



**UNIVERSITÀ
DI TORINO**

Università degli Studi di Torino

Corso di Laurea Magistrale in INFORMATICA

*INTELLIGENZA ARTIFICIALE E SISTEMI INFORMATICI
“PIETRO TORASSO”*

**Leveraging deep neural networks for Text-to-SQL
translation of microbial resources**

Tesi di Laurea Magistrale

Relatori:

Prof. BECCUTI Marco

Prof. DI CARO Luigi

Correlatore:

Dott. CONTALDO Sandro Gepiro

Candidato:

METTA Michele
958226

Anno Accademico 2023/2024

DICHIARAZIONE DI ORIGINALITÀ:

Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata.

Indice

Introduzione	3
1 Background	4
1.1 Elaborazione del linguaggio naturale (NLP)	4
1.2 Apprendimento automatico	5
1.3 Text-to-SQL	7
1.4 Architettura Transformer	13
1.4.1 Tokenization	17
1.4.2 Encoder	18
1.4.2.1 Positional encoding	18
1.4.2.2 Sublayer-Multi-head attention	24
1.4.2.3 Post-layer Normalization	33
1.4.2.4 SubLayer-Feed-Forward Network (FFN) . . .	35
1.4.3 Decoder	38
1.4.3.1 Sublayer-Masked Multi-Head Attention . . .	39
1.4.3.2 Sublayer-Multi-head attention - Decoder . .	42
1.4.3.3 Sublayers Linear e Softmax	43
1.4.4 Training	45
1.4.5 Large Language Model Meta AI (LLaMA)	49
1.5 Fine-tuning (LoRA & QLoRA)	50
2 Database & Dataset	58
2.1 Database biologico	58
2.2 Dataset sperimentali	65
2.2.1 SQL-Create-Context-Instruction	65
2.2.2 BioInfoDataset	67
3 Implementazione applicazione Text-to-SQL	73
3.1 Architettura del Sistema	73
3.1.1 SLURM	79
4 Sperimentazione e Risultati	83
4.1 Valutazione Performance	83
4.1.1 Risultati in the wild	95

5 Conclusioni e Sviluppi futuri	99
Ringraziamenti	104
Bibliografia	106

Introduzione

Questa tesi si concentra sullo studio e l'implementazione di un sistema nel campo del Natural Language Processing, focalizzandosi sul task del text-to-SQL applicato all'ambito biologico. Verranno descritte le moderne tecniche di deep learning per la risoluzione di questo task, con particolare enfasi sull'architettura Transformer. Inoltre, sarà descritto il database contenente informazioni riguardanti le collezioni microbiche italiane, sviluppato all'interno del progetto MIRRI-IT (Microbial Resource Research Infrastructure). Successivamente, saranno presentati i dataset utilizzati per il fine-tuning dei modelli linguistici impiegati nella fase di sperimentazione e l'architettura dell'applicazione web, sviluppata in Next.js, progettata per consentire ai biologi di sfruttare efficacemente il sistema. Infine, verranno discussi i risultati ottenuti e i possibili miglioramenti futuri.

Capitolo 1

Background

1.1 Elaborazione del linguaggio naturale (NLP)

L’elaborazione del linguaggio naturale (NLP) [10] [11] è un campo interdisciplinare che combina linguistica, informatica e intelligenza artificiale per studiare l’interazione tra macchine ed esseri umani. L’obiettivo è consentire ai sistemi automatici di analizzare, rappresentare, interpretare e generare il linguaggio naturale. In particolare, l’obiettivo principale di NLP, è quello di sviluppare algoritmi e modelli in grado di estrarre informazioni dai dati di tipo linguistico, in forma testuale o vocale, automatizzando una vasta gamma di task che, altrimenti, richiederebbero l’intervento umano per poter essere completati. Nel corso del tempo, NLP ha attraversato diverse fasi di evoluzione: dagli approcci simbolici e basati su regole, agli approcci statistici degli anni ’90, fino agli attuali metodi basati su modelli di deep learning e rappresentazioni dense (word embeddings), come ad esempio BERT e GPT. Il linguaggio umano, è per sua natura, incredibilmente complesso, soprattutto a causa dell’eccessiva ambiguità provocata dalla grande quantità di elementi più o meno rilevanti che sono presenti all’interno di una frase, a cui noi esseri umani siamo in grado di assegnare una maggiore o minore importanza anche in base al contesto considerato fino ad un certo istante. Negli ultimi anni, i progressi nei modelli neurali, in particolare quelli ottenuti grazie all’architettura Transformer (che verrà descritta più approfonditamente nella sezione 1.4), hanno permesso di superare molte di queste difficoltà, garantendo prestazioni elevate in diversi task linguistici.

Le applicazioni basate su NLP sono ormai utilizzate in quasi tutti i settori, come ad esempio:

- Servizio Clienti: mediante l’utilizzo di chatbot e assistenti virtuali, le aziende sono in grado di fornire assistenza ai propri clienti, fornendo risposte rapide e pertinenti riducendo il carico di lavoro degli operatori umani.

- Finanziario: utilizzato per accelerare l'estrazione di informazioni da bilanci, relazioni annuali e normative, comunicati stampa o provenienti dai social media.
- Medico: le nuove conoscenze e scoperte mediche possono arrivare tanto velocemente da non consentire agli operatori sanitari di restare al passo. Gli strumenti basati su NLP e AI possono aiutare a velocizzare l'analisi delle cartelle cliniche e dei documenti di ricerca medica, rendendo possibili decisioni mediche più informate o assistendo nel rilevamento o addirittura nella prevenzione delle condizioni mediche.
- Assicurativo: permette di poter creare modelli in grado di analizzare più velocemente le richieste di rimborso da parte dei clienti in modo tale da ottimizzare i tempi di revisione da parte dei dipendenti.
- Risorse Umane: alcune aziende utilizzano l'NLP per analizzare i curriculum dei candidati, identificando le competenze chiave e abbinandole alle descrizioni delle posizioni aperte, ottimizzando in questo modo il processo di selezione.

1.2 Apprendimento automatico

L'apprendimento automatico (o Machine Learning) è una branca dell'intelligenza artificiale che si occupa dello sviluppo di algoritmi in grado di apprendere automaticamente dai dati e migliorare le proprie performance senza essere esplicitamente programmati. Attraverso l'analisi di grandi quantità di dati, i modelli di apprendimento automatico possono riconoscere pattern, fare previsioni e prendere decisioni in modo autonomo. Questo campo, affronta numerosi task (alcuni dei quali appartenenti all'ambito dell'NLP) che possono essere classificati in base alle tipologie di apprendimento utilizzate. I principali approcci sono: apprendimento supervisionato, non supervisionato, semi-supervisionato e self-supervised. Essi saranno descritti di seguito.

Apprendimento Supervisionato:

L'apprendimento supervisionato richiede dati etichettati per addestrare gli algoritmi di Machine Learning. L'obiettivo, è quindi quello di ottenere un modello che sia in grado di predire correttamente l'output dei nuovi esempi non visti durante la fase di addestramento. Uno dei compiti che possono essere affrontati con l'approccio descritto, e che rappresenta anche l'obiettivo principale di questo lavoro di tesi, è il **Text-to-SQL**. Esso verrà descritto in maniera più approfondita nella sezione 1.3.

Apprendimento Non supervisionato:

Questo approccio utilizza dati non etichettati per scoprire schemi nascosti. In questo caso quindi, gli algoritmi di Machine Learning devono essere in grado di riconoscere direttamente dai dati eventuali sottogruppi con determinate caratteristiche distintive rispetto a tutti gli altri.

Esempio:

Un task comune in NLP, che rientra in questa tipologia di apprendimento è quello del **clustering di documenti**. In questo caso, l'obiettivo è quello di raggruppare, all'interno di cluster differenti, documenti che siano semanticamente coerenti tra loro. In questo modo, ad esempio si potrebbero categorizzare automaticamente articoli in base alle tematiche trattate al loro interno (ad es. politica, tecnologia, religione, ecc..). Tutte le tecniche di clustering si basano sul concetto di distanza tra due elementi, di conseguenza, i dati potranno essere raggruppati in modi differenti in base alla metrica di distanza utilizzata.

Apprendimento Semi-supervisionato:

E' una tecnica di Machine Learning che si colloca tra l'apprendimento supervisionato e quello non supervisionato [13]. In questo caso, l'algoritmo, che può essere ad esempio una rete neurale piuttosto che un classificatore classico (ad es. K-Nearest Neighbors, Support Vector Machines o Random Forest), apprende da una combinazione di una piccola quantità di dati etichettati e una grande quantità di dati non etichettati. Questo approccio è particolarmente utile nel momento in cui riuscire ad ottenere dei dati etichettati può essere troppo costoso oppure richiede una quantità di tempo troppo elevata.

Esempio:

Un esempio di apprendimento semi-supervisionato è la tecnica chiamata "Self-training". Essa consiste, nell'addestrare prima un algoritmo su un piccolo set di dati etichettato e dopodichè, una volta ottenuto il modello, quest'ultimo viene utilizzato per predire le etichette dei dati non ancora etichettati. In particolare, se la confidenza del modello, su un certo esempio, supera un certo valore di soglia, allora questo esempio sarà aggiunto al nuovo dataset con la pseudo-etichetta assegnata. Dopodichè, il modello viene riaddestrato sull'insieme combinato di dati etichettati e pseudo-etichettati. Questo processo si ripete fino alla convergenza, e, in genere, il modello riesce a migliorare gradualmente le sue prestazioni. Nel campo dell'NLP, ad esempio Meta ha utilizzato il semi-supervised learning per i suoi modelli di

Speech Recognition riuscendo a ridurre notevolmente il Word Error Rate (WER), ottenendo di fatti un miglioramento significativo [12].

Apprendimento Self-supervised:

E' un approccio basato su dati non etichettati. Sostanzialmente, in questo caso, il modello impara a rappresentare i dati senza utilizzare una supervisione esterna da parte di un essere umano. In questo modo, è il modello stesso a generare i nuovi esempi grazie ai quali avanza nell'apprendimento. Questo metodo è ampiamente utilizzato nell'addestramento di modelli di linguaggio naturale, come GPT o Llama, dove il modello viene pre-addestrato su grandi quantità di testo non etichettato per apprendere rappresentazioni linguistiche profonde.

Esempi:

Gli esempi più noti, sono il Masked Language Modeling (MLM) e il Causal Language Modeling (CLM) che saranno descritti alla fine della sottosezione 1.4.4.

1.3 Text-to-SQL

In questa sezione verrà descritto più nel dettaglio il task del Text-to-SQL.

Formulazione del problema [14]:

Data una query in linguaggio naturale (Natural Language Query, NLQ) su un database relazionale (RDB) con uno schema specifico, l'obiettivo è quello di produrre una query SQL equivalente nel significato, che sia valida per il suddetto RDB e che, quando eseguita, restituirà risultati che corrispondono all'intento dell'utente.

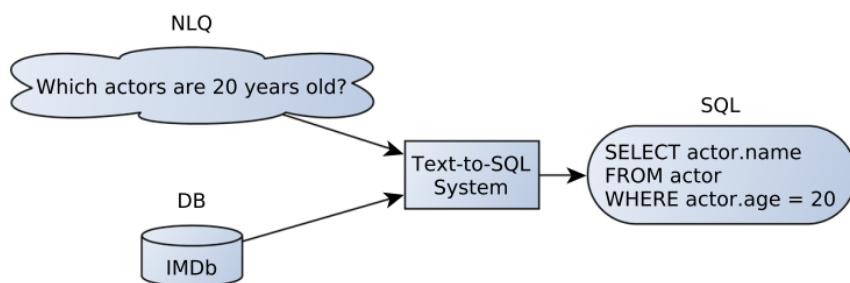


Figura 1.1: Rappresentazione generica del task Text-to-SQL [14]

La query descritta linguaggio naturale, può corrispondere o ad un’espressione completa e fluente (ad esempio “Quali attori hanno più di 20 anni?” mostrata in figura 1.1) oppure può essere costituita solo da alcune parole chiave (ad esempio “Ristoranti italiani a Vienna”). In questa tesi è stato utilizzato un **database contenente informazioni sulle collezioni microbiche italiane**, sviluppato nell’ambito del **progetto SUS-MIRRI.IT** (Strengthening the MIRRI Italian Research Infrastructure for Sustainable Bioscience and Bioeconomy). Il progetto, finanziato dal PNRR, fa parte di **MIRRI-IT**, il **nodo italiano di MIRRI-ERIC** (European Research Infrastructure Consortium). Ulteriori dettagli sulle componenti di tale database saranno descritte all’interno del capitolo 2, in particolare nella sezione 2.1.

Stato dell’arte del text-to-SQL:

Come descritto nell’articolo [15], il deep learning, rappresenta ormai lo stato dell’arte per per il task del text-to-SQL. In letteratura, sono state proposte diverse architetture neurali come risoluzione del task basate su approcci differenti. Nonostante ciò, esse hanno delle componenti chiavi simili tra loro, per questo motivo, secondo gli autori dell’articolo [14], è quindi possibile definire una struttura generica, in grado di rappresentare questi diversi approcci in modo tale da poterli comprendere meglio.

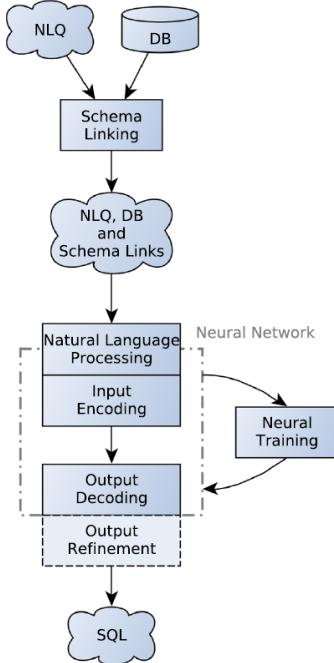


Figura 1.2: Rappresentazione grafica di un sistema generico per il text-to-SQL, basato su reti neurali [14]

Dati in input una query in linguaggio naturale (NLQ) e un database relazionale (RDB) di riferimento, è possibile descrivere il funzionamento di un sistema specializzato per il text-to-SQL, seguendo la struttura mostrata in figura 1.2, con i seguenti step [14] [15]:

1. **Schema linking:** Corrisponde al processo di individuazione delle parti della domanda in linguaggio naturale che fanno riferimento agli elementi specifici del database. Le parti della NLQ che potrebbero riferirsi ad un elemento del database sono chiamate **query candidates**, mentre gli elementi del database che potrebbero essere menzionati nella NLQ sono chiamati **database candidates**. I query candidates possono essere parole o frasi, mentre i database candidates possono essere tabelle, colonne e valori presenti nel database. Una connessione tra un query candidate e un database candidate viene chiamata **schema link**, che può essere ulteriormente classificata come **table link** o **column link** quando il query candidate è mappato su un nome di tabella o di colonna, e come **value link** quando corrisponde a un valore specifico di una colonna. Lo schema-linking è un compito complesso per vari motivi. I query candidates e i database candidates potrebbero non utilizzare lo stesso vocabolario o apparire con la stessa formulazione. Ad esempio, la frase “sang by” nella domanda potrebbe riferirsi alla colonna del database chiamata “singer” (stessa radice della paro-

la, ma formulata diversamente) e, un altro problema da considerare, è quando la NLQ esprime una condizione (cioè un riferimento a un valore del database) in modo diverso rispetto a come quel valore è memorizzato. **Lo schema linking, migliora il contenuto informativo dell'input ma non è strettamente necessario, poichè esistono sistemi, anche piuttosto recenti, che si basano sulle sole componenti neurali per generare la query SQL corretta.**

In particolare, per l'individuazione dei query candidates, in letteratura, sono stati individuati e testati diversi approcci. Uno dei più significativi è quello implementato in ValueNet [16], dove si applica inizialmente il Named Entity Recognition (NER) sulla NLQ per identificare le entità rilevanti. Successivamente, a partire dalle entità identificate, vengono generati nuovi potenziali query candidates cercando valori simili nel database attraverso tecniche di manipolazione delle stringhe. Ad esempio, l'entità “New York” presente nella NLQ può essere mappata su varianti come “N.Y.” o “NY”. Se nel database viene individuata solo la forma abbreviata “NY”, allora sarà quest'ultima ad essere selezionata come input finale. Questo processo potrebbe aiutare la rete a generare correttamente una condizione del tipo “state=NY”.

Le informazioni così ottenute vengono poi fornite in input ad una rete neurale basata su un'architettura encoder-decoder, il cui scopo è generare la query SQL corretta. Questo approccio si è dimostrato altamente efficace, raggiungendo risultati allo stato dell'arte.

Anche per l'individuazione dei database candidates sono stati proposti diversi approcci in letteratura. Uno di essi consiste nel considerare tutti i nomi delle tabelle e delle colonne del database come possibili candidati. Un altro approccio, invece, parte dai query candidates e seleziona solo quelli che hanno una probabilità elevata di riferirsi a valori effettivamente presenti nel database. Questa selezione può avvenire, ad esempio, identificando i valori racchiusi tra virgolette nella NLQ o basandosi su specifiche euristiche. Un ulteriore approccio interessante, utilizzato invece da IRNet [17], è quello basato sui knowledge graph nel caso in cui non si abbia accesso diretto ai contenuti del database. In questo metodo, IRNet sfrutta il grafo della conoscenza ConceptNet per riconoscere i collegamenti ai valori. Come primo passo, vengono considerati tutti gli n-grammi racchiusi tra virgolette singole nella NLQ come possibili query candidates che potrebbero riferirsi a dei valori. Per scoprire, la colonna o la tabella del DB, che potrebbe contenere un valore corrispondente, il sistema ricerca ciascun candidato all'interno del grafo della conoscenza e conserva solo due tipi di risultati: is-type-of e related-terms. Ad esempio, cercando “New York” in ConceptNet, uno dei risultati restituiti è “is-type-of state”. Questo risultato per-

mette a IRNet di collegare “New York” a una colonna denominata “state” o simile. Ciò che distingue questo approccio è il fatto che il collegamento al valore viene scoperto utilizzando un candidato intermedio (il risultato del grafo della conoscenza) in combinazione con i nomi delle colonne del database.

Una volta ottenuti i query candidates e i database candidates c’è bisogno di utilizzare una metodologia per legare queste due componenti e quindi eseguire quello che viene chiamato **candidate matching**.

In questa tesi, per mantenere l’input fornito al modello transformer il più semplice possibile, si è deciso di non implementare a priori un meccanismo di schema-linking, delegando invece questa funzione ai meccanismi di attenzione interni al transformer stesso, poiché questi ultimi si sono rivelati molto efficaci nell’individuare e collegare automaticamente i query candidates e i database candidates. Nonostante ciò, come sarà descritto nella sottosezione 3.1.1, nella fase di post-processing è stata utilizzata la tecnica di candidate matching, chiamata **Fuzzy/approximate string matching**, per mappare eventuali nomi di colonne presenti nella query di output, generati in forma non del tutto corretta dal modello, con i rispettivi elementi appropriati del database.

2. **Natural language representation:** Uno step fondamentale per un sistema generico di text-to-SQL consiste nel trasformare il linguaggio naturale, espresso in forma testuale, in una rappresentazione numerica che la macchina possa elaborare. Fino a poco tempo fa, la tecnica più utilizzata per questo scopo era basata sugli embedding di parole pre-addestrati. Tuttavia, l’introduzione dell’architettura Transformer [1] e il suo impiego nella creazione di grandi **modelli linguistici pre-addestrati (Pretrained Language Models, PLM)** ha cambiato radicalmente il panorama, affermandosi rapidamente come la soluzione di riferimento per la rappresentazione del linguaggio naturale. In particolare, i **modelli encoder-decoder**, grazie all’encoder, sono in grado di costruire le rappresentazioni numeriche contestualizzate per ciascun token della frase di input, e grazie al decoder, sono in grado di sfruttare tali rappresentazioni per generare l’output desiderato (in questo caso la query SQL). In questo lavoro di tesi, sono stati testati alcuni dei modelli pre-addestrati Llama che, come sarà descritto nella sottosezione 1.4.5, in realtà, sono costituiti esclusivamente dallo stack di decoder, mantenendo comunque la possibilità di poter apprendere rappresentazioni interne e quindi risolvere task specifici.
3. **Input encoding:** Un altro aspetto importante è come si decide di strutturare l’input che verrà poi fornito in input all’encoder della rete, in modo tale che quest’ultimo possa elaborarlo in maniera efficace.

L'approccio utilizzato in questa tesi è chiamato **input serialisation**. Esso consiste nell'andare a serializzare tutti gli input in un'unica sequenza che costituirà sostanzialmente il prompt finale. Il motivo di questa scelta è che essa è una tecnica molto comune quando si utilizzano i PLM, poichè questi ultimi sono in grado di creare una rappresentazione contestualizzata dell'intero input. Quindi, qualora ogni input, venisse codificato separatamente, il sistema non potrebbe beneficiare della capacità di contestualizzazione del PLM. Questo approccio da un lato permette di semplificare il processo di codifica sfruttando la robustezza dei PLM ma dall'altro, comporta anche degli svantaggi, come ad esempio la perdita di informazioni sulla struttura dello schema del DB e il fatto di non riuscire a rappresentare facilmente le relazioni tra gli input (ad esempio, relazioni chiave primaria-chiave esterna, collegamenti allo schema, ecc.). Tuttavia, in questa tesi è stato utilizzato l'approccio appena descritto, poichè come verrà discusso nel capitolo 2, la sperimentazione si è basata principalmente su una singola tabella del database per cui non c'è stata la necessità di fare riferimento ad eventuali schemi esterni.

4. **Output decoding:** E' lo step nel quale si decide in che modo vincolare la generazione dell'output prodotta dal decoder della rete. Anche qui esistono diversi approcci, i più noti sono: il Sequence-based, lo Sketch-based slot-filling e il Grammar-based. In particolare, in questo progetto è stato utilizzato il **Sequence-based**, il quale, sfruttando sempre l'enorme potenzialità dei decoder dei PLM, grazie ad una specifica fase di fine-tuning (che sarà descritta nel capitolo 4), è stato possibile vincolare il decoder ad apprendere la corretta generazione delle query SQL.
5. **Neural training:** Un'altra scelta importante, nel momento in cui si decide di costruire un sistema neurale per il text-to-SQL, è quella legata a quale metodologia applicare per addestrare la rete. L'approccio che è stato utilizzato in questa sperimentazione è stato quello del **transfer learning**, poichè, sono stati testati diversi modelli (Llama) pre-addestrati su grandi corpus, per riuscire ad apprendere le rappresentazioni latenti del linguaggio naturale e, dopodichè, tramite l'applicazione del fine-tuning, essi sono stati specializzati per svolgere il task del text-to-SQL. Il motivo di questa scelta è legata al fatto che, il transfer learning, è stato già applicato per la risoluzione di altri differenti task dell'NLP, producendo un notevole aumento delle performance in quasi tutti i casi. I dataset etichettati, che sono stati utilizzati per "specializzare" il modello, saranno descritti nel capitolo 2.
6. **Output refinement:** Infine, l'ultimo step da considerare, è quello legato al come "correggere" l'output ottenuto dal modello, in modo tale

da modificare un’eventuale query SQL generata in maniera errata, per ottenere risultati migliori. Per raggiungere tale obiettivo, nel sistema sviluppato per questa tesi, è stata implementata una **fase di post-processing** specifica che sarà descritta nella sottosezione 3.1.1.

1.4 Architettura Transformer

Nell’articolo “Attention is all your need” [1] è stata presentata una nuova architettura neurale chiamata Transformer. Tale architettura ha portando notevoli miglioramenti nel campo del NLP stabilendo di fatti il nuovo stato dell’arte. I transformer sono stati sviluppati inizialmente come miglioramento delle architetture precedenti, basate principalmente sulle Recurrent Neural Networks (RNN), per risolvere il task di Machine Translation [2]. Successivamente però hanno trovato tantissime applicazioni reali in diversi ambiti, come ad esempio nell’elaborazione del linguaggio naturale su larga scala, nella computer vision (con i vision transformers), nella text categorization, sentiment analysis e nell’apprendimento multimediale.

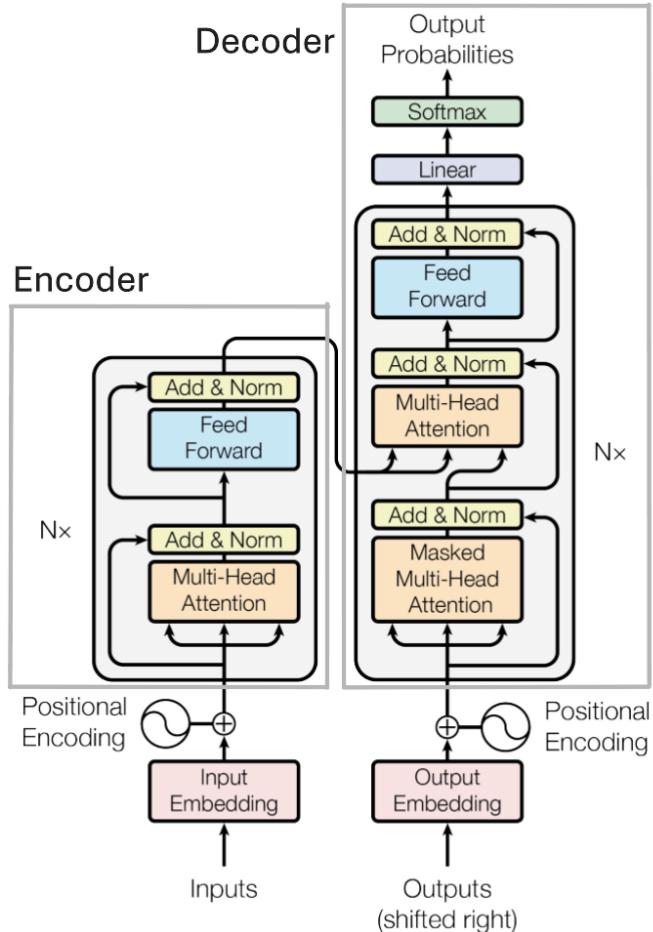


Figura 1.3: Architettura Transformer [1]

Come mostrato in Figura 1.3, l'architettura Transformer è composta da due parti principali chiamate rispettivamente Encoder e Decoder. Per semplicità, nell'immagine vengono mostrati solamente i singoli blocchi di encoding e decoding ma nell'architettura originale in realtà i singoli blocchi cerchiati in grigio vengono ripetuti 6 volte in sequenza formando una pila di 6 layers sia per l'encoder che per il decoder (encoder stack e decoder stack). È possibile riassumere, il funzionamento complessivo di un Transformer con i seguenti punti:

- 1) **Tokenizzazione:** La frase di input viene convertita in una serie di embeddings che sono semplicemente dei vettori numerici rappresentanti i singoli token (unità minime di significato che compongono la frase) attraverso quello che viene chiamato processo di tokenizzazione.
- 2) **Encoding:** Tutti gli embedding ottenuti dalla fase precedente vengono dati in input all'encoder che si preoccuperà di applicare layer dopo layer il meccanismo di self-attention seguito ogni volta da una fully-connected

feed-forward network (FC-FFN) le cui caratteristiche saranno descritte nella sottosezione 1.4.2.4. La funzione principale dell'encoder è quella di fornire in input al decoder una rappresentazione della frase iniziale che tenga conto delle dipendenze che ci sono tra le parole. In particolare, cerca di individuare quali sono i termini all'interno della frase che sono semanticamente legati tra loro e sui quali quindi bisognerà prestare “maggiore attenzione” in modo tale da poter permettere al decoder di generare il prossimo token correttamente.

3) **Decoding:** Il decoder riceve in input degli embedding appartenenti a determinati token che lo aiuteranno sia per iniziare la generazione del testo che per continuarlo. In particolare, possiamo immaginare che all'inizio, quando il decoder non ha ancora generato nessun token, esso prenderà in input sia l'embedding associato ad uno specifico token di start e sia l'output generato dal top encoder e sfruttando tutte queste informazioni genererà in output il token più probabile. Questo token appena generato verrà riportato in input al decoder e quest'ultimo sfruttando, sia la solita rappresentazione ottenuta dal top encoder e sia tutte le informazioni associate a quello che ha già generato, produrrà il token successivo più probabile. Nello step successivo, questo processo continua fino a quando o si raggiunge il numero massimo di tokens generati oppure non viene generato il token di stop. Il meccanismo appena descritto è chiamato **autoregressione**.

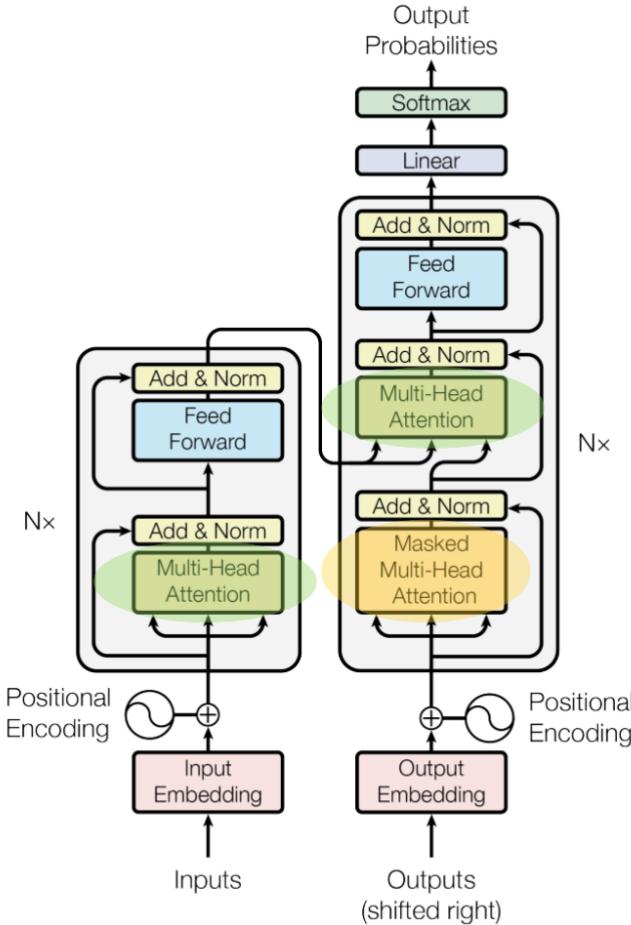


Figura 1.4: Meccanismi di attenzione presenti in un Transformer [1]

Come mostrato in Figura 1.4, all'interno dell'architettura il meccanismo di attenzione è applicato tre volte in modi leggermente differenti tra loro, in particolare:

1) **Multi-Head Attention - Encoder:** è un meccanismo di auto-attenzione (self-attention) che permette al modello di poter individuare le dipendenza tra ogni coppia di parole che costituiscono la frase. Poichè si controlla anche la dipendenza che c'è tra una parola e tutte le altre parole che sono presenti dopo di lei (quindi future), questo meccanismo è detto **anticausale**. Questo può essere fatto poichè nell'encoder il modello ha a disposizione tutta la frase di partenza e le informazioni che si possono acquisire anche dalle dipendenze future possono essere molto utili per creare una rappresentazione corretta della frase di input.

2) **Masked Multi-Head Attention - Decoder:** è una variazione rispetto al meccanismo precedente poichè questa volta sarà **causale**, questo perchè con l'autoregressione il modello può vedere solo come le paro-

le precedenti influenzano quelle successive. Si vedrà più avanti che, per implementare quanto appena descritto, verrà utilizzata una maschera.

3) **Multi-Head Attention - Decoder**: all'interno del decoder è presente anche la multi-head attention che, a differenza di quella presente nell'encoder, è più una sorta di mutua-attenzione perchè essa prende in input sia elementi provenienti dal top encoder e sia elementi provenienti dalla parte sottostante del decoder. La cosa più importante che accade in questo attention-layer è la combinazione tra la rappresentazione della frase iniziale, costruita internamente dall'encoder, e la rappresentazione interna di tutto ciò che il decoder ha generato fino a quel momento con la successiva applicazione del Masked Multi-Head Attention layer descritto precedentemente.

Gli elementi che vengono passati tra le diverse parti che costituiscono l'architettura verranno discusse successivamente.

1.4.1 Tokenization

Adesso, verrà descritto come viene costruito l'input che verrà successivamente fornito all'architettura. Le reti transformer operano su “**subword units**”. Ciò vuol dire che, i loro token possono essere frammenti di parole piuttosto che parole complete. Questi elementi vengono chiamati “**tokens**” e vengono generati automaticamente utilizzando l'algoritmo chiamato “**Byte Pair Encoding**” (**BPE**) che è in grado di effettuare la segmentazione delle parole. L'algoritmo BPE è implementato nel seguente modo:

1. Viene creato un vocabolario di simboli che terrà traccia delle sub-word units permesse. Questo dizionario verrà inizializzato con caratteri singoli incluso il simbolo come “</w>” che indica la fine di una parola. Quindi, all'inizio di tutto, l'algoritmo non parte dai termini ma semplicemente dai singoli caratteri.
2. Dopodichè, l'algoritmo in maniera iterativa conta la frequenza di ogni coppia di simboli (adiacenti) all'interno di un corpus testuale molto grande e andrà a rimpiazzare all'interno del vocabolario (costruito fino a quel momento) i singoli simboli che in coppia sono più frequenti con la concatenazione dei due simboli stessi. Grazie a questo step, i nuovi token ottenuti ad ogni iterazione, verranno via via ingranditi in modo tale da raggruppare insieme tutti quei simboli che sono in genere presenti insieme all'interno delle frasi di una certa lingua. Questo passo viene ripetuto, fino a raggiungere un numero totale di tokens prefissato.

I vantaggi della tokenizzazione, sono fondamentalmente due:

1. **C'è una maggiore copertura per i termini poco frequenti oppure sconosciuti**. Ad esempio, la parola “transformers” fino a qualche anno fa non esisteva o comunque non aveva un uso particolare e

quindi se gli embeddings fossero stati costruiti a partire da dei corpus costruiti precedentemente al 2017, allora la parola “transformers” sarebbe stata molto rara. Tuttavia, il prefisso “transform” e il suffisso “ers”, non erano sicuramente rari anche prima di quell’anno, perché comunque era possibile trovarli già in molte altre parole della lingua inglese. La disponibilità di molti esempi, in cui un certo token compare, è fondamentale per permettere al modello di poter apprendere i possibili contesti di utilizzo e, di conseguenza, il suo significato all’interno delle frasi.

2. **Riduzione dello spazio necessario per la memorizzazione degli embeddings.** È facilmente comprensibile il fatto che la dimensione del vocabolario cresca in base alla dimensione del corpus considerato. Questo perchè, assumendo ad esempio che il nostro vocabolario sia formato da 1 milione di termini, se consideriamo che ciascun embedding, associato a ciascun termine, sia costituito da 512 dimensioni e che ogni dimensione sia un numero reale che occupi 4 bytes, allora bisognerebbe avere **> 2GB** di spazio per memorizzarli tutti. Invece, poichè con la sub-word tokenization, in genere si arriva ad avere, circa 50 mila token, allora lo spazio richiesto in questo caso sarebbe **> 100MB**. Inoltre, un altro aspetto importante da considerare è il fatto che in alcuni casi si utilizzano vettori più grandi di 512 per trattare testi più lunghi e quindi chiaramente questo porta ad un aumento notevole nell’occupazione di memoria necessaria. Quindi, grazie alla sub-word tokenization, per rappresentare tutto l’insieme di parole di un vocabolario di una lingua, **è come se si settasse un limite massimo non superabile in termini di memoria**, senza perdere informazione.

All’interno di tutte le sottosezioni successive, per mantenere una maggiore semplicità durante la spiegazione, si parlerà di come le **“parole”** o i **“termini”** presenti all’interno della frase di input, vengano trasformati in opportuni vettori numerici e successivamente gestiti internamente da un Transformer. In realtà, a livello di implementazione, tutto quello che verrà spiegato, invece di essere applicato a livello di singole parole, viene applicato a livello di singoli token ottenuti dal processo di tokenizzazione appena descritto.

1.4.2 Encoder

In questa sezione verrà descritto ogni modulo presente all’interno dell’encoder di un Transformer.

1.4.2.1 Positional encoding

Ora verrà illustrato il trattamento e la gestione dell’input fornito all’encoder.

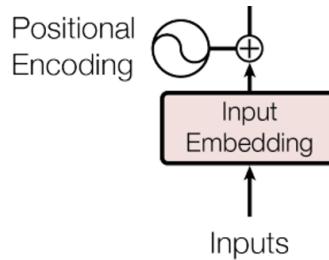


Figura 1.5: positional encoding [1]

Come è possibile notare in Figura 1.5, l'input risultante viene costruito sfruttando 3 elementi fondamentali:

- **Inputs:** Corrisponde alle parole che compongono la frase di input.
- **Input embedding:** E' un sub-layer che produce in output gli embedding dei token (token-embedding). Sempre per questioni di semplicità, in questa trattazione, invece dei token-embedding si considereranno i word-embedding. I word-embeddings sono delle rappresentazioni vettoriali di una certa parola. Ogni embedding è in grado di dirci in quale punto dello spazio a d dimensioni (nell'articolo [1] $d = 512$) essa si trova. L'embedding di una parola viene costruito partendo da un grande quantità di frasi che contengono quella parola e in base al contesto di ogni frase è possibile costruire questa rappresentazione vettoriale. Essi sono molto importanti per **tre ragioni principali**:
 1. Sono rappresentazioni dense e quindi più efficienti, perché ci permettono di risparmiare tempo di calcolo e spazio di memoria rispetto alle tecniche più semplici come la one-hot encoding, che produce vettori sparsi.
 2. Ci permettono di poter rappresentare i sensi delle parole. Questo perchè grazie al fatto di considerare tutte le frasi presenti in un corpus, nelle quali una certa parola è presente, si riescono a considerare tutti i contesti tipici di utilizzo di ciascuna parola e quindi si riesce a codificare parole che hanno significato simile con vettori simili tra loro.
 3. Gli algoritmi di Machine Learning, in particolare le reti neurali, non funzionano con il testo ma hanno bisogno di una rappresentazione numerica.

Nei Transformer, i token embeddings vengono appresi durante la fase di training attraverso il meccanismo della backpropagation, insieme agli altri parametri del modello. All'inizio del pre-training (che sarà descritto più nel dettaglio nella sottosezione 1.4.4), ogni embedding

viene inizializzato con valori casuali per ciascuna dimensione. Al termine del pre-training, tuttavia, ogni token avrà un embedding specifico, appreso in base ai dati di addestramento e rappresentativo del significato del token nel modello.

- **Positional encoding:** Meccanismo che ci permette di poter aggiungere l'informazione posizionale dei termini che compongono la frase, all'interno dei word-embedding. Questo è importante perché, assumendo che la frase di input sia questa: “Bank of America financed a repair of the river bank”. Chiaramente, ci si aspetterebbe che la rete riesca a rappresentare correttamente il fatto che la prima istanza di “bank” faccia riferimento all’ambito finanziario poichè è più vicina al termine “financed” ed è più distante da “river”. Diventa quindi abbastanza chiaro il fatto che per riuscire a modellare correttamente questa distanza tra i termini della frase, la rete debba ricevere in qualche modo in input l'informazione sulla posizione nella quale si trova ciascun termine.

Per comprendere meglio il funzionamento del Positional encoding, come riportato in [3], è possibile partire considerando la seguente frase di input:

“The **black** cat sat on the couch ant the **brown** dog slept on the rug.”

E’ possibile notare come il termine black è in posizione 2 (quindi pos=2) mentre brown in posizione 10 (pos=10). In pratica, bisogna trovare un modo per riuscire ad inserire questa informazione all’interno dei singoli word-embedding dei due termini, i quali, come è già stato detto precedentemente, sono costituiti da 512 dimensioni ($d_{model} = 512$). Per aggiungere l'informazione di cui si ha bisogno, esistono diversi modi [4] ma in questo caso verrà utilizzato l’approccio descritto in [1]. Tale approccio, consiste nell’andare a sommare tra loro queste due matrici:

$$X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1,512} \\ x_{21} & x_{22} & \cdots & x_{2,512} \\ \vdots & \vdots & \ddots & \vdots \\ x_{T1} & x_{T2} & \cdots & x_{T,512} \end{bmatrix}, \quad PEs = \begin{bmatrix} pe_{11} & pe_{12} & \cdots & pe_{1,512} \\ pe_{21} & pe_{22} & \cdots & pe_{2,512} \\ \vdots & \vdots & \ddots & \vdots \\ pe_{T1} & pe_{T2} & \cdots & pe_{T,512} \end{bmatrix},$$

ove:

- **X:** E’ la matrice di embedding dei dati. In pratica ogni riga di questa matrice corrisponde al vettore di embedding di una certa parola presente nella frase di input. ($T \times d_{model}$ dove T è il numero di termini presenti nella frase iniziale).

- **PEs:** E' una matrice che contiene per ciascuna riga, il vettore contenente l'informazione posizionale distribuita in un certo modo per ciascuna dimensione, che verrà sommato all'embedding di riferimento presente nella matrice X. ($\mathbf{T} \mathbf{x} \mathbf{d}_{\text{model}}$)

Quindi, applicando il positional encoding all'input embedding e quindi sommando le due matrici appena descritte, si otterrà questa nuova matrice sempre di dimensione $\mathbf{T} \mathbf{x} \mathbf{d}_{\text{model}}$:

$$X + PEs = \begin{bmatrix} x_{11} + pe_{11} & x_{12} + pe_{12} & \cdots & x_{1,512} + pe_{1,512} \\ x_{21} + pe_{21} & x_{22} + pe_{22} & \cdots & x_{2,512} + pe_{2,512} \\ \vdots & \vdots & \ddots & \vdots \\ x_{T1} + pe_{T1} & x_{T2} + pe_{T2} & \cdots & x_{T,512} + pe_{T,512} \end{bmatrix} \quad (1.1)$$

Per costruire ciascun vettore della matrice PEs, per ogni posizione e per ogni dimensione i-esima, vengono utilizzate le seguenti funzioni seno e coseno:

$$\text{PE}(\text{pos}, 2i) = \sin \left(\frac{\text{pos}}{10000^{\frac{2i}{d_{\text{model}}}}} \right) \text{ per i pari} \quad (1.2)$$

$$\text{PE}(\text{pos}, 2i + 1) = \cos \left(\frac{\text{pos}}{10000^{\frac{2i}{d_{\text{model}}}}} \right) \text{ per i dispari} \quad (1.3)$$

ove:

- **pos:** indica l'indice di posizione della parola che si sta considerando in un certo istante (quindi, ad esempio, nel caso di "black" ci sarà $\text{pos} = 2$).
- **$d_{\text{model}} = 512$:** rappresenta la dimensionalità di ogni positional encoding vector (ovvero di ciascun pe) che si otterrà.
- **i:** indica l'i-esima dimensione nello spazio di embedding (da 0 a 511) del vettore pe.
- **2i:** indica che la funzione seno verrà applicata solamente alle dimensioni pari del pe.
- **2i+1:** indica che la funzione coseno verrà applicata solamente alle dimensioni dispari del pe.

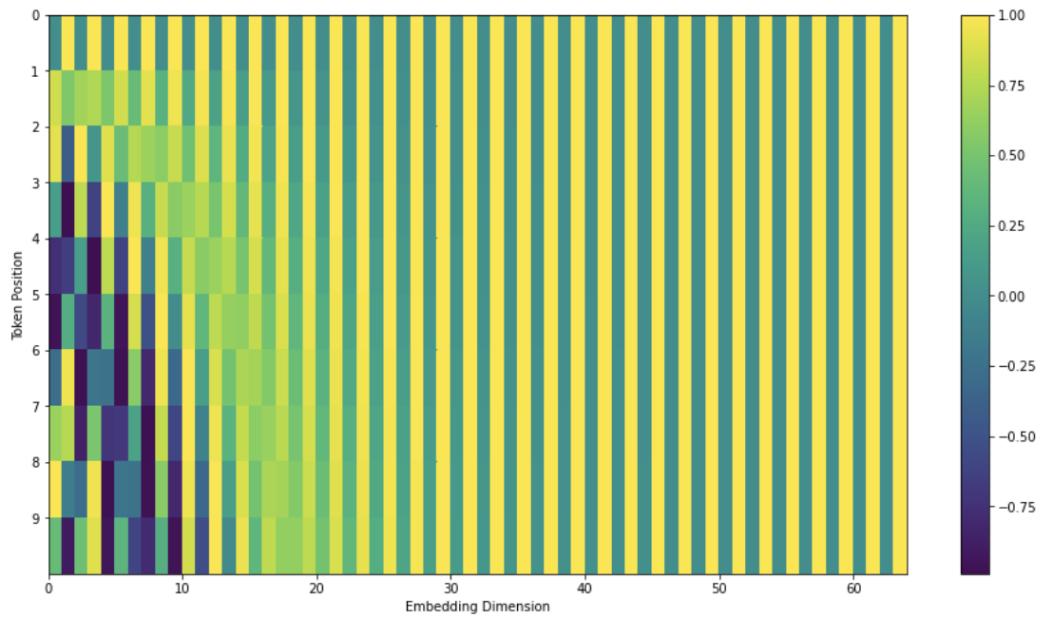


Figura 1.6: Grafico del pattern generato dall'applicazione delle funzioni (1.2) e (1.3), in funzione della posizione di ciascun token e della dimensione corrispondente all'interno del vettore [6]

In Figura 1.6, leggendo il grafico in orizzontale, è possibile notare i valori di ciascun token embedding (dal token in posizione 0 a quello in posizione 9) per ciascuna dimensione (da 0 a 65). Invece, osservando il grafico in verticale, è possibile notare come i valori lungo la prima dimensione cambino con una frequenza maggiore, quelli lungo la seconda con una frequenza leggermente minore, e così via..

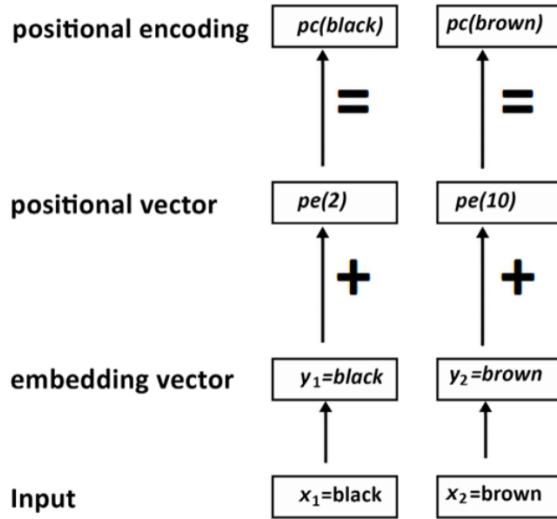


Figura 1.7: Positional encoding per le parole black e brown [3]

In Figura 1.7, vengono mostrati tutti i passaggi che vengono eseguiti per applicare il positional encoding ai termini della frase in input black e brown. Considerando solo il termine black:

- 1) **Input:** E' appunto il termine della frase iniziale che si sta considerando (ad es. black).
- 2) **embedding vector:** Si considera l'embedding corrispondente al termine di input (y_1).
- 3) **positional vector:** Si crea il positional vector calcolando ogni valore per ciascuna dimensione da 0 a 512, applicando la Formula 1.2 per le dimensioni pari e la Formula 1.3 per le dimensioni dispari. In questo caso, poichè black è la seconda parola della frase in input, si fissa **pos=2** e per questo motivo, nell'immagine 1.7, il vettore risultante è chiamato pe(2). Questo pe(2) è esattamente la seconda riga della matrice PEs descritta precedentemente.
- 4) **positional encoding:** A questo punto il vettore finale di positional encoding può essere calcolato con la seguente somma vettoriale:

$$pc(\text{black}) = y_1 + pe(2)$$

Poichè il vettore $pc(\text{black})$, con l'aggiunta dei valori provenienti dal $pe(2)$ potrebbe perdere troppa informazione proveniente dal word-embedding di partenza (y_1), quello che si fa nella pratica è amplificare i valori presenti in (y_1) eseguendo questa moltiplicazione:

$$y_1 = y_1 \cdot \sqrt{d_{\text{model}}}$$

Quindi, in realtà:

$$pc(\text{black}) = (y_1 \cdot \sqrt{d_{\text{model}}}) + pe(2)$$

Lo stesso ragionamento si applica per il calcolo di $pc(brown)$ e ovviamente per tutti gli altri termini della frase, in questo modo si otterrà la matrice finale di positional encoding mostrata precedentemente (1.1).

A questo punto, per avere un'idea complessiva di come il positional encoding vada effettivamente a modificare gli embeddings di partenza, si possono considerare i seguenti valori di similarità del coseno calcolati tra i vettori associati ai termini black e brown (utilizzando sempre la notazione di Figura 1.7). In questo esempio, i word embedding iniziali (y_1, y_2), sono stati ottenuti da Word2Vec [3]:

$$\text{cosine_similarity}(y_1, y_2) = 0.99987495 \quad (1.4)$$

$$\text{cosine_similarity}(pe_1, pe_2) = 0.8600013 \quad (1.5)$$

$$\text{cosine_similarity}(pc(\text{black}), pc(\text{brown})) = 0.9627094 \quad (1.6)$$

E' possibile notare come il valore di similarità tra i word embedding di partenza sia molto elevato (1.4). Questo accade perché essendo i due termini entrambi dei colori, è molto probabile che all'interno del corpus di partenza, utilizzato per costruire gli embedding, essi si trovavano spesso in contesti simili tra loro. E' possibile anche notare però, che il valore di similarità tra i positional vector è invece inferiore (1.5) rispetto a quello dei word-embedding. Infine, è possibile notare come il valore di similarità tra i vettori di positional encoding (1.6) risulta essere più basso rispetto a quelli dei word-embedding. Questo è un chiaro segno del fatto che, all'interno dei positional vectors, è vero che è sempre presente l'informazione sul fatto che le due parole iniziali sono semanticamente simili tra loro, ma poichè queste appaiono nella frase iniziale, ad una distanza non trascurabile l'una dall'altra, allora la codifica di questa informazione aggiuntiva, produce un abbassamento della similarità tra i due vettori ottenuti dopo l'applicazione del positional encoding. E' possibile quindi affermare che, grazie al meccanismo del positional encoding, è possibile aggiungere l'informazione posizionale dei termini che compongono la frase di partenza, all'interno dei word-embedding iniziali.

1.4.2.2 Sublayer-Multi-head attention

Adesso verrà descritto il meccanismo di funzionamento della Multi-head attention, facendo riferimento in particolar modo a quella presente all'interno dello stack di encoder.

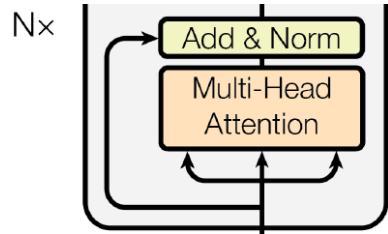


Figura 1.8: Multi-head attention - encoder [1]

Prima di arrivare alla Multi-Head Attention, è necessario comprendere come il meccanismo di attenzione che è implementato all'interno dei Transformers. Questo è implementato mediante il cosiddetto **“Scaled Dot-Product Attention”**.

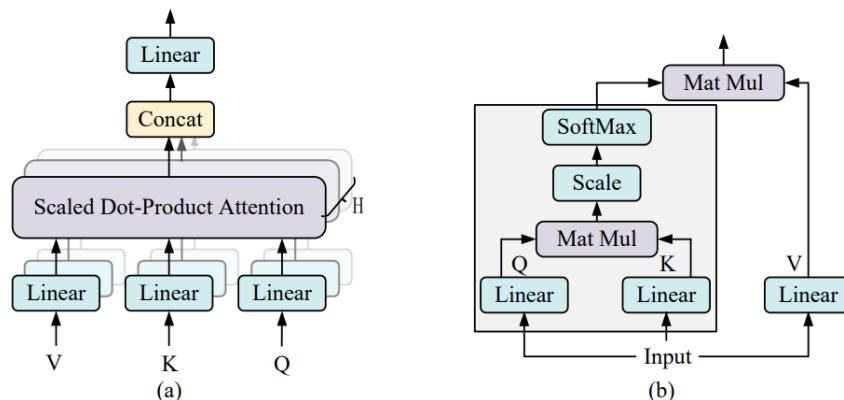


Figura 1.9: (a) Multihead self-attention module. (b) Scaled dot-product attention (Self-Attention) [5]

Come mostrato in Figura 1.8 l'input del Multi-head Attention è composto da 3 elementi. Questi elementi sono esattamente le tre matrici Q , K e V presenti nell'immagine (b) in Figura 1.9. In particolare, Q è detta **matrice delle queries**, K è chiamata **matrice delle chiavi** e V è la **matrice dei valori**. Per calcolare le matrici appena citate si utilizzano altre 3 matrici di pesi, che sono $W^{(Q)}$, $W^{(K)}$ e $W^{(V)}$. Il calcolo avviene nel seguente modo:

$$Q = X \cdot W^{(Q)}$$

$$K = X \cdot W^{(K)}$$

$$V = X \cdot W^{(V)}$$

ove:

- **X:** Corrisponde alla matrice di input, ottenuta dopo l'applicazione del positional encoding (1.1), ha dimensione pari a $(T \times d_{\text{model}})$, con

T pari al numero di termini presenti nella frase di input che si sta considerando.

- $W^{(Q)}$, $W^{(K)}$ e $W^{(V)}$: Matrici apprese durante la fase di pre-training del modello. Nell'articolo originale [1], esse hanno dimensione pari a $(d_{\text{model}} \times d_k)$, con $d_k = 64$.

Il motivo principale per il quale vengono eseguiti questi prodotti, che sono semplicemente delle trasformazioni lineari, è che in questo modo è possibile proiettare l'input (X) all'interno di uno spazio a dimensionalità ridotta. Questa riduzione è possibile notarla osservando direttamente le dimensioni delle singole matrici in gioco.

In particolare, si parte con le matrici:

$$X \in \mathbb{R}^{T \times 512}, \quad W^{(Q)} \in \mathbb{R}^{512 \times 64}, \quad W^{(K)} \in \mathbb{R}^{512 \times 64}, \quad W^{(V)} \in \mathbb{R}^{512 \times 64}$$

Per poi calcolare le seguenti matrici:

$$Q \in \mathbb{R}^{T \times 64}, \quad K \in \mathbb{R}^{T \times 64}, \quad V \in \mathbb{R}^{T \times 64}$$

È quindi facile notare che, partendo da X , dove ogni termine veniva rappresentato su 512 dimensioni, si è passati a Q , K e V , dove quindi ogni riga è adesso rappresentata su 64 dimensioni.

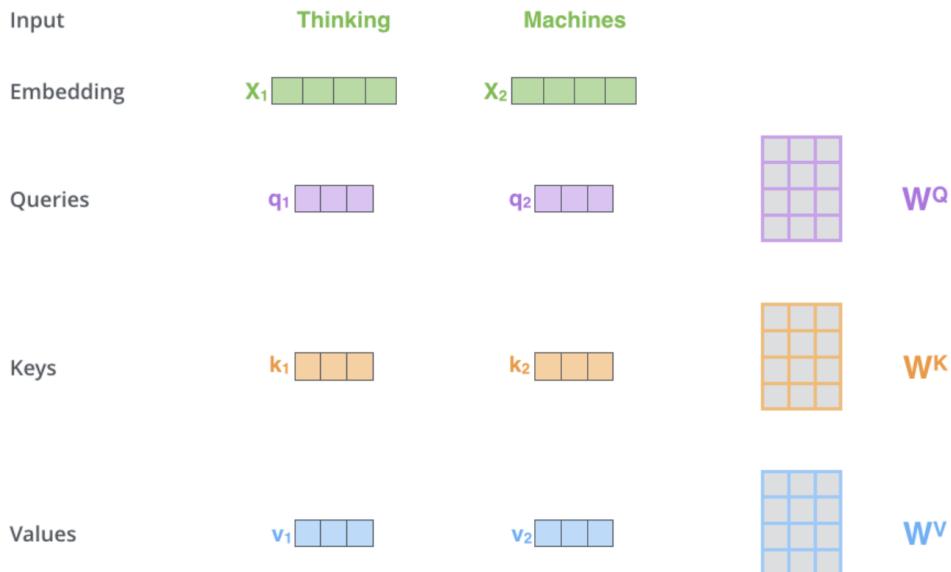


Figura 1.10: Rappresentazione riassuntiva dei vettori delle matrici principali Q , K e V e dei pesi $W^{(Q)}$, $W^{(K)}$ e $W^{(V)}$ [6]

In Figura 1.10, è mostrato un semplice esempio grafico che permette di comprendere meglio quali sono gli elementi di base che vengono costruiti con

la Self-Attention. In particolare, la frase di input considerata è costituita solo da due termini “Thinking” e “Machines”. In verde sono rappresentati gli embeddings di questi due termini, in viola i vettori di query che costituiranno le righe della matrice Q , in arancione quelli che costituiranno le righe della matrice K e in azzurro quelli della matrice V . Per ottenere ad esempio il vettore di query associato al primo termine della frase (“Thinking”) rappresentato da q_1 si moltiplicherà X_1 per $W^{(Q)}$. Per ottenere k_1 si moltiplicherà X_1 per $W^{(K)}$ mentre per ottenere v_1 si moltiplicherà X_1 per $W^{(V)}$. In maniera analoga, utilizzando però il vettore X_2 che è associato al secondo termine della frase, verranno calcolati i vettori q_2 , k_2 e v_2 che costituiranno rispettivamente la seconda riga delle matrici Q , K e V . I vettori “query”, “key” e “value” sono delle astrazioni utili per calcolare e comprendere il meccanismo di attenzione [6].

Una volta calcolate le matrici Q , K e V , si può procedere con il vero e proprio calcolo degli score di attenzione. Per ottenerli, si applica quella che probabilmente è la formula più importante per riuscire a comprendere come un Transformer riesca a rappresentare e sfruttare internamente le informazioni riguardanti le relazioni tra i termini di una frase:

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^\top}{\sqrt{d_k}} \right) V \quad (1.7)$$

La formula appena mostrata, è un modo compatto per riassumere tutti i passaggi descritti nell’immagine (b) in Figura 1.9 che **può essere suddivisa in 4 step fondamentali**:

1. **Moltiplicazione matriciale tra Q e K^\top :**

$$QK^\top = i \begin{bmatrix} Q_i \cdot K_j^\top \end{bmatrix}_j \quad \text{dove } i, j = 1, \dots, T$$

Questo calcolo determina, **per ogni termine della frase di input**, quanto sia importante (o rilevante) ogni altro termine all’interno della frase stessa, incluso il termine corrente. In altre parole, più grande sarà il valore ottenuto dal prodotto scalare tra Q_i e K_j^\top e maggiore sarà la correlazione tra i due termini t_i e t_j . Di conseguenza, il termine t_j fornirà un contributo più significativo nell’aggiornamento del significato del termine t_i . Il risultato di questo prodotto, sarà una matrice di dimensione $(T \times T)$. Questa matrice è anticausale poiché, come menzionato all’inizio di questa sottosezione, essendo parte del blocco di Multi-Head Attention dell’encoder, considera anche le dipendenze tra una parola e quelle che la seguono.

2. **Scaling:** Ogni valore presente nella matrice risultante dallo step precedente, viene diviso per $\sqrt{d_k}$. Questo scaling viene eseguito per garantire una maggiore stabilità dei gradienti, poiché per valori elevati

di d_k , i prodotti scalari eseguiti nello step precedente, potrebbero assumere valori troppo elevati e di conseguenza questo porterebbe la softmax in regioni dove i gradienti risulterebbero essere troppo piccoli riducendo quindi l'efficacia dell'apprendimento [1].

3. **Softmax:** Viene applicata la funzione softmax su ciascuna riga della matrice ottenuta dallo step 1 e sulla quale è stato già applicato lo scaling. In questo modo, ciascuna riga diventerà una distribuzione di probabilità che ci dirà, per ciascuna parola lungo le righe, quanto è probabile che ogni altra parola (compresa se stessa), rappresentata lungo le colonne, sia importante per essa.
4. **Moltiplicazione con V:** Nell'ultimo step viene eseguita sempre la moltiplicazione riga per colonna tra la matrice ottenuta dagli step precedenti e la matrice dei valori V . Il risultato di questo prodotto sarà una matrice di dimensione $(T \times 64)$. Per spiegare meglio questo step, è utile mostrare le matrici in gioco:

$$\begin{aligned} \text{Softmax}(QK_{\text{scaled}}^{\top}) &= \begin{bmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,T} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,T} \\ \vdots & \vdots & \ddots & \vdots \\ p_{T,1} & p_{T,2} & \cdots & p_{T,T} \end{bmatrix} \quad V = \begin{bmatrix} v_{1,1} & v_{1,2} & \cdots & v_{1,64} \\ v_{2,1} & v_{2,2} & \cdots & v_{2,64} \\ \vdots & \vdots & \ddots & \vdots \\ v_{T,1} & v_{T,2} & \cdots & v_{T,64} \end{bmatrix} \\ \Rightarrow Z &= \begin{bmatrix} z_{1,1} & z_{1,2} & \cdots & z_{1,64} \\ z_{2,1} & z_{2,2} & \cdots & z_{2,64} \\ \vdots & \vdots & \ddots & \vdots \\ z_{T,1} & z_{T,2} & \cdots & z_{T,64} \end{bmatrix} \end{aligned} \tag{1.8}$$

Dalla moltiplicazione tra la matrice $\text{Softmax}(QK_{\text{scaled}}^{\top})$ con la matrice V , si ottiene una nuova matrice di dimensione $T \times d_v$, chiamata Z . Poiché il prodotto consiste nel moltiplicare ogni riga della matrice sinistra per ogni colonna della matrice V , ciò che effettivamente si fa è pesare, utilizzando le distribuzioni di probabilità, ogni termine rappresentato lungo le righe di V , nelle sue diverse dimensioni, in cui i termini iniziali x_i erano stati proiettati. Questo accade perché, eseguendo ad esempio il prodotto scalare tra la prima riga della matrice sinistra e la prima colonna di V , il valore di $v_{2,1}$ potrà essere amplificato o meno in base al valore di $p_{1,2}$, a seconda di quanto la seconda parola della frase sia correlata (o rilevante) rispetto alla prima. Quindi, maggiore sarà la dipendenza tra la prima e la seconda parola, più grande sarà il valore di $p_{1,2}$ e quindi maggiore sarà il contributo che $v_{2,1}$ fornirà nel calcolo complessivo dell'attenzione $z_{1,1}$.

Questo contributo maggiore può essere interpretato come un “avvicinare” la prima dimensione, della prima riga della matrice di Self-

Attention (associata al primo termine della frase) al significato rappresentato dalla prima dimensione del secondo termine della frase. Questo perché, spesso le dimensioni degli embedding, tendono a rappresentare le direzioni dello spazio di rappresentazione associandogli un certo significato semantico. Generalizzando quello appena descritto, per tutti i valori $z_{i,j}$, è possibile affermare che:

“Il meccanismo di Self-Attention si pone come obiettivo il fatto di focalizzare maggiormente “l’attenzione” sui values (embedding rappresentati lungo le righe di V) che sono associati alle keys (embedding rappresentati lungo le colonne di K^\top) che sono più simili a ciò che ci interessa in un certo istante (query), ovvero ad un certo termine rappresentato da un embedding lungo una particolare riga di Q”.

$$\begin{array}{c}
 \mathbf{X} \quad \mathbf{W}^{\mathbf{Q}} \quad \mathbf{Q} \\
 \text{---} \times \text{---} = \text{---} \\
 \begin{matrix} \text{green} & \text{purple} & \text{purple} \\ 4 \times 4 & 4 \times 3 & 4 \times 3 \end{matrix}
 \end{array}$$

$$\begin{array}{c}
 \mathbf{X} \quad \mathbf{W}^{\mathbf{K}} \quad \mathbf{K} \quad \mathbf{Q} \quad \mathbf{K}^{\top} \quad \mathbf{V} \\
 \text{---} \times \text{---} = \text{---} \quad \text{---} \times \text{---} = \text{---} \\
 \begin{matrix} \text{green} & \text{orange} & \text{orange} & \text{purple} & \text{orange} & \text{blue} \\ 4 \times 4 & 4 \times 3 & 4 \times 3 & 4 \times 3 & 3 \times 3 & 4 \times 3 \end{matrix}
 \end{array}$$

$$\text{softmax} \left(\frac{\mathbf{Q} \times \mathbf{K}^{\top}}{\sqrt{d_k}} \right) = \mathbf{Z}$$

$$\begin{array}{c}
 \mathbf{X} \quad \mathbf{W}^{\mathbf{V}} \quad \mathbf{V} \\
 \text{---} \times \text{---} = \text{---} \\
 \begin{matrix} \text{green} & \text{blue} & \text{blue} \\ 4 \times 4 & 4 \times 3 & 4 \times 3 \end{matrix}
 \end{array}$$

Figura 1.11: A sinistra: rappresentazione delle matrici principali. A destra: calcolo della self-attention in formato matriciale [6]

Nella Figura 1.11 sono mostrati in maniera compatta, a sinistra, i calcoli matriciali con i quali si ottengono le matrici Q , K e V partendo dalla matrice dei dati X mentre a destra il calcolo riassuntivo per la self-attention dove la matrice Z corrisponde alla matrice “Self-Attention” descritta in precedenza. La lunghezza di ciascun embedding della matrice iniziale X è di 512 (4 quadratini) mentre la lunghezza degli embedding di Q , K , V e Z è 64 (3 quadratini). Il numero di righe delle matrici X , Q , K , V e Z è 2 poiché in questo esempio il numero di parole della frase iniziale erano “Thinking” e “Machines” (Quindi in questo caso $T=2$).

L'ultimo tassello mancante che permette di poter comprendere meglio il funzionamento del meccanismo di attenzione, all'interno dell'architettura Transformer, è il passaggio dalla Self-Attention (immagine (b) in 1.9) alla Multi-Head Attention (immagine (a) in 1.9). Nella pratica, invece di applicare il meccanismo di attenzione una sola volta, si è riscontrato un beneficio nel proiettare linearmente le matrici delle queries (Q), delle chiavi (K) e dei valori (V), H volte mediante l'utilizzo di diverse matrici apprese durante il training. Questo significa che, invece di applicare la self-attention una sola volta, essa viene eseguita diverse volte in parallelo (in particolare, $H=8$ nell'articolo [1]). Una rappresentazione grafica di quello appena menzionato è mostrato nella figura 1.12.

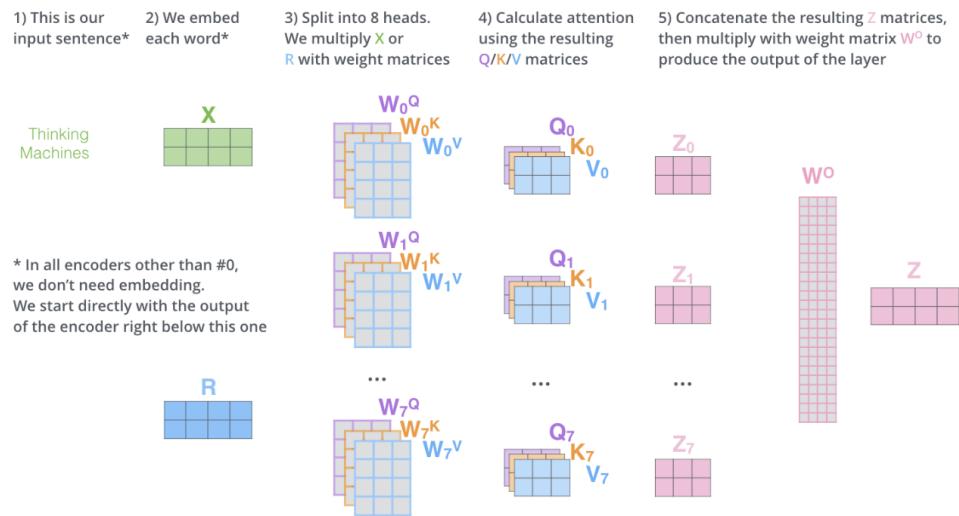


Figura 1.12: Multi-Head Attention [6]

Sostanzialmente quello che accade con la Multi-Head Attention, può essere suddiviso in 5 step differenti (partendo da sinistra nella Figura 1.12):

- 1. Frase di input:** In questo caso si suppone che la frase iniziale sia sempre "Thinking Machines".
- 2. Embedding per ciascun termine:** Dopo aver applicato ovviamente il positional encoding descritto nella sottosezione 1.4.2.1, viene creata la matrice iniziale di embedding chiamata X, dove ciascuna riga costituisce una rappresentazione vettoriale di ciascun termine della frase e anche della sua posizione. Ciascun embedding in questa fase è composto da $d_{\text{model}} = 512$ dimensioni, quindi la matrice X ha dimensioni pari a:

$$X \in \mathbb{R}^{T \times d_{\text{model}}}$$

Poiché $T = 2$, si ottiene:

$$X \in \mathbb{R}^{2 \times 512}$$

Ovviamente, tutti gli encoders che si trovano sopra il primo, non utilizzeranno la matrice di partenza X ma riceveranno come input direttamente l'output ottenuto dall'econder del layer sottostante. Questo input, sempre in Figura 1.12, è rappresentato dalla matrice R ed essa avrà le stesse dimensione della matrice X .

3. **Split in 8 Heads:** La matrice X viene moltiplicata per 8 ($H=8$) differenti matrici di pesi chiamate:

$$W_i^{(Q)} \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_i^{(K)} \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_i^{(V)} \in \mathbb{R}^{d_{\text{model}} \times d_v}, \\ \text{con } i = 0, \dots, 7, \quad d_k = d_v = \frac{d_{\text{model}}}{H} = 64.$$

Ogni prodotto produce quindi 3 differenti matrici Q , K e V che è possibile riassumere nel seguente modo:

$$X \cdot W_i^{(Q)} = Q_i, \\ X \cdot W_i^{(K)} = K_i, \\ X \cdot W_i^{(V)} = V_i,$$

dove:

$$Q_i \in \mathbb{R}^{T \times d_k}, \quad K_i \in \mathbb{R}^{T \times d_k}, \quad V_i \in \mathbb{R}^{T \times d_v}$$

4. **Calcolo dell'attenzione:** Con ciascuna Q_i, K_i, V_i viene eseguito lo stesso calcolo dell'attenzione mostrato con la Formula 1.7 che, con una notazione leggermente differente per esplicitare il fatto che in ciascuna head si utilizzano matrici di pesi differenti, è possibile riscrivere nel modo seguente:

$$\text{Head}_i = \text{Attention}(QW_i^{(Q)}, KW_i^{(K)}, VW_i^{(V)}) \quad (1.9)$$

Come mostrato in Figura 1.12, ciascuna Head_i produrrà una determinata matrice di attenzione Z_i con dimensioni pari a $(T \times d_v)$ ovvero (2×64) .

5. **Concatenazione degli Z_i e moltiplicazione per $W^{(0)}$:** Nell'ultimo step, le 8 matrici di attenzione Z_i vengono concatenate in orizzontale tra loro per formare una nuova matrice, indicata con Z . Successivamente, la matrice Z viene moltiplicata per un'altra matrice di pesi, chiamata $W^{(0)}$. Poichè le dimensioni delle due matrici in gioco sono le seguenti:

$$Z \in \mathbb{R}^{T \times Hd_v}, \quad Z \in \mathbb{R}^{2 \times 512} \\ W^{(0)} \in \mathbb{R}^{Hd_v \times d_{\text{model}}}, \quad W^{(0)} \in \mathbb{R}^{512 \times 512}$$

Allora la matrice risultante della Multi-Head Attention sarà:

$$\text{MultiHead}(Q, K, V) = ZW^{(0)}$$

Di conseguenza, la matrice finale ottenuta con il calcolo $\text{MultiHead}(Q, K, V)$ avrà dimensioni pari a $(T \times d_{model})$, ovvero le stesse dimensioni della matrice di input X considerata nello step 2. Si può quindi osservare come, al termine del calcolo, i dati ritornino alla stessa dimensionalità di partenza. Per queste ragioni, è possibile affermare che, quello che fa il blocco di Multi-Head Attention, è trasformare la sequenza di input (rappresentata con la matrice X) in un'altra sequenza con le stesse identiche dimensioni (rappresentata dalla matrice risultante dal blocco di Multi-Head Attention).

È possibile riassumere tutto il calcolo della Multi-Head Attention, appena descritto, con un'unica formula come descritto nell'articolo [1]:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^{(0)} \quad (1.10)$$

dove:

$$\text{head}_i = \text{Attention}(QW_i^{(Q)}, KW_i^{(K)}, VW_i^{(V)}) \quad (1.11)$$

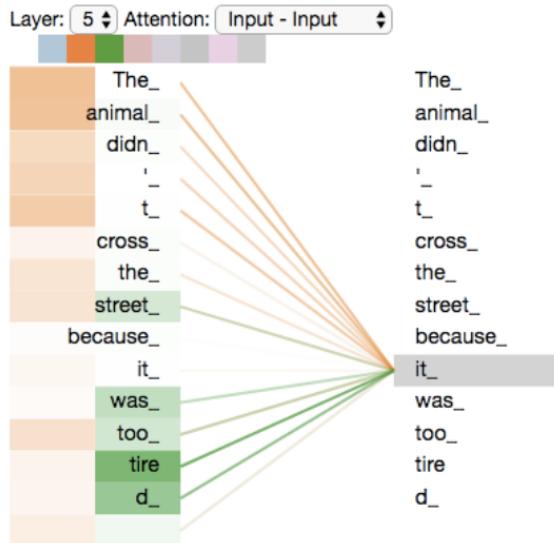


Figura 1.13: Codifica della parola “it” attraverso il contributo fornito dalle diverse heads [6]

I vantaggi principali per i quali nella pratica si implementa la Multi-Head Attention sono i seguenti:

- 1) **Maggiore precisione nelle rappresentazioni:** Aumenta la capacità del modello di poter determinare per ciascuna parola, quali sono gli

aspetti più importanti e rilevanti della sequenza di input che possono aiutarlo a codificare al meglio quella determinata parola. Quindi, permette di poter migliorare il processo di codifica dell'informazione all'interno degli embedding di ciascuna parola, grazie al fatto che con le diverse “heads”, il modello è in grado di poter esaminare l'intero contesto da prospettive diverse e quindi riesce a catturare eventuali relazioni differenti che ci sono tra la parola presa in esame in un certo istante e tutte le altre. Tutto questo è possibile grazie al fatto che, la Multi-Head Attention fornisce più “**sottospazi di rappresentazione**”, poichè, come mostrato in Figura 1.12, ogni head ha un proprio set di matrici di pesi che vengono utilizzate per proiettare gli embedding di input (o i vettori provenienti dagli encoder/decoder inferiori) in un diverso sottospazio di rappresentazione.

In Figura 1.13, è possibile notare come, considerando come frase di input “**The animal, didn't cross the street because it was too tired**”, per la codifica del termine “it”, una head (colorata in arancione) si concentra principalmente su “The animal”, mentre un'altra (colorata in verde) si concentra su “tired”. Quindi ciò significa che per il modello, la rappresentazione della parola “it”, include sia un pò della rappresentazione di “animal” che di “tired”.

2) **Parallelismo:** Grazie al fatto di essere comunque basata sulla Self-Attention, l'approccio Multi-Head rispetto ai modelli sequenziali (come le RNN o LSTM), è altamente parallelizzabile. Questo permette di processare intere sequenze simultaneamente, migliorando l'efficienza computazionale. La Multi-Head Attention, come già descritto nelle pagine precedenti, divide il processo di attenzione in più “heads” e ciascuna testa calcola l'attenzione indipendentemente. Tutti questi calcoli matriciali possono essere quindi eseguiti in parallelo sfruttando le capacità di computazione delle GPUs. Ad esempio, con $H=8$, è possibile eseguire 8 calcoli di attenzione contemporaneamente per poi concatenare i risultati.

1.4.2.3 Post-layer Normalization

Come mostrato in figura 1.8, il sottolivello di Multi-Head Attention è seguito da un **blocco chiamato “Add & Norm”**. Inoltre, è possibile notare come ci sia una quarta freccia che dall'input entra all'interno del blocco appena menzionato. Questa freccia è chiamata “**Residual Connection**”.

In particolare, la funzione “**Add**” del blocco elabora la connessione residua proveniente dal livello di input e, per farlo, somma semplicemente alla matrice di input X (ottenuta sempre dopo l'applicazione del positional encoding) la matrice Z ottenuta in output dal layer di Multi-Head Attention descritto precedentemente. Matematicamente, è possibile quindi scrivere:

$$\text{Add}(X, Z) = X + Z = X'$$

Il motivo per il quale viene applicata questa funzione, è che essa consente al modello sia di preservare le informazioni originali dell'input ma soprattutto, grazie a questo potenziamento del segnale iniziale, si riesce ad evitare la scomparsa del gradiente (vanishing gradient) durante la backpropagation. Quindi di conseguenza, permette di poter migliorare l'addestramento di reti neurali molto profonde come nel caso dei Transformers.

Per quanto riguarda invece la funzione “**Norm**”, essa si preoccupa di applicare la **layer normalization**. Questa normalizzazione è molto utile dal punto di vista del training delle reti neurali, poichè, grazie al fatto che i valori di output dei nodi di ogni layer, nel complesso, dovranno avere per forza media 0 e varianza 1, gli aggiornamenti ottenuti dai gradienti, non potranno apportare grandi modifiche ai pesi della rete. Questo perchè, quando gli output di un certo layer non sono normalizzati, esempi differenti potrebbero generare valori della loss function molto diversi tra loro e questo porterebbe ad un’aggiornamento completamente diverso per i pesi della rete sfavorendo di fatti la convergenza. Poichè quindi, con la layer normalization, i gradienti risulteranno essere più stabili e consistenti, il vantaggio che si avrà, sarà che la rete potrà essere addestrata con meno iterazioni poichè essa riuscirà a convergere più rapidamente [7].

Nel caso dei Transformer, la normalizzazione viene applicata all’output del blocco di Multi-Head Attention secondo i seguenti passaggi:

1. **Input della normalizzazione:** si parte dalla matrice $X' \in \mathbb{R}^{T \times d}$, dove:

- T è la lunghezza della sequenza (numero di parole),
- d è la dimensione dell’embedding (ad esempio, $d = d_{model} = 512$).

2. **Calcolo della media e della deviazione standard per ogni riga di X' :** Per ogni riga i della matrice X' , si calcolano la media μ_i e la deviazione standard σ_i considerando tutti i valori lungo le dimensioni d :

$$\mu_i = \frac{1}{d} \sum_{j=1}^d X'_{ij}, \quad (1.12)$$

$$\sigma_i = \sqrt{\frac{1}{d} \sum_{j=1}^d (X'_{ij} - \mu_i)^2}. \quad (1.13)$$

ove:

- i indica la riga della matrice (ossia la parola i -esima della sequenza),
- j indica la colonna della matrice (ossia la j -esima dimensione dell’embedding).

3. **Normalizzazione:** Ogni valore della matrice X' viene normalizzato sottraendo la media e dividendo per la deviazione standard che sono state calcolate in precedenza per quella determinata parola i-esima:

$$\hat{X}'_{ij} = \frac{X'_{ij} - \mu_i}{\sigma_i + \epsilon}, \quad (1.14)$$

dove ϵ è un valore piccolo (es. 10^{-6}) aggiunto per evitare un'eventuale divisione per zero.

4. **Applicazione dei parametri di scala e offset:** Infine, vengono applicati due vettori di parametri appresi durante l'addestramento, $\gamma \in \mathbb{R}^d$ (fattore di scala) e $\beta \in \mathbb{R}^d$ (fattore di shift), ottenendo l'output finale della normalizzazione:

$$\text{Norm}(X')_{ij} = \gamma_j \cdot \hat{X}'_{ij} + \beta_j. \quad (1.15)$$

Dove:

- γ_j e β_j sono i parametri associati alla dimensione j -esima dell'embedding.

5. **Inizializzazione tipica di γ e β :** In genere, i parametri vengono inizializzati come segue:

- γ viene inizializzato con tutti i valori pari a 1,
- β viene inizializzato con tutti i valori pari a 0.

Ad esempio, se $d = 512$, una parte dei vettori iniziali potrebbe essere rappresentata come:

$$\gamma = [1.0, 1.0, 1.0, 1.0, \dots, 1.0] \in \mathbb{R}^{512},$$

$$\beta = [0.0, 0.0, 0.0, 0.0, \dots, 0.0] \in \mathbb{R}^{512}.$$

L'output normalizzato della Layer Normalization viene infine passato al successivo sottolivello del Transformer, che nel caso dell'encoder è un **Feed-Forward Network (FFN)**.

1.4.2.4 SubLayer-Feed-Forward Network (FFN)

L'ultimo blocco mancante, nella spiegazione dell'encoder, è quello della Feed-Forward Network. Essa è una classica rete neurale fully-connected.

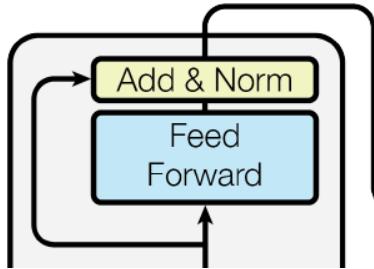


Figura 1.14: Feed-Forward Network (FFN) seguita dal blocco “Add & Norm” [1]

In particolare, come descritto sempre in [1], le caratteristiche principali di questa FFN sono le seguenti:

- L’input layer ha 512 dimensioni ($d_{model}=512$).
- L’hidden layer è composto da 2048 neuroni.
- Ciascun neurone dello strato nascosto, ha come funzione di attivazione la ReLU.
- L’output layer ha la stessa dimensionalità dell’input (d_{model}).

Date le caratteristiche della rete appena citate, e considerando il fatto che ogni singola riga della matrice X' (descritta nella sottosezione precedente) può essere trattata in maniera indipendente dalle altre, poichè in questa fase ogni parola viene considerata singolarmente, è quindi possibile descrivere l’intera rete su un singolo esempio di input nel modo seguente:

$$\text{FFN}(\mathbf{x}) = \max(0, \mathbf{x}W_1 + b_1)W_2 + b_2$$

ove:

- \mathbf{x} : Rappresenta un embedding (associato ad una certa parola) fornito in input alla rete. Quindi esso corrisponde ad una riga della matrice X' ottenuta in output dal blocco “Add & Norm” che è stato descritto nella sottosezione precedente. Quindi, ha dimensione pari a $d_{model}=512$.
- W_1 : Rappresenta la matrice dei pesi che ci sono tra il layer di input e l’hidden layer. Quindi ha dimensione pari a $(d_{model} \times 2048)$ ovvero (512×2048) .
- b_1 : Rappresenta il vettore contenente i bias di tutti i neuroni presenti nello strato nascosto. Quindi, ha dimensione sempre pari a 2048.
- W_2 : Rappresenta la matrice dei pesi che ci sono tra il layer hidden e quello di output. Quindi ha dimensione pari a $(2048 \times d_{model})$ ovvero (2048×512) .

- b_2 : Rappresenta il vettore contenente i bias di tutti i neuroni presenti nel layer di output. Quindi, anch'esso ha dimensione pari a 512.

E' possibile comunque, generalizzare considerando tutti i termini della frase, il risultato dell'intera rete nel seguente modo:

$$\text{FFN}(\mathbf{X}) = \max(0, XW_1 + b_1)W_2 + b_2$$

ove:

- X : Rappresenta in questo caso, l'intera matrice X' ottenuta in output dal blocco "Add & Norm" che è stato descritto nella sottosezione precedente. Quindi ha dimensione pari a $(T \times d_{model})$ ovvero $(T \times 512)$

Matematicamente, è quindi possibile descrivere a livello matriciale il calcolo $XW_1 + b_1$ nel modo seguente:

$$\begin{aligned} H &= XW_1 + b_1 \\ H_{i,j} &= \sum_{k=1}^{d_{model}} X_{i,k} W_{k,j}^{(1)} + b_j^{(1)} \\ H'_{i,j} &= \text{ReLU}(H_{i,j}) \end{aligned}$$

Con:

- i : Varia da 1 a T (scorre le parole della sequenza di input).
- j : Varia da 1 a 2048 (scorre i nodi dello strato nascosto).
- k : Scorre le dimensioni dell'embedding.

Dove:

- H : E' l'output della trasformazione lineare prima della funzione di attivazione.
- $H_{i,j}$: Rappresenta l'output del nodo j -esimo del layer nascosto (senza l'applicazione della funzione ReLU), ottenuto con il prodotto scalare tra la riga i -esima della matrice x (i -esima parola della sequenza) e la colonna j -esima della matrice W_1 (che contiene i pesi entranti nel nodo j -esimo dello strato nascosto).
- $H'_{i,j}$: Output del neurone j -esimo dello strato nascosto, dopo l'applicazione della ReLU, per la parola i -esima della sequenza di input.

Il calcolo complessivo della FFN(X) sarà quindi:

$$\begin{aligned} \text{FFN}(X) &= Y = H'W_2 + b_2 \\ Y_{i,m} &= \sum_{j=1}^{2048} H'_{i,j} W_{j,m}^{(2)} + b_m^{(2)} \end{aligned}$$

Con:

- m : Varia da 1 a $d_{model} = 512$ (scorre i neuroni dello strato di output).

Dove:

- H' : Matrice di output dell'hidden layer ($T \times 2048$) che contiene quindi, per ogni riga, quindi per ciascuna parola della frase iniziale, i valori di output di ogni neurone dello strato nascosto.
- Y : Matrice di output finale della Feed-Forward Network (FFN). Ciascuna riga Y_i , conterrà, i valori del layer di output per la parola i -esima della frase di input.
- $Y_{i,m}$: Output del nodo m -esimo del layer di output, ottenuto con il prodotto scalare tra la riga i -esima della matrice H' e la colonna m -esima della matrice W_2 , sommando il bias b_2 .

Come illustrato in Figura 1.14, l'output della FFN viene dato in input ad un nuovo blocco “Add & Norm” che, sempre con l'applicazione della residual connection, in maniera analoga a quanto già descritto nella sottosezione precedente, eseguirà:

1. $\text{Add}(X, Y) = X + Y = Y'$
2. $\text{Norm}(Y')_{ij} = \gamma_j \cdot \hat{Y}'_{ij} + \beta_j$

La matrice Y' ottenuta in output dall'encoder dopo il modulo “Add & Norm”, avrà sempre dimensione pari a ($T \times d_{model}$).

È quindi possibile affermare che, nel complesso, l'encoder convertirà la sequenza di input (rappresentata dalla matrice X iniziale) in una nuova sequenza (rappresentata dalla matrice Y'), che manterrà la stessa dimensionalità della matrice di partenza. Tuttavia, terrà conto della self-attention, modellando le dipendenze mutue tra i termini della frase di input. In particolare, grazie all'approccio anticausale, per ogni parola verranno individuate possibili dipendenze non solo con parole precedenti, ma anche con quelle successive.

1.4.3 Decoder

In questa sottosezione verranno descritti i vari moduli presenti all'interno del decoder dell'architettura. Tuttavia, prima di analizzare ciascun modulo specifico, è utile fornire una descrizione generale degli elementi che compongono gli stack di encoder e decoder in un Transformer.

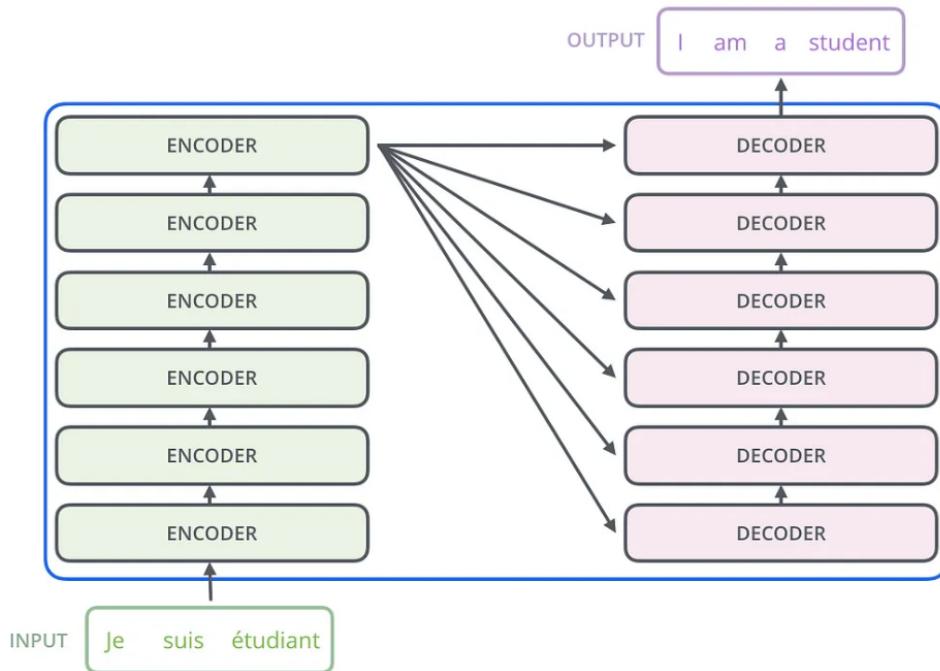


Figura 1.15: Stack di encoder e decoder [6]

Come già accennato all'inizio della sezione 1.4, in realtà ciascun encoder e decoder viene ripetuto n volte, in particolare, nell'articolo originale [1] $n=6$. Ciascun encoder quindi, produrrà in output la propria matrice Y' e dopodichè questa verrà fornita in input all'encoder del livello superiore. Questo stack è stato implementato per permettere a ciascun encoder di poter apprendere una rappresentazione differente per la frase di input, tramite il meccanismo della Multi-Head Attention. In questo modo quindi, ciascun encoder, modificherà man mano, in un certo modo, l'insieme degli embeddings associati ai termini di input. Inoltre, come mostrato in Figura 1.15, l'output prodotto dall'ultimo encoder dello stack verrà fornito in input a ciascun decoder.

1.4.3.1 Sublayer-Masked Multi-Head Attention

In questa sottosezione verrà descritto il funzionamento del Masked Multi-head Attention, il primo modulo presente in ciascun decoder.

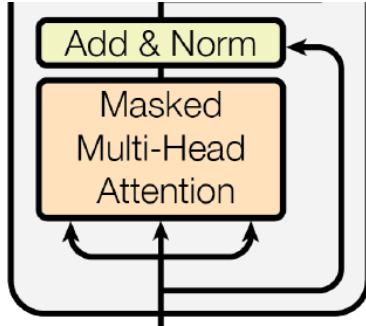


Figura 1.16: modulo di Masked Multi-Head Attention [1]

Il meccanismo **Masked Multi-head attention**, differisce leggermente da quello descritto nella sottosezione 1.4.2.2, in quanto, con “Masked” si intende dire che questa volta il meccanismo di Multi-head attention, implementata nel decoder, sarà **causale**. Sostanzialmente, questo significa che, durante il calcolo delle dipendenze che ci sono tra una certa parola e tutti gli altri termini della frase, **non bisogna permettere al decoder di poter prendere in considerazione i termini successivi alla parola considerata in quel determinato momento**. Questo viene fatto poichè, in fase di inferenza, il compito del decoder sarà appunto quello di generare la parola successiva avendo a disposizione solo tutti i termini già generati fino a quel momento, senza poter quindi considerare i termini successivi poichè essi non saranno stati ancora generati. Quanto appena detto, dal punto di vista del calcolo, si traduce nel fatto che, la formula 1.7, viene modificata nel modo seguente:

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^\top}{\sqrt{d_k}} - M \right) V \quad (1.16)$$

Quello che viene fatto quindi, è **sottrarre dalla matrice** $(\frac{QK^\top}{\sqrt{d_k}})$ **la matrice M** , dove quest’ultima è definita in questo modo (considerando direttamente il “-”):

La matrice $M \in \mathbb{R}^{T \times T}$ è detta **matrice di mascheramento causale** ed è costruita in modo tale che, per $i, j \in \{1, \dots, T\}$:

$$M_{ij} = \begin{cases} 0 & \text{se } j \leq i \\ -\infty & \text{se } j > i \end{cases}$$

Questo significa che, per ogni posizione i (quindi per ogni termine i -esimo), si impedisce di considerare i termini successivi, impostando a $-\infty$ le posizioni $j > i$. Una rappresentazione esemplificativa di M per $T = 4$ è la seguente:

$$M = \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Quindi, tutti gli elementi sopra la diagonale principale avranno valore pari a $-\infty$ mentre quelli al di sotto, saranno 0. Di conseguenza, il prodotto $\text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} - M\right)$ assicura che il decoder possa considerare solo le parole precedenti o uguali a quella corrente, garantendo il comportamento causale richiesto.

Poichè, $\text{softmax}(-\infty) = 0$, dopo aver sottratto la matrice M e applicato la softmax, si otterrà la seguente **matrice triangolare inferiore**:

$$\text{Softmax}\left(\frac{QK^\top}{\sqrt{d_k}} - M\right) = \begin{bmatrix} p_{1,1} & 0 & 0 & 0 \\ p_{2,1} & p_{2,2} & 0 & 0 \\ p_{3,1} & p_{3,2} & p_{3,3} & 0 \\ p_{4,1} & p_{4,2} & p_{4,3} & p_{4,4} \end{bmatrix}$$

A questo punto, quando si moltiplicherà la matrice appena descritta, con la matrice V per ottenere come risultato finale la matrice Z (mostrata in 1.8), ciascun embedding (ciascuna riga di Z), associato ad una certa parola della frase iniziale, non terrà conto dell'importanza (probabilità) che i termini successivi a quella parola avevano nei suoi confronti. Quindi, in questo modo, si riesce ad implementare il meccanismo causale. Invece, per quanto riguarda il modulo di “Add & Norm”, esso è implementato nello stesso modo descritto nella sottosezione 1.4.2.3.

1.4.3.2 Sublayer-Multi-head attention - Decoder

In questa sottosezione, verrà discusso in cosa consiste il passaggio dell'informazione proveniente dal top encoder al decoder.

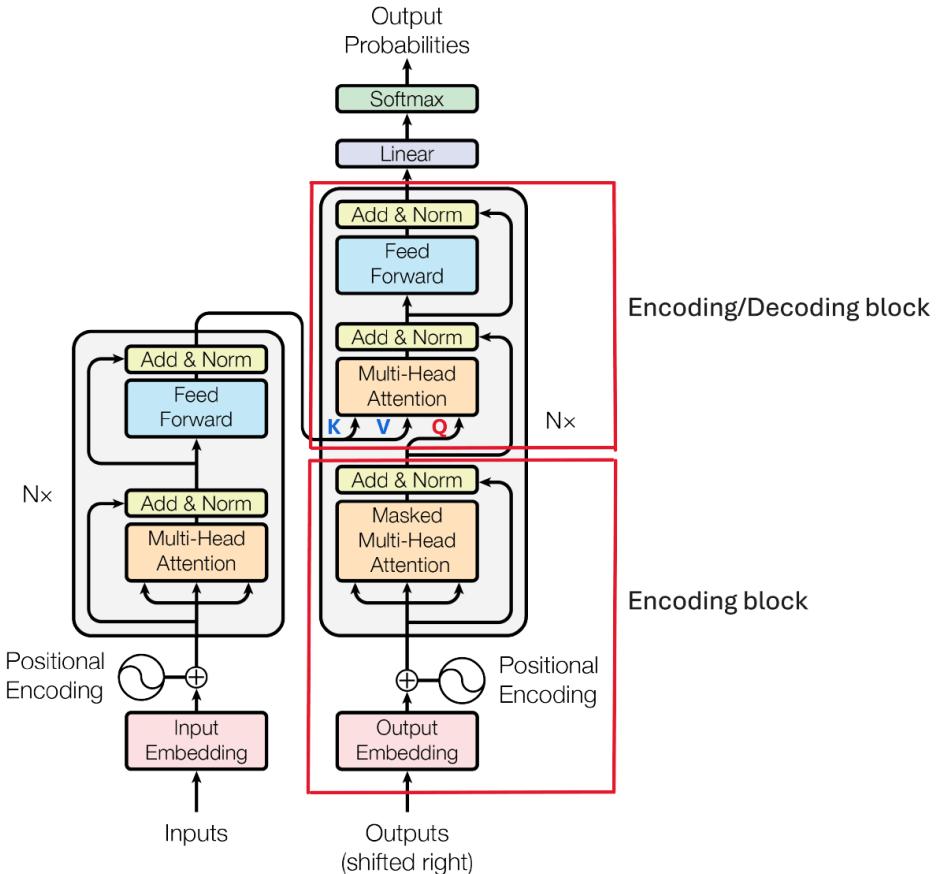


Figura 1.17: Passaggio informazioni dall'encoder (sx) al decoder (dx) [1]

La Figura 1.17, descrive l'input che verrà gestito dal sottolivello di Multi-head attention del decoder. In particolare, è possibile notare come il decoder può essere suddiviso in 2 blocchi principali:

- 1. Encoding block:** Permette al Transformer di poter costruire la rappresentazione interna (embeddings) di tutti i termini che il decoder ha generato fino a quel momento con l'applicazione della Masked Multi-Head Attention descritta nella sottosezione 1.4.3.1. In particolare quindi, come mostrato in figura 1.17, questo blocco fornisce in input al blocco superiore la matrice delle query Q.
- 2. Encoding/Decoding block:** Permette al decoder di poter combinare la rappresentazione della frase iniziale costruita dall'encoder e

la rappresentazione interna ottenuta dall'encoding block sfruttando il meccanismo della Multi-Head Attention. In questo modo, il decoder potrà considerare quali saranno i termini della frase di partenza più importanti per i termini che ha generato nella risposta fino a quel momento focalizzandosi maggiormente su di essi. Come mostrato sempre in figura 1.17, questo blocco riceve in input le matrici Q, K e V, dove:

- Q: Viene fornita dal blocco di encoding del decoder.
- K e V: Vengono calcolate sfruttando l'output del top encoder. Questo perchè in realtà, queste due matrici vengono costruite sfruttando altre due matrici di pesi, chiamate $W_{\text{enc}/\text{dec}}^{(K)}$ e $W_{\text{enc}/\text{dec}}^{(V)}$, nel seguente modo:

$$K_{\text{enc}/\text{dec}} = O_{\text{enc}} W_{\text{enc}/\text{dec}}^{(K)} \quad (1.17)$$

$$V_{\text{enc}/\text{dec}} = O_{\text{enc}} W_{\text{enc}/\text{dec}}^{(V)} \quad (1.18)$$

ove:

- $O_{\text{enc}} \in \mathbb{R}^{T \times 512}$: Output del top encoder.
- $W_{\text{enc}/\text{dec}}^{(K)}, W_{\text{enc}/\text{dec}}^{(V)} \in \mathbb{R}^{512 \times 64}$: Matrici di pesi per la trasformazione dell'output del top encoder, apprese sempre durante la fase di addestramento.
- $K_{\text{enc}/\text{dec}}, V_{\text{enc}/\text{dec}} \in \mathbb{R}^{T \times 64}$: Sono le matrici K e V mostrate in 1.17.

Quindi, utilizzando le matrici Q, $K_{\text{enc}/\text{dec}}$ e $V_{\text{enc}/\text{dec}}$, verrà applicato il meccanismo di Multi-head attention tramite le formule 1.10 e 1.11 descritte nella sottosezione 1.4.2.2. Dopodichè, per il modulo “Add & Norm” successivo verranno eseguiti di nuovo gli stessi passi descritti nella sottosezione 1.4.2.3 prendendo in considerazione in questo caso, come residual connection, l'output generato dal modulo di “Add & Norm” del sublayer sottostante (ovvero quello subito dopo il modulo Masked). Successivamente, l'output del modulo appena descritto verrà fornito in input ad un'altra FFN ripetendo gli stessi passi già descritti nella sottosezione 1.4.2.4. Infine, l'output ottenuto dal blocco encoding/decoding del decoder, diventerà l'input del sottolivello lineare.

1.4.3.3 Sublayers Linear e Softmax

In Figura 1.17, è possibile notare come il decoder abbia due livelli finali prima di individuare davvero quale sarà la nuova parola da generare. In particolare essi vengono utilizzati per queste ragioni:

- Linear:** E' una semplice rete neurale fully-connected che permette di proiettare il vettore finale (prodotto dallo stack di decoders) contenente dei valori reali, all'interno di un vettore a dimensionalità maggiore chiamato **"logits vector"**. Quindi, ad esempio, se l'architettura viene addestrata considerando come vocabolario di output un insieme di 30k parole univoche della lingua Inglese, allora il logits vector sarà composto proprio da 30k celle e ogni cella conterrà un valore di score per un certo termine specifico.
- Softmax:** Consiste nell'applicare la funzione softmax al logits vector in modo tale da ottenere una **"distribuzione di probabilità su tutti i termini del vocabolario"** preso in considerazione.

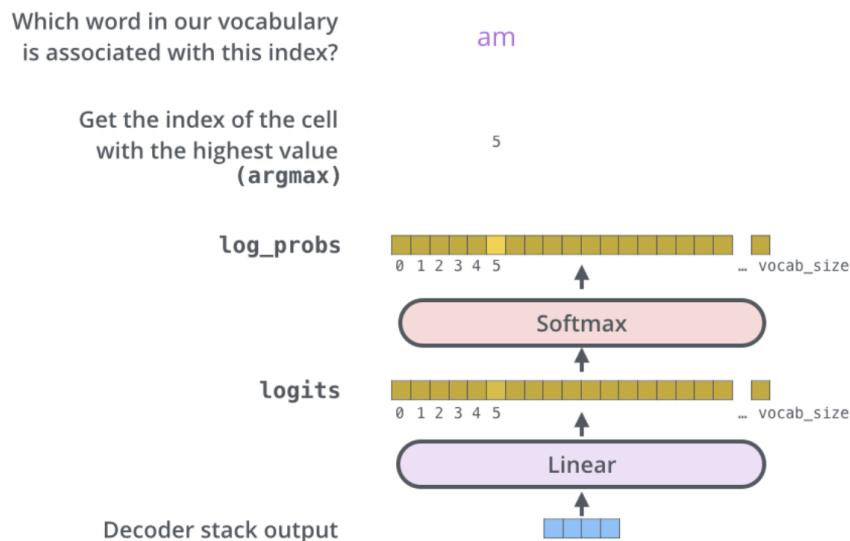


Figura 1.18: Passi per ricavare la nuova parola generata dal decoder [6]

La Figura 1.18, mostra che, una volta ottenuto l'output dal layer di softmax, a partire dalla cella con il valore di probabilità più alto, verrà ricavato l'indice del termine corrispondente nel vocabolario e con quest'ultimo verrà quindi individuata la nuova parola generata dal decoder. Dopodichè, il nuovo termine appena generato verrà riportato in input al decoder che, considerando tutta la sequenza generata fino a quel momento e sfruttando sempre le stesse informazioni provenienti dal top encoder genererà la nuova parola e così via. Proprio perchè ad ogni step, il decoder riceve in input tutto quello generato fino allo step precedente, in Figura 1.17, sotto il blocco chiamato "Output Embedding", è presente "(Shifted right)". In questo caso, il blocco chiamato "Output Embedding", durante la generazione, corrisponderà alla matrice degli embeddings dei termini che sono già stati generati dal decoder, mentre durante l'addestramento, essa corrisponderà alla matrice dei termini

corretti della frase di output (che si desidera che il transformer generi) con l'applicazione del mascheramento descritto nella sottosezione 1.4.3.1.

La generazione terminerà non appena verrà generato un token di terminazione. L'approccio appena descritto è chiamato **Greedy Search**. Questo approccio però, non è in grado di individuare la sequenza di output a livello globale veramente più probabile poiché ad ogni step prende solo il token più probabile in quel momento e quindi non esplora altre possibili sequenze per trovare quella con la probabilità complessiva massima. Per queste ragioni, esiste un altro possibile approccio chiamato **Beam Search**. In questo caso, ad ogni step, si mantengono i primi k termini più probabili che quindi producono le prime k sequenze di output più probabili. Dopodichè, per ciascuna sequenza di output costruita fino a quel momento, si aggiorna la rispettiva distribuzione di probabilità congiunta tramite l'utilizzo della seguente formula:

$$P(y_1, y_2, \dots, y_N) = \prod_{n=1}^N P(y_n | y_1, y_2, \dots, y_{n-1})$$

Dove:

- y_n è il termine i -esimo nella sequenza generata.
- $P(y_n | y_1, y_2, \dots, y_{n-1})$ è la probabilità condizionata del termine y_n , dato il contesto formato dai termini precedenti y_1, y_2, \dots, y_{n-1} .
- N è la lunghezza totale della sequenza considerata.

La formula, quindi, moltiplica le probabilità condizionate di ogni termine per ottenere la probabilità complessiva della sequenza di output \mathbf{y} . In sostanza, l'obiettivo diventa quindi quello di **massimizzare** tale formula considerando tutti i percorsi possibili che sono stati mantenuti fino alla fine. Ad ogni step, per mantenere sempre e solo le prime k sottosequenze più probabili, le sottosequenze con il valore di probabilità più basso vengono scartate, quindi non è detto che un certo percorso, scelto inizialmente, venga mantenuto fino alla fine della generazione. Alla fine, una volta che ciascuna sequenza avrà generato il token di terminazione, la sequenza con la distribuzione di probabilità congiunta maggiore, sarà effettivamente quella finale che il decoder mostrerà in output.

1.4.4 Training

Adesso che sono stati descritti tutti gli elementi principali, che permettono il passaggio dell'informazione all'interno di un Transformer nel così detto “Forward-pass”, ovvero nel processo di propagazione in avanti dell'informazione che permette al modello di poter generare un nuovo token, in questa sottosezione verrà descritto come avviene l'addestramento dell'intera rete.

Durante il training della rete, quando il modello non è ancora stato addestrato completamente, viene eseguito esattamente lo stesso Forward-pass ma con una differenza sostanziale: poichè per addestrare il Transformer si utilizza un dataset etichettato, quello che viene fatto è che **ogni volta che verrà generata una nuova parola in output, questa verrà confrontata con la parola realmente corretta e se non coincideranno, allora i pesi delle matrici W verranno aggiornati di conseguenza tramite quello che viene chiamato “Backward-pass”**.

Quindi, all'inzio, tutti i pesi della rete saranno inizializzati più meno casualmente e, tramite il passaggio all'indietro, essi verranno man mano aggiornati in funzione dell'errore che è stato commesso.

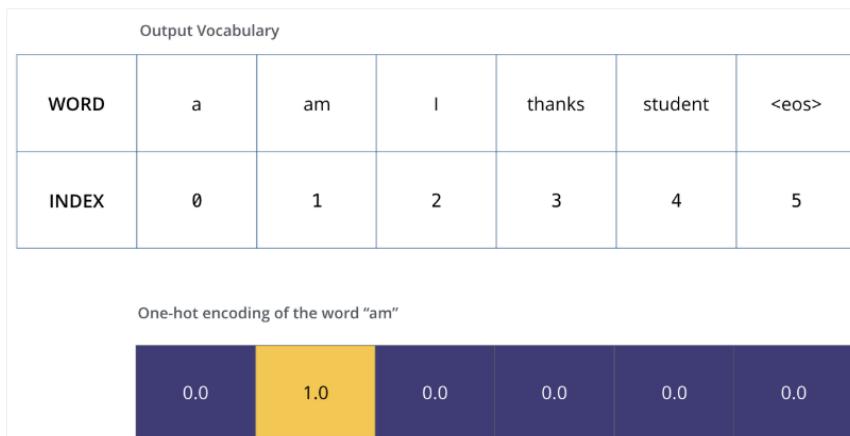


Figura 1.19: One-hot encoding per la parola “am” nel vocabolario [6]

È possibile supporre ad esempio di avere un vocabolario composto da sei termini come quelli mostrati in Figura 1.19 e di associare un vettore one-hot a ciascuno di essi.

A questo punto, per riuscire a confrontare l'output generato dalla rete e quello realmente corretto, si utilizza una **Loss-function**. Essa, indipendentemente da quale funzione specifica si decida poi di utilizzare nella pratica, ci permette di poter stimare correttamente la distanza tra l'output generato dalla rete e quello realmente corretto. Esistono diverse funzioni di loss che possono essere utilizzate, le più note sono ad esempio la Cross-Entropy [8] e la Kullback–Leibler divergence [9]. In particolare, la risposta generata dal modello sarà una distribuzione di probabilità su tutte le possibili parole del vocabolario di partenza mentre la risposta realmente corretta corrisponderà ad un vettore one-hot che avrà il valore 1 in corrispondenza del termine corretto e che quindi vogliamo che la rete generi. Nel caso dell'esempio, la lunghezza di questi vettori è 6, ma nella realtà tipicamente hanno 30k o 50k dimensioni.

All'inizio dell'addestramento chiaramente la rete non produrrà delle distribuzioni di probabilità adeguate, ma, grazie alla funzione di loss e in particolare all'algoritmo di **backpropagation**, i pesi verranno modificati in modo da rendere gli output prodotti dal modello più simili a quelli attesi.



Figura 1.20: Output della rete all'inizio dell'addestramento [6]



Figura 1.21: Confronto tra la distribuzione di probabilità desiderata (sx) e quella generata dal modello (dx) [6]

Supponendo, di voler tradurre la frase francese “Je suis étudiant”, in inglese “I am a student”, in Figura 1.21, nell’immagine di sinistra è rappresentata la distribuzione di probabilità desiderata per la generazione di ogni termine della frase in inglese (per ciascuna posizione), mentre l’immagine a destra, mostra quella generata dal modello ad ogni step. In questo esempio, si sta assumendo che, l’addestramento del modello, sia in una fase piuttosto avanzata, motivo per cui le distribuzioni di probabilità risultano essere così simili tra loro. Nonostante ciò, ci sono ancora dei valori di probabilità per certe posizioni che non sono completamente nulli, questo è in realtà un

vantaggio che ci viene fornito dalla softmax, grazie al quale il processo di addestramento può continuare a progredire.

E' possibile suddividere il processo di training delle reti transformer in 2 step distinti:

1. **Pre-training:** è una tipologia di apprendimento **self-supervised** che permette di pre-addestrare la rete transformer su una grande quantità di dati testuali in modo da permettergli di **apprendere le rappresentazioni (o pattern) generali che sono alla base del linguaggio naturale e indipendenti dalle specifiche applicazioni**. In questo modo, l'architettura impara rappresentazioni contestualizzate, cioè rappresentazioni che catturano il significato di un termine (in realtà di un token) in relazione agli altri termini (altri token) circostanti. Ci sono diverse tecniche, le più note sono il **Masked Language Modeling (MLM)** e il **Causal Language Modeling (CLM)**. In particolare, nel primo caso, durante l'addestramento, una parte dei token di input (tipicamente il 15%) vengono mascherati e il modello deve predire i token mancanti basandosi sui token che si trovano sia a sinistra che a destra rispetto a quello mascherato, e quindi, per questo motivo, questo approccio è detto **bidirezionale**. Questo metodo è ottimo per apprendere rappresentazioni semantiche più ricche, infatti, è utilizzato da BERT che è un modello di linguaggio, composto solamente dall'encoder, che al termine dell'addestramento è in grado di fornire un embedding contestuale associato a ciascun token [22]. Il MLM però non è adatto alla generazione di testo, poichè, in fase di generazione, il modello non ha a disposizione i token successivi perché questi non saranno stati ancora generati. Per questa ragione, i modelli di linguaggio generativi (come ad esempio GPT e Llama) utilizzano il CLM, poichè, essendo un approccio **causale**, data una certa sequenza di token, il modello viene forzato a predire il token successivo (mascherato) solo sulla base di quelli precedenti, in modo tale da non provocare discrepanze tra la fase di training e quella di generazione.
2. **Fine-tuning:** è invece, una tipologia di apprendimento **supervisionata** che permette a queste reti di poter **apprendere i pattern necessari per risolvere dei task specifici** andando a "raffinare" i pesi del modello. I task specifici più noti sono ad esempio: Text Classification, Question Answering, Name Entity Recognition, Machine Translation, Sentiment Analysis, **Text2SQL**, ecc.. Nel capitolo 2, verrà descritto quali sono stati i dataset costruiti ad hoc per risolvere il task principale di questo lavoro di tesi, ovvero il Text2SQL.

1.4.5 Large Language Model Meta AI (LLaMA)

I modelli Llama, sviluppati da Meta AI, sono una famiglia di modelli linguistici autoregressivi (LLM) di grandi dimensioni. Hanno come caratteristica principale il fatto di essere dei modelli **open weights**, il che significa che i pesi del modello sono accessibili liberamente al pubblico per il download (ad esempio tramite la piattaforma Hugging Face Hub) ma sono soggetti a limitazioni in termini di uso e distribuzione. In particolare, ad esempio per i modelli Llama 2 e Llama 3, Meta non ha rilasciato specifiche riguardanti i dataset che sono stati utilizzati per il pre-training così come anche il codice utilizzato per implementarli.

In questa tesi, sono stati testati i seguenti modelli Llama [18] [19] [20]:

- Llama-2-13B-hf
- Llama-3-8B-Instruct
- Llama-3-70B-Instruct

Le differenze sostanziali tra i modelli Llama-2 e Llama-3 sono le seguenti:

1. **Date di rilascio e numero di parametri:** I modelli Llama-2 sono stati rilasciati nel Luglio del 2023 con dimensioni pari a 7B, 13B e 70B, mentre i Llama-3 sono stati rilasciati nell'Aprile del 2024 con dimensioni pari a 8B e 70B.
2. **Dimensione del dataset di addestramento:** I Llama-2 sono stati pre-addestrati su 1.8 trilioni di token, mentre i Llama-3 su 15 trilioni (circa 8 volte superiore), permettendo a questi ultimi di avere una rappresentazione del parlato più approfondita e una maggiore capacità di generare risposte accurate. Tutti i dati sono stati raccolti da fonti pubbliche.
3. **Finestra di contesto:** I modelli Llama-2 supportano una finestra di contesto fino a 4.096 token mentre i Llama-3 fino a 8000 token, consentendo una gestione più efficace di testi più lunghi.

L'architettura di base per tutti i modelli è il Transformer, che è stato descritto nella sottosezione precedente, con qualche differenza:

1. **Decoder-only:** I modelli Llama sono composti solamente da uno stack di decoder, quindi sfruttano solamente la parte destra presente nell'immagine 1.3. Di conseguenza, poiché l'encoder non è presente, il blocco centrale di Multi-Head Attention seguito dal blocco di Add&Norm non è implementato nell'architettura.
2. **Pre-normalizzazione:** Per migliorare la stabilità dell'addestramento, la normalizzazione viene applicata prima di ciascun sottolivello del

transformer, a differenza di quanto descritto nella sottosezione 1.4.2.3 dove la normalizzazione veniva applicata dopo ciascun modulo di attenzione e di FFN.

3. **Funzione di attivazione SwiGLU:** Nei modelli Llama le FFN utilizzano come funzioni di attivazione la SwiGLU, a differenza della ReLU utilizzata nell’architettura originale. Questo perchè, dal punto di vista sperimentale, è stato dimostrato che la SwiGLU, produce un miglioramento delle performance.
4. **Rotary Embeddings:** Il positional encoding è stato sostituito dal dal rotary positional encoding (RoPE), descritto nell’articolo [21].

In particolare il modello Llama-2-13B-hf, a differenza degli altri due, non è stato ottimizzato appositamente per seguire le istruzioni descritte all’interno del prompt. Sostanzialmente, i modelli Llama-8B-Instruct e Llama-70B-Instruct, a differenza del primo, sono stati sottoposti ad un processo di fine-tuning supervisionato, utilizzando dataset di istruzioni e risposte, per migliorare la loro capacità di comprendere e rispondere in modo coerente alle richieste degli utenti. Nonostante questo, si è deciso comunque di testarlo in questa sperimentazione per avere un modello con una dimensione che fosse a metà strada tra gli 8B e i 70B.

1.5 Fine-tuning (LoRA & QLoRA)

I dataset utilizzati durante la sperimentazione saranno descritti nella sezione 2.2, mentre in questa sezione verranno illustrate le principali tecniche adottate per il fine-tuning dei modelli descritti nella sottosezione precedente (1.4.5). In particolare, il raffinamento dei modelli è stato ottenuto tramite le tecniche **LoRA** e **QLoRA**.

LoRA (Low-Rank Adaptation):

È una tecnica introdotta da Microsoft che consente di effettuare il fine-tuning di un modello pre-addestrato senza modificare direttamente tutti i suoi pesi [29]. È utile descrivere brevemente alcuni concetti fondamentali che serviranno durante la spiegazione della tecnica LoRA:

- **Rango di una matrice:** è il numero complessivo di righe o colonne che sono linearmente indipendenti. Ad esempio, data una certa matrice, se una colonna non può essere descritta attraverso una combinazione lineare delle colonne precedenti, allora vuol dire che questa è una colonna linearmente indipendente della matrice iniziale.
- **Matrice a rango completo:** indica che il rango della matrice corrisponde al numero minore tra le sue righe o colonne.

- **Matrice a basso rango:** in questo caso il rango della matrice è notevolmente inferiore sia al numero di righe che a quello delle colonne.

La tecnica consiste nell'andare a congelare i pesi pre-addestrati e introdurre matrici addestrabili a basso rango in ogni livello dell'architettura Transformer. Questo approccio consente di adattare il modello a nuovi compiti con un consumo di risorse computazionali significativamente ridotto. L'ipotesi di base è che, sebbene i modelli di linguaggio hanno ormai raggiunto dimensioni enormi, è possibile riuscire a specializzarli su task specifici, riuscendo ad ottenere performance simili al fine-tuning completo, sfruttando matrici di pesi a basso rango che saranno le uniche ad essere modificate durante l'intero processo di fine-tuning.

Sfruttando ad esempio le colonne linearmente indipendenti di una matrice, è possibile decomporre quest'ultima nel prodotto di due matrici più piccole, e il rango della matrice decomposta, rappresenterà sia il numero di colonne della prima matrice che il numero di righe della seconda matrice. Quindi, con un rank piccolo, è possibile ottenere un'approssimazione a basso rango di una matrice di partenza W , che è esattamente il principio sfruttato da LoRA per ridurre il numero di parametri da addestrare durante il fine-tuning di modelli di linguaggio di grandi dimensioni.

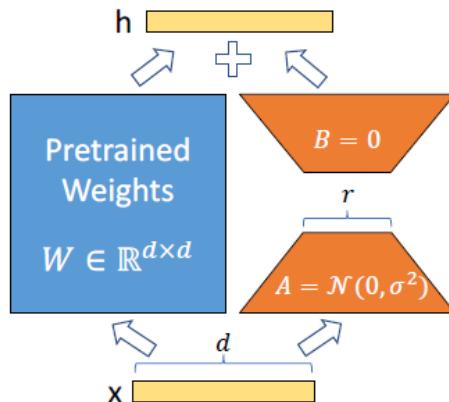


Figura 1.22: Illustrazione del meccanismo di LoRA: i pesi pre-addestrati W restano congelati, mentre solamente le matrici a basso rango A e B (inserite in parallelo rispetto a W) vengono aggiornate durante l'addestramento [29]

Come mostrato in figura 1.22, **L'idea principale di LoRA** è quella di adattare i pesi pre-addestrati W senza modificarli direttamente. Questo viene fatto aggiungendo a questi ultimi la perturbazione appresa (ΔW), calcolata con il prodotto tra le due matrici a basso rango (A e B).

A livello implementativo, le matrici A e B corrispondono ai pesi che collegano i due layer della seguente rete neurale feed-forward:

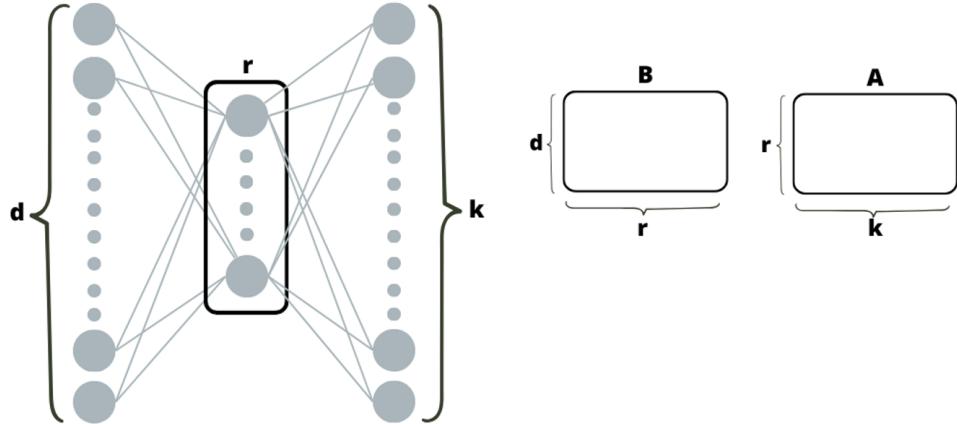


Figura 1.23: **A sinistra:** rappresentazione della rete feed-forward (FFN) utilizzata nell'addestramento con LoRA. **A destra:** struttura delle matrici A e B , che contengono i pesi delle connessioni tra i due layer della FFN. In questo caso, si assume $d = k$.

Quindi, ricapitolando:

Assumendo che sia possibile aggiornare la matrice dei pesi pre-addestrata W , utilizzando una seconda matrice dei pesi chiamata (ΔW), per adattare un modello per la risoluzione di un task specifico:

$$W_{update} = W + \Delta W$$

L'ipotesi principale di LoRA è che la matrice ΔW sia a basso rango e per questo motivo possa essere decomposta in due matrici più piccole A e B :

$$\Delta W = BA$$

dove:

- $W \in \mathbb{R}^{d \times k}$ sono i pesi pre-addestrati della rete neurale, che rimangono **congelati** durante il fine-tuning.
- $B \in \mathbb{R}^{d \times r}$ è una matrice inizializzata a zero ($B = 0$), quindi inizialmente non si introduce alcuna modifica al modello iniziale (pre-addestrato). In altre parole, il modello parte dal comportamento originale e si adatta gradualmente man mano che LoRA apprende i valori dei nuovi pesi.

- $A \in \mathbb{R}^{r \times k}$ è una matrice inizializzata casualmente i cui valori seguono la distribuzione normale $\mathcal{N}(0, \sigma^2)$, in modo tale che neuroni differenti possano apprendere caratteristiche differenti che sono utili per l'adattamento del modello.

Con l'ipotesi descritta pocanzi, in pratica si assume che le dimensioni veramente rilevanti, che dovranno essere considerate durante il raffinamento del modello, siano relativamente poche. **Il punto chiave è che, scegliendo un valore di rango $r \ll \min(d, k)$, le matrici A e B risulteranno significativamente più piccole rispetto alla matrice W , rendendo l'intera rappresentazione molto più efficiente con una conseguente riduzione dello spazio di memoria richiesto durante il fine-tuning.**

Al termine del fine-tuning, quindi dopo che A e B sono state modificate per risolvere il task specifico l'output sarà generato nel modo seguente:

$$h = Wx + \frac{\alpha}{r} \Delta Wx = Wx + \frac{\alpha}{r} BAx$$

dove:

- $x \in \mathbb{R}^d$: rappresenta l'input della rete neurale.
- BAx : rappresenta la variazione appresa tramite LoRA durante il fine-tuning, calcolata a partire dal prodotto delle matrici a basso rango B e A , applicato all'input x .
- $\frac{\alpha}{r}$: è un **fattore di scaling** che viene inserito per regolare l'intensità della modifica introdotta da LoRA. Un valore più alto amplifica la variazione BAx , mentre un valore più basso la attenua, mantenendo il modello più vicino ai pesi originali.
- $h \in \mathbb{R}^d$: è l'**output del forward-pass dopo l'applicazione di LoRA**.

I vantaggi principali della tecnica LoRA sono i seguenti:

1. **Riduzione del numero di parametri da modificare e memorizzare per il fine-tuning**, poiché, assumendo $d = 1000$ e $k = 5000$:

$$W \in \mathbb{R}^{1000 \times 5000}$$

il numero totale di parametri da considerare sarebbe **5,000,000**.

Invece, con $r = 64$:

$$B \in \mathbb{R}^{1000 \times 64}, \quad A \in \mathbb{R}^{64 \times 5000}$$

il numero totale di parametri da aggiornare sarà **384,000**, ovvero una **riduzione di circa 13 volte** rispetto al numero totale di parametri della matrice originale W . Gli autori dell'articolo [29], hanno mostrato che, applicando LoRA al modello **GPT-3 175B**, il consumo di **VRAM durante il training** è stato ridotto da 1.2TB a 350GB, con un risparmio di circa il 70.8% in termini di memoria.

2. **Maggiore velocità durante il backward-pass**, poiché vengono aggiornati solo i gradienti delle matrici A e B , evitando il calcolo, la memorizzazione dei gradienti e l'aggiornamento degli stati dell'ottimizzatore, per i pesi congelati del modello.
3. **Nessuna latenza aggiuntiva durante l'inferenza**, poiché, una volta completato il fine-tuning, sfruttando questo passaggio:

$$h = Wx + BAx = x(W + BA)$$

risulta evidente come sia utile andare a pre-calcolare l'intero set di pesi del modello pre-addestrato in questo modo:

$$W_{\text{update}} = W + BA$$

e memorizzarlo direttamente. In questo modo, durante l'inferenza, si utilizzerà direttamente W_{update} senza la necessità di calcoli aggiuntivi. Nelle ultime due formule mostrate, non è stato inserito il fattore di scaling per semplicità di annotazione, ma ovviamente nell'implementazione bisognerà considerarlo moltiplicandolo per BA .

4. **Possibilità di passare rapidamente da un task all'altro con un basso costo computazionale**, poiché è sempre possibile **sostituire i pesi LoRA** senza dover ricaricare o riaddestrare l'intero modello.

In particolare, per ripristinare la matrice dei pesi originali W , basta rimuovere la modifica introdotta da LoRA:

$$W = W_{\text{update}} - BA$$

e successivamente, per adattare il modello ad un nuovo task, è sufficiente caricare un nuovo set di matrici LoRA A' e B' e applicarle ai pesi originali:

$$W_{\text{update}} = W + B'A'$$

Questo approccio consente di commutare rapidamente tra diversi task, mantenendo inalterati i pesi pre-addestrati W e riducendo significativamente il costo computazionale rispetto ad un fine-tuning completo del modello.

A questo punto, è facilmente intuibile che aumentando il **valore di r** , le matrici B e A cattureranno informazioni più complesse durante il fine-tuning, ma l'efficienza in termini di memoria diminuirà. Al contrario, con un valore basso di r , l'efficienza aumenterà, ma le matrici dei pesi potrebbero non adattarsi correttamente al task specifico. Gli autori dell'articolo [29], consigliano di aumentare il valore di r (ad esempio settandolo a 64, 128 o 256) se il task differisce significativamente dai dati utilizzati durante il pre-training del modello. In caso contrario, sarà sufficiente settarlo a valori più bassi (1, 2, 4, 8, 16). Per quanto riguarda invece il **valore di α** , non esiste ancora una regola generale certa, poichè nell'articolo principale [29], viene settato al primo valore di r utilizzato negli esperimenti (quindi ad 1), in modo tale da poter testare le performance dei modelli solo al variare di questo iperparametro. Tuttavia, in letteratura è stato osservato che, per alcuni task, impostare $\alpha = 2r$ sia la scelta migliore in termini di performance. **Durante la fase di sperimentazione di questa tesi**, come sarà mostrato nella sezione 4.1, sono stati utilizzati i valori $r = 64$ e $\alpha = 16$ in modo tale da impostare il fattore di scaling a 0.25 per permettere al modello finale di poter conservare le capacità di interpretazione del linguaggio naturale apprese durante il pre-training.

In linea di principio, la tecnica **LoRA** può essere applicata a qualsiasi matrice di pesi in una rete neurale per ridurre il numero di parametri addestrabili. Infatti, all'interno della libreria **PEFT** di Hugging Face (utilizzata in questa tesi), nei modelli Transformer, LoRA può essere applicata sia ai pesi delle **proiezioni delle matrici di attenzione** (W^Q, W^K, W^V, W^O) all'interno di ciascuna head presente in ogni blocco di Multi-head Attention (descritto nella sottosezione 1.4.2.2), sia ai pesi delle reti feed-forward (FFN). Tuttavia, nelle sperimentazioni di questa tesi, che saranno descritte nella sezione 4.1, la tecnica LoRA è stata applicata esclusivamente alle matrici W^Q e W^V . Questo perché, come mostrato nell'articolo [29], la modifica di queste due sole matrici si è dimostrata un compromesso efficace tra la qualità dell'adattamento e l'efficienza computazionale del fine-tuning.

QLoRA (Quantized Low-Rank Adaptation):

È una tecnica che combina la quantizzazione dei parametri del modello pre-addestrato con LoRA [30]. In questo metodo, tutti i pesi del modello pre-addestrato vengono inizialmente quantizzati e memorizzati a **4 bit**, riducendo significativamente l'occupazione di memoria. Successivamente, l'addestramento degli adattatori LoRA avviene con **precisione a 16 bit floating point (FP16)**.

Durante il **forward-pass** e il **backward-pass**, i pesi del modello quantizzato vengono **temporaneamente dequantizzati**, tornando nel formato FP16 **esclusivamente per eseguire i calcoli necessari**. Questo consente di calcolare in modo più preciso le predizioni del modello (forward-pass) e i gradienti **solo per i pesi introdotti da LoRA** (backward-pass). La dequantizzazione viene applicata in modo **intelligente**, ovvero solo quando strettamente necessario, evitando di sovraccaricare inutilmente la memoria della GPU. Di conseguenza, durante tutto il processo di fine-tuning, gli unici pesi mantenuti in FP16 sono quelli degli adattatori LoRA (le matrici A e B), che, come già descritto in precedenza, occupano uno spazio nettamente inferiore rispetto all'intero modello pre-addestrato.

Un aspetto cruciale della dequantizzazione è che essa **introduce inevitabilmente degli errori**, poiché i valori quantizzati a 4 bit vengono **mappati su un numero finito di 16 possibili intervalli**, riducendo la precisione rispetto al formato FP16. Tuttavia, è stato dimostrato sperimentalmente che il processo di fine-tuning è **in grado di compensare questi errori**, permettendo di ottenere dei modelli finali, come ad esempio dei chatbot, **che raggiungono prestazioni comparabili o addirittura superiori** rispetto allo stato dell'arte [30]. Questa compensazione avviene in maniera del tutto naturale, poiché con la tecnica QLoRA, durante il fine-tuning, la backpropagation calcola i gradienti in base alla derivata della funzione di perdita, ma poiché quest'ultima è influenzata dai pesi del modello pre-addestrato (che contengono gli errori di quantizzazione), di conseguenza, i gradienti dei pesi adattivi di LoRA, saranno calcolati considerando tali errori, permettendo a questi pesi di compensarli per risolvere il task specifico. Quindi la minimizzazione della funzione di perdita avverà anche considerando gli errori di dequantizzazione. Per questo motivo, QLoRA rappresenta una strategia efficace che **concilia l'efficienza computazionale della quantizzazione con la qualità dell'adattamento**, riducendo il consumo di memoria senza compromettere le prestazioni del modello.

In questa tesi, per caricare, addestrare ed eseguire le inferenze con i LLM descritti nella sottosezione 1.4.5, è stata utilizzata la **libreria Transformers** del framework Hugging Face che si basa su **PyTorch** per la gestione dei calcoli tensoriali. Per applicare la tecnica **LoRA**, è stata utilizzata la libreria **PEFT (Parameter-Efficient Fine-Tuning)**, sempre di Hugging

Face, che implementa un insieme di tecniche per l'ottimizzazione del fine-tuning dei modelli linguistici di grandi dimensioni. Invece, per applicare la tecnica **QLoRA**, è stata utilizzata la libreria PEFT in combinazione con **BitsAndBytes**. Quest'ultima si occupa dell'effettiva **quantizzazione dei pesi a 4 bit**, riducendo drasticamente l'uso di memoria senza comprometterne la qualità, mentre PEFT gestisce l'integrazione della tecnica LoRA per il fine-tuning efficiente del modello quantizzato.

Capitolo 2

Database & Dataset

2.1 Database biologico

In questa sezione sarà descritta la struttura del database biologico basato sull’**“Italian Culture Collection Catalogue (ItCCC)”,** costruito durante il progetto “SUS-MIRRI.IT”, il quale mira a rafforzare l’infrastruttura di ricerca MIRRI-IT, che rappresenta il nodo italiano di MIRRI-ERIC. Poichè il diagramma logico completo risulta essere piuttosto grande e complesso, per questioni di chiarezza, si è deciso di suddividerlo in 7 sottodiagrammi e descrivere separatamente ciascuno di essi.

I 7 sottodiagrammi logici saranno i seguenti [25]:

1. Descrive gli attributi delle tabelle Strain, Collection, Microorganism e Virus.
2. Si focalizza sulle associazioni che ci sono tra gli strain e altre entità che descrivono alcune delle loro proprietà.
3. Descrive le associazioni tipiche degli strain, che sono specificamente dei microorganismi e che quindi non sono né virus né linee cellulari.
4. Fa riferimento alla letteratura scientifica.
5. È dedicato alle origini geografiche degli strain.
- 6-7. Descrivono i ruoli relativi alle persone che hanno contribuito alla raccolta degli strain presenti nel catalogo.

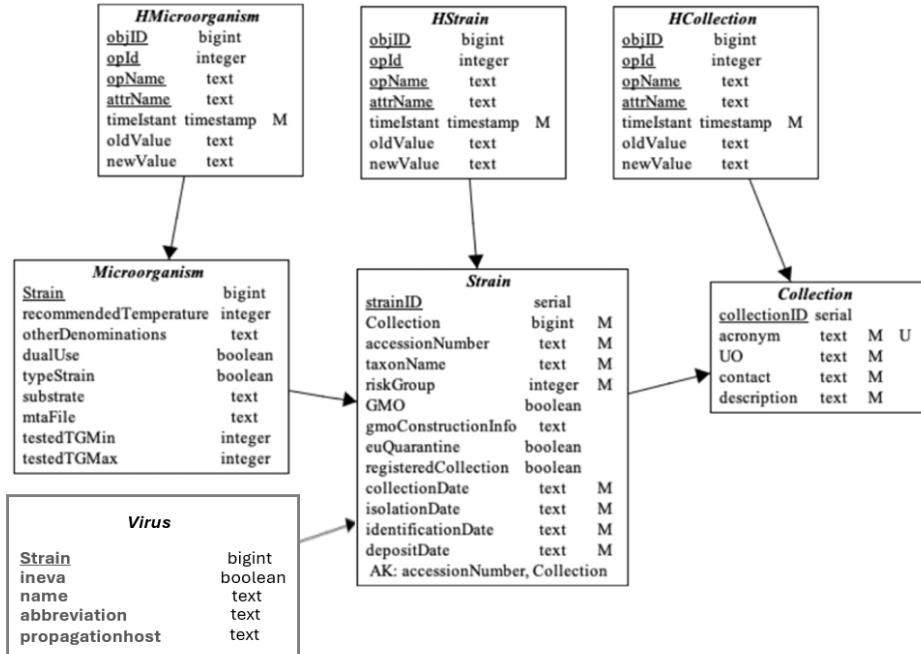


Figura 2.1: Il primo sottodiagramma logico dell'ItCCC [25]

Nel primo sottodiagramma logico mostrato in Figura 2.1, è possibile notare come all'interno dell'entità Strain ci sia il campo **strainID** che permette di identificare in maniera univoca un ceppo all'interno del database. Tale identificatore viene assegnato ad un singolo ceppo non appena questo viene caricato per la prima volta e dopodichè non potrà più essere riutilizzato o modificato. **L'accessionNumber** invece, è l'identificatore univoco obbligatorio del ceppo all'interno del catalogo. Secondo le specifiche MIRRI-IS, è un codice che non può includere spazi. Il **taxonName** include il genere, la specie e i nomi delle varianti degli strain, descrivendoli tramite l'utilizzo di termini conosciuti in ambito biologico. **GMO** è un attributo booleano che descrive se il ceppo è geneticamente modificato (le informazioni testuali, riguardanti la costruzione di un ceppo geneticamente modificato sono archiviate in **gmoConstructionInfo**). **RiskGroup** è il gruppo di rischio secondo la Direttiva UE 2000/54/CE, mentre **EUQuarantine** (di tipo booleano) indica se il ceppo è soggetto a quarantena secondo la Direttiva europea 2000/29/CE e i suoi emendamenti e correzioni. Gli attributi **collectionDate**, **isolationDate**, **identificationDate** e **depositDate**, come suggeriscono i nomi scelti, sono utilizzati per memorizzare informazioni sulla data di raccolta del ceppo, la data di isolamento del ceppo, la data di identificazione del ceppo e la data in cui il ceppo è stato depositato nella collezione. Secondo le specifiche MIRRI-IS, questi attributi non sono date classiche, ma possono essere semplicemente l'anno (come numero intero)

oppure mese-giorno-anno (data completa). Per queste ragioni, come sarà descritto nella sottosezione (2.2.2), all'interno dei dataset per il fine-tuning dei modelli, sono stati inseriti degli esempi specifici per permettere ai modelli di poter gestire tali combinazioni in maniera corretta. La tabella **Microorganism**, è sostanzialmente una specializzazione di Strain e quindi contiene al proprio interno dei campi specifici che li differenziano dai **Virus**. Alcuni di questi campi sono ad esempio i **TestedTGMin** e **TestedTGMax** che rappresentano rispettivamente la temperatura più bassa e quella più alta (entrambe sono numeri interi, in gradi Celsius) a cui il ceppo è stato testato per la crescita. Il valore booleano **dualUse** permette di specificare se il ceppo ha il potenziale per un uso dannoso secondo il Regolamento del Consiglio UE 2000/1334/CE e i suoi emendamenti e correzioni. Diversamente, **otherDenominations** è un attributo testuale utilizzato per rappresentare nomi non ufficiali che sono spesso utilizzati per il ceppo, ad esempio nelle pubblicazioni, o un nome dato al ceppo dall'isolatore prima del suo deposito nella collezione. Le entità che iniziano con la lettera **H** vengono introdotte per storizzare i dati. Un'occorrenza di HStrain, ad esempio, è identificata dal ceppo a cui si riferisce (objID), da un intero (opID) che corrisponde al tipo di operazione (inserimento, aggiornamento o eliminazione) e dall'attributo a cui si riferiscono oldValue e newValue (in formato testuale). La cosa da mettere in evidenza qui, è che **oldValue** e **newValue** non possono essere uguali e non possono essere entrambi nulli per ipotesi. Quindi, **timeInstant** è l'ora in cui è stata applicata l'operazione. Considerazioni simili si applicano a **HMicrorganism**, **HCollection** e alle altre entità di storizzazione che saranno introdotte di seguito.

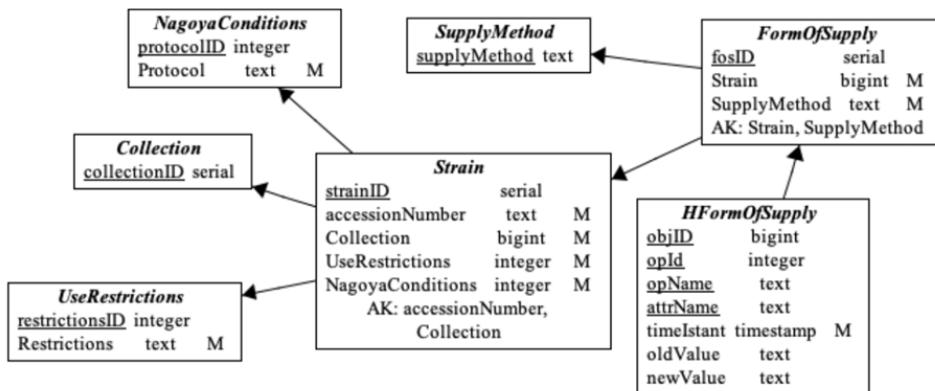


Figura 2.2: Il secondo sottodiagramma logico dell'ItCCC [25]

Nel secondo sottodiagramma logico mostrato in Figura 2.2, è utile mettere in evidenza il fatto che ogni strain sia obbligatoriamente collegato ad una singola occorrenza dell'entità **NagoyaCondition** (utilizzata per descrivere

la situazione del ceppo in relazione al protocollo Nagoya), ad una singola occorrenza dell'entità **UseRestrictions** (che segnala invece se il ceppo può essere utilizzato per lo sviluppo commerciale o meno) e ad uno o più SupplyMethod (che corrispondono alle forme di fornitura degli utenti agli utenti finali). Un'altra cosa da specificare è che l'attuale linea guida del MIRRI.IS riporta le seguenti forme di fornitura: Agar, Cryo, Dry Ice, Liquid Culture Medium, Lyo, Oil e Water. Ovviamente, questo elenco potrebbe essere facilmente esteso in base a nuove regolamentazioni.

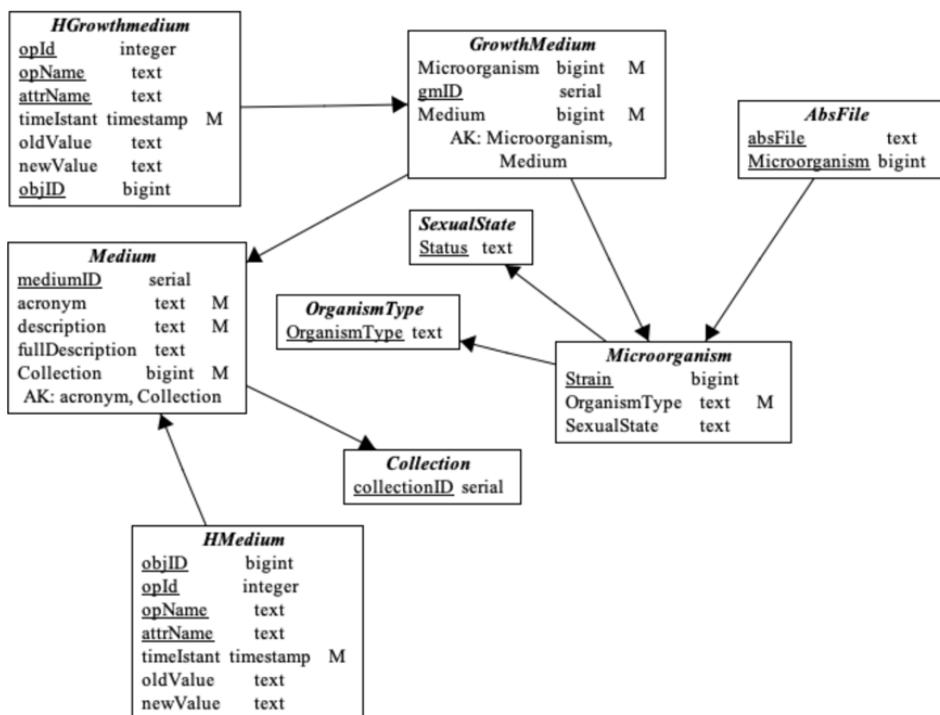


Figura 2.3: Il terzo sottodiagramma logico dell'ItCCC [25]

Nel terzo sottodiagramma logico mostrato in Figura 2.3, è possibile notare il fatto che, per rappresentare il tipo di un microrganismo, ogni occorrenza della tabella Microorganism, è collegata ad un'occorrenza di **OrganismsType**. Secondo le specifiche MIRRI-IS, vengono considerati come tipi: Algae, Archaea, Bacteria, Cyanobacteria, Filamentous Fungi, Yeasts, e Microalgae. Ogni terreno di coltura è rappresentato come un'occorrenza dell'entità **Medium**. Ciascun terreno è identificato nel database tramite **mediumID**. Inoltre, il terreno è identificato da un acronimo, univoco nella raccolta. I terreni di coltura consigliati sono quindi rappresentati tramite l'entità **GrowthMedium**. **ABSfile** invece, è un Uniform Resource Locator (URL) dei Certificati di conformità riconosciuti a livello internazionale (IRCC) che fornisce la prova che il ceppo è stato consultato in conformi-

tà con il consenso informato preventivo (PIC) e i termini reciprocamente concordati (MAT). Invece, **SexualState** definisce le informazioni sullo stato sessuale/tipo di accoppiamento del ceppo. I suoi valori possono essere: typesMata, Matalpha, Mata/Matalpha, Mata, Matb, Mata/Matb, MTLa, MTLalpha, MTLa/MTLalpha, MAT1-1, MAT1-2, MAT1, MAT2, MT+, MT-. Ovviamente, anche questa lista potrebbe essere facilmente estesa in base a nuove normative.

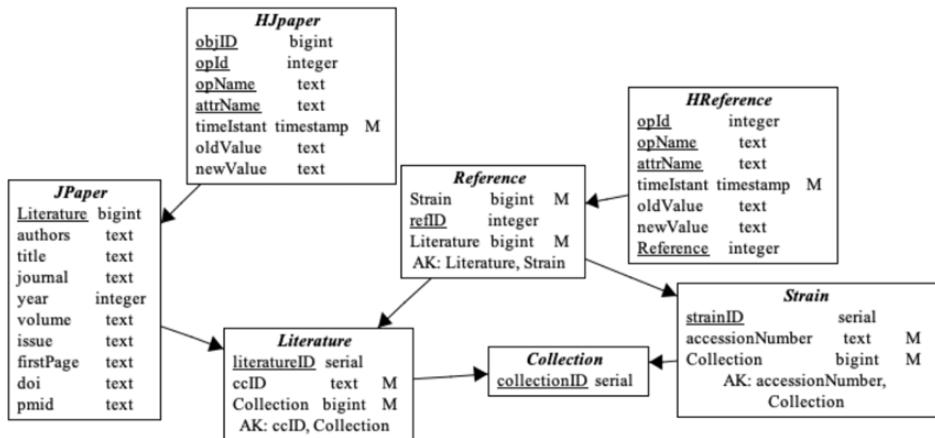


Figura 2.4: Il quarto sottodiagramma logico dell'ItCCC [25]

Il quarto sottodiagramma logico mostrato in Figura 2.4, permette di poter codificare le informazioni sulla letteratura scientifica relative ad un ceppo (indipendentemente da quale sia il suo tipo). La tabella **Reference**, permette di poter collegare ceppi e occorrenze dell'entità **Literature** direttamente alla tabella **Strain**. Ogni occorrenza di Literature è identificata sia da un identificatore assegnato da ItCCC (literatureID), sia da un identificatore univoco nella raccolta (ccID). Un'altra cosa da notare è che, tenendo conto dei risultati dell'analisi iniziale dei dati gestiti dalle raccolte, si è deciso di introdurre l'entità **JPaper**, utilizzata per rappresentare le informazioni principali di un articolo scientifico pubblicato su una rivista. Il significato dei suoi attributi è abbastanza autoesplicativo. E' stato deciso di rappresentare le informazioni in questo modo per poter avere un progetto facilmente estensibile, qualora si presentasse la necessità di gestire altre fonti di informazioni (ad es. brevetti, documenti di conferenze, relazioni tecniche e così via).

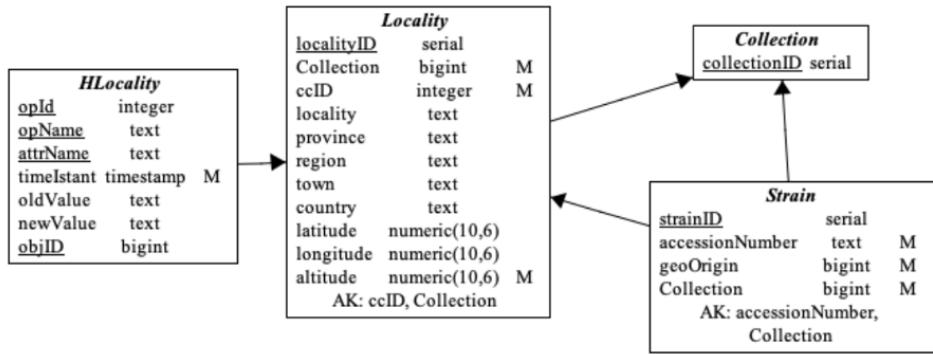


Figura 2.5: Il quinto sottodiagramma logico dell'ItCCC [25]

All'interno del quinto diagramma logico, mostrato in Figura 2.5, è presente la tabella **Locality**. Essa è stata inserita per poter mantenere l'informazione sull'origine geografica di ciascun ceppo presente nel database. Ciascuna tupla presente al suo interno è identificata sia da un attributo interno assegnato da ItCCC (**LocalityID**) e sia da un identificatore univoco nella collezione (**Collection**). La tabella, ha diversi attributi autoesplicativi, tra questi, è possibile notare: **latitude**, **longitude** e **altitude**, che, se disponibili, permetteranno la geolocalizzazione dell'origine del ceppo.

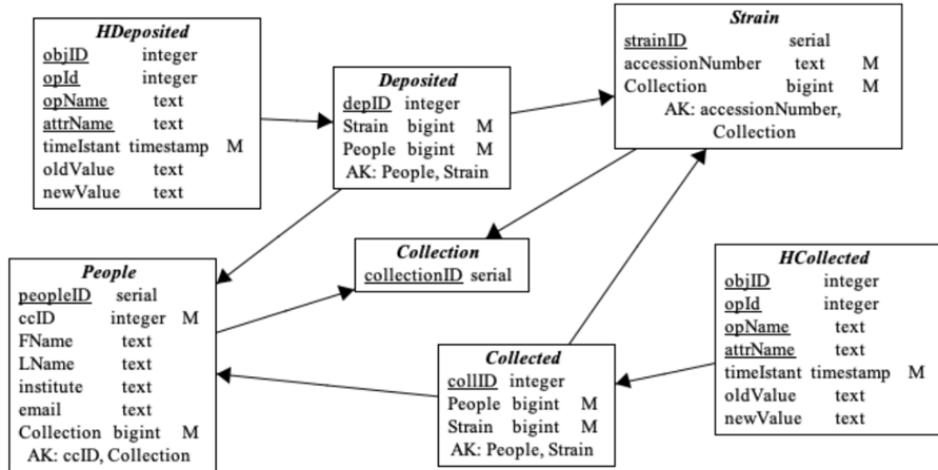


Figura 2.6: Il sesto sottodiagramma logico dell'ItCCC [25]

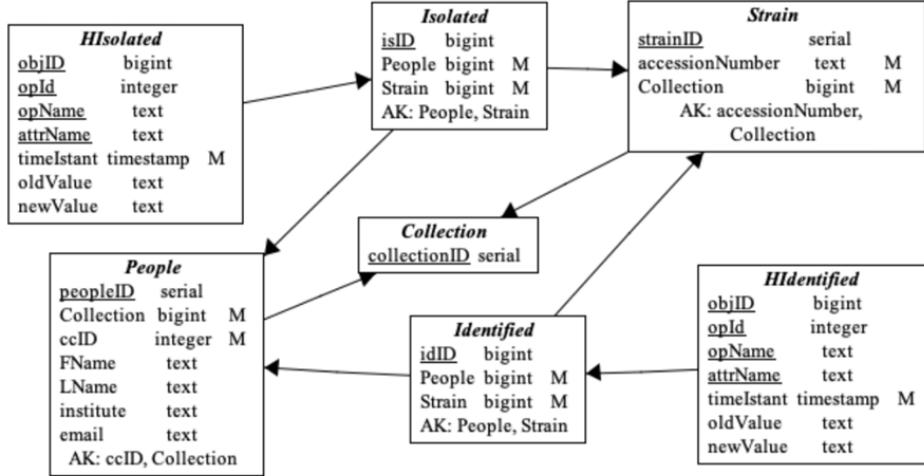


Figura 2.7: Il settimo sottodiagramma logico dell'ItCCC [25]

Un'altra informazione fondamentale, che il catalogo ItCCC deve gestire, riguarda il ruolo che le persone hanno svolto relativamente alla raccolta dei vari strain. Per questo motivo è stata introdotta la tabella **People** (Fig. 2.6, Fig 2.7). Le occorrenze di **People** sono collegate alle occorrenze di **Strain** in base al loro ruolo. L'attributo **ccID** rappresenta l'identificativo di ogni persona nella raccolta.

In particolare, durante la fase di sperimentazione di questa tesi, è stata utilizzata una vista, del database appena descritto, chiamata “**Detailed-strain**”. Essa contiene in totale **73 colonne** che rappresentano le informazioni più rilevanti per un generico strain. I campi sono mostrati nella seguente tabella (la maggior parte di essi sono stati già descritti precedentemente):

strainid	collection	accessionnumber
gmo	gmoconstructioninfo	registeredcollection
collectiondate	isolationdate	identificationtechnique
depositdate	otherdenominations	typestrain
inclusiondate	remarks	pathogenicity
beforeenagoya	availablefordis	otherccnumbers
commentontaxonomy	mutantinformation	userrestrictions
nagoyaconditions	geoorigin	cacronym
confirmed	organismtype	riskgroup
euquarantine	strain	recommendedtemperature
dualuse	substrate	testedtgmin
testedsdtgmax	organismtype	sexualstate
mtafile	prodofmetabolites	genus
species	meuquarantine	isolationhabitat
status	infrasubspecificnames	qps
axenic	genotype	mriskgroup
historyofdeposit	enzymeproduction	applications
plasmids	plasmidscollectionsfields	ploidy
interspecifichybrid	scientificname	ineva
inmirri	othernames	isolationhost
name	abbreviation	propagationhost
cultivar	lyticcicle	pathotypeserotypetype
storageconditions	symptomatologyonisolationhost	transmissionby
contamination	infectivitytested	restrictions
nagoyaproto		

Tabella 2.1: Campi della vista **Detailedstrain** presente nel database biologico.

2.2 Dataset sperimentali

Per la sperimentazione dei modelli di linguaggio descritti nella sottosezione 1.4.5, sono stati utilizzati diversi dataset contenenti informazioni sia sulla struttura della vista **Detailedstrain**, sia sulle questioni in linguaggio naturale e sulle relative query SQL. I dataset, che saranno descritti successivamente, sono stati impiegati per affinare i modelli durante il fine-tuning e per valutarli durante il testing, entrambe le fasi saranno descritte nel capitolo 4.

2.2.1 SQL-Create-Context-Instruction

E' un dataset [26] che è stato utilizzato nella fase di fine-tuning, in particolare per poter permettere ai modelli di poter apprendere una rappresentazione migliore e più completa del linguaggio SQL. E' stato costruito a partire dal dataset chiamato **SQL-Create-Context** [27] che a sua volta è basato sui

dati presenti in altre due dataset che sono **WikiSQL** [23] e **Spider** [24], entrambi abbastanza conosciuti nell’ambito del text-to-SQL.

In particolare [14] [15]:

- **WikiSQL:** E’ un’ampio dataset che raccoglie circa 80.000 coppie di domande in linguaggio naturale e query SQL create tramite crowdsourcing e che fanno riferimento a circa 25.000 tabelle estratte tramite HTML da Wikipedia. Quindi, ciascun esempio, contiene: una tabella di riferimento con le proprie colonne, una query descritta in NL e la corrispondente query SQL. La complessità delle query presenti in WikiSQL è bassa, poichè, ciascuna query fa riferimento ad una singola tabella e non ad un database relazionale e inoltre nelle query non ci sono clausole complesse come ad esempio JOIN, GROUP BY, ORDER BY, UNION, e INTERSECTION.
- **Spider:** E’ un dataset pittosto grande per il parsing text-to-SQL cross-domain che è stato scritto e revisionato da 11 studenti di computer science madrelingua inlgese. Contiene più di 10.000 domande in linguaggio naturale e oltre 5.000 query SQL uniche su 200 database appartenenti a 138 domini diversi. Le query presenti al suo interno, possono essere suddivise in quattro livelli in base alla loro difficoltà: facile, medio, difficile ed estremamente difficile, utilizzando tutti gli elementi comuni del linguaggio SQL, inclusa la nidificazione. Queste caratteristiche, insieme all’elevata qualità del dataset, poiché è stato annotato a mano e ricontrollato, hanno portato i ricercatori a fare ampio affidamento su di esso per sviluppare sistemi in grado di generare query SQL piuttosto complesse.

Nello specifico, il dataset SQL-Create-Context-Instruction, contiene al suo interno 78.577 esempi ed è stato progettato avendo come obiettivo proprio il fatto di specializzare gli LLM sul task text-to-SQL, in modo tale da prevenire la generazione errata di nomi di colonne e tabelle, un problema comune nei dataset associati al text-to-SQL.

Per creare il dataset, sono stati effettuati processi di pulizia e data augmentation sui dati combinati di WikiSQL e Spider. È stato utilizzato SQLGlot [28] per analizzare le query provenienti da Spider e WikiSQL, suddividendole in tabelle e colonne differenti.

I tipi di dati delle colonne sono stati inferiti in base all’uso degli operatori $>$ e $<$, nonché all’impiego delle funzioni MIN(), MAX(), AVG() e SUM() sulle colonne. Sebbene questo metodo non sia perfetto, aumenta la probabilità di determinare correttamente il tipo di dato di una colonna; in caso contrario, le colonne sono state impostate di default come VARCHAR.

Successivamente, queste tabelle e colonne sono state utilizzate per generare le istruzioni “CREATE TABLE” con i tipi di dato dedotti. Infine,

SQLGlot è stato nuovamente impiegato per garantire che sia le query SQL che le istruzioni “CREATE TABLE” venissero interpretate senza errori.

Ciascun esempio presente in questo dataset è basato su questa struttura (che segue lo stile dei prompt dei modelli Llama):

[INST] Instruction/context [/INST] Question-QuerySQL

dove:

- **[INST]** e **[/INST]**: Indicano l'inizio delle istruzioni (o contesto) fornite al modello.
- **Instruction/context**: E' la parte del prompt che contiene le istruzioni fondamentali sulle quali il modello baserà tutta la sua generazione. Al suo interno ci sono:
 1. Una piccola descrizione su quello che il modello dovrà fare (ovvero generare una query SQL), lo schema del database di riferimento che potrà descrivere una o più tabelle, ciascuna con le informazioni sui propri campi (nome, tipo) e la question.
 2. Question in linguaggio naturale.

Esempio:

“Write SQLite query to answer the following question given the database schema. Please wrap your code answer using Schema:.. Question:..”

- **Model output**: Conterrà prima una piccola descrizione per aggiungere maggiore contesto e, inoltre, verrà reinserita la question in NL con la speranza che il modello possa associare correttamente la query SQL corretta alla domanda di partenza.

Esempio:

“Here is the SQLite query to answer the question:.. (QuerySQL)”

2.2.2 BioInfoDataset

In questo lavoro di tesi, sono stati costruiti dei dataset specifici per il text-to-SQL basati sulla vista di riferimento “Detailedstrain” descritta alla fine della sezione 2.1. Essi sono stati suddivisi in: **Training set (per il Fine-tuning)**, **Validation set** e **Test set**. Per riuscire a costruire in maniera automatizzata i dataset appena menzionati, sono state prima individuate 5 categorie specifiche di query e dopodichè, per ciascuna di esse, sono stati costruiti dei **template** specifici che definiscono **un'insieme di strutture**

sia per le question che per le query appartenenti a quella particolare categoria. Durante la costruzione dei tre dataset, ogni template di ciascuna categoria è stato selezionato in modo casuale, evitando la generazione di duplicati sia all'interno di ciascun dataset che tra dataset diversi. **Sia i template che i dataset sono in formato JSON.**

Le categorie di query individuate sono le seguenti:

- **C1:** queries semplici - su un solo campo e COUNT.
- **C2:** queries più complesse – su due o più campi con AND e OR.
- **C3:** queries su vista, campi e valori scritti in maniera errata.
- **C4:** queries sui campi data (collectionDate, isolationDate, identificationDate e depositDate).
- **C5:** queries con WHERE, GROUP BY e HAVING.

Esempio di un template della categoria C2 del test set:

```
{
  "index_template": 0,
  "question_template": "Give me all strains that have {column1}
    equal to {value1} and {column2} equal to {value2}.",
  "sql_template": "SELECT * FROM detailedstrain WHERE {column1}
    = {value1} AND {column2} = {value2};",
  "possible_sql_queries": [
    "SELECT * FROM detailedstrain WHERE {column1} = {
      value1} AND {column2} = {value2};",
    "SELECT * FROM detailedstrain WHERE {column2} = {
      value2} AND {column1} = {value1};"
  ]
}
```

Ciascun template è composto da:

1. **index_template:** rappresenta l'indice del template all'interno della categoria.
2. **question_template:** rappresenta la struttura generale della question in NL.
3. **sql_template:** rappresenta la query SQL di riferimento alla question.
4. **possible_sql_queries:** è una lista contenente diverse possibili query SQL che un modello potrebbe generare in risposta alla question. La vera utilità di questo campo sarà presentata durante la descrizione dei risultati ottenuti, ovvero nella sezione 4.1.

Sono stati creati anche dei template per il training set e il validation set (che seguono la stessa struttura di quello mostrato in precedenza), con alcune piccole differenze nei termini utilizzati nelle question in linguaggio naturale rispetto al test set. Questo è stato fatto per simulare, almeno in parte, una situazione reale, poiché è logico aspettarsi che gli utenti dell'applicazione usino termini diversi da quelli presenti nel training set.

Esempio di un template della categoria C2 utilizzato per costruire sia il training set che il validation set:

```
{
    "index_template": 1,
    "question_template": "Select all strains where {column1}
        is {value1} and {column2} is {value2}.",
    "sql_template": "SELECT * FROM detailedstrain WHERE {
        column1} = {value1} AND {column2} = {value2};",
    "possible_sql_queries": [
        "SELECT * FROM detailedstrain WHERE {column1} = {
            value1} AND {column2} = {value2};"
    ]
}
```

Dal template di esempio appena mostrato, è possibile notare come in corrispondenza del campo “possible_sql_queries” è stata inserita un’unica query SQL, questo perchè durante la costruzione dei dataset di training e validation non è necessario specificare diverse varianti della stessa query.

All’interno della tabella sottostante sono mostrati, per ciascuna categoria di query, il numero di template per il training set, il validation set e il test set:

Categoria	Training Set	Validation Set	Test Set
C1	5	5	2
C2	4	4	6
C3	4	4	6
C4	6	6	6
C5	4	4	4

Tabella 2.2: Numero di template definiti per ciascuna categoria di query nei diversi dataset.

Il numero di esempi presenti in ciascun dataset è riportato nella tabella sottostante. **Ciascun dataset contiene esempi univoci, rappresentati da coppie (question, querySQL).** I dataset creati includono anche esempi in italiano, così da permettere agli utenti finali di formulare le question sia in questa lingua che in inglese.

Dataset	Totale	Inglese	Italiano
Training Set	500	250	250
Validation Set	300	150	150
Test Set	300	150	150

Tabella 2.3: Numero di esempi presenti in ciascun dataset suddivisi per lingua.

Nella tabella seguente, invece, sono riassunti il numero di esempi per ciascuna categoria all'interno di ciascun dataset:

Categoria	Training Set	Validation Set	Test Set
C1	100 (50 IT, 50 EN)	60 (30 IT, 30 EN)	60 (30 IT, 30 EN)
C2	100 (50 IT, 50 EN)	60 (30 IT, 30 EN)	60 (30 IT, 30 EN)
C3	100 (50 IT, 50 EN)	60 (30 IT, 30 EN)	60 (30 IT, 30 EN)
C4	100 (50 IT, 50 EN)	60 (30 IT, 30 EN)	60 (30 IT, 30 EN)
C5	100 (50 IT, 50 EN)	60 (30 IT, 30 EN)	60 (30 IT, 30 EN)

Tabella 2.4: Numero totale di esempi per ciascuna categoria nei diversi set, suddivisi per lingua (IT = Italiano, EN = Inglese).

È possibile notare dalla tabella 2.4 che i dataset sono completamente bilanciati per ciascuna categoria e contengono lo stesso numero di esempi sia per l'italiano che per l'inglese. Un'altra informazione importante è che, in ciascun dataset, le question in linguaggio naturale e le rispettive query SQL si basano sugli stessi template per entrambe le lingue. Queste simmetrie sono state mantenute per garantire una distribuzione uniforme dei dati sia tra le categorie che tra le lingue.

Il **Training set** è stato utilizzato per il fine-tuning dei modelli e quindi per la modifica di una parte dei loro pesi. Il **Validation set** è stato utilizzato per interrompere il riaddestramento di un modello nel momento in cui il valore della funzione di loss non continuasse a scendere (criterio di Early-Stopping) per evitare l'overfitting. Invece, il **Test set** è stato utilizzato per testare le capacità di generalizzazione di ciascun modello nello svolgere il task di text-to-SQL avendo sempre come punto di riferimento il database descritto all'inizio del capitolo 2.

La struttura di ciascun esempio (prompt), che viene fornito in input al modello, è identica a quella del dataset “SQL-Create-Context-Instruction” descritto nella sottosezione 2.2.1, con l'unica differenza che questa volta, lo schema non conterrà informazioni su una o più tabelle generiche, ma bensì avrà tutte le informazioni riguardanti le colonne (nomi e tipi) della vista **Detailedstrain** che sono stati mostrati nella tabella 2.1.

Ovviamente, i prompt che sono stati costruiti a partire dagli esempi presenti nel test set, a differenza di quelli per il training o il validation, non contengono la query SQL corretta perchè si presume che sia il modello a doverla generare.

Infine, per quanto riguarda la gestione dei campi rappresentanti delle date (collectionDate, isolationDate, identificationDate e depositDate) sfruttando i template della categoria C4, all'interno dei tre dataset appena descritti, sono stati inseriti esempi specifici per poter permettere ai modelli di poter apprendere correttamente la gestione dei due formati possibili:

- **Mese-Giorno-Anno** (Formato-1)
- **Anno** (Formato-2)

Nello specifico, le query SQL della categoria C4, indipendentemente da quale sia il formato della data inserita nella question in NL, sono state strutturate in modo tale da considerare sempre le tuple che hanno il valore in entrambi i formati.

Ad esempio, supponendo che nella **question venga specificata la data completa (formato-1)**:

Question con data nel Formato-1

“Give me all strains that have the {column} field equal to {value} and the {column_data} field with a date before {date}.”

Query SQL partendo dal Formato-1 con l'aggiunta del controllo per il Formato-2

```
SELECT * FROM detailedstrain
WHERE {column} = {value}
AND (
    (LENGTH({column_data}) = 10
     AND SUBSTR({column_data}, 7, 4) || '-' ||
        SUBSTR({column_data}, 4, 2) || '-' ||
        SUBSTR({column_data}, 1, 2) < {date})
    OR
    (LENGTH({column_data}) = 4
     AND {column_data} <= {year}))
);
```

Invece, supponendo che nella **question** venga specificato solo l'anno (**formato-2**):

Question con data nel Formato-2

“Give me all strains that have the {column} field equal to {value} and the {column_data} field with a date before {year}.”

Query SQL partendo dal Formato-2 con l'aggiunta del controllo per il Formato-1

```
SELECT * FROM detailedstrain WHERE {column} = {value} AND
    ((LENGTH({column_data}) = 10 AND
        SUBSTR({column_data}, 7, 4) || '-' ||
        SUBSTR({column_data}, 4, 2) || '-' ||
        SUBSTR({column_data}, 1, 2) < {year}-01-01) OR
    (LENGTH({column_data}) = 4 AND {column_data} < {year}));
```

Un ragionamento simile è stato applicato sia alle question che impongono una condizione su un valore successivo ad una certa data, ad esempio utilizzando il termine “after” invece di “before”, sia a quelle che definiscono un intervallo tra due anni differenti, sfruttando espressioni che includono “...between {start_year} and {end_year}”.

Capitolo 3

Implementazione applicazione Text-to-SQL

3.1 Architettura del Sistema

In questa sezione verranno descritte sia le componenti che costituiscono l'architettura del sistema text-to-SQL, sia i passi eseguiti dall'applicazione: dalla frase in linguaggio naturale inserita dall'utente, alla produzione della query SQL da parte del modello, fino al rendering dei risultati sull'interfaccia grafica.

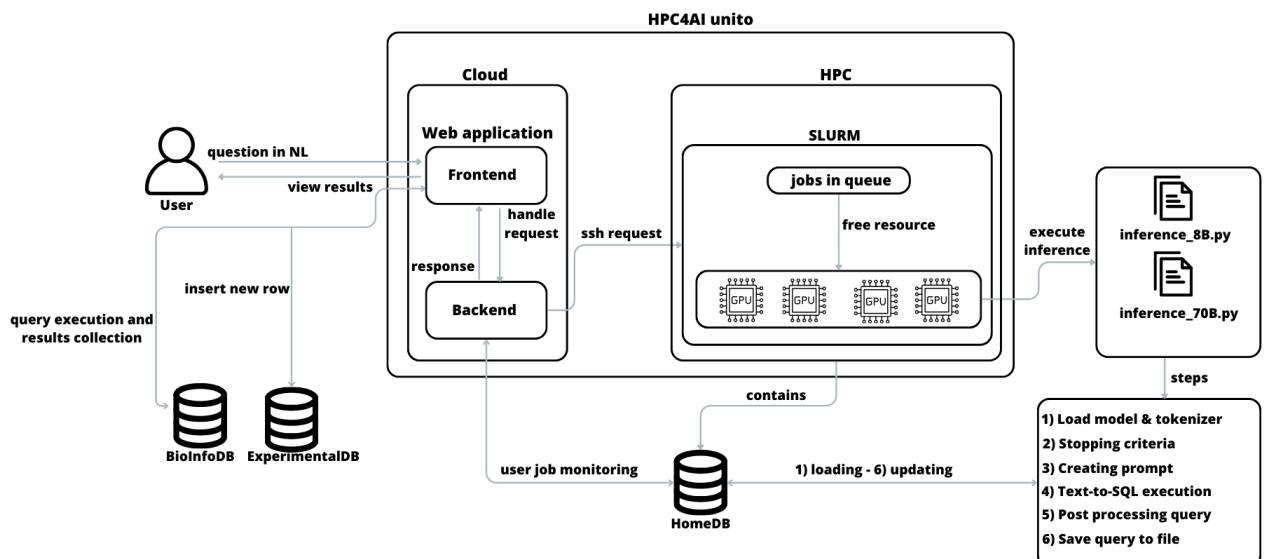


Figura 3.1: Architettura generale del sistema

La Figura 3.1 mostra l'architettura completa del sistema Text-to-SQL sviluppato in questo lavoro di tesi.

L’infrastruttura hardware che ospita l’intero sistema è basata sul centro dati ad accesso aperto **High-Performance Computing for Artificial Intelligence (HPC4AI)**¹ dell’Università di Torino. Questa struttura all'avanguardia è stata recentemente istituita grazie a un investimento iniziale di **4,5 milioni di euro**, ottenuto tramite un bando competitivo finanziato dalla **Regione Piemonte** attraverso il programma **EU POR-FESR 2014-2020**. HPC4AI implementa un’architettura avanzata che integra **HPC e cloud**, ridefinendo l’utilizzo tradizionale di tali tecnologie per supportare applicazioni di intelligenza artificiale. In particolare, il cloud fornisce un’interfaccia moderna per l’HPC, mentre l’HPC agisce da acceleratore per il cloud.

Osservando nuovamente la Figura 3.1, partendo da sinistra, si può notare come l’utente, dopo aver selezionato il modello di linguaggio per la generazione della query SQL, possa digitare la domanda in linguaggio naturale (italiano o inglese) direttamente nell’interfaccia grafica, all’interno di una text box. Proseguendo verso destra, è possibile osservare che l’architettura del sistema è suddivisa in due componenti principali, entrambe eseguite su HPC4AI. La prima componente, denominata **Web application**, è stata sviluppata con il framework Next.js e include due moduli, **Frontend** e **Backend**, che verranno descritti a breve. Questi moduli sono gestiti dal sistema cloud di HPC4AI. La seconda componente si occupa della gestione delle inferenze dei modelli, che vengono eseguite su nodi di calcolo gestiti con **SLURM** (Simple Linux Utility for Resource Management). SLURM è un sistema di gestione delle risorse e di scheduling dei job su cluster HPC (High-Performance Computing). Questa seconda componente sarà descritta più nel dettaglio nella sottosezione 3.1.1.

I **moduli della web application**, possono essere descritti nel modo seguente:

- **Frontend:** implementa l’interfaccia grafica e tutto ciò che è legato alla gestione di visualizzazione dei dati. In particolare, una volta che l’utente cliccherà sul button per richiedere la generazione della query SQL, il frontend si preoccuperà di inviare una richiesta al backend (“**handle request**”) inviando sia la question in NL che il nome del modello selezionato dall’utente. Successivamente, l’istanza associata a quel determinato client attenderà la ricezione del risultato (indicato dalla freccia “**response**” in Figura 3.1). Una volta ottenuta la risposta, qualora venisse effettivamente restituita la query SQL, questa verrà eseguita sul database “**BioInfoDB**” (contenente le informazioni microbiche) e dopodichè i risultati verranno mostrati all’utente in formato tabellare.

¹<https://hpc4ai.unito.it>

Il database chiamato “**ExperimentalDB**” è utilizzato invece per memorizzare diverse informazioni riguardanti tutte le question che sono state inviate da parte degli utenti al sistema. In particolare, esso contiene **un'unica tabella (chiamata querytest)** con le seguenti colonne:

- **Question**: rappresenta la question in NL scritta dall'utente.
- **QuerySQL**: rappresenta l'eventuale query SQL generata come risposta alla question.
- **ResponseExecuteQuery**: rappresenta lo stato di esecuzione della query (può essere un errore oppure una conferma che la query è stata eseguita con successo in BioInfoDB).
- **Model**: rappresenta il modello selezionato dall'utente per eseguire la generazione.

Questo database è stato creato per consentire, in futuro, un'analisi più approfondita sulle tipologie di question maggiormente richieste dagli utenti e sulle query generate dai modelli. Ciò permetterà di individuare eventuali criticità e di addestrare nuovamente i modelli di generazione.

- **Backend**: implementa la logica di controllo all'interno della Web application. Nello specifico, il backend è suddiviso in due sottomoduli principali:

1. **Gestione delle richieste utente**:

A ciascun client è assegnata una componente dedicata, che gestisce la richiesta (*job*) proveniente dal frontend, assegnandole un identificatore univoco (**sessionId**) e inoltrandola a SLURM. Il sistema attende quindi l'esito dell'elaborazione, gestendo eventuali errori o fallimenti. Una volta ottenuto il risultato, questo viene quindi inviato al frontend e successivamente viene inviata una notifica al secondo sottomodulo, che provverà a rimuovere il job dalla struttura dati.

2. **Monitoraggio centralizzato dei job**:

Un thread, avviato subito dopo l'esecuzione dell'applicazione Next.js lato server, monitora lo stato delle richieste inoltrate utilizzando gli identificatori univoci assegnati dal primo sottomodulo. Il suo compito è verificare l'avanzamento dei job su SLURM e, in caso di completamento senza errori, aggiungere l'informazione relativa alla query SQL generata. Questo consente al primo sottomodulo, legato al client, di leggere il risultato e aggiornare correttamente lo stato del proprio job. Il thread utilizza un dizionario per tracciare i job attivi e, ad ogni ciclo di polling, rimuove quelli completati.

Entrambi i sottomoduli del backend comunicano con SLURM tramite protocollo SSH. Grazie all'uso di **Docker**, l'intera web application è stata containerizzata in un ambiente isolato, mentre i database Bio-InfoDB ed ExperimentalDB sono stati containerizzati separatamente, consentendo un'esecuzione indipendente sul server.

Di seguito saranno mostrate alcune **immagini dell'interfaccia grafica** della web application. In particolare, i due modelli che l'utente potrà selezionare, corrispondono ai modelli migliori (chiamati **LLaMA-3-8B-Large-Small-Dataset-DoubleFT (LoRA)** e **LLaMA-3-70-Large-Small-Dataset-DoubleFT (QLoRA)**) che sono stati ottenuti al termine della fase di sperimentazione che sarà descritta nella sezione 4.1.

The screenshot shows the 'SQL Query Generator' web application. At the top, there are logos for the European Union, the Italian Ministry of Education, University, and Research (MIUR), and the project SUS-MIRRI. Below the header, the title 'SQL Query Generator' is centered. A sub-instruction reads: 'First select the model to be used during inference, then write the query in natural language (English/Italian) and finally click on the button to generate the SQL query and obtain the results.' A dropdown menu titled 'Choose LLM model:' contains two options: 'Model 8B' (selected) and 'Model 70' (disabled). Next to it is a note: 'Lower accuracy (92%-English, 89%-Italian) - Lower average waiting time (30 seconds)'. Below the dropdown is a text input field with placeholder text 'Inserisci query in linguaggio naturale'. At the bottom right of the input field is a blue 'Generate SQL' button.

Figura 3.2: Selezione del modello **LLaMA-3-8B-Large-Small-Dataset-DoubleFT (LoRA)** con relative informazioni riguardanti l'**accuratezza** ottenuta sul test set e il **tempo medio** di inferenza richiesto.

SQL Query Generator

First select the model to be used during inference, then write the query in natural language (English/Italian) and finally click on the button to generate the SQL query and obtain the results.

Choose LLM model:

Higher accuracy (96%-English, 93%-Italian) - Longer average waiting time (70 seconds)

Inserisci query in linguaggio naturale

Figura 3.3: Selezione del modello **LLaMA-3-70-Large-Small-Dataset-DoubleFT (QLoRA)** con relative informazioni riguardanti l'**accuratezza** ottenuta sul test set e il **tempo medio** di inferenza richiesto.

First select the model to be used during inference, then write the query in natural language (English/Italian) and finally click on the button to generate the SQL query and obtain the results.

Choose LLM model:

Give me all strains

SQL query generated:

```
SELECT syntheticstrain.* FROM syntheticstrain WHERE strainid IN (SELECT strainid FROM detailedstrain);
```

Time taken for response:
31.01 seconds

Query Results (Displayed 100 of 28707 results):

collection	accessionnumber	fullaccessionnumber	organismtype	taxonname	strainid
DBVPG	10530	DBVPG 10530	Yeasts	Aureobasidium pullulans	19015
DBVPG	5090	DBVPG 5090	Yeasts	Goffeauzyma gastrica	16357
EMCC	0385	EMCC 0385	Bacteria	Bacillus amyloliquefaciens	390
BNSS	10497	BNSS 10497	Bacteria	Sphingomonas paucimobilis	12012

Figura 3.4: Query SQL generata dal modello-8B, tempo di risposta e prima parte dei risultati in formato tabellare.

SQL Query Generator

First select the model to be used during inference, then write the query in natural language (English/Italian) and finally click on the button to generate the SQL query and obtain the results.

Choose LLM model:

Model 70B

Give me all strains

Generate SQL

SQL query generated:

```
SELECT syntheticstrain.* FROM syntheticstrain WHERE strainid IN (SELECT strainid FROM detailedstrain);
```

Time taken for response:
65.99 seconds

Query Results (Displayed 100 of 28707 results):

collection	accessionnumber	fullaccessionnumber	organismtype	taxonname	strainid
DBVPG	10530	DBVPG 10530	Yeasts	Aureobasidium pullulans	19015
DBVPG	5090	DBVPG 5090	Yeasts	Goffeauzyma gastrica	16357
EMCC	0385	EMCC 0385	Bacteria	Bacillus amyloliquefaciens	390
BNSS	10497	BNSS 10497	Bacteria	Sphingomonas paucimobilis	12012

Figura 3.5: Query SQL generata dal modello-70B,tempo di risposta e prima parte dei risultati in formato tabellare.

Give me all strains

Generate SQL

SQL query generated:

```
SELECT syntheticstrain.* FROM syntheticstrain WHERE strainid IN (SELECT strainid FROM detailedstrain);
```

Time taken for response:
31.81 seconds

Query Results (Displayed 200 of 28707 results):

collection	accessionnumber	fullaccessionnumber	organismtype	taxonname	strainid
TUCC	MUT00000570	TUCC MUT00000570	Filamentous Fungi	Purpureocillium lilacinum	23912
ITEM	8597	ITEM 8597	Filamentous Fungi	Fusarium graminearum	6760
DBVPG	1475	DBVPG 1475	Yeasts	Saccharomyces cerevisiae	13752
TUCC	TUCC00000579	TUCC TUCC00000579	Yeasts	Malassezia pachydermatis	28652
DBVPG	10731	DBVPG 10731	Yeasts	Pseudotremella sp.	19210
ITEM	8812	ITEM 8812	Yeasts	Hanseniaspora uvarum	6937
ITEM	2039	ITEM 2039	Filamentous Fungi	Fusarium tricinctum	2333
EMCC	0614	EMCC 0614	Bacteria	Alcaligenes faecalis	600
TUCC	MUT00005494	TUCC MUT00005494	Filamentous Fungi	Cladosporium ramotenellum	22508
DBVPG	5332	DBVPG 5332	Yeasts	Naganishia vaghanmartiniae	16585

Figura 3.6: Possibilità di scrolling dei risultati.

The screenshot shows a user interface for generating an SQL query from a natural language question. At the top, there's a dropdown menu labeled "Choose LLM model:" with "Model 8B" selected. Below it is a text input field containing the Italian sentence: "Dammi gli strain con organismtype uguale a bacteria e otherdenomination uguale a mvpc". A "Generate SQL" button is located below the input field. Underneath, the generated SQL query is displayed:

```
SELECT syntheticstrain.* FROM syntheticstrain WHERE strainid IN (SELECT strainid FROM detailedstrain WHERE organismtype ILIKE '%bacteria%' AND otherdenominations ILIKE '%mvpc%');
```

Below the query, the time taken for response is listed as "23.70 seconds". The final section, "Query Results (Displayed 100 of 139 results):", contains a table with the following data:

collection	accessionnumber	fullaccessionnumber	organismtype	taxonname	strainid
EMCC	0122	EMCC 0122	Bacteria	Burkholderia ambifaria	128
EMCC	0166	EMCC 0166	Bacteria	Burkholderia cenocepacia	172
EMCC	0189	EMCC 0189	Bacteria	Burkholderia cenocepacia	195
EMCC	0106	EMCC 0106	Bacteria	Burkholderia ambifaria	112
EMCC	0185	EMCC 0185	Bacteria	Burkholderia ambifaria	191
EMCC	0171	EMCC 0171	Bacteria	Burkholderia cenocepacia	177
EMCC	0146	EMCC 0146	Bacteria	Burkholderia ambifaria	152
EMCC	0156	EMCC 0156	Bacteria	Burkholderia ambifaria	162

Figura 3.7: Visualizzazione risultati di una **question scritta in italiano**.

3.1.1 SLURM

Slurm gestisce diversi cluster di macchine all'interno del sistema HPC4AI del Dipartimento di Informatica dell'Università di Torino. Tuttavia, in questo lavoro di tesi è stata utilizzata esclusivamente la partizione composta da quattro nodi, ciascuna dotata di una GPU H100. Una volta ricevuta la richiesta da parte del backend della web application, Slurm inserirà il nuovo job all'interno della propria coda in attesa che una delle 4 risorse sia disponibile. Non appena una GPU sarà libera, il job verrà mandato in esecuzione e, in base all'LLM selezionato dall'utente inizialmente, potrà essere eseguito lo script associato al modello selezionato. Questi script fanno riferimento ai modelli chiamati **LLaMA-3-8B-Large-Small-Dataset-DoubleFT (LoRA)** e **LLaMA-3-70-Large-Small-Dataset-DoubleFT (QLoRA)** che, come sarà descritto nel capitolo 4, durante la sperimentazione, sono risultati essere i modelli migliori dopo l'applicazione del fine-tuning. In particolare, come mostrato in Figura 3.1, entrambi gli script eseguono i seguenti step:

1. **Load model & tokenizer:** caricamento dei pesi del modello di riferimento (LLaMA-3-8B-Large-Small-Dataset-DoubleFT (LoRA) o LLaMA-3-70-Large-Small-Dataset-DoubleFT (QLoRA)) e del tokenizzatore.

2. **Stopping criteria:** applicazione del criterio di stop per terminare la generazione del testo, da parte del modello, nel momento in cui verrà generato un certo token. Questo è importante, poichè in questo modo, si evita la possibilità che il LLM possa continuare a generare inutilmente altro testo, dopo aver già completato la generazione della query SQL.
3. **Creating prompt:** creazione del prompt da fornire in input al modello. Esso viene costruito con la stessa struttura descritta nella sottosezione 2.2.1 sempre con riferimento alla vista Detailedstrain (campi mostrati in tabella 2.1).
4. **Text-to-SQL-execution:** generazione da parte del modello della query SQL di riferimento.
5. **Post processing query:** applicazione del post processing sulla query ricevuta in output dal modello. Questo step è particolarmente rilevante, in quanto si è rivelato determinante nel migliorare l'accuratezza generale del sistema in ambiente di produzione. I passi eseguiti in questa fase possono essere riassunti nel modo seguente:
 - Sanificazione della query SQL per prevenire attacchi di SQL Injection. Se la query contiene elementi potenzialmente dannosi, l'elaborazione verrà immediatamente interrotta e la query sarà respinta, restituendo un errore.
 - Rimozione di eventuali caratteri inutili presenti nella query (ad es. '[]' o il ';' finale).
 - Rimozione di eventuali clausole diverse da WHERE. Questo perché in questa prima versione dell'applicazione si è preferito utilizzare il modello in production solo per generare query con la clausola citata pocanzi.
 - Eventuale modifica delle clausole SELECT e FROM in modo tale che contengano rispettivamente i valori **strainid** e **detailedstrain**. Per quanto riguarda il **vincolo su strainid**, è stato inserito poichè, come verrà descritto nell'ultimo punto, la query generata dal modello sarà innestata come una sub-query. Affinchè la query finale restituisca un resultset corretto, è necessario che la colonna presente nella **SELECT della sub-query, sia identica a quella utilizzata nella condizione WHERE della query principale**. Invece, per quanto riguarda il **mapping forzato su detailedstrain**, questo risulta fondamentale nel caso in cui il modello per un qualche motivo generasse un nome di tabella o vista differente, causando inevitabilmente un errore nella query. Quindi, grazie a questo step, l'errore verrà corretto, producendo una query coerente con la struttura del database.

- Eventuale sostituzione della colonna **collection** con **cacronym** qualora il valore assegnato alla prima non fosse numerico. Questa modifica è necessaria poiché, come citato nel punto precedente, la query finale sarà composta da una query principale ed una innestata.

In particolare, **nella query principale si farà riferimento alla vista “syntheticstrain”**, mentre **nella query innestata si farà riferimento alla vista “detailedstrain”**. Tuttavia, la colonna “collection” è presente in entrambe le viste, **ma con tipi di dato differenti**: in detailedstrain è di tipo **bigint**, mentre in syntheticstrain è di tipo **text**. Fortunatamente, in detailedstrain è presente la colonna **cacronym**, anch’essa di tipo **text**, che corrisponde esattamente alla colonna **collection** di syntheticstrain.

Pertanto, qualora l’utente inviasse una **question ambigua**, come ad esempio:

“Dammi gli strain con organismtype uguale a Virus appartenenti alla collezione ‘EMCC’ ”

dove chiaramente EMCC è una stringa, e il modello generasse una query del tipo:

SELECT strainid FROM detailedstrain
WHERE collection = 'EMCC' ...

sarà necessario sostituire la colonna **collection** con **cacronym**. In questo modo, non solo si eviterà un errore sintattico, ma si garantirà il corretto mapping delle tuple di **detailedstrain** con quelle di **syntheticstrain**.

- Eventuale modifica dei nomi di colonne errati tramite l’utilizzo della distanza di Levenshtein. Questa metrica misura la differenza tra due stringhe, calcolando il numero minimo di operazioni elementari (inserimenti, cancellazioni o sostituzioni di caratteri) necessarie per trasformare una stringa nell’altra. In questo passaggio viene applicata la tecnica di **Fuzzy/approximate string matching**, menzionata nella sezione 1.3 al termine della spiegazione dello schema linking.
- Partendo dalla query SQL iniziale, viene creata una nuova query, chiamata **“ILIKE-query”**, sostituendo ogni espressione della forma [column = 'value'] con [column ILIKE '%value%'] solo se ‘value’ non contiene numeri.

- Successivamente, la query originale ottenuta e l’“ILIKE-query” verranno innestate come sub-query all’interno di due query finali:

```
final_query = (
    "SELECT syntheticstrain.* "
    "FROM syntheticstrain "
    "WHERE strainid IN {original_subquery};"
)
```

```
ilike_final_query = (
    "SELECT syntheticstrain.* "
    f"FROM syntheticstrain "
    f"WHERE strainid IN {ilike_subquery};"
)
```

6. Le due query costruite al termine del post processing, verranno memorizzate nel database di Slurm, chiamato **HomeDB**, all’interno di un file identificato dal codice di sessione che era stato assegnato all’utente di riferimento dalla web application. In questo modo, la seconda componente del backend (descritta nella sezione precedente) potrà modificare lo stato del job e quindi la prima componente leggerà il risultato e lo restituirà al frontend.

Il motivo principale per cui viene costruita anche la “**ILIKE-query**” sarà descritto nella sottosezione 4.1.1. Per quanto riguarda invece l’uso della query generata come subquery, questa scelta è stata adottata perché i biologi, principali fruitori del sistema, necessitano di visualizzare specifiche colonne della **vista syntheticstrain** presente nel database biologico. In particolare, le colonne di interesse sono: **strainid**, **collection**, **accesionnumber**, **fullaccesionnumber** (identificatore univoco del ceppo, più specifico dell’accesionnumber), **organismtype** e **taxonname**.

Capitolo 4

Sperimentazione e Risultati

4.1 Valutazione Performance

In questa sezione saranno presentati gli esperimenti eseguiti e i relativi risultati, ottenuti applicando il fine-tuning con LoRA e QLoRA (descritte nella sezione 1.5) ai modelli illustrati nella sottosezione 1.4.5 e utilizzando i dataset riportati nella sezione 2.2. Prima di arrivare alla descrizione dei test, è utile descrivere brevemente quali sono stati i valori assegnati agli **iperparametri più importanti** sia per quanto riguarda il **fine-tuning** che per l'**inferenza**. Il linguaggio di programmazione utilizzato per eseguire tutti gli esperimenti è stato Python.

Iperparametri Fine-tuning:

- Iperparametri LoRA (libreria PEFT):
 - **LORA_R**: 64
Rappresenta il rank dell'adattatore LoRA. Un valore più alto permette una maggiore capacità di adattamento ma aumenta il numero di parametri da aggiornare.
 - **LORA_ALPHA**: 16
È l'elemento che determina il fattore di scaling utilizzato nel meccanismo di adattamento di LoRA. Un valore più alto amplifica l'impatto delle matrici di adattamento sulla rete.
 - **LORA_DROPOUT**: 0.1
Indica la probabilità di dropout applicata ai pesi degli adattatori LoRA per evitare l'overfitting.
- Iperparametri QLoRA (libreria PEFT e BitsAndBytes):
 - **LOAD_IN_4bit**: True
Indica che il modello di base viene caricato con una precisione a

4 bit, riducendo il consumo di memoria e migliorando l'efficienza computazionale.

- **BNB_4bit_COMPUTE_DTYPE:** torch.float16
Specifica il tipo di dato utilizzato per la computazione dopo la dequantizzazione temporanea. In questo caso, viene usata la precisione FP16 per migliorare la stabilità numerica.
- **BNB_4bit_QUANT_TYPE:** nf4
Definisce il tipo di quantizzazione adottata. L'opzione nf4 (Normalized Float 4) è una variante avanzata che migliora la rappresentazione dei valori riducendo la perdita di precisione [29].
- **BNB_4bit_USE_DOUBLE_QUANT:** False
Se impostato a True, abilita una seconda fase di quantizzazione per comprimere ulteriormente i pesi. In questo caso è disabilitato per evitare ulteriori perdite di precisione.

- **Iperparametri Training (libreria Transformers):**

- **NUM_TRAIN_EPOCHS:** 20
Numero massimo di epoche di addestramento raggiungibili poichè è utilizzato in combinazione con il criterio di Early Stopping descritto più in basso (negli iperparametri di SFT).
- **FP16:** True
Abilita il training a 16 bit in floating point (FP16), riducendo il consumo di memoria e migliorando la velocità di calcolo.
- **PER_DEVICE_TRAIN_BATCH_SIZE:** 1, 4, 8
Definisce la dimensione dei batch durante l'addestramento. Sono stati testati 3 differenti valori, rispettando la dimensione massima di memoria della GPU H100 (80 GB) per individuare il modello migliore.
- **GRADIENT_ACCUMULATION_STEPS:** 1
Numero di step di accumulo dei gradienti prima di aggiornare i pesi. Poichè è stato impostato a 1, l'aggiornamento sarà eseguito dopo ogni batch.
- **LEARNING_RATE:** 2e-4
Tasso di apprendimento per l'ottimizzatore AdamW. Un valore troppo piccolo provoca un aggiornamento dei pesi molto basso e questo può portare ad un rallentamento in termini di apprendimento della rete. Al contrario, un valore più alto accelera il training ma può compromettere la stabilità del processo di convergenza verso la soluzione, in quanto l'aggiornamento di un peso per un certo esempio sarà molto grande. Questo però potrebbe ridurre la capacità del modello di generalizzare correttamente sugli esempi successivi.

- **OPTIM**: paged_adamw_32bit
Ottimizzatore utilizzato per il fine-tuning, una variante di AdamW ottimizzata per la gestione della memoria.
- **LR_SCHEDULER_TYPE**: cosine
Tipo di scheduler del learning rate. La funzione coseno riduce progressivamente il valore del learning rate nel tempo.
- Iperparametri per il Supervised Fine-Tuning (SFT) con la classe SFTTrainer della libreria TRL (Transformers Reinforcement Learning):
 - **DEVICE_MAP**: cuda:0
Specifica su quale dispositivo caricare il modello. In questo caso, il modello sarà caricato interamente su una **GPU NVIDIA H100**. Gli altri valori possibili sono:
 - “**cpu**”: carica il modello sulla CPU, utilizzando la RAM del sistema anzichè la memoria della GPU. Questa opzione è utile in assenza di una GPU, ma comporta tempi di inferenza più elevati.
 - “**auto**”: consente la gestione automatica dell’allocazione del modello, distribuendo i pesi tra CPU e GPU disponibili in base alle risorse presenti sul nodo di calcolo. Questo approccio è particolarmente utile quando la memoria di una singola GPU non è sufficiente per ospitare l’intero modello, permettendo di suddividerlo su più dispositivi per ottimizzare l’uso delle risorse.
 - **EARLY_STOPPING_PATIENCE**: 2
Numero di valutazioni consecutive della **funzione di loss sul validation set** senza miglioramenti, dopo le quali il training sarà interrotto automaticamente.
 - **EARLY_STOPPING_THRESHOLD**: 0.0
Miglioramento minimo richiesto per la loss del validation set affinché il contatore di early stopping venga resettato. Se il miglioramento è inferiore a questa soglia, il training continua a monitorare la convergenza senza azzerare il conteggio delle iterazioni senza miglioramento.

Iperparametri di inferenza:

- Iperparametri per rendere deterministiche le risposte generate dai modelli (libreria Transformers):
 - **TEMPERATURE**: 1
Controlla la casualità nella generazione del testo. Valori bassi (vicini a 0) rendono il modello più deterministico, amplificando la

differenza tra le probabilità dei token e favorendo sempre quello più probabile. Valori alti (>1) aumentano la casualità distribuendo le probabilità in modo più uniforme, rendendo più competitivi anche token meno probabili. Con **Temperature = 1**, il modello mantiene la distribuzione originale senza alterarla, garantendo un comportamento neutrale.

- **TOP_K:** 1

Specifica il numero massimo di token tra cui scegliere ad ogni passo di generazione. Il modello eseguirà tale scelta, considerando solamente i primi **k** token con probabilità più elevata. Settando **Top_k = 1**, verrà sempre scelto il token con la probabilità massima, rendendo l'output completamente deterministico.

- **TOP_P:** 0.0

Definisce la soglia di probabilità cumulativa per selezionare il sottinsieme di token tra cui scegliere. È una tecnica dinamica in cui il modello seleziona il minimo numero di token che raggiungono almeno una probabilità cumulativa pari a **Top_p**. Con **Top_p = 0.0**, il modello ignora questa soglia, ma poiché **Top_k = 1**, questa impostazione diventa irrilevante.

- **MAX_NEW_TOKENS:** 200

Numero massimo di token generati nella risposta del modello. Imposta un limite alla lunghezza dell'output per evitare generazioni troppo lunghe e inutili. Per il task di riferimento di questa tesi, tale valore è risultato essere abbastanza bilanciato per poter dare spazio ai modelli di generare anche le query SQL più lunghe.

I valori di Temperature, Top_k e Top_p così configurati, garantiscono che i modelli siano completamente deterministici e che i tempi di generazione di una stessa risposta rimangano costanti. Queste due caratteristiche sono fondamentali sia per ottenere risultati generali nel task di text-to-SQL, sia per garantirne la replicabilità.

Di seguito è mostrata la **tabella** contenente la **percentuale di parametri** addestrabili per ciascun modello grazie all'applicazione della tecnica LoRA:

Nella tabella 4.1 è riportata un'analisi relativa alla riduzione significativa del numero di parametri addestrabili durante il fine-tuning ottenuta con la tecnica LoRA. Tali riduzioni, sono state rese possibili grazie alla scelta del valore $r = 64$ e alla restrizione dell'adattamento dei pesi alle sole matrici W^Q e W^V . È stato utilizzato $\alpha = 16$ in modo da ottenere il rapporto $\frac{\alpha}{r} = 0.25$, evitando così modifiche eccessive ai pesi delle matrici originali.

Di seguito è mostrata la **tabella dei risultati** ottenuti durante la fase di sperimentazione:

Modello&Metodo $(r = 64, \alpha = 16)$	# Totale parametri	# Totale parametri addestrabili	% Parametri addestrabili
LLaMA-2-13B (LoRA)	13.068.293.120	52.428.800	0.4012%
LLaMA-3-8B (LoRA)	8.057.524.224	27.262.976	0.3384%
LLaMA-3-70B (QLoRA)	70.684.778.496	131.072.000	0.1854%

Tabella 4.1: Numero totale di parametri, parametri addestrabili e percentuale di parametri addestrabili per ciascun modello grazie alla tecnica LoRA.

Modello&Metodo	Inglese	Italiano	Totale	Batch size	Epoche
	Acc. (%)	Acc. (%)	Acc. (%)	(best-small-dataset)	(small-dataset)
LLaMA-3-8B-Instruct	41.3	37.3	39.3	-	-
LLaMA-3-70B-Instruct	52.0	40.7	46.3	-	-
LLaMA-2-13B-Small-Dataset-SingleFT (LoRA)	75.3	63.3	69.3	4	5
LLaMA-3-8B-Small-Dataset-SingleFT (LoRA)	88.0	88.0	88.0	1	4
LLaMA-3-70-Small-Dataset-SingleFT (QLoRA)	94.0	90.7	92.3	1	4
LLaMA-2-13B-Large-Small-Dataset-DoubleFT (LoRA)	78.0	78.0	78.0	1	4
LLaMA-3-8B-Large-Small-Dataset-DoubleFT (LoRA)	92.0	88.7	90.3	1	4
LLaMA-3-70-Large-Small-Dataset-DoubleFT (QLoRA)	96.0	93.3	94.6	1	5

Tabella 4.2: Accuratezza dei modelli testati.

In tabella 4.2 sono riportati i risultati, espressi in termini di **accuratezza esatta**, ottenuti dai modelli testati. In questo contesto, per **accuratezza esatta** si intende un criterio che tiene conto delle difficoltà legate alla valutazione delle performance dei modelli nel task di text-to-SQL. La valutazione di questo task è particolarmente complessa, sia considerando l'aspetto sintattico che semantico. Per mitigare tale complessità, si è deciso di sfruttare il più possibile i vantaggi offerti dai template costruiti per il test set, descritti nella sottosezione 2.2.2. In particolare, per ciascun template, all'interno del campo “**possible_sql_queries**”, è stato inserito un insieme di possibili query SQL **sintatticamente corrette** che il modello potrebbe generare. Questo approccio permette di evitare che un modello venga penalizzato nel caso in cui, invece di restituire esattamente l'unica query SQL considerata corretta dal punto di vista sintattico, ne produca una leggermente diversa ma ugualmente valida rispetto alla question di partenza.

All'interno della tabella, l'etichetta **Small-Dataset-SingleFT** indica che il modello è stato fine-tunato utilizzando il training set e il validation set, descritti nella sottosezione 2.2.2. L'addestramento è stato effettuato adottando il criterio di Early Stopping sulla funzione di loss del validation set, al fine di prevenire l'overfitting. È stato utilizzato un valore di *patience* pari a 2.

Invece, con **Large-Small-Dataset-DoubleFT** si intende che il modello è stato sottoposto ad un primo fine-tuning sul dataset più grande, descritto nella sottosezione 2.2.1, seguito da un secondo fine-tuning applicando lo stesso approccio appena descritto per i modelli a singolo fine-tuning.

Per entrambe le tipologie di modelli ottenuti, **il calcolo dei valori di accuratezza** è stato eseguito sul **test set descritto nella sottosezione 2.2.2**.

La colonna **Batch size** indica la dimensione del batch utilizzata durante il fine-tuning per il miglior modello ottenuto in termini di accuratezza sul test set. Per ciascun modello sono stati testati tre valori di batch size: 1, 4 e 8.

Invece, la colonna **Epoche** rappresenta il numero di epoche di addestramento sul validation set (descritto nella sottosezione 2.2.2) necessarie per raggiungere il valore di *patience*, determinando così l'interruzione del fine-tuning.

Per il fine-tuning sul dataset più grande è stata utilizzata una sola epoca di addestramento, poiché un aumento di questo valore non ha portato a miglioramenti significativi per giustificare un maggiore impiego di risorse.

Nella tabella, è possibile osservare che i **modelli base LLaMA-3-8B-Instruct** e **LLaMA-3-70B-Instruct** non sono in grado di raggiungere un'accuratezza adeguata per risolvere il task del text-to-SQL. Al contrario, i modelli **LLaMA-2-13B-Small-Dataset-SingleFT (LoRA)**, **LLaMA-3-8B-Small-Dataset-SingleFT (LoRA)** e **LLaMA-3-70-Large-Small-Dataset-DoubleFT (QLoRA)**, fine-tunati utilizzando come training e validation set i dataset descritti in 2.2.2, ottengono performance nettamente superiori, dimostrando di aver appreso correttamente i pattern necessari per il task di riferimento.

Infine, è possibile notare come i modelli **LLaMA-2-13B-Large-Small-Dataset-DoubleFT (LoRA)**, **LLaMA-3-8B-Large-Small-Dataset-DoubleFT (LoRA)** e **LLaMA-3-70-Small-Dataset-SingleFT (QLoRA)**, sottoposti ad un doppio fine-tuning, prima utilizzando come training set l'intero dataset descritto in 2.2.1, e successivamente i dataset di training e validation in 2.2.2, superano nettamente i modelli a singolo fine-tuning. Questo risultato evidenzia come, grazie ad un dataset di training più ampio, i modelli siano stati in grado di apprendere in modo più efficace e generalizzato le caratteristiche salienti del linguaggio SQL, ottenendo un'accuratezza superiore sul test set.

Una piccola osservazione aggiuntiva che vale la pena di menzionare è che il modello **LLaMA-2-13B-Large-Small-Dataset-DoubleFT (LoRA)**, grazie al doppio fine-tuning, riesce a migliorare in maniera significativa sulla lingua italiana anche se il dataset più grande contiene solamente esempi in lingua inglese, raggiungendo praticamente la stessa accuratezza di questa lingua. Inoltre, si potrebbe affermare che il modello in questione, pur non essendo un modello di tipo “Instruct”, abbia in qualche modo acquisito

caratteristiche simili a questa tipologia, soprattutto grazie al doppio fine-tuning. Nonostante questo però, le performance complessive lo collocano piuttosto indietro rispetto agli altri due modelli testati in questa sperimentazione. Per queste ragioni, i modelli **LLaMA-3-8B-Large-Small-Dataset-DoubleFT (LoRA)** e **LLaMA-3-70-Large-Small-Dataset-DoubleFT (QLoRA)** sono stati scelti come modelli finali disponibili, mediante l'utilizzo della web application descritta nel capitolo 3, per la gestione delle inferenze richieste da parte degli utenti.

Qui di seguito è riportata la tabella riassuntiva dei risultati di accuratezza per ciascuna categoria di query, relativa ai tre modelli migliori con doppio fine-tuning menzionati in precedenza:

Modello&Metodo	C1(%)		C2(%)		C3(%)		C4(%)		C5(%)	
	EN	IT								
LLaMA-2-13B-DoubleFT (LoRA)	93.3	90.0	83.3	90.0	70.0	60.0	53.3	53.3	90.0	96.7
LLaMA-3-8B-DoubleFT (LoRA)	100	100	96.7	93.3	70.0	56.7	93.3	93.3	100	100
LLaMA-3-70-DoubleFT (QLoRA)	93.3	96.3	100	100	90.0	80.0	96.7	93.3	100	96.7

Tabella 4.3: Accuratezza dei modelli testati per ciascuna categoria di query in inglese (EN) e italiano (IT).

Dalla tabella 4.3 è possibile osservare che il modello da 70B ottiene complessivamente le migliori performance su quasi tutte le categorie. Tuttavia, anche il modello da 8B mostra prestazioni competitive nella maggior parte dei casi, ad eccezione della categoria C3, in cui registra un'accuratezza sensibilmente inferiore soprattutto rispetto alla lingua italiana. Considerando che la categoria C3 include query contenenti errori nel nome della vista detailedstrain, delle sue colonne o per i suoi valori, è plausibile che la minore capacità parametrica del modello da 8B rispetto al modello da 70B (8 miliardi di contro 70 miliardi di parametri) lo renda meno robusto nell'interpretare e correggere input errati.

Gli stackplot riportati di seguito mostrano il numero di query SQL del test set che sono state generate correttamente o in modo errato da ciascun modello a doppio fine-tuning, suddivise per tipologia di query, sia per la lingua inglese che per la lingua italiana:

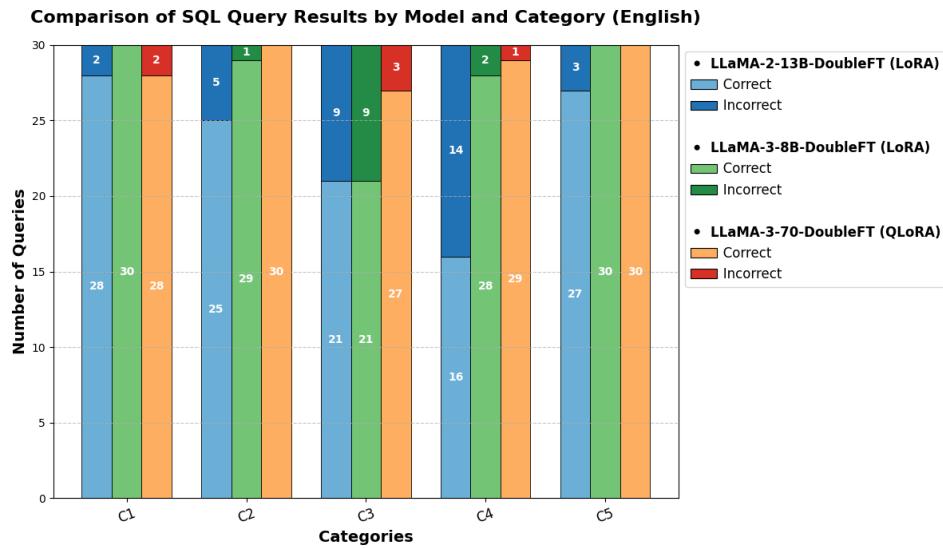


Figura 4.1: Stackplot risultati su ciascuna categoria per la lingua inglese.

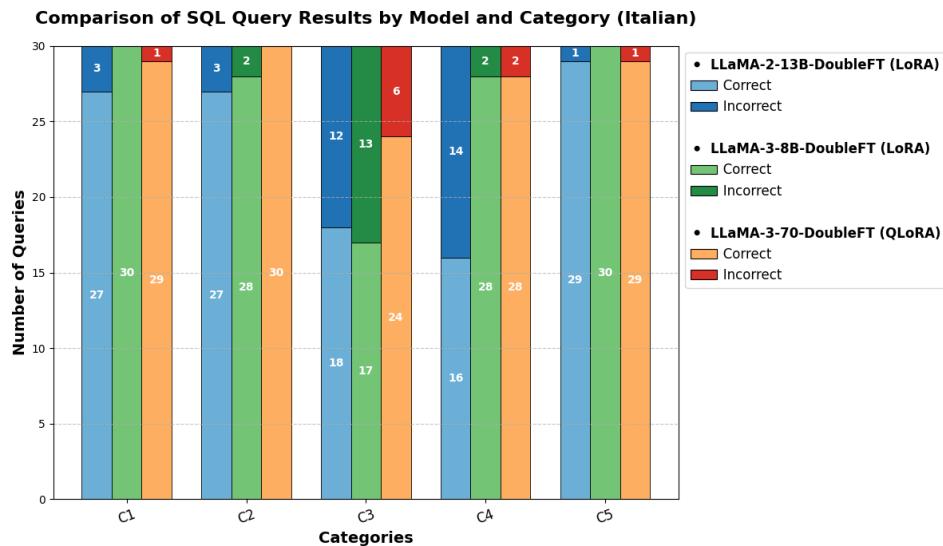


Figura 4.2: Stackplot risultati su ciascuna categoria per la lingua italiana.

Gli stackplot riportati di seguito mostrano i tempi di inferenza medi e la deviazione standard per la generazione delle query SQL presenti

nel test set, per ciascun modello a doppio fine-tuning, suddivise per tipologia di query, sia per la **lingua inglese** che per la **lingua italiana**:

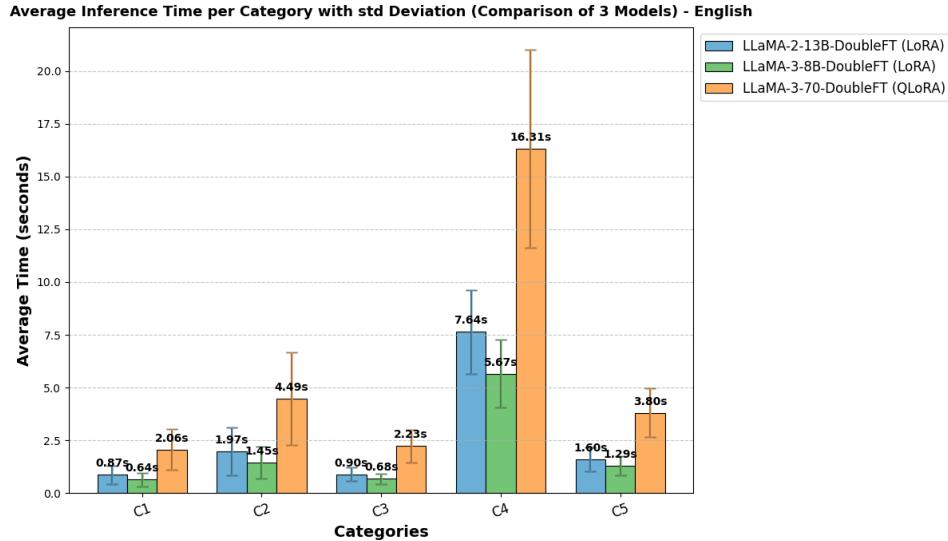


Figura 4.3: Stackplot con tempi medi e deviazione standard relativi alla generazione delle query SQL per la lingua inglese.

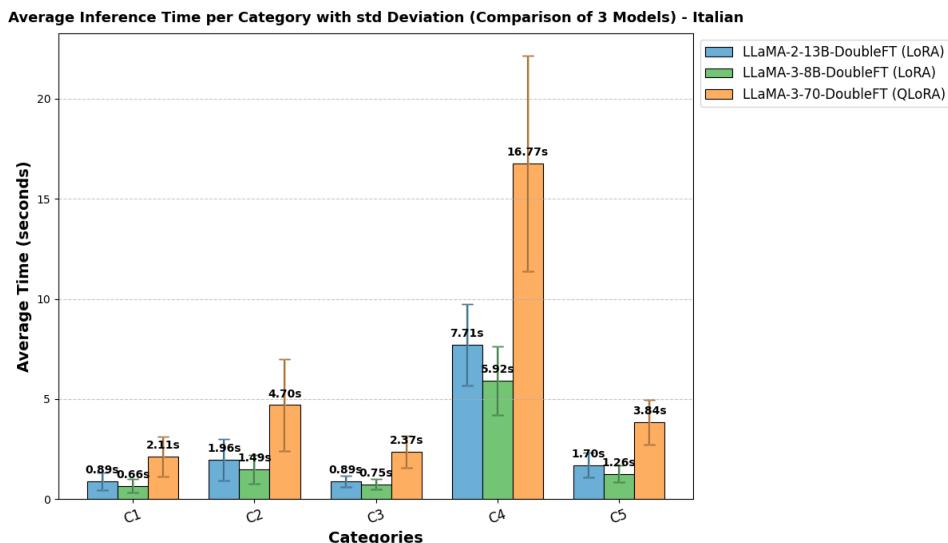


Figura 4.4: Stackplot con tempi medi e deviazione standard relativi alla generazione delle query SQL per la lingua italiana.

Dalle immagini 4.3 e 4.4 è possibile notare come i modelli con dimensioni maggiori (13B e 70B) tendono ad avere dei valori medi di inferenza

superiori rispetto al modello più piccolo (8B). Questo è normale dato che il tempo di generazione è direttamente proporzionale al numero di parametri dei modelli. Inoltre, è possibile notare come tutti e 3 i modelli impiegano un tempo medio nettamente superiore per la generazione delle query di categoria C4. Questa è una normale conseguenza legata al fatto che le query di questa categoria sono più lunghe rispetto alle altre e quindi richiedono la generazione di un numero nettamente maggiore di token.

Nei successivi grafici invece, sono riportati **i tempi richiesti dal primo e dal secondo processo di fine-tuning** sempre per ciascun modello:

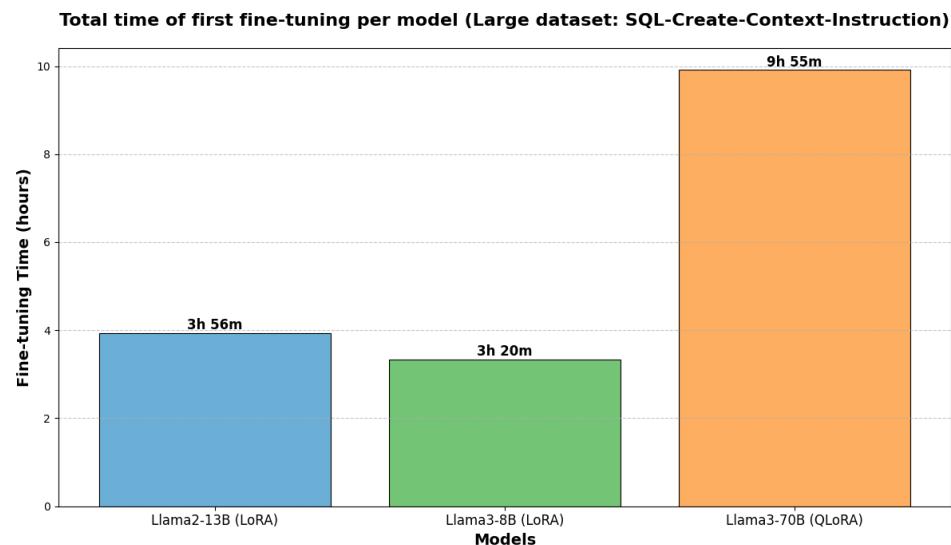


Figura 4.5: Grafico del tempo richiesto da ciascun modello per eseguire il primo fine-tuning sul dataset più grande chiamato **SQL-Create-Context-Instruction**.

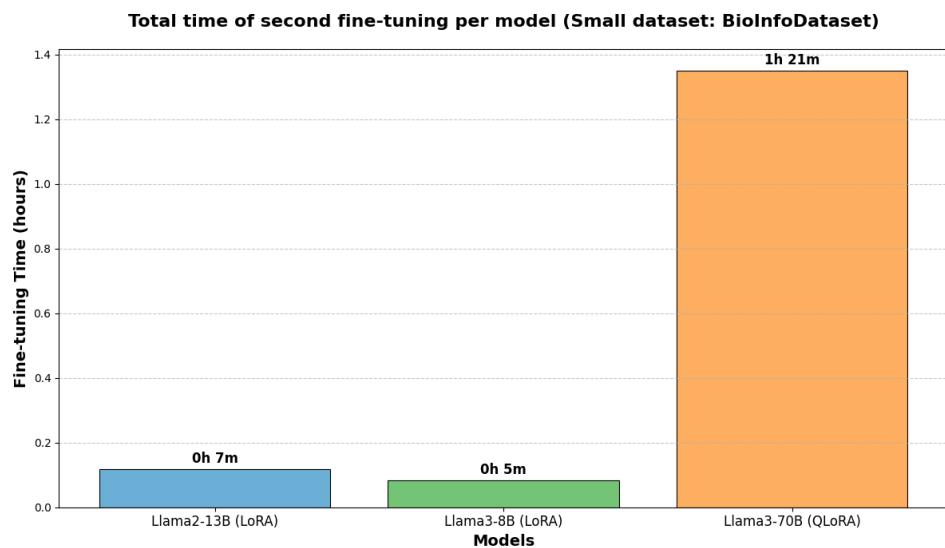


Figura 4.6: Grafico del tempo richiesto da ciascun modello per eseguire il secondo fine-tuning sul dataset più piccolo chiamato **BioInfoDataset** costruito a partire dai dati presenti nel database biologico descritto nella sezione 2.1.

Negli ultimi due grafici, sono riportati i dati sull'occupazione di memoria (media e deviazione standard) della GPU sia per il primo che per il secondo fine-tuning, per ciascun modello:

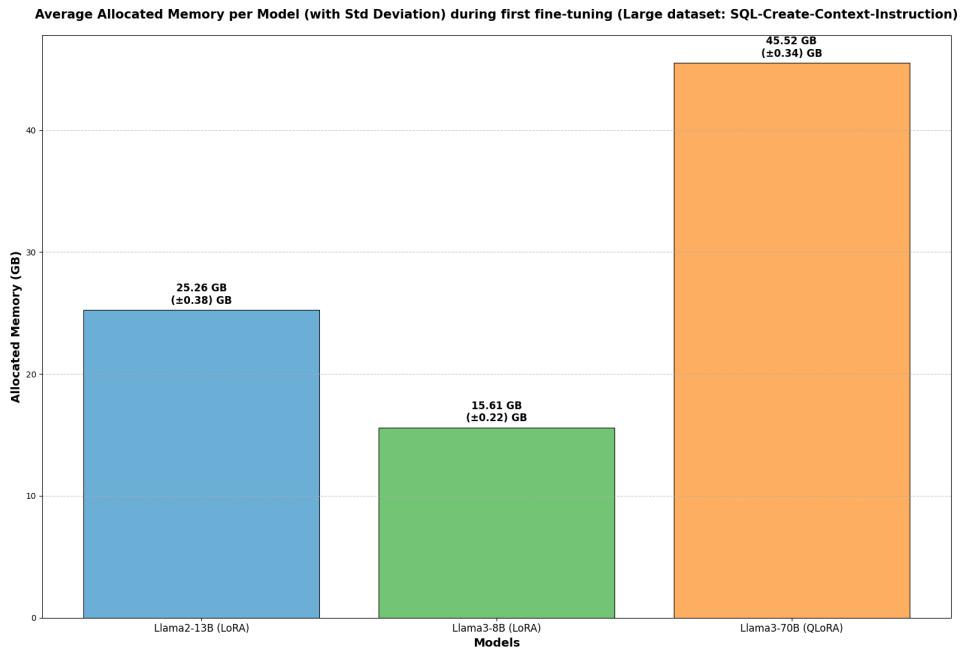


Figura 4.7: Grafico relativo all'occupazione della VRAM, richiesta da ciascun modello per eseguire il primo fine-tuning sul dataset più grande.

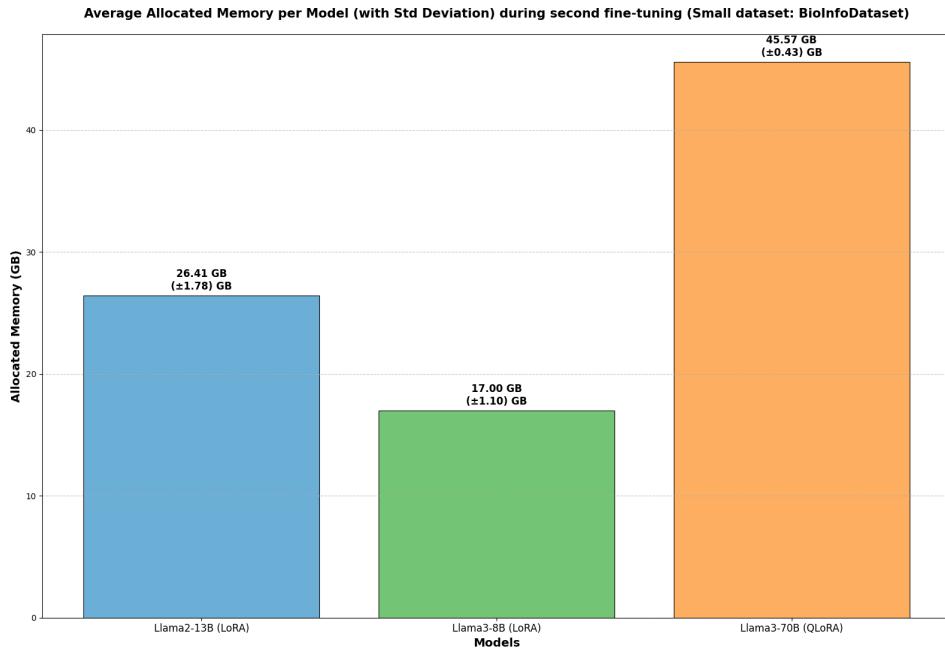


Figura 4.8: Grafico relativo all’occupazione della VRAM, richiesta da ciascun modello per eseguire il secondo fine-tuning sul dataset più piccolo.

4.1.1 Risultati in the wild

Per approfondire la valutazione dei due modelli finali selezionati (**LLaMA-3-8B-Large-Small-Dataset-DoubleFT (LoRA)** e **LLaMA-3-70-Large-Small-Dataset-DoubleFT (QLoRA)**), è stata condotta un’analisi aggiuntiva con l’obiettivo di individuare eventuali punti deboli. **I risultati in termini di accuratezza ottenuti dai due modelli sul dataset *in the wild*** sono riportati nelle seguenti tabelle:

Modello&Metodo	Accuratezza Totale (%)
LLaMA-3-8B-Large-Small-Dataset-DoubleFT (LoRA)	80.8
LLaMA-3-70-Large-Small-Dataset-DoubleFT (QLoRA)	84.6

Tabella 4.4: Accuratezza dei modelli sul dataset *in the wild*.

Modello&Metodo	Accuratezza Totale (%)
LLaMA-3-8B-Large-Small-Dataset-DoubleFT (LoRA)	86.5
LLaMA-3-70-Large-Small-Dataset-DoubleFT (QLoRA)	90.3

Tabella 4.5: Accuratezza dei modelli sul dataset in the wild con l'applicazione del post-processing.

Il dataset **in the wild** contiene un totale di **52** esempi. Dai risultati riportati in Tabella 4.4, si osserva come l'accuratezza di entrambi i modelli sia diminuita rispetto a quella ottenuta sul test set (Tabella 4.2). Questo comportamento è normale, in quanto, quando un modello di Machine Learning viene testato in un ambiente di produzione non controllato, tende generalmente a mostrare un calo delle performance.

Da questi risultati emerge che il **modello da 8B ottiene un'accuratezza inferiore di circa il 3.8% rispetto al modello da 70B**. Questo risultato è coerente con quelli ottenuti durante la fase di sperimentazione (riportati in Tabella 4.2). Tuttavia, rimane significativo, poiché dimostra che, nonostante il numero nettamente inferiore di parametri, il modello da 8B è comunque in grado di garantire prestazioni adeguate anche in un contesto di produzione.

A seguito della sperimentazione in the wild, è emerso che la generazione di una seconda query, sfruttando la parola chiave **ILIKE** del linguaggio SQL, in alcuni casi potesse consentire di correggere la query SQL prodotta inizialmente, generandone una variante leggermente differente. Per comprendere meglio questo aspetto, si consideri il seguente esempio.

Si supponga che la domanda posta dall'utente (question) e la corrispondente query SQL generata dal modello siano le seguenti:

Question

“Dammi tutti gli strain con **organismtype** uguale a **bacteria**.”

Query SQL

```
SELECT * FROM detailedstrain
WHERE organismtype = 'bacteria';
```

A questo punto, la query SQL generata, una volta eseguita sul database biologico, produce un resultset vuoto poichè in realtà all'interno del database è presente il valore ‘Bacteria’ e non ‘bacteria’. Di conseguenza, la query corretta, che restituirebbe risultati validi, dovrebbe essere la seguente:

Query SQL corretta

```
SELECT * FROM detailedstrain  
WHERE organismtype = 'Bacteria';
```

Per risolvere questo problema, è stato sufficiente integrare nel post-processing (descritto nella sottosezione 3.1.1) un'operazione aggiuntiva che genera una nuova query, sostituendo tutte le espressioni della forma [column = `value`] con [column ILIKE `%value%`] solo se `value` non contiene numeri, evitando così altre possibili fonti di errore. In questo modo, se la query originale restituisce un resultset non vuoto, la versione modificata con ‘ILIKE’ viene ignorata. Al contrario, se la query originale non produce risultati, si assume che possa contenere un errore nei valori di confronto, e quindi viene eseguita anche la versione con ‘ILIKE’. Se quest’ultima genera un resultset **non vuoto**, allora il database restituisce questo risultato all’utente insieme alla query con ‘ILIKE’; in caso contrario, viene comunque mantenuta e mostrata la query originale.

Qui sotto è mostrata l’ILIKE query che in questo caso sarà generata dal sistema (dopo il post-processing) e che produrrà un result set non vuoto:

ILIKE Query SQL generata dal post-processing

```
SELECT * FROM detailedstrain  
WHERE organismtype ILIKE '%bacteria%';
```

Grazie a **tutti gli step** seguiti nella **fase di post-processing** (descritti nella sottosezione 3.1.1), dalla Tabella 4.5 si osserva che questa fase ha contribuito a un significativo aumento dell’accuratezza per entrambi i modelli. In particolare, entrambi hanno registrato un **incremento del 5.7%**. Questi risultati suggeriscono che ulteriori ottimizzazioni potrebbero migliorare ulteriormente le prestazioni dei modelli, rendendoli sempre più adatti a un impiego in contesti reali.

Sempre durante la sperimentazione *in the wild*, è emersa la necessità di individuare un metodo per fornire al modello anche la conoscenza sui valori attribuibili alle colonne della vista **detailedstrain** (mostrata in 2.1). In particolare, è possibile supporre che la domanda posta dall'utente e le corrispondenti query (generata e reale) siano le seguenti:

Question

“Give me all the Aspergillus strains with Marine sediment”

Query SQL generata dai modelli

```
SELECT * FROM detailedstrain  
WHERE genus = 'Aspergillus' AND  
isolationhabitat = 'Marine sediment';
```

Query SQL realmente corretta

```
SELECT * FROM detailedstrain  
WHERE genus = 'Aspergillus' AND  
substrate = 'Marine sediment';
```

Questa tipologia di query risulta più ambigua rispetto alla norma, poiché nella question non vengono specificati esplicitamente i nomi dei campi su cui imporre le condizioni. Nonostante ciò, si può osservare come entrambi i modelli siano in grado di disambiguare correttamente la prima condizione, associando in modo corretto `genus = 'Aspergillus'`. Tuttavia, lo stesso non accade per la seconda condizione, poiché i modelli assegnano erroneamente il valore `'Marine sediment'` alla colonna `isolationhabitat` anziché a `substrate`.

Per affrontare questo problema, nel capitolo successivo saranno descritti due possibili approcci che potranno essere implementati in futuro.

Capitolo 5

Conclusioni e Sviluppi futuri

In questo lavoro di tesi è stato approfondito il task del text-to-SQL, con l'obiettivo di sviluppare un'applicazione basata su modelli di linguaggio in grado di fornire una possibile soluzione automatizzata a tale problema. È stato notato durante la sperimentazione descritta nel capitolo 4, che grazie al fine-tuning è possibile specializzare gli LLM della famiglia Llama, sulla generazione delle query SQL corrette, sfruttando le informazioni riguardanti la struttura di un database. Grazie a tecniche come LoRA e QLoRA, è stato possibile affinare questi modelli anche con risorse computazionali significativamente inferiori rispetto a quelle impiegate per l'addestramento dei modelli pre-addestrati. Inoltre, è stato notato che grazie ad un **singolo fine-tuning** su un dataset specifico relativo al database biologico di riferimento, le **performance**, in termini di accuratezze dei modelli, sono **migliorate in modo significativo di circa il 48.7%** per il Llama-3-8B e **di circa il 46%** per il modello Llama-3-70B rispetto ai propri modelli base. Dopodichè, **grazie ad un doppio fine-tuning**, prima su un dataset più ampio relativo al text-to-SQL e poi su quello più specifico, è stato notato un **ulteriore incremento di circa il 2.3%** sia per il modello ad 8B che per quello a 70B rispetto alle versioni a singolo fine-tuning. Infine, nella **sperimentazione “in the wild”**, è stato osservato come la fase di post-processing (descritta nella sottosezione 3.1.1) abbia permesso di **incrementare le performance** di entrambi i modelli del **5.7%**. Tali risultati, mettono in evidenza le grandi capacità di apprendimento nascoste all'interno di questi grandi modelli statistici del linguaggio naturale, suggerendo un potenziale sempre maggiore per il loro futuro utilizzo nella risoluzione, in maniera sempre più precisa, del task del text-to-SQL.

Sono stati individuati due principali limitazioni nella versione attuale dell'applicazione: un **limite architetturale** e un **limite sulla conoscenza dei valori**. Di seguito, ciascuno di essi verrà descritto con maggiore dettaglio e saranno proposte possibili soluzioni per cercare di mitigarne gli effetti

e migliorare le performance dell'intero sistema.

Limite Architetturale:

Come descritto nel capitolo 3, il sistema utilizza 4 nodi gestiti da SLURM, tale approccio però presenta fondamentalmente due problemi:

1. Un nodo potrebbe non essere disponibile in un certo momento con conseguente aumento dei tempi di attesa per gli utenti che fanno richiesta al sistema.
2. Ogni volta che un utente invia una richiesta, il modello selezionato deve essere caricato all'interno della VRAM della GPU disponibile in quel momento. Questa operazione rappresenta il principale collo di bottiglia nella gestione delle richieste, in quanto il tempo necessario per il caricamento del modello incide significativamente sui tempi complessivi di risposta. Nello specifico, questo ritardo si traduce in un tempo medio di circa 30 secondi per il modello da 8B e 70 secondi per il modello da 70B. Dai grafici relativi ai tempi di inferenza per ciascuna categoria (Figura 4.3 e Figura 4.4), è possibile osservare che, ad eccezione del modello da 70B sulla categoria C4 (più complessa), i tempi di inferenza effettivi sono notevolmente inferiori rispetto ai valori medi di attesa appena citati.

Questo limite potrebbe essere superato disponendo delle risorse necessarie per dedicare un server ad uso esclusivo del sistema. In questo modo, i modelli potrebbero essere caricati una sola volta e mantenuti sempre attivi, riducendo drasticamente i tempi di risposta complessivi. Tuttavia, ciò richiederebbe di riservare una GPU al sistema in modo permanente, con un conseguente spreco di risorse nei momenti di inattività.

Limite sulla conoscenza dei valori:

Come mostrato alla fine del capitolo precedente, dal punto di vista dei modelli, il principale problema emerso, in particolare durante la fase di sperimentazione *in the wild*, riguarda la loro attuale mancanza di conoscenza sui valori dei campi presenti nella vista **Detailedstrain**. Per affrontare questa limitazione, si potrebbero seguire due approcci distinti:

1. Ampliare il contesto nei dati di addestramento.

Un possibile intervento consiste **nell'includere informazioni più dettagliate nei prompt di addestramento e validazione**. Tuttavia, questo approccio è vincolato dalla dimensione della finestra di contesto dei modelli, il che potrebbe rappresentare un limite significativo. Inoltre, l'aggiunta di informazioni specifiche sui valori delle

colonne potrebbe compromettere la generalizzazione del modello sul test set, poichè un eccesso di dettagli rischierebbe di far perdere al modello la capacità di adattarsi a nuove query, riducendo l'abilità di generalizzazione sul task del text-to-SQL.

Un'alternativa per mitigare questa limitazione, consiste nel selezionare un sottoinsieme di valori associati alle colonne ritenute più rilevanti dagli esperti di dominio (in questo caso, i biologi), in modo tale che questi possano essere inclusi negli esempi di input, evitando un sovraccarico informativo.

2. Implementazione di un algoritmo deterministico basato su un meccanismo di sostituzione.

Struttura dati da costruire: si propone la costruzione di un **dizionario valore-colonna/e**, in cui ogni chiave corrisponde ad un possibile valore presente in almeno una colonna della vista **Detailedstrain**. Il valore mappato può essere costituito dal nome della colonna o delle colonne in cui tale valore compare. Per ottimizzare ulteriormente la ricerca, nel caso in cui un valore sia presente in più colonne, si potrebbe selezionare la colonna in cui esso è più frequente, definendola come **colonna più probabile**. Questa potrebbe quindi corrispondere ad esempio alla colonna che, tra tutte quelle che contengono quel valore, lo presenta con maggiore frequenza nel database.

Meccanismo di sostituzione: a questo punto, **per ciascun valore associato ad una colonna errata** nella query SQL generata dal modello, il dizionario valore-colonna/e può essere sfruttato per **sostituire tale colonna con la prima colonna più probabile per quel valore**.

Un ulteriore perfezionamento di questo meccanismo potrebbe prevedere l'uso di una metrica di similarità, ad esempio la **distanza di Levenshtein**, per gestire i casi in cui il valore identificato nella query generata non corrisponda esattamente ad un valore presente nel dizionario. In tal caso, si potrebbe mappare il valore sulla sua versione più simile all'interno del dizionario, migliorando la robustezza del sistema.

I principali vantaggi di questo approccio sono:

- Il dizionario può essere facilmente precalcolato.
- Non è necessario riaddestrare i modelli, permettendo un miglioramento delle prestazioni senza richiedere risorse computazionali aggiuntive per il fine-tuning.

Un possibile svantaggio potrebbe essere un aumento dell'overhead nell'elaborazione dell'inferenza, dovuto all'integrazione del meccanismo

di sostituzione all'interno della fase di post-processing. Tuttavia, tale costo non dovrebbe risultare significativo grazie all'accesso diretto fornito dal dizionario.

Infine, per migliorare ulteriormente i modelli, si potrebbero modificare e/o aggiungere nuovi template nei dataset di training, validation e test, al fine di generare dati aggiuntivi, possibilmente ancora più generici, su cui eseguire un ulteriore fine-tuning. Questo permetterebbe di rendere i modelli più efficaci nel task di riferimento, enfatizzando allo stesso tempo le caratteristiche specifiche del database biologico.

Ovviamente l'ulteriore sperimentazione potrà estendersi anche ad altri modelli pre-addestrati magari anche non appartenenti alla famiglia dei modelli Llama ma basati comunque sulla medesima architettura transformer (ad esempio: GPT, Mistral, Falcon, Gemini, Claude, Qwen, Gemma, ecc..). Sarebbe interessante non solo confrontare le prestazioni di questi modelli sul task del text-to-SQL, ma anche analizzare i costi, sia computazionali che economici, legati al loro utilizzo. Un'analisi approfondita su questo aspetto potrebbe essere utile per riuscire ad individuare il miglior compromesso tra accuratezza e costi, soprattutto in riferimento al deploy in ambienti di produzione.

In conclusione, è possibile affermare che l'obiettivo principale di questo lavoro di tesi sia stato raggiunto. Tuttavia, ulteriori miglioramenti all'intero sistema saranno necessari per riuscire a perfezionarlo in modo tale da aumentarne l'efficacia in un contesto di utilizzo reale.

Considerazioni finali:

Questa tesi evidenzia, qualora ce ne fosse ancora bisogno, l'incredibile capacità dei Large Language Model nell'interpretazione del linguaggio naturale. Molte sono le sfide che restano ancora da affrontare per migliorare tali modelli e per comprendere in modo più preciso i meccanismi che gli permettono di svolgere compiti che fino a pochi anni fa sembravano irraggiungibili per i sistemi di intelligenza artificiale. Questo discorso si potrebbe estendere ai numerosi modelli recentemente sviluppati, che si applicano non solo alla generazione di testo, ma anche ad altri ambiti come la creazione di audio e video. Una parte del fascino di queste tecnologie risiede anche nella loro crescente utilità all'interno della società, in tutte le discipline sia scientifiche che umanistiche. Ad esempio, in questo lavoro di tesi abbiamo visto come un sistema di AI possa supportare i biologi nell'effettuare ricerche su un database, pur non avendo una conoscenza diretta né del linguaggio SQL né della struttura dei dati. Tuttavia, sarà fondamentale istruire persone con background culturali differenti affinché comprendano anche i limiti di tali strumenti. Inoltre, bisognerà sviluppare regolamentazioni adeguate che

ne garantiscano l'uso e la diffusione in modo etico e coerente con i nostri principi morali. Gli elementi fondanti per costruire una società più evoluta ci sono tutti, ora spetta a noi sviluppare la consapevolezza e la volontà di utilizzarli a beneficio dell'umanità.

Ringraziamenti

Arrivato al termine di questo percorso, sento il bisogno di fermarmi un attimo, pensare a tutto ciò che mi ha portato fin qui e ringraziare tutte quelle persone, che in un modo o nell'altro, hanno reso possibile il raggiungimento di questo traguardo. Questo lavoro di tesi per me rappresenta la chiusura di un cerchio, la conclusione di un percorso di vita che mi ha permesso di crescere e maturare non solo dal punto di vista delle conoscenze ma anche personale. Gli anni universitari sono stati un banco di prova importante, un'opportunità per mettermi in gioco e dimostrare, prima di tutto a me stesso, che con passione e costanza anche gli obiettivi che sembrano impossibili possono diventare realtà. Le difficoltà affrontate lungo il cammino, più o meno complesse che siano state, mi hanno permesso di cadere, rialzarmi e imparare a scoprire aspetti del mio carattere che non conoscevo.

Desidero ringraziare i miei relatori Marco, Luigi e Sandro per avermi accompagnato in questa avventura. Le vostre idee e il vostro supporto sono stati fondamentali per lo sviluppo di questa tesi, non avrei potuto chiedere di meglio.

Il mio pensiero più profondo va a tutta la mia famiglia. A voi che mi avete cresciuto con amore, che avete sempre creduto in me, anche quando io stesso faticavo a farlo. Siete stati la mia forza nei momenti difficili, il mio rifugio sicuro e la mia più grande motivazione. Ogni sacrificio che avete fatto per me non è mai passato inosservato, ogni vostro incoraggiamento mi ha dato la spinta per andare avanti. Questo traguardo non è solo mio, ma Nostro.

Un ringraziamento speciale va a tutti i miei nonni: Dante, Clotilde, Michele e Teresa. Per essermi stati accanto con il loro spirito in ogni istante di vita durante questi anni. Non ho mai avuto bisogno di vedervi fisicamente, perché so che siete stati sempre al mio fianco, pronti a sostenermi.

Grazie a tutti i miei amici, che hanno reso questo viaggio più leggero. Forse non lo sapete perchè spesso cerco di nasconderlo, ma siete stati la mia risata anche nei giorni più difficili.

E infine, un ringraziamento, che potrebbe determinare la mia sopravvivenza dopo la consegna di questa tesi: alla donna della mia vita Agnese. Grazie per aver creduto in me ogni giorno, per aver condiviso le mie ansie, le mie paure ma anche i momenti di felicità e spensieratezza. Grazie per la tua pazienza, che so essere veramente tanta.. Forse non te l'ho mai detto abbastanza, ma per me sei lo stimolo fondamentale per il raggiungimento di qualsiasi obiettivo. Grazie per rendermi ogni giorno un uomo migliore. Sei il mio ombrello giallo che auguro a qualsiasi uomo di poter trovare nella propria vita.

Questo percorso rimarrà per sempre un tassello importante della mia vita.
Un tassello...



Bibliografia

- [1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin. “Attention is All You Need.” *Advances in Neural Information Processing Systems*, 2017.
URL: <https://arxiv.org/abs/1706.03762>.
- [2] Transformer (deep learning architecture). URL:
[https://en.wikipedia.org/wiki/Transformer_\(deep_learning_architecture\)](https://en.wikipedia.org/wiki/Transformer_(deep_learning_architecture)).
- [3] Rothman, D., & Gulli, A. (2022). Transformers for Natural Language Processing. Packt Publishing Ltd.
- [4] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning. arXiv preprint
URL: <https://arxiv.org/abs/1705.03122>, 2017.
- [5] Bearing Fault Diagnosis for Time-Varying System Using Vibration-Speed Fusion Network Based on Self-Attention and Sparse Feature Extraction - Scientific Figure on ResearchGate. URL:
https://www.researchgate.net/figure/a-Multihead-self-attention-module-b-Scaled-dot-product-attention_fig3_364435023
- [6] The Illustrated Transformer. URL:
<https://jalammar.github.io/illustrated-transformer/>
- [7] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. “Layer Normalization.” *arXiv preprint arXiv:1607.06450*, 2016. URL:
<https://arxiv.org/abs/1607.06450>.
- [8] Visual Information Theory. URL:
<https://colah.github.io/posts/2015-09-Visual-Information/>.
- [9] Kullback-Leibler Divergence Explained. URL:
<https://www.countbayesie.com/blog/2017/5/9/>

[kullback-leibler-divergence-explained](#).

[10] Elaborazione del linguaggio naturale. URL: https://it.wikipedia.org/wiki/Elaborazione_del_linguaggio_naturale.

[11] Cos’è l’NLP? URL:
<https://www.ibm.com/it-it/topics/natural-language-processing>.

[12] Self-Training for End-to-End Speech Recognition. URL:
<https://arxiv.org/abs/1909.09116v2>.

[13] Semi-Supervised Learning, Explained with Examples. URL:
<https://www.altexsoft.com/blog/semi-supervised-learning/>.

[14] A survey on deep learning approaches for text-to-SQL. URL:
<https://link.springer.com/article/10.1007/s00778-022-00776-8>.

[15] A Survey on Text-to-SQL Parsing: Concepts, Methods, and Future Directions. URL: <https://arxiv.org/abs/2208.13629>.

[16] Brunner, U., Stockinger, K.: Valuenet: “A neural text-to-sql architecture incorporating values (2020).” URL:
<https://arxiv.org/abs/2006.00888>.

[17] Guo, J., Zhan, Z., Gao, Y., Xiao, Y., Lou, J.-G., Liu, T., Zhang, D.: “Towards complex Text-to-SQL in cross-domain database with intermediate representation (2019).”
URL:<https://arxiv.org/abs/1905.08205>.

[18] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet Marie-Anne Lachaux, Timothee Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin Edouard Grave, Guillaume Lample: “LLaMA: Open and Efficient Foundation Language Models.” URL:<https://arxiv.org/abs/2302.13971>.

[19] Hugo Touvron et al.: “Llama 2: Open Foundation and Fine-Tuned Chat Models.” URL: <https://arxiv.org/abs/2307.09288>.

[20] Llama Team, AI @ Meta: “The Llama 3 Herd of Models.” URL:
<https://arxiv.org/abs/2407.21783>.

[21] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. 2021. “Roformer: Enhanced transformer with rotary position embedding.” URL:<https://arxiv.org/abs/2104.09864>.

- [22] Jacob Devlin Ming-Wei Chang Kenton Lee Kristina Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” URL:<https://arxiv.org/abs/1810.04805>.
- [23] Victor Zhong, Caiming Xiong, Richard Socher. “Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning.” URL:<https://arxiv.org/abs/1709.00103>.
- [24] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, Dragomir Radev. “Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task.” URL:<https://aclanthology.org/D18-1425/>.
- [25] “Report D3.4 Design of the ITCCC Catalogue.”
- [26] “SQL-Create-Context-Instruction.” URL:<https://huggingface.co/datasets/bugdaryan/sql-create-context-instruction>.
- [27] “SQL-Create-Context.” URL:<https://huggingface.co/datasets/b-mc2/sql-create-context>.
- [28] “SQLGlot.” URL:<https://github.com/tobymao/sqlglot>.
- [29] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen. “LoRA: Low-Rank Adaptation of Large Language Models.” URL:<https://arxiv.org/abs/2106.09685>.
- [30] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, Luke Zettlemoyer. “QLoRA: Efficient Finetuning of Quantized LLMs.” URL:<https://arxiv.org/abs/2305.14314>.