 3.22: Selezione degli embeddings per un'intera sequenza: una matrice di one-hot O moltiplicata per E produce una matrice $N \times d$ contenente gli embeddings lessicali della finestra di contesto [4].

Perché servono gli embeddings posizionali

Gli embeddings di token, da soli, *non* codificano la posizione: il token *bill* ha lo stesso vettore sia che compaia all'inizio sia che compaia alla fine della frase. Tuttavia, nei modelli di linguaggio l'ordine è essenziale: molte regolarità sintattiche e semantiche dipendono dalla struttura sequenziale. Inoltre, la self-attention confronta token tramite similarità tra query e key; senza un segnale di posizione, il modello avrebbe difficoltà a distinguere configurazioni che differiscono solo per permutazione [4]. Per questo motivo, al token embedding si affianca un embedding di posizione.

Posizione assoluta e composizione dell'input

La soluzione più semplice è usare **posizione assoluta**: si associa a ogni posizione i un vettore $p_i \in \mathbb{R}^d$, apprendibile durante l'addestramento, raccolto in una matrice:

$$E_{\text{pos}} \in \mathbb{R}^{N \times d}, \qquad p_i = E_{\text{pos}}[i].$$

L'input finale per il token in posizione i è la somma:

$$x_i = e(w_i) + p_i, \tag{3.55}$$

e impilando tutti i vettori si ottiene:

$$X = \begin{bmatrix} x_1^\top \\ \vdots \\ x_N^\top \end{bmatrix} \in \mathbb{R}^{N \times d}.$$

In altre parole, X **è una matrice di embeddings compositi** (token + posizione), che costituisce l'input al primo blocco Transformer [4]. La Figura 3.23 visualizza l'operazione: a ogni token embedding viene sommato il

corrispondente embedding di posizione, producendo un vettore della stessa dimensionalità.

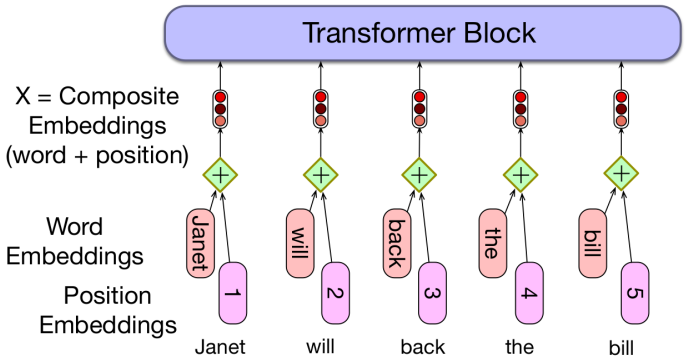

Figura 3.23: Composizione dell'input: l'embedding del token viene sommato all'embedding della posizione assoluta, producendo un vettore di input x_i in $\mathbb{R}^d$. L'insieme dei vettori forma la matrice $X \in \mathbb{R}^{N \times d}$ [4].

Input del Transformer
La matrice $X \in \mathbb{R}^{N \times d}$ contiene una riga per token, ed è costruita come $x_i = E[w_i] + E_{\text{pos}}[i]$. I blocchi Transformer trasformano poi X in rappresentazioni contestuali sempre più ricche, mantenendo costante la dimensionalità d per poter essere impilati.

3.9.5 Limiti della posizione assoluta e alternative: sinusoidale e posizione relativa

L'approccio a posizione assoluta è concettualmente semplice e funziona bene, ma presenta un potenziale limite: le posizioni più "lontane" (verso la massima lunghezza N) possono essere viste meno spesso durante l'addestramento, e quindi i relativi embeddings p_i rischiano di essere appresi in modo meno robusto e di generalizzare peggio [4]. Per mitigare questo problema, sono state proposte due famiglie di alternative.

Embeddings posizionali sinusoidali. Una prima alternativa consiste nel sostituire i vettori posizionali appresi con una funzione deterministica che mappa l'indice di posizione i in un vettore reale, ad esempio usando combinazioni di seni e coseni a frequenze diverse (come nel Transformer originale).

Questa scelta ha due vantaggi: (i) non richiede parametri addizionali per ogni posizione, e (ii) introduce regolarità geometriche che riflettono relazioni tra posizioni, rendendo più naturale rappresentare che posizioni vicine (es. i e $i+1$) siano più correlate di posizioni molto distanti [4].

Posizione relativa. Una seconda linea di lavoro, più espressiva, mira a modellare non la posizione assoluta di un token, ma la sua **posizione relativa** rispetto agli altri token. In questo caso, l'informazione posizionale viene spesso incorporata direttamente nel meccanismo di attenzione a ogni strato, in modo che i pesi di attenzione possano dipendere esplicitamente dalla distanza (o dall'offset) tra le posizioni [4]. Questa idea è particolarmente utile quando si vogliono gestire sequenze molto lunghe e favorire la generalizzazione oltre le lunghezze viste in training.

Queste tecniche completano la definizione dell'input del Transformer: una volta costruita la matrice X, il modello può applicare in parallelo i blocchi Transformer e, alla fine, utilizzare le rappresentazioni prodotte per il compito di language modeling tramite una proiezione sul vocabolario (testa di output), che introduciamo nella prossima sezione.

3.9.6 La *language modeling head*

Nelle sezioni 9.2–9.4 abbiamo costruito tutti gli ingredienti necessari per un Transformer causale: (i) un input $X \in \mathbb{R}^{N \times d}$ ottenuto da token+posizione, e (ii) una pila di blocchi Transformer che trasforma tali vettori in rappresentazioni contestuali sempre più ricche. A questo punto resta un ultimo passaggio fondamentale per ottenere un **modello di linguaggio**: dobbiamo trasformare l'output del Transformer in una **distribuzione di probabilità sul vocabolario**, cioè in una stima di

$$P(w_{N+1} \mid w_{1:N}).$$

Questo è esattamente il ruolo della *language modeling head*, ovvero il circuito "in cima" al Transformer che prende una rappresentazione vettoriale e la converte in probabilità sui token possibili [4].

Cosa entra e cosa esce: dal vettore h_N^L alle probabilità sul vocabolario

Indichiamo con $h_i^L \in \mathbb{R}^d$ l'output (rappresentazione contestuale) del token in posizione i dopo l'ultimo strato L della pila di Transformer blocks. Nel caso autoregressivo (causale), l'informazione rilevante per predire il prossimo token è contenuta nel vettore dell'ultima posizione disponibile, cioè h_N^L. La language modeling head implementa allora una mappa:

$$h_N^L \in \mathbb{R}^d \quad \longrightarrow \quad y \in \mathbb{R}^{|V|}$$

dove y_k rappresenta la probabilità assegnata al token k del vocabolario.

La Figura 3.24 visualizza questo processo: dall'ultimo stato h_N^L si producono dei punteggi (*logits*) per tutti i token, e poi una softmax li normalizza in una distribuzione di probabilità [4].

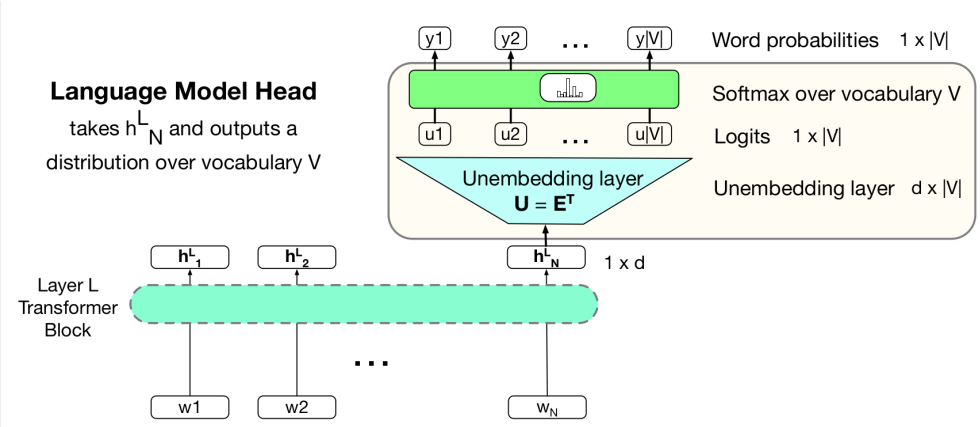

Figura 3.24: La *language modeling head*: mappa l'output dell'ultimo token all'ultimo strato (h_N^L) in una distribuzione sul vocabolario tramite (i) un livello lineare (*unembedding*) e (ii) una softmax [4].

Logits e livello di *unembedding*

Il primo modulo della head è un **livello lineare** che proietta il vettore h_N^L (dimensione d) in un vettore di punteggi $u \in \mathbb{R}^{|V|}$, uno per ogni token del vocabolario. Questi punteggi si chiamano **logits**: non sono probabilità, ma valori reali che misurano quanto il modello "preferisce" ciascun token come prossimo elemento.

Formalmente, esiste una matrice di proiezione $U \in \mathbb{R}^{d \times |V|}$ tale che:

$$u = h_N^L U, \qquad u \in \mathbb{R}^{|V|}. \tag{3.56}$$

Weight tying: perché spesso $U = E^\top$. In molti Transformer per language modeling, invece di apprendere U come matrice indipendente, si usa **weight tying**: si impone che U coincida con la trasposta della matrice di embedding dei token usata in input, cioè $E \in \mathbb{R}^{|V| \times d}$ vista nella sezione precedente. In questo caso:

$$U = E^\top \in \mathbb{R}^{d \times |V|}.$$

L'interpretazione è elegante: all'inizio il modello usa E per passare da un id discreto a un vettore denso (*embedding*); alla fine usa $E^\top$ per "tornare indietro" da un vettore denso a punteggi sul vocabolario (*unembedding*). Durante l'addestramento, la stessa matrice E viene quindi ottimizzata per essere utile in entrambe le direzioni [4]. Con weight tying:

$$u = h_N^L E^\top. \tag{3.57}$$

Softmax: da logits a probabilità

Per ottenere una distribuzione di probabilità, i logits vengono normalizzati con una softmax:

$$y = \text{softmax}(u), \qquad y_k = \frac{e^{u_k}}{\sum_{t=1}^{|V|} e^{u_t}}. \tag{3.58}$$

La softmax ha due effetti essenziali: (i) rende tutti i valori non-negativi e (ii) li normalizza a sommare a 1, così da poter interpretare y come $P(\cdot \mid w_{1:N})$ [4].

Riassunto operativo della head
Dato l'ultimo vettore contestuale h_N^L, la language modeling head calcola i logits $u = h_N^L E^\top$ (o $u = h_N^L U$ in generale) e poi le probabilità $y = \text{softmax}(u)$. Il token successivo w_{N+1} viene scelto campionando da y (o con decodifica greedy/beam, ecc.).

Dal modello alla generazione: scegliere il prossimo token

Una volta ottenuto y, il modello può:

- scegliere il token a massima probabilità (*greedy decoding*);
- campionare secondo y (con varianti come temperature, top-k, nucleus sampling, ecc.).

Qualunque strategia si adotti, il token selezionato diventa w_{N+1} e viene aggiunto al contesto; il processo si ripete in modo autoregressivo.

Visione d'insieme: un *decoder-only* che impila blocchi

La Figura 3.25 mostra l'intera pipeline per una posizione i: input encoding $\rightarrow$ pila di blocchi Transformer $\rightarrow$ language modeling head $\rightarrow$ predizione del token successivo [4].

3.9.7 Nota: *logit lens* e terminologia *decoder-only*

Una conseguenza utile dell'aver separato nettamente "corpo" del Transformer e "testa" di output è che il livello di unembedding $E^\top$ può essere riutilizzato come strumento di interpretabilità. In particolare, si può prendere un vettore interno h_i^ℓ a uno strato intermedio ℓ e applicare la stessa proiezione verso il vocabolario:

$$u^{(\ell)} = h_i^\ell E^\top, \qquad y^{(\ell)} = \mathrm{softmax}(u^{(\ell)}).$$

In questo modo si ottiene una distribuzione "come se" quel vettore fosse già pronto a predire un token. Questa tecnica è nota come **logit lens**: non è garantito che funzioni perfettamente (il modello non è addestrato esplicitamente per rendere ogni strato direttamente decodificabile), ma spesso fornisce una finestra qualitativa su quali ipotesi stiano emergendo negli strati interni [4].

Nota terminologica: *decoder-only*. Un Transformer causale usato per language modeling viene spesso chiamato **decoder-only model**. Il motivo è storico e architetturale: nell'encoder–decoder Transformer (usato, ad esempio, in traduzione), il *decoder* è proprio la parte che genera token in modo autoregressivo con maschera causale. I language model moderni (stile GPT) corrispondono, in sostanza, a questa sola metà generativa [4].

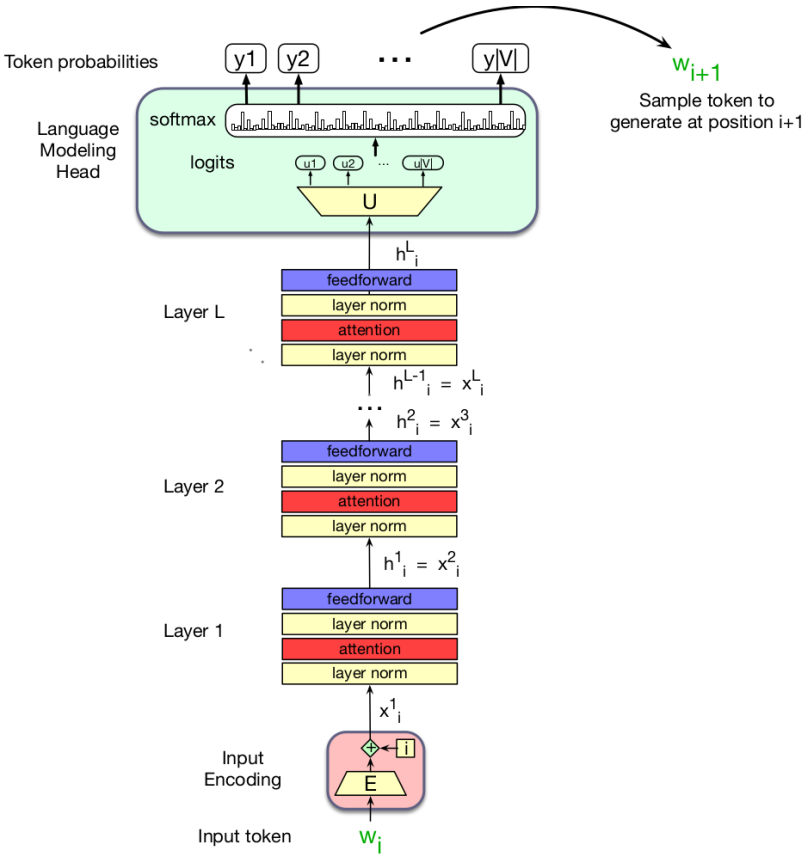

Figura 3.25: Un Transformer per language modeling (*decoder-only*): impila blocchi Transformer e usa la language modeling head per mappare h_i^L in una distribuzione sul prossimo token w_{i+1} [4].

3.10 Large Language Models

Nelle sezioni precedenti abbiamo visto come il Transformer (in particolare nella variante *decoder-only*) costruisca rappresentazioni contestuali tramite self-attention causale e, attraverso la *language modeling head*, produca una distribuzione di probabilità sul token successivo. A questo punto il passo concettuale naturale è chiedersi cosa succede quando questa architettura viene addestrata (*pretrained*) su quantità enormi di testo: nasce così la famiglia dei **Large Language Models** (LLM).

L'idea di fondo si collega direttamente all'ipotesi distribuzionale discussa all'inizio del capitolo: se gran parte della conoscenza linguistica (e una quota non banale di conoscenza sul mondo) è contenuta nelle regolarità statistiche del testo, allora un modello addestrato a predire i token successivi su corpora massivi può assorbire tali regolarità e riutilizzarle in molti compiti diversi. In questa prospettiva, la conoscenza non viene "inserita" manualmente, ma emerge come *sottoprodotto* dell'addestramento auto-supervisionato.

Pretraining (auto-supervision)
Un LLM viene addestrato a predire il token successivo in moltissimi contesti diversi. Il testo stesso fornisce l'etichetta: dato un prefisso, il token successivo osservato nel corpus è il target.

Questa capacità di predizione, quando combinata con finestre di contesto molto ampie, rende gli LLM particolarmente efficaci nei compiti in cui serve **generare testo** (riassunti, traduzione, risposta a domande, assistenti conversazionali, ...). Nel seguito formalizziamo come si usa un Transformer come LLM e perché molti task di NLP possono essere reinterpretati come *word prediction*.

3.10.1 Large Language Models con Transformer: generazione condizionata

Un Transformer *decoder-only* implementa un modello autoregressivo: legge un prefisso di token e stima la distribuzione del token successivo. In pratica, l'uso tipico avviene in forma di **generazione condizionata**: forniamo al modello un testo di input (spesso chiamato *prompt*) e chiediamo di continuare a generare token uno alla volta, condizionandosi sia sul prompt sia sui token già generati.

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i \mid w_{<i}), \tag{3.59}$$

dove, grazie alla *causal mask*, in posizione i il modello può usare soltanto informazione proveniente da posizioni $< i$ (niente "futuro" nella sequenza). Questo è esattamente il vincolo che rende coerente la predizione del token successivo.

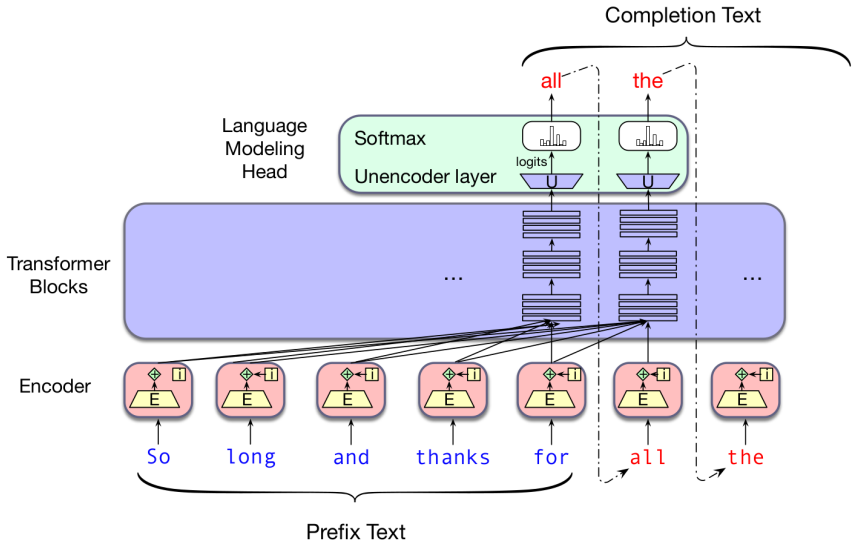

Figura 3.26: Completamento autoregressivo left-to-right: i token generati vengono riaggiunti al contesto e diventano parte del prefisso per la predizione successiva.

Perché "predire parole" è utile per tanti task. La chiave è che moltissimi problemi di NLP possono essere riformulati come scelta del token (o della sequenza di token) più probabile dato un contesto. Ad esempio:

- **Classificazione via prompting:** possiamo chiedere al modello di completare una frase istruttiva e confrontare la probabilità di token alternativi (es. *positivo* vs *negativo*).
- **Question answering:** forniamo una domanda e un indicatore del tipo `A:` e lasciamo che il modello generi l'inizio (e poi il resto) della risposta, massimizzando coerenza col contesto.

- **Compiti generativi lunghi:** come riassunto o traduzione, in cui la risposta richiede più token, non uno solo.

Esempio: riassunto come generazione condizionata. Nel riassunto, l'input è un testo lungo e l'output è una sua versione più breve. Un modo pratico per "dire" al modello cosa vogliamo è aggiungere un delimitatore o un cue testuale (per esempio `tl;dr`), che segnala l'inizio della sintesi. Il modello, avendo visto in addestramento molte occorrenze simili, impara ad associare quel pattern alla produzione di un riassunto.

Original Article

The only thing crazier than a guy in snowbound Massachusetts boxing up the powdery white stuff and offering it for sale online? People are actually buying it. For $89, self-styled entrepreneur Kyle Waring will ship you 6 pounds of Boston-area snow in an insulated Styrofoam box – enough for 10 to 15 snowballs, he says.

But not if you live in New England or surrounding states. "We will not ship snow to any states in the northeast!" says Waring's website, ShipSnowYo.com. "We're in the business of expunging snow!"

His website and social media accounts claim to have filled more than 133 orders for snow – more than 30 on Tuesday alone, his busiest day yet. With more than 45 total inches, Boston has set a record this winter for the snowiest month in its history. Most residents see the huge piles of snow choking their yards and sidewalks as a nuisance, but Waring saw an opportunity.

According to Boston.com, it all started a few weeks ago, when Waring and his wife were shoveling deep snow from their yard in Manchester-by-the-Sea, a coastal suburb north of Boston. He joked about shipping the stuff to friends and family in warmer states, and an idea was born. [...]

Summary

Kyle Waring will ship you 6 pounds of Boston-area snow in an insulated Styrofoam box – enough for 10 to 15 snowballs, he says. But not if you live in New England or surrounding states.

Figura 3.27: Esempio di testo e relativo riassunto (scenario tipico di summarization). Il compito può essere visto come generazione condizionata sul documento.

Nota sul passo successivo (ponte verso BERT). L'architettura descritta fin qui è *unidirezionale*: per costruzione non incorpora informazione dai token futuri, perché è pensata per generare da sinistra a destra. Per arrivare a BERT, il passaggio narrativo naturale è:

> *se invece di generare testo vogliamo "capire" un testo completo, possiamo permetterci attenzione bidirezionale e un obiettivo di addestramento diverso (masked language modeling), ottenendo embeddings contestuali che vedono sia sinistra sia destra.*

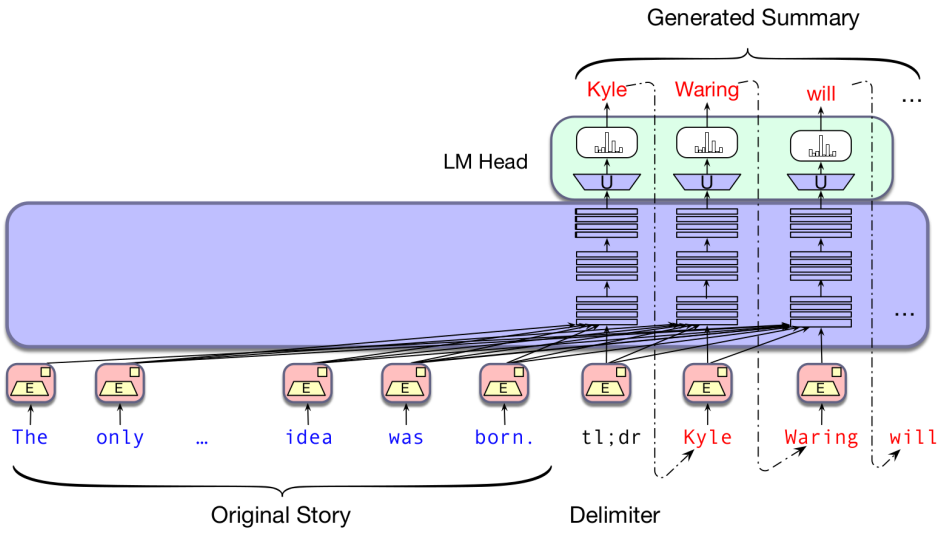

Figura 3.28: Riassunto tramite prompting: il token (o stringa) `tl;dr` agisce da segnale che innesca la generazione di una sintesi, sfruttando la lunga finestra di contesto del Transformer.

Questo è esattamente il cambio di paradigma che introdurrai quando passerai dai modelli causali (decoder-only) ai modelli encoder bidirezionali come BERT.

3.11 Sampling per la generazione con LLM

Nelle sezioni precedenti abbiamo visto come un transformer (decoder-only) produca, per ogni posizione t, un vettore di *logit* $u_t \in \mathbb{R}^{|V|}$ che, tramite softmax, diventa una distribuzione di probabilità y_t sul vocabolario V. A questo punto, per *generare* testo dobbiamo rispondere a una domanda operativa:

> dato il contesto (prompt + token già generati), *quale token scegliamo come prossimo?*

Il nome generico di questa scelta è **decoding**. Nel caso più comune per gli LLM, la generazione è **autoregressiva** (o **causale**): si genera da sinistra a destra, e ogni nuovo token viene aggiunto al contesto per predire il successivo. Formalmente, al passo t campioniamo

$$w_t \sim p(w_t \mid w_{<t}),$$

dove $p(\cdot)$ è la distribuzione predetta dal modello.

3.11.1 Perché non basta il campionamento "puro"

Un'idea naturale sarebbe: "campiono direttamente dalla softmax". In pseudocodice:

Random sampling (idea base)
$i \leftarrow 1$
campiona $w_i \sim p(w)$
while $w_i \neq$ EOS **do**
  $i \leftarrow i + 1$
  campiona $w_i \sim p(w_i \mid w_{<i})$
end while

Il problema è che una distribuzione su $|V|$ parole ha una "coda" lunga: moltissimi token rarissimi hanno probabilità individualmente piccole, ma *complessivamente* possono accumulare una massa non trascurabile. Questo fa sì che, abbastanza spesso, il modello selezioni token poco appropriati, producendo frasi strane o incoerenti.

Per questo, in pratica, si usano strategie che *limitano* o *rimodellano* la distribuzione, bilanciando due obiettivi:

- **Qualità/coerenza** (testo più corretto e stabile, ma rischia di essere ripetitivo),
- **Diversità/creatività** (testo meno banale, ma più incline a errori o incoerenze).

3.11.2 Top-k sampling

Nel **top-k sampling** non si considera l'intero vocabolario: si tengono solo i k token più probabili nel contesto, si rinormalizza e poi si campiona tra questi.

Sia $V_k \subseteq V$ l'insieme dei k token con probabilità più alta secondo $p(\cdot \mid w_{<t})$. Definiamo la distribuzione troncata:

$$\tilde{p}(w \mid w_{<t}) = \begin{cases} \dfrac{p(w \mid w_{<t})}{\sum_{w' \in V_k} p(w' \mid w_{<t})} & \text{se } w \in V_k \\ 0 & \text{altrimenti.} \end{cases}$$

Quindi si campiona $w_t \sim \tilde{p}(\cdot \mid w_{<t})$.

Osservazione importante: con $k = 1$ si ottiene la scelta deterministica del token più probabile (greedy decoding). Aumentando k, cresce la varietà delle scelte possibili, ma anche il rischio di selezionare token meno adatti.

3.11.3 Top-p (nucleus) sampling

Il limite del top-k è che k è fisso, ma la "forma" della distribuzione cambia da contesto a contesto: a volte poche parole coprono quasi tutta la massa di probabilità, altre volte la distribuzione è più piatta.

Nel **top-p sampling** (o **nucleus sampling**) non si fissa il numero di token, ma la *massa di probabilità* da conservare. Si ordina il vocabolario per probabilità decrescente e si prende il più piccolo insieme $V^{(p)}$ tale che:

$$\sum_{w \in V^{(p)}} p(w \mid w_{<t}) \;\geq\; p.$$

Poi si rinormalizza su $V^{(p)}$ e si campiona. L'effetto pratico è adattivo: quando il modello è molto sicuro, $V^{(p)}$ può essere piccolo; quando è incerto, l'insieme si allarga.

3.11.4 Temperature sampling

Un'altra idea non è "tagliare" la distribuzione, ma **rimodellarla** tramite la **temperatura**. Ricordando che

$$y_t = \text{softmax}(u_t),$$

si introduce un parametro $\tau > 0$ e si calcola:

$$y_t = \text{softmax}\left(\frac{u_t}{\tau}\right).$$

Intuizione:

- **Bassa temperatura** ($0 < \tau < 1$): la distribuzione si *concentra* sui token più probabili (più "greedy").
- **Alta temperatura** ($\tau > 1$): la distribuzione si *appiattisce*, aumentando la diversità.

Nella pratica, molti sistemi combinano temperature e un filtro (top-p o top-k), per evitare la coda lunga e, allo stesso tempo, controllare quanto la generazione sia conservativa o creativa.

3.12 Pretraining dei Large Language Models

Le sezioni precedenti hanno descritto *come* un large language model (LLM) genera testo: dato un contesto, produce una distribuzione di probabilità sul prossimo token e poi, tramite una strategia di decoding (greedy, sampling, ecc.), sceglie il token da emettere. In questa sezione chiudiamo il cerchio e descriviamo *come* si addestra un transformer affinché questa distribuzione sia utile: l'obiettivo del pretraining è rendere il modello un buon *predittore del prossimo token* su una grande varietà di testi, così da acquisire regolarità linguistiche e conoscenze statistiche riutilizzabili in molti compiti.

3.12.1 Setup, notazione e obiettivo di language modeling

Sia V il vocabolario (insieme dei token) e $|V|$ la sua cardinalità. Un testo viene rappresentato come sequenza di token

$$W = (w_1, w_2, \ldots, w_T), \qquad \text{con } w_t \in V,$$

dove T è la lunghezza della sequenza (numero di token). Un transformer decoder-only, con maschera causale, definisce un modello probabilistico autoregressivo:

$$p_\theta(W) \;=\; \prod_{t=1}^{T} p_\theta(w_t \mid w_{<t}),$$

dove $w_{<t} = (w_1, \ldots, w_{t-1})$ e θ indica l'insieme dei parametri del modello.

In pratica, al passo t il modello produce un vettore di *logit*

$$u_t \in \mathbb{R}^{|V|},$$

che viene normalizzato con una softmax per ottenere una distribuzione di probabilità sul vocabolario:

$$\hat{y}_t = \text{softmax}(u_t) \in \Delta^{|V|-1},$$

dove $\Delta^{|V|-1}$ denota il simplesso delle distribuzioni di probabilità su $|V|$ elementi. L'elemento $\hat{y}_t[v]$ è quindi la probabilità assegnata dal modello al token $v \in V$ come *prossimo token*.

3.12.2 Self-supervision e funzione obiettivo

Il pretraining degli LLM è tipicamente **self-supervised**: non servono etichette manuali, perché il testo stesso fornisce la supervisione. L'idea è semplice: dato un prefisso $w_{\leq t}$, il "target" naturale è il token successivo w_{t+1}.

Per formalizzare la loss, definiamo la distribuzione corretta (target) come one-hot sul token vero w_{t+1}:

$$y_t[v] = \begin{cases} 1 & \text{se } v = w_{t+1}, \\ 0 & \text{altrimenti.} \end{cases}$$

La cross-entropy tra target y_t e predizione $\hat{y}_t$ è:

$$L_{\text{CE}}(\hat{y}_t, y_t) \;=\; -\sum_{v \in V} y_t[v] \log \hat{y}_t[v].$$

Poiché y_t è one-hot, la somma collassa sul token corretto:

$$L_{\text{CE}}(\hat{y}_t, y_t) \;=\; -\log \hat{y}_t[w_{t+1}] \;=\; -\log p_\theta(w_{t+1} \mid w_{\leq t}).$$

Per una sequenza di lunghezza T, si minimizza la media (o somma) delle cross-entropy ai vari passi. Usando la media:

$$L(W;\theta) \;=\; \frac{1}{T-1} \sum_{t=1}^{T-1} -\log p_\theta(w_{t+1} \mid w_{\leq t}).$$

Minimizzare questa loss equivale a massimizzare la log-verosimiglianza (maximum likelihood) del corpus di training.

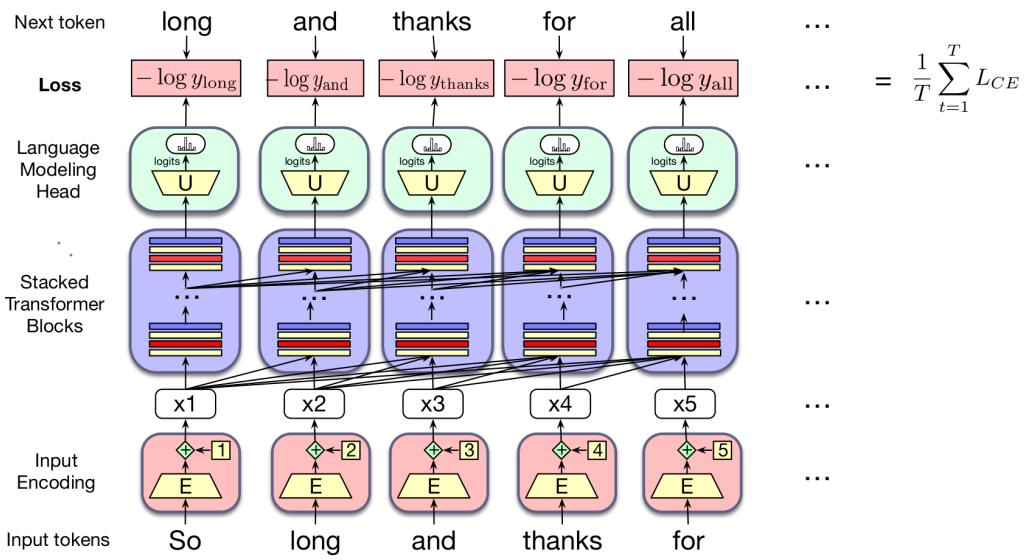

Figura 3.29: Addestramento di un transformer come language model: per ogni posizione si predice il token successivo e si calcola una loss di cross-entropy; la loss totale è la media (o somma) sulle posizioni.

3.12.3 Teacher forcing

Nel training autoregressivo si adotta quasi sempre la strategia detta **teacher forcing**. Anziché far avanzare la sequenza alimentando al modello i token che lui stesso ha predetto, durante l'addestramento si fornisce sempre il prefisso *corretto* dal dataset. In altre parole, per calcolare la loss al tempo t, il modello condiziona su $w_{\leq t}$ (vero), non su una storia generata $\tilde{w}_{\leq t}$.

Teacher Forcing
Nel teacher forcing il modello è addestrato a predire w_{t+1} avendo come contesto il prefisso vero $w_{\leq t}$. Durante la generazione, invece, il contesto include i token effettivamente generati dal modello. Questa differenza (training con storia vera vs inference con storia generata) è una delle ragioni per cui la qualità del decoding e delle strategie di campionamento è cruciale: in generazione gli errori possono propagarsi perché il modello si condiziona su output potenzialmente imperfetti.

Questo ha due conseguenze pratiche:

1. stabilizza l'ottimizzazione: la rete vede input realistici (prefissi reali),

quindi il gradiente è meno rumoroso;

2. permette il calcolo della loss su tutte le posizioni in modo efficiente (in particolare con i transformer), perché per ogni token della finestra si conosce già il target successivo.

3.12.4 Efficienza computazionale: parallelismo nei transformer

Nei modelli ricorrenti classici, il calcolo degli stati nascosti è intrinsecamente seriale: per ottenere lo stato al tempo t serve quello al tempo $t-1$. Nei transformer, invece, (con maschera causale) ogni posizione può essere processata in parallelo *all'interno di una stessa forward pass*, perché la dipendenza dal passato è implementata dalla self-attention mascherata: ogni token t può "leggere" solo posizioni $\leq t$, ma tutte le posizioni della finestra vengono comunque elaborate simultaneamente come matrici. In pratica, si costruiscono batch che contengono finestre di lunghezza T (spesso chiamata anche *context length* o *sequence length*) e si ottimizza la loss media sul batch mediante gradient descent (o varianti). Quando un documento è più corto della finestra, si possono concatenare più documenti separandoli con token speciali (ad es. fine-documento/testo), così da sfruttare al massimo la capacità computazionale.

3.12.5 Dati di pretraining: fonti e filtraggio

La qualità e la copertura del pretraining dipendono in larga misura dal corpus. Gli LLM moderni usano corpora enormi che combinano:

- testo dal web (crawl) in grandi quantità,
- sorgenti relativamente curate (enciclopedie, articoli, libri),
- talvolta altre tipologie (dialoghi, forum, materiale tecnico, ecc.).

L'intuizione è che, su scala molto grande, il modello incontri naturalmente una varietà di stili e contenuti sufficiente a imparare regolarità linguistiche generali e molti pattern utili (domande/risposte, definizioni, esempi, spiegazioni, ...).

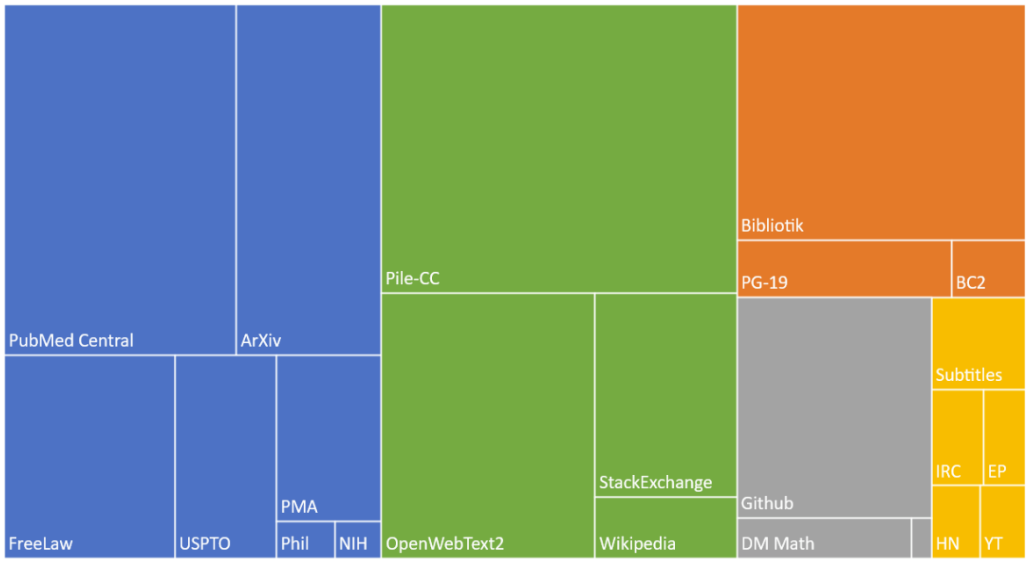

Figura 3.30: Esempio di composizione di un grande corpus di pretraining (treemap): diverse sorgenti contribuiscono con proporzioni differenti.

Filtri di qualità e sicurezza. Poiché il web contiene rumore, duplicazioni e contenuti problematici, i corpus vengono in genere *pre-processati*. Due famiglie di scelte sono particolarmente comuni:

- **Qualità**: deduplicazione (a livello documento o porzioni di testo), rimozione di boilerplate e pagine a bassa densità informativa, esclusione di porzioni non linguistiche quando indesiderate.
- **Sicurezza/privacy**: tentativi di ridurre contenuti tossici e la presenza di informazioni personali (PII).

Questi filtri migliorano spesso la resa del modello, ma introducono anche trade-off: criteri troppo aggressivi possono rimuovere esempi utili o introdurre bias; criteri troppo permissivi possono lasciare rumore e contenuti problematici.

Aspetti etici e legali (panoramica). L'uso di corpora web solleva questioni importanti (copyright, consenso dei siti, privacy). Qui basta notare che la costruzione del dataset non è un dettaglio secondario: è parte integrante del progetto di un LLM e influenza sia le capacità sia i rischi del modello.

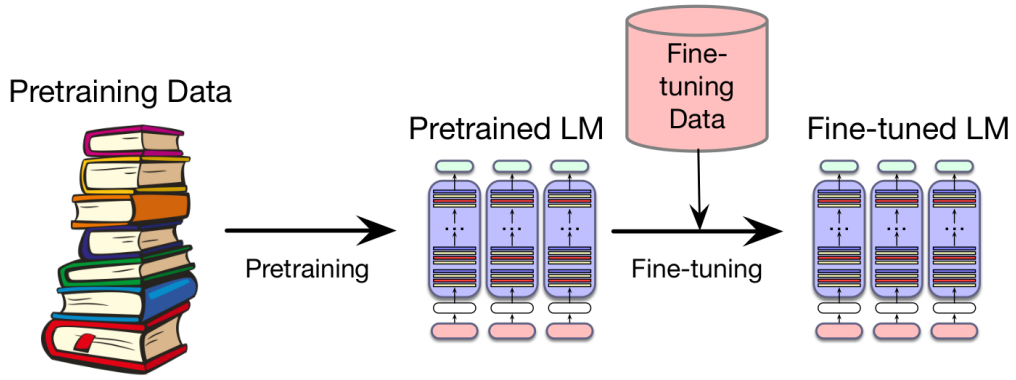

Figura 3.31: Schema concettuale: pretraining su grandi corpora → modello generale → adattamento su dati specifici (finetuning) → modello specializzato.

3.12.6 Dal pretraining all'adattamento: finetuning

Il pretraining produce un modello "generale", ma molte applicazioni richiedono adattamento: a un dominio (es. medico/legale), a uno stile, a una lingua specifica, o a un formato di risposta desiderato. In senso ampio, **finetuning** indica il processo di partire da un modello pre-addestrato e continuare l'ottimizzazione su dati aggiuntivi più mirati.

Tipi di adattamento (senza anticipare modelli specifici). Nel seguito del capitolo e nei capitoli successivi incontreremo più strategie di post-training/finetuning. Per ora è sufficiente distinguere *cosa* si modifica:

- **Aggiornamento completo**: si aggiornano (quasi) tutti i parametri del modello su un nuovo corpus, spesso mantenendo lo stesso obiettivo di next-token prediction (*continued pretraining*).
- **Aggiornamento parziale**: si congelano molti pesi e si aggiornano solo sottoinsiemi piccoli o moduli aggiuntivi, per ridurre costi e memoria (*parameter-efficient finetuning*, che descriveremo più avanti).
- **Adattamenti supervisionati**: in alcune impostazioni di post-training si usano dataset costruiti con coppie input/risposta (o input/etichetta) per ottenere comportamenti più desiderabili (ad esempio seguire istruzioni, rispondere a domande, ecc.). Entreremo nei dettagli nelle sezioni successive del corso/testo.

Collegamento alle sezioni successive. Una volta completato (pre-)training e post-training, restano due domande centrali. La prima è come misurare in modo affidabile ciò che il modello ha imparato: non basta guardare la loss, ma serve valutare qualità linguistica, coerenza, robustezza e (quando rilevante) factuality, con metriche e benchmark appropriati. La seconda è come la scala influenzi queste capacità: aumentando dati, parametri e risorse computazionali cambiano sia le prestazioni sia le modalità con cui emergono nuove abilità, con implicazioni pratiche su costi e progettazione. Le prossime sezioni affrontano quindi prima la valutazione dei large language models e poi le sfide e i principi legati alla scalabilità.

Capitolo 4

Disentangling Dense Embeddings with Sparse Autoencoders

4.1 Motivazione e contesto

I modelli di linguaggio basati su architetture Transformer producono embeddings contestuali estremamente ricchi ed efficaci dal punto di vista empirico. Tuttavia, queste rappresentazioni collocano il testo in spazi vettoriali ad alta dimensionalità le cui dimensioni non risultano direttamente interpretabili dal punto di vista semantico. Il significato è codificato in modo distribuito e fortemente entangled, rendendo difficile analizzare, spiegare o controllare le rappresentazioni interne del modello.

Questa mancanza di interpretabilità rappresenta un limite rilevante, in particolare in applicazioni che richiedono trasparenza, analisi qualitativa o manipolazione controllata del significato. Ne deriva l'esigenza di trasformare embeddings densi in rappresentazioni latenti più semplici e strutturate, in cui le componenti corrispondano a fattori semantici coerenti e, per quanto possibile, indipendenti. Questo obiettivo è comunemente indicato come *disentanglement* delle rappresentazioni.

In questo capitolo analizziamo il metodo proposto nel lavoro *Disentangling Dense Embeddings with Sparse Autoencoders* [3], che affronta il problema del disentanglement applicando sparse autoencoders agli embeddings prodotti da modelli di linguaggio pre-addestrati. L'idea centrale è che l'introduzione di vincoli di sparsità nello spazio latente favorisca l'emergere di feature interpretabili, consentendo di scomporre rappresentazioni dense ed entangled

in componenti semanticamente significative. Di seguito viene illustrata la metodologia proposta.

4.2 Metodologia e Architettura

Per ottenere il disentanglement degli embeddings densi, la metodologia adottata si basa sull'utilizzo di *Sparse Autoencoders* (SAE). A differenza degli autoencoder tradizionali, che comprimono l'input in uno spazio latente di dimensione inferiore (*bottleneck*), i SAE proiettano l'input in uno spazio latente sovradimensionato ($n \gg d$), imponendo tuttavia un forte vincolo di sparsità sulle attivazioni. Questa scelta architetturale consente di rappresentare ciascun embedding come combinazione lineare di un numero limitato di feature latenti, favorendo l'emergere di componenti semanticamente interpretabili e riducendo il fenomeno di superposizione delle informazioni.

4.2.1 Definizione del Modello

Sia $x \in \mathbb{R}^d$ un vettore di embedding in input, prodotto da un modello di linguaggio pre-addestrato, e sia $h \in \mathbb{R}^n$ la rappresentazione latente, dove n indica il numero totale di feature latenti apprese dal modello. In pratica, n viene scelto come multiplo della dimensionalità originale d, così da ottenere una base latente sovracompleta.

L'architettura del SAE è composta da un encoder e un decoder, definiti come segue:

$$\text{Encoder:} \quad h = f_\theta(x) = \sigma(W_e x + b_e) \tag{4.1}$$

$$\text{Decoder:} \quad \hat{x} = g_\phi(h) = W_d h + b_d \tag{4.2}$$

dove:

- $W_e \in \mathbb{R}^{n \times d}$ e $W_d \in \mathbb{R}^{d \times n}$ sono rispettivamente le matrici dei pesi di codifica e decodifica;
- $b_e \in \mathbb{R}^n$ e $b_d \in \mathbb{R}^d$ sono i vettori di bias;
- $\sigma(\cdot)$ è una funzione di attivazione non lineare, tipicamente una ReLU.

Ogni colonna della matrice di decoder W_d può essere interpretata come una *feature latente*, ovvero una direzione nello spazio degli embedding che corrisponde a un concetto semantico appreso dal modello. La ricostruzione di un embedding avviene quindi come combinazione lineare di un piccolo sottoinsieme di queste direzioni.

4.2.2 Vincolo di Sparsità *k-Sparse*

Un elemento cruciale della metodologia è il meccanismo con cui viene imposta la sparsità. Invece di adottare una regolarizzazione L_1 standard, che può introdurre effetti indesiderati di *shrinkage* (riduzione sistematica dell'ampiezza delle attivazioni), viene utilizzato un vincolo *Top-k*.

Per ciascun input x, solo le k attivazioni più elevate nel vettore latente h vengono mantenute, mentre tutte le altre sono poste a zero. Questo garantisce che ogni embedding venga rappresentato attraverso un numero fisso e limitato di feature attive, rendendo la rappresentazione più interpretabile e favorendo un disentanglement più netto dei fattori semantici.

4.2.3 Funzione di Costo e Addestramento

L'obiettivo dell'addestramento del SAE è minimizzare l'errore di ricostruzione preservando al contempo la sparsità delle attivazioni. La funzione di perdita globale è definita come:

$$\mathcal{L}(\theta, \phi) = \frac{1}{d}\|x - \hat{x}\|_2^2 + \lambda \mathcal{L}_{\text{sparse}}(h) + \alpha \mathcal{L}_{\text{aux}}(x, \hat{x}) \tag{4.3}$$

Oltre al termine di ricostruzione principale, viene introdotta una *auxiliary loss* ($\mathcal{L}_{\text{aux}}$), ispirata alla tecnica dei *ghost gradients*. Questa componente ha lo scopo di mitigare il problema dei *dead latents*, ovvero feature che non si attivano mai durante il processo di addestramento.

Dato l'errore di ricostruzione del modello principale $e = x - \hat{x}$, la perdita ausiliaria è calcolata modellando l'errore residuo tramite un sottoinsieme di latenti inermi:

$$\mathcal{L}_{\text{aux}}(x, \hat{x}) = \|e - \hat{e}\|_2^2 \tag{4.4}$$

dove $\hat{e}$ rappresenta la ricostruzione ottenuta utilizzando esclusivamente tali latenti. Questo meccanismo forza il modello a riutilizzare feature altrimenti inattive, migliorando la copertura dello spazio semantico.

4.3 Interpretazione Automatizzata delle Feature

Una volta addestrato il SAE, è necessario assegnare un significato semantico alle feature latenti apprese. Poiché il numero di feature può raggiungere decine o centinaia di migliaia, un'annotazione manuale risulta impraticabile. Per questo motivo viene adottato un approccio di interpretazione automatizzata basato su Large Language Models (LLM), utilizzati come *Interpreter*.

Per una data feature i, il processo di etichettatura avviene nei seguenti passaggi:

1. **Selezione degli esempi:** vengono individuati i documenti che producono le attivazioni più elevate per la feature i (*max activating examples*), insieme a documenti che non attivano la feature.

2. **Generazione dell'interpretazione:** l'LLM analizza entrambi gli insiemi di testi e identifica il concetto comune presente nei testi attivi ma assente negli altri.

3. **Astrazione:** il concetto individuato viene sintetizzato in una breve etichetta testuale, che rappresenta l'interpretazione semantica della feature.

Questo processo consente di collegare le direzioni vettoriali astratte dello spazio latente a concetti comprensibili dall'uomo.

4.4 Feature Families e Struttura Gerarchica

Le feature apprese da un SAE non sono indipendenti, ma mostrano relazioni di co-occorrenza e organizzazione gerarchica. Per catturare tali relazioni viene introdotto il concetto di *Feature Families*, ovvero gruppi di feature semanticamente correlate.

Una famiglia di feature è composta da una feature *genitore*, caratterizzata da un'elevata densità di attivazione e associata a un concetto più astratto e generale, e da più feature *figlie*, più sparse e specializzate, che rappresentano sottocategorie o istanze più specifiche del concetto genitore. È importante distinguere questo fenomeno dal *feature splitting*, che descrive l'evoluzione delle feature al variare della capacità del modello e non relazioni interne a un singolo SAE.

4.5 Feature Families e Struttura Gerarchica

Le feature apprese da uno Sparse Autoencoder non sono indipendenti, ma mostrano strutture di co-occorrenza e relazioni gerarchiche che riflettono l'organizzazione latente dei concetti semantici nello spazio degli embedding. Per analizzare e strutturare tali relazioni, viene introdotto il concetto di *Feature Families*, ovvero insiemi di feature semanticamente correlate che rappresentano uno stesso concetto a diversi livelli di astrazione.

Una feature family è composta da una feature *genitore*, associata a un concetto più generale e astratto, e da un insieme di feature *figlie*, più sparse e specializzate, che catturano sottocategorie o istanze più specifiche del concetto genitore. È importante distinguere questa struttura gerarchica interna a un singolo SAE dal fenomeno del *feature splitting*, che descrive invece come una stessa direzione semantica possa frammentarsi al crescere della capacità del modello e non riguarda relazioni tra feature all'interno dello stesso autoencoder.

Criterio di identificazione delle feature genitore e figlie La distinzione tra feature genitore e feature figlie non è determinata a partire da considerazioni semantiche esplicite, ma emerge da una proprietà statistica osservata nei dati, ovvero la *densità di attivazione* delle feature. Per ciascuna feature i viene definita la frequenza di attivazione

$$f_i = \sum_k A_{ik}, \tag{4.5}$$

che misura il numero di esempi del dataset sui quali la feature risulta attiva. Intuitivamente, feature con valori elevati di f_i tendono ad attivarsi in un ampio numero di contesti e quindi a rappresentare concetti più generali,

mentre feature con densità inferiore si attivano in contesti più ristretti e catturano concetti più specifici.

Questa osservazione viene utilizzata per indurre una gerarchia: date due feature connesse nel grafo di co-occorrenza, la feature con densità di attivazione maggiore viene interpretata come *genitore*, mentre quella con densità minore come *figlia*. In questo modo, il ruolo gerarchico delle feature è determinato in modo automatico e riproducibile a partire dalle statistiche di attivazione, senza ricorrere a supervisione semantica.

4.5.1 Costruzione del Grafo di Co-occorrenza

Per identificare le feature families, viene costruito un grafo che cattura i pattern di co-attivazione delle feature nel dataset. Per ogni coppia di feature i e j vengono calcolate le seguenti metriche:

- **Matrice di co-occorrenza:**

$$C_{ij} = \sum_k A_{ik} A_{jk}, \tag{4.6}$$

  dove $A_{ik} = 1$ se la feature i è attiva sull'esempio k, e 0 altrimenti. Questo termine misura quante volte le due feature si attivano congiuntamente.

- **Matrice di similarità delle attivazioni:**

$$D_{ij} = \sum_k B_{ik} B_{jk}, \tag{4.7}$$

  dove B_{ik} rappresenta il valore scalare dell'attivazione latente. Questa metrica tiene conto non solo della presenza congiunta delle feature, ma anche dell'intensità delle loro attivazioni.

Poiché feature molto frequenti tendono a co-attivarsi con molte altre in modo spurio, la matrice di co-occorrenza viene normalizzata rispetto alla frequenza f_i della feature i:

$$C_{ij}^{\text{norm}} = \frac{C_{ij}}{f_i + \epsilon}, \tag{4.8}$$

ottenendo una misura asimmetrica che approssima la probabilità che la feature j sia attiva dato che la feature i è attiva.

4.5.2 Identificazione delle Feature Families

Sulla matrice di co-occorrenza normalizzata viene applicata una soglia τ per eliminare relazioni deboli o rumorose. Nel lavoro originale viene utilizzato $\tau = 0.1$, valore che rappresenta un compromesso empirico tra robustezza delle relazioni individuate e copertura dello spazio semantico.

A partire dalla matrice sogliata, l'identificazione delle feature families avviene attraverso i seguenti passaggi:

1. costruzione di un *Maximum Spanning Tree* (MST) sul grafo di co-occorrenza, al fine di preservare le relazioni più forti evitando cicli;
2. orientamento degli archi confrontando la densità di attivazione dei nodi connessi, dirigendo ciascun arco dalla feature più densa (genitore) verso quella meno densa (figlia);
3. identificazione delle famiglie tramite una ricerca in profondità (DFS) a partire dai nodi radice, ovvero feature prive di archi entranti;
4. rimozione iterativa delle feature genitore e ricostruzione dell'MST per far emergere famiglie più fini o parzialmente sovrapposte.

Questo procedimento consente di ricostruire una struttura gerarchica dello spazio semantico latente, nella quale concetti astratti emergono come nodi ad alta connettività che aggregano progressivamente sottocategorie più specifiche, riflettendo l'organizzazione interna degli embeddings densi appresi dal modello.

Capitolo 5

Prisma

5.1 Introduzione

Prisma è un'applicazione nata con l'obiettivo di rendere interpretabili gli *embeddings* testuali, generalizzando questo processo a qualsiasi insieme di documenti. Gli *embeddings*, pur essendo rappresentazioni ricche dal punto di vista semantico, tendono a condensare molteplici concetti in uno spazio denso e difficilmente esplorabile, rendendo complessa l'analisi del loro contenuto.

L'idea alla base di Prisma richiama il comportamento di un prisma ottico: così come un fascio di luce bianca viene scomposto nelle sue componenti cromatiche fondamentali, allo stesso modo un *embedding* semanticamente denso può essere "rifratto" nei concetti che lo compongono. Il software si propone di decomporre queste rappresentazioni compatte in dimensioni concettuali interpretabili, fornendo una visione strutturata del contenuto semantico. Un obiettivo centrale è l'accessibilità: Prisma mira ad abbassare la barriera d'ingresso per utenti non specialisti, fornendo un'interfaccia intuitiva per esplorare rappresentazioni semantiche complesse.

5.2 Architettura

L'applicazione è sviluppata in Python utilizzando il framework Django come nucleo centrale per la gestione della logica di business e dell'interfaccia web. Il sistema si appoggia a un database SQLite locale per la persistenza dei dati e interagisce con il framework Ollama per l'esecuzione dei modelli di linguaggio (LLM) necessari all'interpretazione.

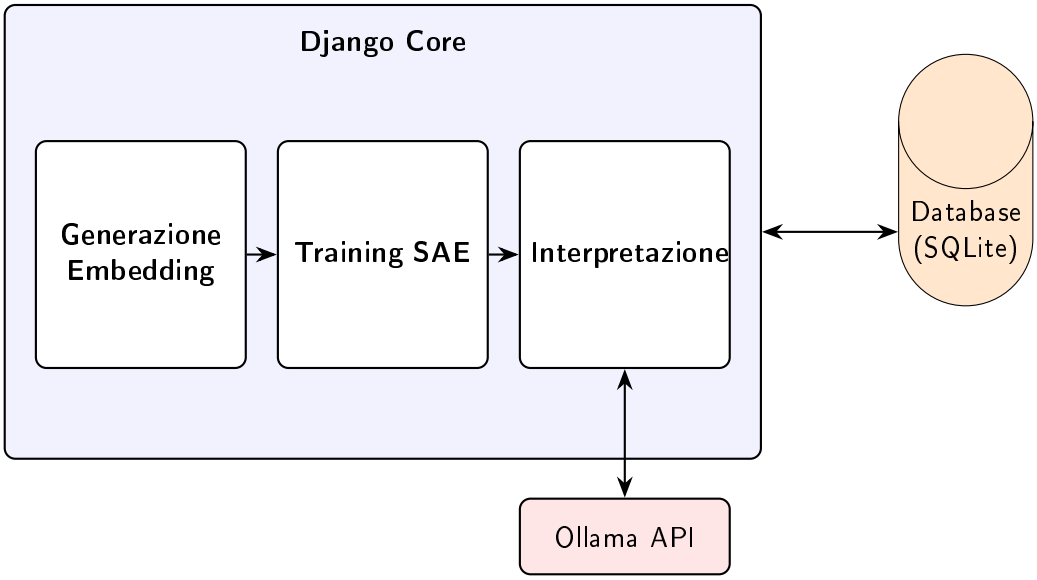

Figura 5.1: Schema semplificato dell'architettura di Prisma.

5.3 Generazione Embeddings

Il flusso operativo inizia con la creazione di un `Dataset` e la successiva trasformazione dei documenti in vettori numerici. L'applicazione permette di selezionare diversi modelli di codifica, tra cui modelli specializzati per il dominio clinico (come *medbit*) o modelli multilingua performanti (come la famiglia *GTE*).

5.3.1 Gestione di documenti lunghi: strategia *chunk-and-average*

Un aspetto strutturale del sistema Prisma riguarda la gestione di documenti testuali la cui lunghezza eccede il limite massimo di input dei modelli Transformer utilizzati per la generazione di embeddings. In particolare, molti modelli della famiglia BERT-like accettano in input una sequenza di lunghezza massima pari a

$$L_{\max} = 512 \text{ token}.$$

Di conseguenza, un documento con lunghezza superiore a $L_{\max}$ non può essere processato integralmente in un singolo forward pass. Una soluzione naïve consiste nella troncatura (*truncation*) del testo, che mantiene solo i primi $L_{\max}$ token; tuttavia, questa scelta può eliminare porzioni informative

rilevanti (ad esempio conclusioni, diagnosi o follow-up riportati in coda al documento). Per preservare l'intero contenuto informativo, Prisma implementa una strategia di segmentazione denominata *chunk-and-average*, che consente di codificare testi arbitrariamente lunghi producendo un singolo embedding globale per documento.

Tokenizzazione e vincoli di lunghezza Dato un documento testuale D, esso viene prima trasformato in una sequenza di token tramite un tokenizer associato al modello scelto. Indichiamo tale sequenza come

$$T = (t_1, t_2, \ldots, t_N),$$

dove N è il numero totale di token ottenuti dalla tokenizzazione e i token t_i possono rappresentare parole intere o sotto-parole (subword), a seconda del vocabolario del modello. I modelli Transformer richiedono tipicamente l'aggiunta di token speciali che delimitano la sequenza e ne abilitano alcune funzionalità interne. Ad esempio, nei modelli tipo BERT è comune inserire un token di classificazione `[CLS]` all'inizio e un token di separazione `[SEP]` alla fine. Indichiamo con N_{special} il numero di token speciali necessari. La lunghezza massima effettivamente disponibile per il contenuto testuale (cioè per i token provenienti dal documento) è quindi

$$W_{\text{eff}} = L_{\max} - N_{\text{special}}.$$

Segmentazione in chunk contigui Se $N \leq W_{\text{eff}}$, il documento può essere processato direttamente. Se invece $N > W_{\text{eff}}$, la sequenza T viene suddivisa in K segmenti contigui (*chunks*) ciascuno di lunghezza al più W_{eff}:

$$C_k = (t_{(k-1)W_{\text{eff}}+1}, \ldots, t_{\min(kW_{\text{eff}}, N)}), \qquad k = 1, \ldots, K,$$

dove

$$K = \left\lceil \frac{N}{W_{\text{eff}}} \right\rceil.$$

Ciascun chunk viene poi confezionato come input valido per il modello aggiungendo i token speciali richiesti (ad esempio `[CLS]` e `[SEP]`), in modo che la lunghezza totale dell'input sia $\leq L_{\max}$.

Embedding per chunk e concetto di pooling Sia f_θ il modello Transformer selezionato (con parametri θ). Dato un chunk C_k composto da m_k

token (dopo l'aggiunta dei token speciali), il forward pass del modello produce una sequenza di vettori nascosti (hidden states), uno per ciascun token:

$$f_\theta(C_k) = (\mathbf{h}_1^{(k)}, \mathbf{h}_2^{(k)}, \ldots, \mathbf{h}_{m_k}^{(k)}), \qquad \mathbf{h}_j^{(k)} \in \mathbb{R}^d,$$

dove d è la dimensionalità interna della rappresentazione (hidden size) del modello. Per ottenere un singolo vettore che rappresenti l'intero chunk, Prisma applica un'operazione di *pooling*. Con il termine pooling si intende una funzione di aggregazione che combina la sequenza di vettori token-level in un unico vettore a livello di segmento (segment-level). Nel caso più comune utilizzato in Prisma, si adotta il *mean pooling*, ovvero la media aritmetica dei vettori dei token (tipicamente escludendo eventuali token speciali, a seconda dell'implementazione):

$$\mathbf{e}_k = \mathrm{MeanPool}\big(f_\theta(C_k)\big) = \frac{1}{m_k} \sum_{j=1}^{m_k} \mathbf{h}_j^{(k)}, \qquad \mathbf{e}_k \in \mathbb{R}^d.$$

Intuitivamente, il mean pooling produce un vettore che riassume il contenuto semantico del chunk distribuendo il contributo su tutti i token che lo compongono, senza privilegiare una posizione specifica della sequenza.

Aggregazione a livello documento Una volta ottenuti gli embedding dei singoli chunk $\mathbf{e}_1, \ldots, \mathbf{e}_K$, Prisma costruisce un embedding globale per il documento D combinandoli tramite media aritmetica:

$$\mathbf{v}_D = \frac{1}{K} \sum_{k=1}^{K} \mathbf{e}_k, \qquad \mathbf{v}_D \in \mathbb{R}^d.$$

Questa scelta consente di ottenere una rappresentazione unica e confrontabile tra documenti, anche quando le loro lunghezze originali sono molto diverse. Inoltre, la procedura garantisce che tutte le parti del testo contribuiscano alla rappresentazione finale, evitando la perdita sistematica di informazione tipica della troncatura.

5.4 Training SAE

Una volta generati gli *embeddings*, il sistema procede alla loro scomposizione tramite il modulo *Sparse Autoencoders* (SAE). In questa fase, l'utente configura un'istanza di `SAERun` definendo parametri quali l'*expansion factor* (per

determinare la dimensione del livello latente) e la *k-sparsity* (per regolare il numero di neuroni attivi contemporaneamente).

Il training produce un modello capace di ricostruire l'input densificato attraverso una rappresentazione sparsa. Al termine, Prisma genera automaticamente analisi visive, tra cui mappe di calore della similarità delle matrici e della co-occorrenza, che permettono di valutare statisticamente la qualità delle feature estratte prima dell'interpretazione semantica.

5.5 Interpretazione

L'ultima fase del processo trasforma le feature matematiche in concetti comprensibili. Il modulo *Explorer* utilizza le attivazioni più significative di ogni feature per estrarre "evidenze" testuali dai documenti originali.

Queste evidenze vengono inviate all'API locale di Ollama, che genera un'etichetta (*label*) e una descrizione testuale per ogni feature. Prisma supporta la gestione di interpretazioni multiple per la stessa feature e permette di organizzare i concetti in `FeatureFamily`, ricostruendo una gerarchia semantica che va dai concetti generali a quelli più specifici.

Capitolo 6

Esperimenti e risultati

6.1 Introduzione

In questo lavoro di tesi sono stati condotti tre esperimenti principali. Il primo è stato chiamato Pedianet e riguarda dati medici che verranno in seguito presentati e descritti, il secondo basato su un dataset di abstracts di paper scientifici generale, il terzo invece riguarda un dataset di papers scientifici di PubMed. Verranno di seguito descritti e verranno presentati i risultati ottenuti.

6.2 Pedianet

Il progetto Pedianet è una rete italiana di pediatria di famiglia fondata nel 1998, con l'obiettivo di raccogliere, strutturare e analizzare i dati clinici pediatrici derivanti dalla normale attività ambulatoriale dei pediatri di libera scelta (PLS). L'iniziativa nasce per colmare una lacuna storica nella ricerca clinica pediatrica, ovvero la scarsa disponibilità di dati real-world relativi alla popolazione infantile. A differenza di altri database europei che includono anche adulti, Pedianet è uno dei pochi progetti esclusivamente focalizzati sull'età pediatrica, risultando così un archivio unico nel panorama della ricerca biomedica europea. La rete è composta da oltre 400 pediatri distribuiti su tutto il territorio italiano, che utilizzano la stessa piattaforma elettronica (Junior Bit) per la gestione delle cartelle cliniche dei pazienti. Circa 3700 pediatri utilizzano questo software, e sono potenziali candidati ad aderire alla rete. Tutti i dati raccolti vengono trasmessi tramite connessione protetta a

un server centrale situato a Padova, dove vengono validati, standardizzati e archiviati. I dati raccolti sono nella forma di cartelle cliniche pediatriche, ma presentano il problema di non essere strutturati, ovvero non organizzati e non seprati in classi. Con la nascita dei large language models questi dati hanno assunto un valore completamente nuovo perché se correttammente lavorati e organizzati, dal momento che vanno a ricoprire un dominio molto specifico e verticale, si prestano benissimo a funzionare come dati di addestramento nel campo di modelli di NLP. Un'altra motivazione per cui è necessario intervenire con strumenti di NLP è quello di svolgere task di classificaione per destinare questi dati al mondo della ricerca. Per esempio si vorrebbe raccogliere tutte quelle cartelle che hanno visto casi di *bronchiolite*, ma non bastano metodi di regulary expression dal momento che ci sono cartelle che invece presentano casi di *sospetta* bronchiolite. Si proverà quindi ad intervenire con la metodologia presentata nei precedenti capitoli per processare questi dati. Ecco il testo della sottosezione in formato paragrafo unico:

6.2.1 Scelta del modello di embedding

Per la generazione delle rappresentazioni vettoriali è stato selezionato il modello MedBIT [6], progettato per colmare il divario tecnologico nelle lingue meno fornite di risorse biomediche come l'italiano. Nel lavoro di riferimento, gli autori mettono a confronto due approcci distinti: BioBIT, basato sulla traduzione automatica neurale di milioni di abstract di PubMed per privilegiare la quantità dei dati , e MedBIT, che raffina tale conoscenza attraverso l'uso di un corpus di alta qualità composto da libri di testo medici scritti nativamente in italiano. La scelta di MedBIT per il progetto Pedianet risulta ottimale innanzitutto per la sua natività linguistica; poiché i dati di Pedianet sono redatti interamente in italiano dai medici di base , il modello garantisce una rappresentazione più naturale e sintatticamente corretta rispetto ai sistemi basati su traduzioni automatiche, riducendo le ambiguità semantiche. Queste caratteristiche rendono MedBIT lo strumento più affidabile per catturare le delicate sfumature cliniche presenti nei diari dei pediatri italiani, permettendo di distinguere efficacemente tra diagnosi accertate e semplici sospetti.

6.2.2 Esperimento

Per la conduzione dell'esperimento sono state processate circa 200.000 cartelle cliniche provenienti dal database Pedianet. Durante la fase preliminare di

analisi, è emersa una criticità significativa legata alla lunghezza dei testi: una porzione rilevante delle note cliniche superava il limite di lunghezza massima (L_{max}) imposto dall'architettura del modello MedBIT (tipicamente 512 tokens). La troncatura del testo (truncation) avrebbe comportato la perdita di informazioni potenzialmente cruciali situate nella parte finale delle note cliniche, compromettendo la qualità della rappresentazione semantica. Per ovviare a questo problema senza perdere contenuto informativo, è stata adottata una strategia di *chunk-and-average*. Formalmente, dato un documento D costituito da una sequenza di token $T = t_1, t_2, \ldots, t_N$, dove $N > L_{max}$, il testo viene suddiviso in K segmenti (chunks) contigui $C_1, C_2, \ldots, C_K$. La lunghezza effettiva utilizzabile per ogni chunk è definita come $W_{eff} = L_{max} - N_{special}$, dove $N_{special}$ rappresenta il numero di token speciali richiesti dal modello (es. `[CLS]` e `[SEP]`). Il documento viene quindi partizionato in modo tale che ogni chunk C_k contenga una sottosequenza di token di lunghezza al più W_{eff}:

$$C_k = t_i, \ldots, t_{i+W_{eff}-1}$$

Per ogni segmento k, viene calcolato un embedding denso $\mathbf{e}k \in \mathbb{R}^d$ attraverso il modello $f\theta$:

$$\mathbf{e}k = \text{Pooling}(f\theta(C_k))$$

dove la funzione di pooling (nel nostro caso, *mean pooling*) aggrega gli stati nascosti dell'ultimo layer del modello per ottenere una singola rappresentazione vettoriale per il segmento. Infine, la rappresentazione vettoriale globale dell'intero documento clinico $\mathbf{v}_D$ è ottenuta calcolando la media aritmetica degli embedding dei singoli segmenti. Questo approccio permette di condensare l'informazione distribuita lungo tutto il testo in un unico punto nello spazio latente:

$$\mathbf{v}D = \frac{1}{K} \sum k = 1^K \mathbf{e}_k$$

Questa metodologia garantisce che ogni parte della cartella clinica contribuisca equamente alla rappresentazione finale, preservando dettagli sintomatologici o diagnostici che potrebbero trovarsi in qualsiasi punto della narrazione medica. A valle di questo processo di codifica, i vettori risultanti sono stati utilizzati per addestrare gli autoencoder sparsi (SAE) oggetto dello studio.

6.3 Abstracts

6.4 PubMed

Bibliografia

[1] Dor Bank, Noam Koenigstein e Raja Giryes. "Autoencoders". In: *CoRR* abs/2003.05991 (2020). arXiv: 2003.05991. URL: https://arxiv.org/abs/2003.05991.

[2] Umberto Michelucci. *An Introduction to Autoencoders*. 2022. arXiv: 2201.03898 [cs.LG]. URL: https://arxiv.org/abs/2201.03898.

[3] Xin Wang et al. *Disentangled Representation Learning*. 2024. arXiv: 2211.11695 [cs.LG]. URL: https://arxiv.org/abs/2211.11695.

[4] Daniel Jurafsky e James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd. Online manuscript released August 24, 2025. 2025. URL: https://web.stanford.edu/~jurafsky/slp3/.

[5] Mark Davies. *The Wikipedia Corpus: 4.6 million articles, 1.9 billion words*. https://www.english-corpora.org/wiki/. Adapted from Wikipedia. 2015.

[6] Tommaso Mario Buonocore et al. "Localizing in-domain adaptation of transformer-based biomedical language models". In: *Journal of Biomedical Informatics* 144 (2023), p. 104431. ISSN: 1532-0464. DOI: https://doi.org/10.1016/j.jbi.2023.104431. URL: https://www.sciencedirect.com/science/article/pii/S1532046423001521.