Quantum Transformer:
Quantum Self-Attention in Transformer Models for Token Generation

Bernardo Manfriani, Edoardo Cappelli

University of Florence

Abstract

Questo progetto esplora un'architettura ibrida quantistico-classica basata su Transformer per la generazione molecolare condizionata. L'approccio consiste nell'integrare circuiti quantistici parametrizzati all'interno del meccanismo di self-attention, utilizzandoli per calcolare i prodotti scalari tra vettori *query* e *key*, mentre la moltiplicazione con la matrice dei valori (*value*) e il resto dell'architettura rimangono classici. L'obiettivo è valutare il contributo dei circuiti quantistici nell'implementazione del meccanismo di attenzione nei transformer, dimostrando la loro applicabilità.

Il lavoro si articola in due parti: nella prima (sezioni 1, 2) introduciamo gli obiettivi della ricerca, la struttura dei Transformer classici e le basi della computazione quantistica. Nella seconda parte (sezioni 3, 4, 5, 6) descriviamo l'implementazione concreta, i dataset utilizzati e i risultati ottenuti e l'estensione di dominio.

In questo lavoro presentiamo un Quantum Transformer facendo riferimento al paper Smaldone et. al.[5] dove la similarità tra gli elementi della sequenza (token codificati con informazioni posizionali) non viene calcolata tramite un prodotto scalare classico, ma stimata attraverso circuiti quantistici parametrizzati (PQC).

Nel paper analizzato il modello è stato inizialmente addestrato sul dataset QM9[4], che contiene molecole rappresentate in formato SMILES[7], con proprietà chimico-fisiche utilizzate come condizioni in fase di generazione (da noi rimosse per semplificazione). Noi, per ragioni di complessità computazionale abbiamo comunque strutturato la fase di training ma sfruttato i pesi da loro messi a disposizione e applicato l'architettura adatta per la fase di inferenza. Abbiamo poi sperimentato un cambio di dominio, addestrando il modello su una collezione di terzine tratte dall'*Inferno* di Dante[1], con l'obiettivo di generare testo in stile dantesco.

Contents

1 Introduzione: Obiettivi e approccio

Negli ultimi anni, **machine learning** e **quantum computing** sono diventati due pilastri della ricerca scientifica contemporanea. Da un lato, lo sviluppo di modelli capaci di interpretare, analizzare, predire e generare dati; dall'altro, l'esplorazione di nuovi paradigmi computazionali basati sui principi della meccanica quantistica e sulla costruzione di hardware dedicato. La convergenza tra questi ambiti ha dato origine al campo del **Quantum Machine Learning** (QML)[2], che mira a sfruttare le peculiarità dello spazio di Hilbert per potenziare l'efficienza e la capacità espressiva dei modelli di apprendimento automatico.

Tra le architetture più promettenti per questa integrazione ci sono i *Transformer*[1], largamente utilizzati in NLP, computer vision, bioinformatica e chimica computazionale. L'idea alla base di questo lavoro è quella di sostituire parzialmente i componenti classici di un Transformer con *Quantum Variational Circuits*[2], per costruire un'architettura ibrida che sfrutti il meglio di entrambi i mondi.

In questo progetto, abbiamo studiato e riprodotto in versione semplificata il lavoro *"A Hybrid Transformer Architecture with a Quantized Self-Attention Mechanism Applied to Molecular Generation"*[5]. Il paper propone un **Transformer decoder ibrido** (quantum-classical) per la generazione di molecole in formato SMILES[3].

La differenza chiave rispetto a un Transformer classico è che il calcolo del prodotto scalare tra *query* e *key* ($QK^\top$) viene effettuato tramite circuiti quantistici, utilizzando uno *Hadamard test* modificato. La successiva moltiplicazione con il valore (V) resta classica. In questo modo si implementa una *self-attention quantistica*, mantenendo inalterata l'architettura complessiva del decoder Transformer.

A differenza di approcci precedenti che richiedevano un numero di qubit pari alla dimensione dell'embedding d, qui si ottiene una scalabilità più efficiente grazie all'uso di soli $O(\log d)$ qubit, riducendo il costo computazionale senza sacrificare la qualità della generazione.

L'obiettivo principale di questo lavoro è stato la progettazione e l'implementazione di un *Transformer* ibrido per la generazione molecolare (e, in prospettiva, testuale), capace di apprendere da sequenze simboliche (SMILES). Il nostro approccio include i seguenti passaggi:

- **Tokenizzazione SMILES** con aggiunta dei token speciali `[CLS]` e `[EOS]`;
- **Conversione delle stringhe SMILES** in sequenze numeriche tramite espressioni regolari, con costruzione del vocabolario e padding;
- **Integrazione di embedding multipli** — token, posizionali e proprietà chimico-fisiche — normalizzati e scalati per l'iniezione nel modello;
- **Calcolo delle self-attention scores** tramite circuiti quantistici per Q e K, lasciando classico il calcolo di V;

[1] Modelli basati sul meccanismo di self-attention, introdotti da Vaswani et al. (2017)[6], capaci di catturare relazioni a lungo raggio nelle sequenze. Sono diventati lo standard de facto in NLP e oltre.

[2] Circuiti quantistici parametrizzati (PQC) che contengono gate dipendenti da parametri ottimizzabili e permettono di apprendere trasformazioni nello spazio di Hilbert.

[3] **S**implified **M**olecular **I**nput **L**ine **E**ntry **S**ystem: una notazione lineare che descrive molecole mediante una stringa di caratteri ASCII rappresentanti atomi, legami, cicli e ramificazioni.

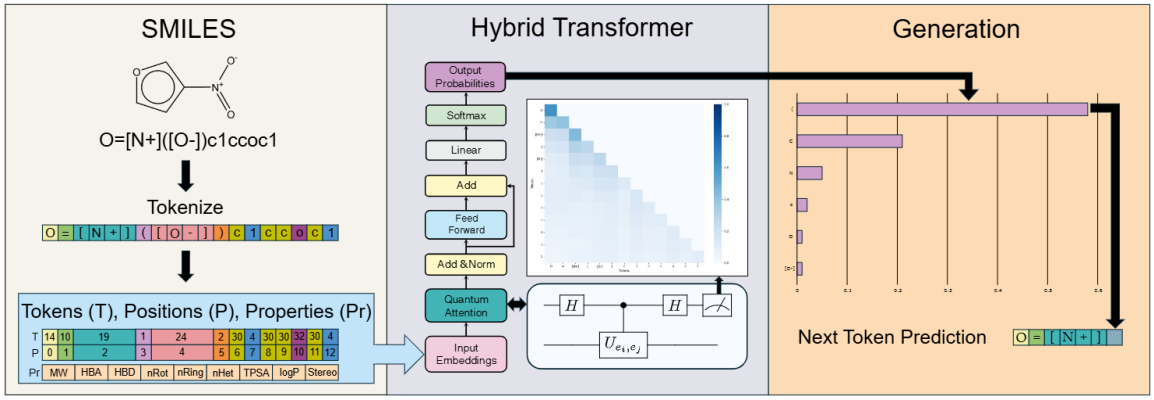


Figure 1: Il modello ibrido quantum-classical genera molecole a partire da stringhe SMILES, combinando embedding di token, posizione e proprietà chimico-fisiche. L'attenzione è calcolata tramite circuiti quantistici, mentre il resto del decoder Transformer resta classico, permettendo una generazione condizionata su proprietà target.

- **Ottimizzazione dei parametri quantistici** tramite l'algoritmo SPSA (*Simultaneous Perturbation Stochastic Approximation*).
- **Cambio di dominio:** valutazione della generalizzazione del modello addestrandolo su un dominio testuale non molecolare, tramite un dataset ispirato all'*Inferno* di Dante[1]. L'obiettivo è verificare se l'architettura ibrida mantiene buone capacità generative anche in contesti fuori dominio.

---

2 Background: SMILES, Transformer e Quantum circuits

Un'introduzione ai principali elementi teorici dell'architettura: rappresentazione SMILES, Transformer e circuiti quantistici.

2.1 SMILES: una rappresentazione testuale delle molecole

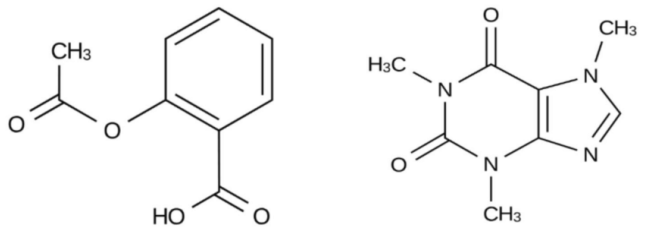


Figure 2: Esempio di rappresentazione tramite SMILES

Il formato SMILES (*Simplified Molecular Input Line Entry System*) è una notazione testuale compatta e lineare che descrive la struttura di una molecola attraverso una sequenza di caratteri ASCII. Ogni stringa SMILES codifica in modo univoco atomi, legami, cicli e ramificazioni,

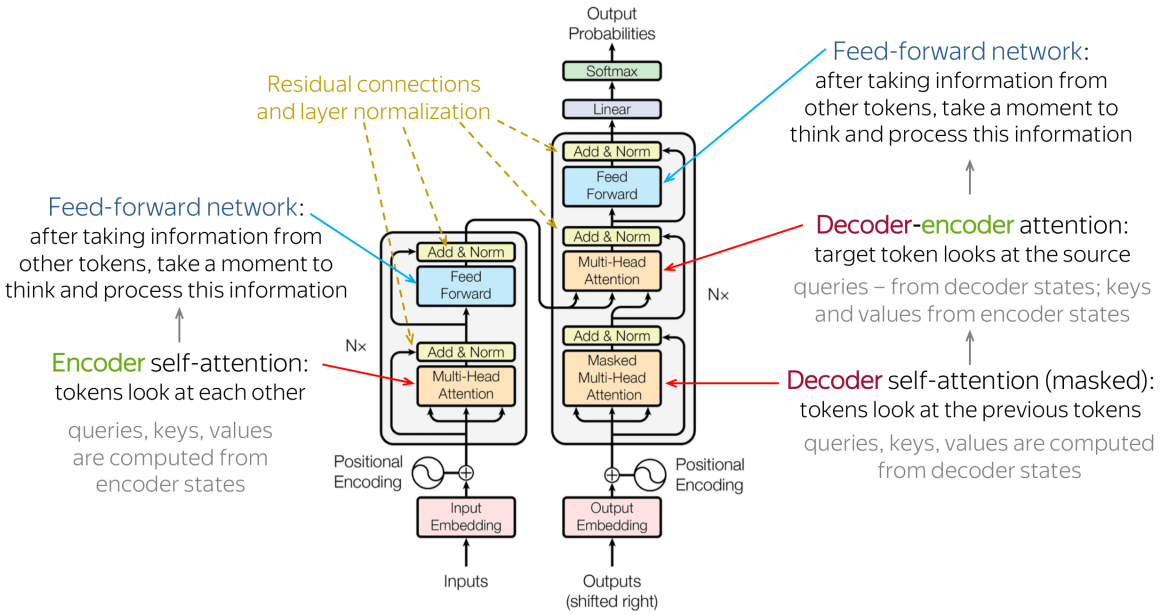
risultando particolarmente adatta per essere processata da modelli di *machine learning* e *natural language processing*.

In questo contesto, le stringhe SMILES costituiscono l'input principale del modello Transformer. Per renderle compatibili con l'elaborazione sequenziale, ogni stringa viene sottoposta a una pipeline di pre-elaborazione composta dai seguenti passaggi:

- **Canonicalizzazione**: viene utilizzata la libreria RDKit per standardizzare la rappresentazione molecolare ed eliminare eventuali duplicati;
- **Tokenizzazione**: una espressione regolare segmenta la stringa SMILES in unità atomiche e strutturali (es. atomi, legami, parentesi, cariche);
- **Inserimento di token speciali**: si aggiungono token come `[CLS]` all'inizio e `[EOS]` alla fine della sequenza [CLS]CC(=O)Oc1ccccc1C(=O)0[EOS];
- **Costruzione del vocabolario**: ogni token viene mappato a un indice numerico attraverso una tabella di corrispondenza;
- **Padding**: le sequenze vengono uniformate in lunghezza tramite l'aggiunta del token speciale `<pad>`.

Questa rappresentazione simbolica permette di trattare le molecole come vere e proprie sequenze linguistiche, rendendo possibile l'applicazione di modelli sequenziali come i Transformer all'analisi e alla generazione di strutture chimiche.

2.2 Transformer architecture



I modelli *Transformer* costituiscono oggi uno standard consolidato per la modellazione di sequenze in una vasta gamma di domini, dal linguaggio naturale alla chimica computazionale. Introdotti da Vaswani et al.[6], si sono rapidamente affermati grazie alla loro capacità di processare sequenze in parallelo, superando le limitazioni delle reti ricorrenti nella gestione delle dipendenze a lungo termine.

Il cuore dell'architettura è rappresentato dal meccanismo di *self-attention*, che consente a ogni token della sequenza di interagire con tutti gli altri, modulando la propria rappresentazione in base al contesto globale. Questo approccio permette di costruire rappresentazioni dinamiche e contestualizzate, fondamentali per catturare relazioni semantiche complesse.

In questa sezione presentiamo una panoramica delle componenti strutturali essenziali dei Transformer, descrivendone il funzionamento generale indipendentemente dal dominio di applicazione. In seguito, questa struttura verrà applicata a due casi studio: la generazione di stringhe molecolari (SMILES), e la generazione di testo poetico ispirato all'*Inferno* di Dante.

La trattazione evidenzierà la flessibilità dell'architettura Transformer, capace di adattarsi a contesti semantici differenti e di integrare efficacemente anche moduli computazionali non convenzionali, come i circuiti quantistici.

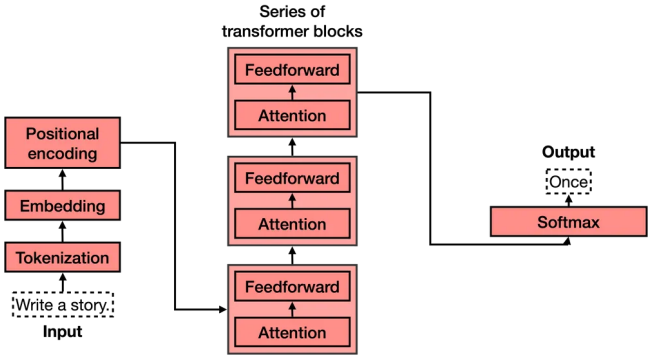


Figure 3: Architettura generale del Transformer.

I Transformer si articolano in cinque componenti fondamentali:

- **Tokenizzazione**, che converte l'input testuale o simbolico in una sequenza discreta di unità chiamate *token*;
- **Embedding**, che trasforma ciascun token in un vettore denso di dimensione fissa;
- **Positional encoding**, che introduce esplicitamente l'informazione sulla posizione dei token nella sequenza;
- **Blocchi Transformer**, che implementano l'alternanza tra meccanismi di self-attention e reti neurali feed-forward, eventualmente replicati in profondità;
- **Softmax finale**, che proietta l'output nello spazio del vocabolario per produrre la distribuzione di probabilità sul token successivo.

2.2.1 Tokenizzazione

La *tokenizzazione* è il primo passo del processo di elaborazione. Essa suddivide l'input in unità discrete chiamate *token*, che possono essere parole, sottoparti di parola (subword), caratteri o simboli, a seconda del tokenizer utilizzato.

Formalmente, data una stringa $T \in \mathcal{A}^*$, dove $\mathcal{A}$ è l'alfabeto di riferimento (ad esempio ASCII o Unicode), la tokenizzazione può essere definita come una funzione:

$$\tau(T) = [t_1, t_2, \ldots, t_n], \quad t_i \in \mathcal{V}$$

dove $\mathcal{V}$ è il vocabolario dei token e n è la lunghezza della sequenza.

Ogni token viene successivamente associato a un indice numerico tramite una funzione di codifica $\phi : \mathcal{V} \to \mathcal{N}$, rendendo la sequenza compatibile con gli strati successivi del modello neurale.

Esempio di tokenizzazione

Consideriamo la molecola: `O=[N+]([O-])c1ccoc1`.

Dopo la tokenizzazione e l'aggiunta dei token speciali, la sequenza risultante è:

`[CLS], O, =, [, N, +, ], (, [, O, -, ], ), c, 1, c, c, o, c, 1, [EOS]`

Ogni token viene successivamente mappato in un indice intero secondo il vocabolario costruito, producendo ad esempio una sequenza numerica del tipo:

`14, 10, 19, 1, 2, 30, 30, 30, 30, 44, 32, 24, ...`

dove i numeri rappresentano gli indici dei token nel dizionario. Questa forma vettoriale consente il trattamento sequenziale della molecola, abilitando operazioni di embedding, self-attention e predizione token-by-token tramite il modello Transformer.

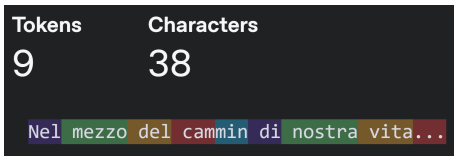


Figure 4: Esempio di tokenizzazione eseguita da GPT-4o.

2.2.2 Embedding

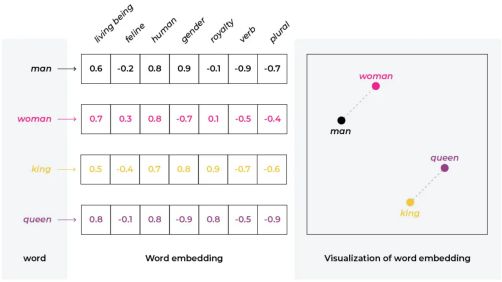


Figure 5: Embedding scheme

A ciascun token viene associato un vettore numerico continuo, chiamato *embedding*, che ne rappresenta il significato in uno spazio vettoriale $\mathcal{R}^d$, dove d è la dimensione dell'embedding. La funzione di embedding è definita come:

$$\phi : \mathcal{V} \to \mathcal{R}^d, \quad \phi(t_i) = \mathbf{e}_i$$

ed è implementata mediante una matrice $E \in \mathcal{R}^{|\mathcal{V}| \times d}$, in cui ogni riga corrisponde all'embedding di un token.

L'obiettivo è che la distanza tra vettori rifletta una relazione semantica: token con significati simili avranno vettori prossimi, mentre token distinti risulteranno distanti.

Ad esempio, una frase come:

> «*Nel mezzo del cammin di nostra vita...*»

viene trasformata in una sequenza di vettori $\mathbf{e}_i$, ciascuno dei quali appreso durante l'addestramento. Un possibile vettore di embedding per il token "Nel" potrebbe essere:

$$\mathbf{e}_{\text{“Nel”}} = \begin{bmatrix} -0.068 & 0.083 & -0.105 & \dots & 0.047 \end{bmatrix} \in \mathcal{R}^d$$

2.2.3 Positional Encoding

Poiché i Transformer non possiedono una struttura sequenziale intrinseca, è necessario fornire al modello l'informazione sulla posizione di ciascun token. Questo viene realizzato tramite un vettore di *positional encoding* $\mathbf{p}_i \in \mathcal{R}^d$, che viene sommato all'embedding del token:

$$\mathbf{z}_i = \mathbf{e}_i + \mathbf{p}_i$$

Tale somma è eseguita elemento per elemento. In questo modo, lo stesso token assumerà rappresentazioni diverse in base alla sua posizione nella sequenza.

Per esempio:

- embedding di "Oscura" $= [1, 2]$
- positional encoding per posizione $1 = [0.1, 0.2]$
- risultato $= [1.1, 2.2]$

Questo consente al modello di distinguere tra sequenze con gli stessi token ma in ordine differente.

2.3 Transformer Block

Dopo la fase di tokenizzazione, embedding e positional encoding, ogni sequenza viene elaborata da una serie di blocchi neurali chiamati *Transformer blocks*. Questi blocchi costituiscono il nucleo computazionale dell'architettura Transformer e sono progettati per trasformare le rappresentazioni intermedie dei token, arricchendole con informazioni contestuali.

Ogni Transformer block è composto da due componenti principali:

- un modulo di **Self-Attention**, che consente a ciascun token di interagire con tutti gli altri della sequenza;
- una **Feed-Forward Network** (FFN), che applica trasformazioni non lineari indipendenti su ogni token.

Ciascuna di queste due componenti è preceduta da una normalizzazione *layer-wise* e seguita da una *residual connection*(2.2), ovvero un collegamento additivo tra l'input e l'output del blocco, per facilitare la propagazione del gradiente e la stabilità numerica durante l'addestramento.

2.3.1 Attention

Il meccanismo di *attention*[6] è progettato per risolvere un problema classico nel trattamento del linguaggio naturale: la **polisemia**. Una stessa parola può assumere significati diversi a seconda del contesto in cui si trova. Ad esempio, nell'*Inferno* di Dante, il termine *ombra* può rappresentare:

- un'anima disincarnata (*ombra* come entità ultraterrena),
- una semplice proiezione visiva,
- una metafora dell'identità, del ricordo o della sofferenza.

I modelli con *embedding statici*, come Word2Vec[3], rappresentano ogni parola sempre con lo stesso vettore $\mathbf{e}_w \in \mathcal{R}^d$, ignorando completamente il contesto in cui la parola appare. Al contrario, il meccanismo di self-attention permette al modello di costruire rappresentazioni contestualizzate dinamicamente, adattando il significato di ciascun token alla sequenza che lo circonda.

2.3.2 Formulazione

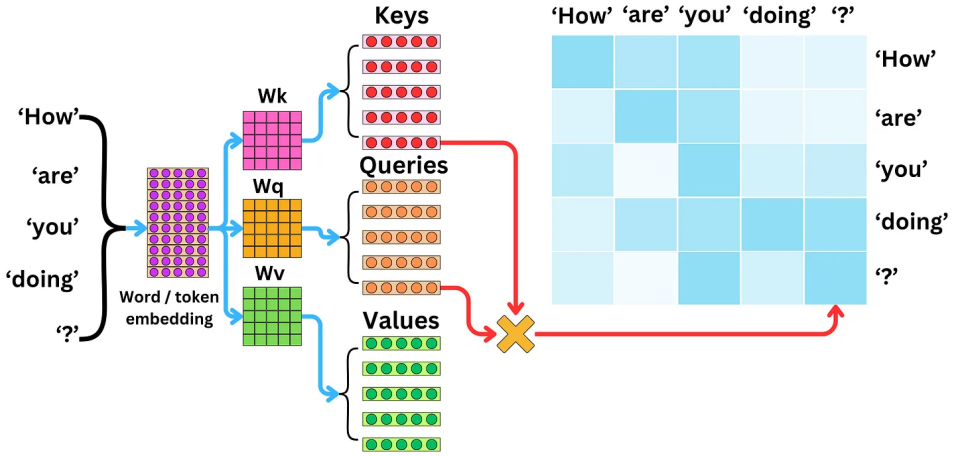


Figure 6: Attention Matrices

Dato un input $X = [\mathbf{x}_1, \ldots, \mathbf{x}_n] \in \mathcal{R}^{n \times d}$, l'attention calcola tre proiezioni lineari:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

dove $W^Q, W^K, W^V \in \mathcal{R}^{d \times d_k}$ sono matrici di peso apprese (6). L'output dell'attention è dato da:

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

Questo prodotto scalare tra query e key rappresenta una misura di affinità tra token, mentre la moltiplicazione successiva per V consente al modello di aggregare le informazioni pertinenti.

Nel *Canto III* dell'Inferno, Dante scrive:

> «Tu che 'ntrasti lasciando ogne speranza,
> **vedrai le ombre dolorose e varie**
> che l'aere nero in eterno si lamenta.»

Il termine *ombre* viene qui compreso, grazie all'attenzione, in relazione a termini come *dolorose*, *eterne*, *lamento*, suggerendo la sua interpretazione come anime dannate.

2.3.3 Masking

Durante la generazione autoregressiva, il modello predice ogni token basandosi esclusivamente sui precedenti, senza mai "sbirciare" nel futuro. Questo vincolo temporale viene imposto tramite un meccanismo noto come **masking**.

Nel dettaglio, la matrice di attenzione $QK^\top$ viene alterata mediante una maschera triangolare inferiore, che annulla tutte le interazioni tra un token e i suoi successivi. Gli elementi sopra la diagonale vengono impostati a valori molto negativi ($-\infty$), così che la funzione softmax li trasformi in zeri. In questo modo, il modello può guardare solo indietro, mai avanti.

$$\text{masked\_score}_{i,j} = \begin{cases} \frac{q_i \cdot k_j}{\sqrt{d_k}} & \text{se } j \leq i \\ -\infty & \text{se } j > i \end{cases}$$

> «*Non è licito al mortal sapere*
> *ciò che ancor non fu disvelato in terra*»

Questo equivale a impedire al poeta di conoscere versi futuri prima che siano composti. Il Transformer, come Dante nella selva oscura, avanza un passo alla volta, ispirato solo dai canti precedenti.

Esempio con una molecola SMILES Consideriamo la stringa SMILES:

`O=[N+]([O-])c1ccoc1`

Dopo la tokenizzazione e l'aggiunta dei token speciali [CLS] e [EOS], otteniamo la seguente sequenza:

$$[\texttt{[CLS]}, \texttt{O}, \texttt{=}, \texttt{[}, \texttt{N}, \texttt{+}, \texttt{]}, \texttt{(}, \ldots, \texttt{[EOS]}]$$

Il modello, durante l'addestramento o la generazione, apprende a predire ogni token basandosi solo sui precedenti. Ad esempio, quando il Transformer si trova al quarto passo (token = `[`), può accedere solo a:

$$[\texttt{[CLS]}, \texttt{O}, \texttt{=}]$$

ma non ai successivi `N`, `+`, ..., `[EOS]`. La matrice di attenzione associata è quindi mascherata come segue (esempio semplificato):

$$\texttt{Mask} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Dove ogni riga indica i token a cui ciascun token può "prestare attenzione". Il masking assicura che il modello apprenda a costruire la molecola un token alla volta, come un chimico che la disegna passo dopo passo senza conoscere la fine della formula.

Nel contesto delle molecole, questo significa che la generazione avviene in maniera coerente con la struttura chimica: ogni nuovo atomo o simbolo è predetto in base ai legami già formati, evitando configurazioni chimiche impossibili.

2.3.4 Feed-Forward Network

Dopo il modulo di attenzione, ogni token viene ulteriormente trasformato da una rete neurale feed-forward applicata in modo indipendente a ciascuna posizione. Questa è costituita da due layer lineari separati da una funzione di attivazione non lineare (tipicamente ReLU o GELU):

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

L'output della FFN viene poi sommato all'input originale tramite una connessione residua, seguita da una normalizzazione layer-wise.

2.3.5 Composizione del blocco

Ciascun Transformer block può quindi essere riassunto come:

1. LayerNorm → Self-Attention → Residual Add
2. LayerNorm → FeedForward → Residual Add

Questi blocchi sono impilati più volte (tipicamente da 6 a 96 nel caso di modelli di grandi dimensioni) per raffinare progressivamente la rappresentazione interna della sequenza. L'output finale viene quindi proiettato sul vocabolario tramite un layer lineare seguito da una funzione softmax per produrre la distribuzione del prossimo token.

2.3.6 Softmax Layer

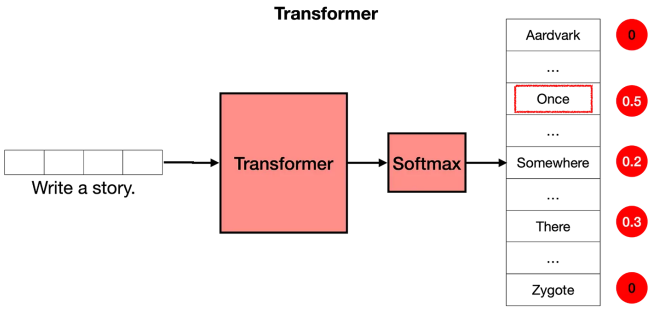


Figure 7: Softmax layer

Al termine della sequenza di *Transformer blocks*, il modello produce una rappresentazione vettoriale per ciascun token della sequenza. L'output corrispondente all'ultima posizione viene proiettato, tramite un layer lineare, nello spazio del vocabolario, producendo un vettore di *logits*, ovvero punteggi reali non normalizzati, uno per ciascun token del vocabolario.

$$\text{logits} = \mathbf{h}_n W^T + b$$

dove $\mathbf{h}_n \in \mathcal{R}^d$ è il vettore di attivazione finale, $W \in \mathcal{R}^{|\mathcal{V}| \times d}$ è la matrice di proiezione e $b \in \mathcal{R}^{|\mathcal{V}|}$ è il bias.

Per ottenere una distribuzione di probabilità $P(w \mid \text{contesto})$ sul prossimo token, si applica la funzione *softmax*:

$$P(w_i \mid \mathbf{h}_n) = \frac{\exp(\text{logit}_i)}{\sum_{j=1}^{|\mathcal{V}|} \exp(\text{logit}_j)}$$

Il token con la probabilità più alta rappresenta quello che il modello considera più plausibile come successivo nella sequenza. A partire da questa distribuzione, è possibile selezionare il *next token* tramite strategie diverse:

- **Greedy decoding**: si seleziona il token con probabilità massima.
- **Sampling**: si campiona casualmente un token secondo la distribuzione.
- **Top-k / Top-p sampling**: si limita il campionamento a un sottoinsieme più probabile del vocabolario.

Questo processo viene ripetuto iterativamente: il token generato viene concatenato ai precedenti e fornito nuovamente come input al modello, che predice il successivo, e così via. Si tratta di una procedura autoregressiva che consente al modello di generare sequenze coerenti un token alla volta.

2.4 Quantum Circuits

In questa sezione si fa un'introduzione ai concetti fondamentali della computazione quantistica necessari per comprendere l'architettura del Quantum Transformer. Vengono presentati gli

elementi costitutivi del calcolo quantistico: i qubit, le operazioni unitarie (quantum gates), i circuiti parametrizzati (PQC) e il test di Hadamard.

L'obiettivo è fornire una base concettuale solida per interpretare correttamente le trasformazioni quantistiche applicate nel meccanismo di attenzione, senza approfondire aspetti formali non direttamente rilevanti per l'implementazione analizzata. La trattazione è quindi orientata all'applicazione e strutturata in modo da riflettere il flusso computazionale adottato nel modello ibrido.

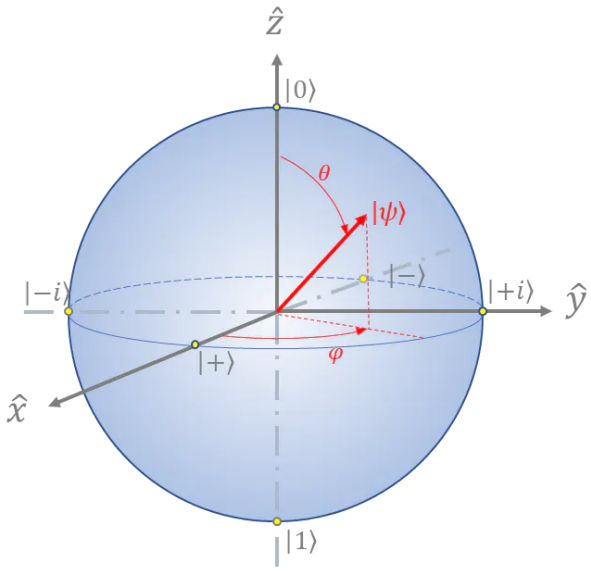


Figure 8: Bloch Sphere

2.4.1 Qubits and Quantum Information

Un *qubit* (bit quantistico) è l'unità fondamentale dell'informazione quantistica, analoga al bit classico. A differenza dei bit classici, che possono trovarsi solo nello stato 0 oppure 1, un qubit può esistere in una combinazione lineare (o *sovrapposizione*) di entrambi:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \quad \text{con } |\alpha|^2 + |\beta|^2 = 1$$

Qui, α e β sono ampiezze di probabilità complesse, e lo stato del qubit $|\psi\rangle$ è rappresentato geometricamente come un punto sulla superficie della *sfera di Bloch* (figura 8), definito da due angoli: l'angolo polare θ e l'angolo azimutale ϕ.

Questo permette ai qubit di codificare un'informazione più ricca rispetto ai bit classici.

2.4.2 Gate Quantistici

I qubit vengono manipolati attraverso trasformazioni unitarie dette *gate quantistici*.

I gate quantistici sono i blocchi fondamentali dei circuiti quantistici. Ogni gate corrisponde a un operatore unitario U che agisce su uno o più qubit, garantendo che la trasformazione

preservi la norma dello stato quantistico e sia reversibile. Questa reversibilità rappresenta una differenza cruciale rispetto a molti gate logici classici, che sono spesso irreversibili.

I gate utilizzati in questo lavoro sono:

- **Gate di Pauli (X, Y, Z)**: Questi gate a un solo qubit corrispondono alle matrici di Pauli e realizzano rotazioni di π radianti attorno agli assi corrispondenti della sfera di Bloch. Ad esempio, il gate Pauli-X agisce come l'equivalente quantistico del gate classico NOT:
$$X|0\rangle = |1\rangle, \quad X|1\rangle = |0\rangle$$

- **Gate di Hadamard (H)**: Questo gate crea stati in sovrapposizione trasformando gli stati della base computazionale come segue:
$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$
È essenziale negli algoritmi che richiedono ampiezze di probabilità uguali, come nelle fasi di inizializzazione di molti algoritmi quantistici.

- **Gate di Rotazione ($R_x(\theta)$, $R_y(\theta)$, $R_z(\theta)$)**: Questi gate realizzano rotazioni di un angolo θ attorno agli assi corrispondenti della sfera di Bloch. Sono definiti come:
$$R_x(\theta) = e^{-i\theta X/2}, \quad R_y(\theta) = e^{-i\theta Y/2}, \quad R_z(\theta) = e^{-i\theta Z/2}$$
Questi gate parametrizzati sono fondamentali nei PQC, permettendo trasformazioni adattabili durante il processo di training.

- **Gate CNOT (Controlled-NOT)**: Un gate a due qubit che inverte lo stato del qubit bersaglio se il qubit di controllo si trova nello stato $|1\rangle$. La sua azione è definita da:
$$\text{CNOT}|a\rangle|b\rangle = |a\rangle|a \oplus b\rangle$$
dove $\oplus$ indica la somma modulo 2. Il gate CNOT è cruciale per generare entanglement tra qubit.

Questi gate vengono combinati per costruire circuiti quantistici.

2.4.3 Circuiti Quantistici Parametrizzati (PQC)

I *Parameterized Quantum Circuits* (PQC) sono circuiti quantistici composti da gate il cui funzionamento dipende da parametri continui e ottimizzabili — tipicamente angoli di rotazione. Questi parametri vengono ottimizzati durante il training e consentono ai circuiti di codificare dati classici in stati quantistici.

In questo lavoro, i PQC vengono utilizzati per trasformare le rappresentazioni classiche dell'identità del token, della posizione e delle proprietà chimico-fisiche in embedding quantistici. I circuiti seguono un'architettura a *ansatz* fissa composta da uno strato di rotazioni $R_y(\theta)$ su ciascun qubit, seguito da uno strato di gate CNOT entangling.

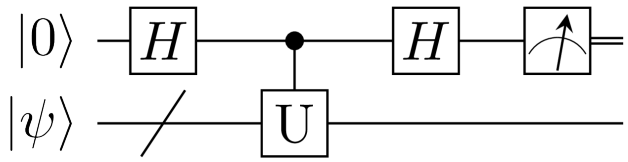
I parametri allenabili (θ_{ei}, θ_{pi}, θ_q, ecc.) sono ottimizzati tramite l'algoritmo SPSA (Simultaneous Perturbation Stochastic Approximation), poiché i metodi basati su gradienti classici come il backpropagation non sono applicabili ai sistemi quantistici.

I PQC nel Transformer ibrido vengono utilizzati nei seguenti modi:

- U_e, U_p: codificano gli embedding dei token e della posizione come stati quantistici.
- U_q, U_k: generano gli stati query e key a partire dagli embedding combinati.

Questi circuiti formano l'interfaccia computazionale tra le rappresentazioni di input classiche e il meccanismo di self-attention quantistico descritto nella sezione successiva.

2.4.4 Hadamard Test



L'*Hadamard test* è una subroutine quantistica utilizzata per stimare la parte reale del prodotto interno tra due stati quantistici. In questa architettura, viene impiegata per calcolare il coefficiente scalare di attenzione tra gli stati query e key $|q_i\rangle$ e $|k_j\rangle$.

La procedura utilizza un qubit ancilla inizializzato in $|0\rangle$, un gate di Hadamard, un'operazione unitaria controllata, un ulteriore gate di Hadamard, e infine una misura. La probabilità di misurare $|0\rangle$ è direttamente correlata a $\Re(\langle q_i|k_j\rangle)$.

Questo prodotto interno sostituisce il prodotto scalare classico nel meccanismo di attenzione e costituisce la base per la costruzione della matrice di attenzione quantistica.

3 Quantum Transformer

Analisi dell'architettura del tranformer quantistico

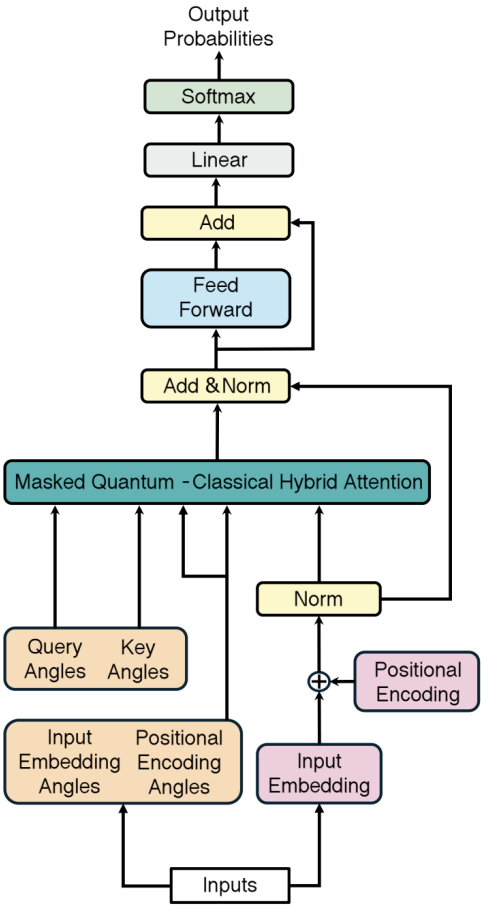


Figure 9: Architettura del transformer decoder ibrido.

Dopo aver introdotto l'architettura classica del Transformer e i principi fondamentali della computazione quantistica, in questa sezione analizziamo il modello ibrido denominato *Quantum Transformer*, proposto da Smaldone et. al.[5]. In tale architettura, alcune componenti classiche e in particolare il meccanismo di *self-attention* vengono reinterpretate e implementate tramite circuiti quantistici.

L'obiettivo è sfruttare la capacità rappresentativa degli stati quantistici per migliorare la modellazione di sequenze molecolari e la generalizzazione del modello. Le restanti componenti dell'architettura, come i layer `feed-forward`, il `value path` e il decoder autoregressivo, restano invariati rispetto al framework classico.

Nelle sezioni successive (4, 5, 6) verranno approfonditi gli aspetti implementativi principali.

Analizzando la Figura 9, si nota come la struttura generale ricalchi quella di un Transformer standard, con l'unica differenza sostanziale nel calcolo dell'attenzione. In particolare, il prodotto

scalare tra vettori *query* e *key* viene sostituito da un test di Hadamard modificato, eseguito su stati quantistici codificati tramite circuiti parametrizzati.

Il circuito quantistico utilizzato per il calcolo del singolo *attention score* è illustrato nella Figura 10.

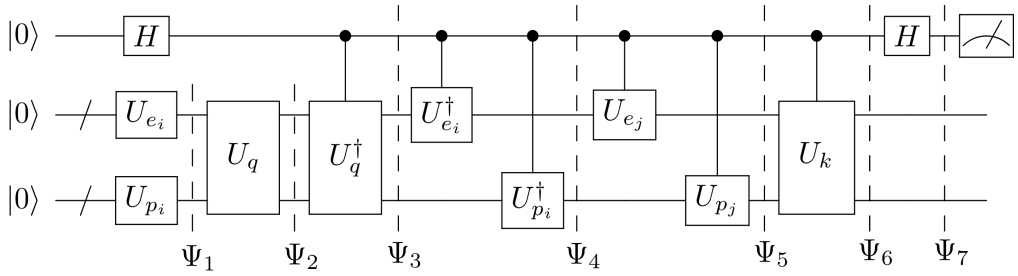


Figure 10: Circuito quantistico per il calcolo dello *attention score* tra due token quantistici. Le unità unitarie U_{e_i}, U_{e_j} e U_{p_i}, U_{p_j} codificano rispettivamente le informazioni di token e posizione dei token i-esimo e j-esimo in stati quantistici. Successivamente, U_q e U_k vengono applicate per costruire le rappresentazioni di query e key a partire dagli embedding combinati token-posizione. Gli stati quantistici risultanti, da Ψ_1 a Ψ_7, descrivono il flusso delle trasformazioni. La parte reale del prodotto interno $\mathrm{Re}\langle q_i|k_j\rangle$ è ottenuta mediante il valore atteso misurato sul qubit ancilla.

3.1 Input processing

Come descritto nella Sezione 2.2.1, l'input al modello è una molecola rappresentata in formato SMILES e tokenizzata in una sequenza $\{x_1, \ldots, x_n\}$. Ogni token x_i viene convertito in un embedding numerico $\mathbf{e}_i$ tramite una matrice di embedding appresa durante l'addestramento.

3.1.1 Tokenization

Si prendono le molecole dal csv che le contiene in formato SMILE. Dopo aver rimosso i duplicati, vengono riscritte in modo canonicalizzato usando la libreria *rdkit*. A questo punto si sfrutta un'espressione regolare per suddividere le stringhe SMILES in unità atomiche, legami e strutture funzionali:

```
self.smiles_regex = re.compile(
    "(\[[^\]]+]|Br?|Cl?|N|O|S|P|F|I|b|c|n|o|s|p|\(|\)
       |\.|=|#|-|\+|\\\\|\/|:|~|@|\?|>|\*|\$|\%[0-9]{2}|[0-9]|<pad>|[CLS
       ]|[EOS])"
)
```

Si definiscono le funzioni per mappare i token in numeri interi `stoi` e per il mapping inverso `itos`.

```
self.stoi = {ch: i for i, ch in enumerate(self.vocab)}
self.itos = {i: ch for i, ch in enumerate(self.vocab)}
```

Si creano le sequenze input_sequence e target_sequence necessarie per la predizione del next token da parte del modello.

```
input_sequence = [self.stoi[s] for s in self.smiles_regex.findall(smiles)[:-1]]
target_sequence = [self.stoi[s] for s in self.smiles_regex.findall(smiles)[1:]]
```

La lunghezza delle sequenze viene standardizzata mediante padding con il token speciale `<pad>`.

3.2 Embedding

Per preservare l'informazione posizionale, a ciascun embedding di token $\mathbf{e}_i$ viene sommato un positional embedding $\mathbf{p}_i$. Il risultato è un vettore $\mathbf{z}_i$ che rappresenta la combinazione delle due componenti:

$$\mathbf{z}_i = \mathbf{e}_i + \mathbf{p}_i$$

Nel paper originale, gli autori introducono anche un insieme di *embedding addizionali* $\mathbf{c}_{i,v}$, che incorporano proprietà chimico-fisiche della molecola per condizionare la generazione. Tuttavia, in questa implementazione, tale condizionamento è stato omesso per semplificare l'architettura [4].

Tutti i vettori hanno dimensione 64 e vengono sommati elemento per elemento. L'intera sequenza di input è così rappresentata dalla matrice $Z \in \mathcal{R}^{n \times d}$:

$$Z = \begin{bmatrix} \mathbf{z}_1^\top \\ \mathbf{z}_2^\top \\ \vdots \\ \mathbf{z}_n^\top \end{bmatrix}$$

A partire da Z, si ottengono le tre matrici fondamentali per il calcolo dell'attenzione classica attraverso proiezioni lineari:

- $Q = ZW^Q$, Matrice delle query
- $K = ZW^K$ Matriche delle chiavi
- $V = ZW^V$ Matrice dei valori

dove W^Q, W^K, e W^V sono matrici di peso apprese. Nel modello quantistico, le matrici Q e K vengono invece sostituite da stati quantistici calcolati tramite circuiti PQC, come descritto nella sezione successiva.

3.3 Quantum Attention

Il calcolo dell'attenzione quantistica si basa su un approccio ibrido in cui gli stati *query* e *key* sono rappresentati da stati quantistici, mentre la matrice dei valori V e le operazioni successive rimangono classiche. A differenza del transformer tradizionale, dove il prodotto scalare $\mathbf{q}_i \cdot \mathbf{k}_j$

[4] Nel caso completo, ogni vettore di input può essere rappresentato come:

$$\mathbf{z}_i = \mathbf{e}_i + \mathbf{p}_i + \sum_{v=1}^{\kappa} \mathbf{c}_{i,v}$$

viene calcolato direttamente tra vettori reali, qui si ottiene la parte reale del prodotto interno quantistico $\langle q_i|k_j\rangle$ attraverso un Hadamard test modificato.

L'intero processo è suddiviso in sette stati quantistici, ciascuno corrispondente a una trasformazione intermedia dello stato quantistico del sistema.

1. **Codifica quantistica dei token e delle posizioni**:

   (a) Ogni token x_i viene mappato in un vettore θ_{e_i}, i cui elementi sono scalati nell'intervallo $[0, \pi]$. Questi valori rappresentano i parametri di rotazione di un circuito parametrico $U_e(\theta_{e_i})$, il quale applicato allo stato iniziale $|0\rangle^{\otimes \log d}$ produce lo stato quantistico $|e_i\rangle$.

   (b) Allo stesso modo, le informazioni posizionali vengono codificate attraverso un circuito $U_p(\theta_{p_i})$, che genera lo stato $|p_i\rangle$.

   (c) I due stati vengono combinati in uno stato complessivo tramite prodotto tensoriale:

$$\Psi_1 = |z_i\rangle = |e_i\rangle \otimes |p_i\rangle$$

2. **Apprendimento dello stato *query***:

   Lo stato $|z_i\rangle$ viene trasformato in uno stato $|q_i\rangle$ mediante un PQC parametrico U_q:

$$\Psi_2 = U_q\,|z_i\rangle = |q_i\rangle$$

3. **Preparazione del test di Hadamard – Parte I**:

   Un qubit ancilla[5] viene inizializzato nello stato $|+\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$. Viene quindi applicato un circuito controllato $CU_q^\dagger$ che esegue l'inverso di U_q solo se l'ancilla è in stato $|1\rangle$:

$$\Psi_3 = CU_q^\dagger\left(|+\rangle \otimes |q_i\rangle\right) = \frac{1}{\sqrt{2}}\left(|0\rangle\,|q_i\rangle + |1\rangle\,|z_i\rangle\right)$$

4. **Cancellazione della codifica dello stato $|z_i\rangle$**:

   Per garantire che il ramo con ancilla $|1\rangle$ contenga uno stato "pulito", vengono applicati $CU_e^\dagger$ e $CU_p^\dagger$ per annullare le codifiche di token e posizione, riportando il registro a $|0\rangle^{\otimes \log d}$:

$$\Psi_4 = CU_p^\dagger CU_e^\dagger\left(\Psi_3\right) = \frac{1}{\sqrt{2}}\left(|0\rangle\,|q_i\rangle + |1\rangle\,|0\rangle^{\otimes \log d}\right)$$

5. **Preparazione controllata dello stato $|z_j\rangle$**:

   Viene ricostruito lo stato $|z_j\rangle$ nel ramo in cui l'ancilla è $|1\rangle$ applicando le versioni controllate dei circuiti U_{e_j} e U_{p_j}:

$$\Psi_5 = CU_{e_j} CU_{p_j}\left(\Psi_4\right) = \frac{1}{\sqrt{2}}\left(|0\rangle\,|q_i\rangle + |1\rangle\,|z_j\rangle\right)$$

[5] Un *qubit ancilla* è un qubit ausiliario utilizzato per eseguire operazioni controllate o per mediare misurazioni senza alterare direttamente lo stato del sistema principale. Viene inizializzato in uno stato noto (spesso $|0\rangle$ o $|+\rangle$) e interagisce con altri registri quantistici secondo lo schema del circuito.

6. **Generazione dello stato *key***:

   Lo stato $|z_j\rangle$ viene trasformato nello stato $|k_j\rangle$ tramite il circuito controllato CU_k:

$$\Psi_6 = CU_k\left(\Psi_5\right) = \frac{1}{\sqrt{2}}\left(|0\rangle\,|q_i\rangle + |1\rangle\,|k_j\rangle\right)$$

7. **Test di Hadamard e misura dell'ancilla**:

   Si applica una porta Hadamard al qubit ancilla, ottenendo:

$$\Psi_7 = (H \otimes I)\Psi_6 = \frac{1}{2}\left(|0\rangle\left(|q_i\rangle + |k_j\rangle\right) + |1\rangle\left(|q_i\rangle - |k_j\rangle\right)\right)$$

   Infine, si esegue una misura dell'osservabile Pauli-Z sull'ancilla. Il valore atteso risultante è:

$$\langle Z\rangle = \langle\Psi_7|\, Z \otimes I\,|\Psi_7\rangle = \mathrm{Re}\langle q_i|k_j\rangle$$

   Questo valore rappresenta la parte reale del prodotto interno tra lo stato *query* $|q_i\rangle$ e lo stato *key* $|k_j\rangle$, ovvero l'equivalente quantistico dello *attention score* classico.

3.4 Generazione delle probabilità in output

Una volta calcolati gli *attention score* tramite i circuiti quantistici, questi vengono assemblati nella matrice di attenzione $A \in \mathbb{R}^{n\times n}$, dove ogni elemento $a_{i,j}$ rappresenta $\mathrm{Re}\langle q_i|k_j\rangle$, ovvero la parte reale del prodotto interno tra gli stati quantistici *query* e *key*.

Per garantire coerenza con il meccanismo di attenzione classico, i punteggi ottenuti vengono scalati per mantenere una varianza unitaria:

$$a_{i,j}^{\text{scaled}} = \sqrt{d_k} \cdot \mathrm{Re}\langle q_i|k_j\rangle$$

A ciascuna riga della matrice viene poi applicata la funzione softmax per normalizzare i pesi di attenzione:

$$\alpha_{i,j} = \frac{\exp(a_{i,j}^{\text{scaled}})}{\sum_k \exp(a_{i,k}^{\text{scaled}})}$$

Questa matrice normalizzata viene infine moltiplicata per la matrice dei *value* V, che è ottenuta in modo classico da:

$$V = ZW^V$$

Il risultato è una nuova rappresentazione della sequenza, in cui ciascun token incorpora informazioni pesate in base alla sua rilevanza rispetto agli altri:

$$\text{Attention}(Q, K, V) = \alpha \cdot V$$

Questa rappresentazione viene successivamente elaborata dai moduli di normalizzazione, feed-forward e proiezione finale per ottenere le distribuzioni di probabilità sui token successivi. La generazione avviene quindi in modo autoregressivo, token per token, come in un Transformer standard.

Nota sull'estensione ad embedding addizionali. Nel lavoro originale, gli autori propongono un'estensione in cui, oltre ai token e alla posizione, anche le proprietà chimico-fisiche delle molecole vengono codificate quantisticamente. Questo richiede la suddivisione del registro quantistico in sottospazi di pari dimensione, uno per ciascun tipo di embedding (token, posizione, proprietà). Nel nostro caso, tale estensione non è stata implementata per semplicità, ma può essere integrata seguendo lo schema riportato nelle equazioni 15 e 16 di Smaldone et. al.[5].

---

4 Attention Mechanism Implementation

Dalla preparazione dell'input al calcolo della matrice di attenzione.

In questa sezione analizziamo in dettaglio l'implementazione del meccanismo di attenzione quantistico, dalla preparazione dei parametri, alla definizione dei circuiti quantistici. Vedremo come vengono utilizzati i gates R_y e CNOT, e come sia possibile sfruttare la presenza di più QPU per svolgere una computazione distribuita.

4.0.1 Preparazione dei parametri classici e quantistici

Durante il training e durante l'inferenza, il modello riceve dei tensori di interi, ovvero le molecole in formato SMILE tokenizzate. Questi indici (`idx`) vengono convertiti in embedding, assieme al calcolo degli embedding posizionali, come nei transformer classici.

```
B, T = idx.size() # B = Batch size, T = Sequence length
x_token = self.token_embed(idx)
x_pos = self.position_embed[:, :T, :]  # [1, T, 64]
```

Successivamente si recuperano i parametri associati a ciascun token e a ciascuna posizione. Non sono vettori nello stesso spazio degli embedding classici, ma set di angoli di rotazione che verranno utilizzati dai gate R_y nei circuiti quantistici, motivo per cui vengono scalati da 0 a π.

```
self.token_embed_quantum_parameters = nn.Embedding(
    vocab_size, num_params_per_register
)

# ... scale ...

self.position_embed_quantum_parameters = nn.Parameter(
    torch.zeros(1, block_size, num_params_per_register)
)
```

```
token_angles = self.token_embed_quantum_parameters(idx)
position_embedding_angles = self.position_embed_quantum_parameters[:, :T, :
    ].expand(B, -1, -1)
angles = torch.stack([token_angles, position_embedding_angles], dim=0)
```

Il layer AttentionQuantumLayer ha dei parametri apprendibili chiamati *query_angles* e *key_angles* che rappresentano rispettivamente la parte Query e Key nel dominio quantistico. Vengono espansi per ottenere un set di angoli Q e K per ogni token nella sequenza e nel batch. In questo modo ogni token ha il proprio set di angoli Q e K apprendibili. Si calcola anche la V in modo classico applicando un layer lineare agli embedding classici x.

```
# Dentro AttentionQuantumLayer.__init__(...)
total_VQC_params = ansatz_layers * num_qubits # parametri necessari per l'
    intero circuito
self.query_angles = nn.Parameter(torch.zeros(1, 1, total_VQC_params))
self.key_angles = nn.Parameter(torch.zeros(1, 1, total_VQC_params))
```

```
# --- Dentro AttentionQuantumLayer.forward(self, x: Tensor, angles: Tensor,
    ...) ---
broadcasted_query_angles = self.query_angles.expand(B, T, -1)
broadcasted_key_angles = self.key_angles.expand(B, T, -1)

v = self.value(x)
```

4.0.2 Combinazione dei parametri per il circuito quantistico

L'idea è creare un unico grande tensore (*circuit_parameters*) contenente, per ogni possibile coppia di posizioni *(i,j)* nella sequenza, tutti gli angoli necessari per calcolare l'attenzione tra l'elemento i (query) e l'elemento j (key). Questo tensore verrà poi passato alla funzione *AttentionQuantumFunction.apply* per eseguire i circuiti quantistici, come vedremo più avanti in dettaglio.

```
# tok_angles: Angoli token (B, T, P_tok)
# pos_angles: Angoli posizione (B, T, P_pos)
# broadcasted_query_angles: Angoli Q apprendibili espansi (B, T, P_q)
# broadcasted_key_angles: Angoli K apprendibili espansi (B, T, P_k)

circuit_parameters = prepare_attention_inputs(
    tok_angles,
    pos_angles,
    broadcasted_query_angles, # 'Query' per elemento i
    broadcasted_key_angles    # 'Key' per elemento j
)
```

4.0.3 Rimozione dei circuiti ridondanti e Masking

Quello che vogliamo fare è calcolare l'attenzione tra ogni possibile coppia di token (i, j), questo però può essere computazionalmente costoso. Visto che alcune coppie potrebbero condividere gli stessi parametri, l'idea è di eseguire i circuiti quantistici per le sole coppie uniche, evitando operazioni ridondanti. Successivamente, come ogni altro modello auto-regressivo, dobbiamo fare in modo che ogni token possa prestare attenzione solo ai token precedenti e a se stesso, senza poter "guardare" al futuro. A tal proposito applichiamo una maschera in modo che si considerino solo le coppie (i, j) in cui $i \leq j$, attribuendo valore $-\infty$ a quelle sopra la diagonale.

```
# Maschera triangolare
initial_mask = torch.tril(
    torch.ones(n, n, dtype=torch.bool, device=mask_device)
)

parameter_tensor_masked = full_batch_parameter_tensor.clone()
inverted_expanded_mask = ~mask.expand(batch_size, n, n, groups, param_count
    )

```

```
# Applico la maschera
parameter_tensor_masked[inverted_expanded_mask] = float("-inf")

# Appiattisco il tensore per elaborare ogni coppia (i, j) individualmente
target_shape = (batch_size * n * n, groups, param_count)
flattened_tensor = parameter_tensor_masked.view(target_shape)

# Identifico gli indici del triangolo inferiore (non -inf) e superiore (-inf)
is_inf_tensor = torch.isinf(flattened_tensor)

# Una entry e' considerata "superiore" se tutti i suoi parametri sono -inf
is_upper_triangle = is_inf_tensor.all(dim=(1, 2))

# Indici flatten corrispondenti agli elementi da calcolare (lower) e da ignorare (upper)
lower_triangle_indices = torch.nonzero(~is_upper_triangle, as_tuple=True)[0]
upper_triangle_indices = torch.nonzero(is_upper_triangle, as_tuple=True)[0]

# Seleziono solo i parametri validi (i<=j)
lower_triangle_parameters = flattened_tensor[lower_triangle_indices]

# Trovo i set di parametri unici tra quelli validi
unique_lower_triangle_params, unique_index_mapping = torch.unique(
    lower_triangle_parameters, return_inverse=True, dim=0
)

# unique_index_mapping mappa ogni indice presente in lower_triangle_indices all'indice corrispondente in unique_lower_triangle_params

return (
    unique_lower_triangle_params, # I soli parametri da passare al circuito
    unique_index_mapping,         # Come mappare i risultati indietro
    lower_triangle_indices,       # Indici originali validi
    upper_triangle_indices,       # Indici originali invalidi/mascherati
)
```

4.0.4 Esecuzione del circuito quantistico

Abbiamo preparato i seguenti parametri:

- token_i: parametri quantistici (3 angoli) derivati dall'embedding del token i-esimo della sequenza.
- pos_i: parametri quantistici (3 angoli) derivati dall'embedding della posizione del token i-esimo della sequenza.
- query_combined: costituito concatenando i parametri apprendibili query_token_register e query_pos_register. Sono parametri globali che vengono appresi durante il training e rappresentano la trasformazione quantistica relativa all'operazione "Query".
- token_j: parametri quantistici (3 angoli) derivati dall'embedding del token j-esimo della sequenza.
- pos_j: parametri quantistici (3 angoli) derivati dall'embedding della posizione del token

j-esimo della sequenza.

- key_combined: costituito concatenando i parametri apprendibili key_token_register e key_pos_register. Sono parametri globali che vengono appresi durante il training e rappresentano la trasformazione quantistica relativa all'operazione "Key".

Siamo pronti per eseguire una serie di operazioni quantistiche con l'obiettivo finale di calcolare il prodotto $Q^T K$ richiesto dal meccanismo di attenzione. Come descritto nel paper, questo viene fatto implementando un Hadamard Test modificato. In generale, il test di Hadamard è un algoritmo quantistico che viene utilizzato per creare una variabile casuale il cui valore atteso è la parte reale $Re\langle\psi|U|\psi\rangle$, dove $|\psi\rangle$ è uno stato quantistico e U è un operatore unitario.

Vediamo ora come costruire un test di Hadamard (illustrato nella figura 2.4.4) che ci permetta di ottenere come risultato proprio il prodotto interno $\langle q_i | k_i \rangle$, richiesto per il calcolo dell'attenzione.

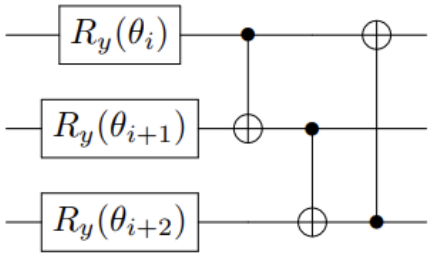
```
@cudaq.kernel

ancilla = cudaq.qubit()
register = cudaq.qvector(num_working_qubits[0])

# Circuito quantistico
h(ancilla)
unitary(register, token\_i, subsystem_size, 0, layers)
unitary(register, position_i, subsystem_size, 1, layers)
unitary(register, query, subsystem_size, -1, layers)
controlled\_adjoint\_unitary(ancilla, register, query, subsystem_size, -1,
    layers)
controlled\_adjoint\_unitary(ancilla, register, position_i, subsystem_size,
     1, layers)
controlled\_adjoint\_unitary(ancilla, register, token\_i, subsystem_size, 0
    , layers)
controlled_unitary(ancilla, register, token_j, subsystem_size, 0, layers)
controlled_unitary(ancilla, register, position_j, subsystem_size, 1, layers
    )
controlled_unitary(ancilla, register, key, subsystem_size, -1, layers)
h(ancilla)
```

- ancilla: un singolo qubit ausiliario necessario per il test di Hadamard. La sua misura finale fornirà il risultato.
- register: un registro di 6 qubit dove vengono codificati gli stati quantistici di Query e Key, per cui vogliamo calcolare il prodotto interno.
- h(ancilla): applica l'Hadamard gate all'ancilla mettendo il qubit, inizialmente nello stato $|0\rangle$, in una sovrapposizione $\frac{(|0\rangle+|1\rangle)}{\sqrt{2}}$.
- – unitary(register, token_i, subsystem_size, 0, layers):

```
start = 0
end = 3

for layer in range(layers):
    for i in range(start, end):
        idx_in_theta = i - start + layer * (end - start)
        angle = theta[idx_in_theta]
        ry(angle, qubits[i])
```

```
9
10 for i in range(start, end):
11     if i < end - 1:
12         x.ctrl(qubits[i], qubits[i + 1])
13     else:
14         x.ctrl(qubits[i], qubits[start])
```

Si applica l'ansatz layer parametrizzato da token_i alla prima metà del registro. In particolare, si utilizzano 3 gate R_y per ruotare i qubit del registro e si applicano tre CNOT in modo circolare per mettere in entanglement il primo qubit con il secondo, il secondo con il terzo, e il terzo con il primo, come si vede in figura.



– unitary(register, position_i, subsystem_size, 1, layers):

```
1  start = 3
2  end = 6
3
4  for layer in range(layers):
5      for i in range(start, end):
6          idx_in_theta = i - start + layer * (end - start)
7          angle = theta[idx_in_theta]
8          ry(angle, qubits[i])
9
10 for i in range(start, end):
11     if i < end - 1:
12         x.ctrl(qubits[i], qubits[i + 1])
13     else:
14         x.ctrl(qubits[i], qubits[start])
```

Si applica l'ansatz layer parametrizzato da position_i alla seconda meta del registro.

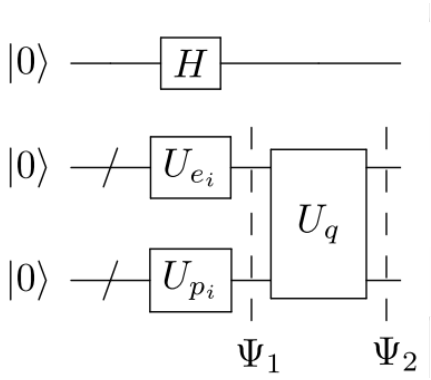
– unitary(register, query, subsystem_size, -1, layers):

```
1  start = 0
2  end = 6
3
4  for layer in range(layers):
5      for i in range(start, end):
6          idx_in_theta = i - start + layer * (end - start)
7          angle = theta[idx_in_theta]
8          ry(angle, qubits[i])
9
10 for i in range(start, end):
11     if i < end - 1:
12         x.ctrl(qubits[i], qubits[i + 1])
13     else:
```

```
14          x.ctrl(qubits[i], qubits[start])
```

Si applica l'ansatz layer parametrizzato da query a tutto il registro.

Queste tre operazioni preparano lo stato quantistico Ψ_2 nel registro.



- Parte centrale dell'Hadamard test:
  - controlled_adjoint_unitary(ancilla, register, query, subsystem_size, -1, layers):

```
start = 0
end = 6

for layer in range(layers - 1, -1, -1):
    if (end - start) > 1:
        x.ctrl(control, qubits[end - 1], qubits[start])
        for i in range(end - 2, start - 1, -1):
            x.ctrl(control, qubits[i], qubits[i + 1])

for i in range(end - 1, start - 1, -1):
    angle_index = i - start + layer * (end - start)
    angle = -theta[angle_index]
    ry.ctrl(angle, control, qubits[i])
```

Se l'ancilla qubit è nello stato $|1\rangle$ si applica l'inverso dell'operazione unitary relativa a query. Si applicano i CNOT in ordine inverso e si utilizzano i gate R_y con gli angoli di segno opposto annullando di fatto l'operazione di unitary.

  - controlled_adjoint_unitary(ancilla, register, position_i, subsystem_size, 1, layers):

```
start = 3
end = 6

for layer in range(layers - 1, -1, -1):
    if (end - start) > 1:
        x.ctrl(control, qubits[end - 1], qubits[start])
        for i in range(end - 2, start - 1, -1):
            x.ctrl(control, qubits[i], qubits[i + 1])

for i in range(end - 1, start - 1, -1):
    angle_index = i - start + layer * (end - start)
    angle = -theta[angle_index]
    ry.ctrl(angle, control, qubits[i])
```

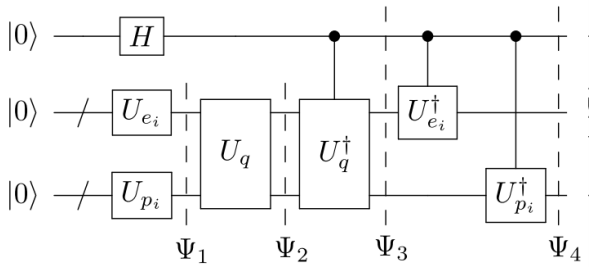
Se l'ancilla qubit è nello stato $|1\rangle$ si applica l'inverso dell'operazione unitary relativa a position_i, di fatto annullandola.

– controlled_adjoint_unitary(ancilla, register, token_i, subsystem_size, 0, layers):

```
start = 0
end = 3

for layer in range(layers - 1, -1, -1):
    if (end - start) > 1:
        x.ctrl(control, qubits[end - 1], qubits[start])
        for i in range(end - 2, start - 1, -1):
            x.ctrl(control, qubits[i], qubits[i + 1])

for i in range(end - 1, start - 1, -1):
    angle_index = i - start + layer * (end - start)
    angle = -theta[angle_index]
    ry.ctrl(angle, control, qubits[i])
```

Se l'ancilla qubit è nello stato $|1\rangle$ si applica l'inverso dell'operazione unitary relativa a token_i, di fatto annullandola.

Queste tre operazioni preparano lo stato quantistico Ψ_4 nel registro.



– controlled_unitary(ancilla, register, token_j, subsystem_size, 0, layers):

```
start = 0
end = 3

for layer in range(layers - 1, -1, -1):
    if (end - start) > 1:
        x.ctrl(control, qubits[end - 1], qubits[start])
        for i in range(end - 2, start - 1, -1):
            x.ctrl(control, qubits[i], qubits[i + 1])

for i in range(end - 1, start - 1, -1):
    angle_index = i - start + layer * (end - start)
    angle = -theta[angle_index]
    ry.ctrl(angle, control, qubits[i])
```

Se l'ancilla qubit è nello stato $|1\rangle$ si applica l'operazione unitary relativa a token_i.

– controlled_unitary(ancilla, register, position_j, subsystem_size, 1, layers):

```
start = 3
end = 6

```

```
for layer in range(layers - 1, -1, -1):
    if (end - start) > 1:
        x.ctrl(control, qubits[end - 1], qubits[start])
        for i in range(end - 2, start - 1, -1):
            x.ctrl(control, qubits[i], qubits[i + 1])

for i in range(end - 1, start - 1, -1):
    angle_index = i - start + layer * (end - start)
    angle = -theta[angle_index]
    ry.ctrl(angle, control, qubits[i])
```

Se l'ancilla qubit è nello stato $|1\rangle$ si applica l'operazione unitary relativa a position_i.

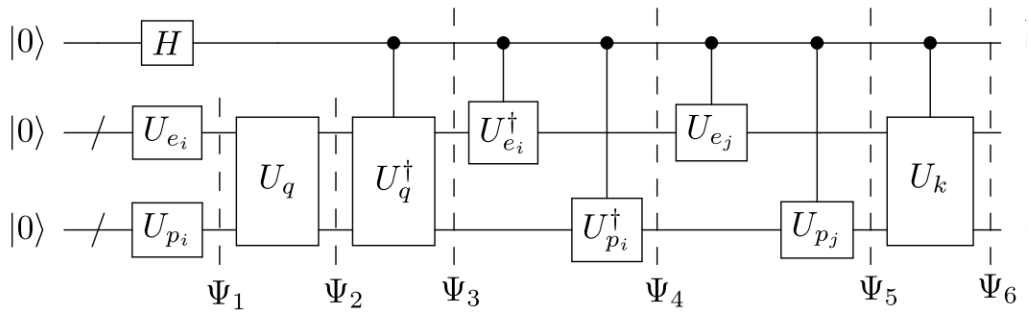
– controlled_unitary(ancilla, register, key, subsystem_size, -1, layers):

```
start = 0
end = 6

for layer in range(layers - 1, -1, -1):
    if (end - start) > 1:
        x.ctrl(control, qubits[end - 1], qubits[start])
        for i in range(end - 2, start - 1, -1):
            x.ctrl(control, qubits[i], qubits[i + 1])

for i in range(end - 1, start - 1, -1):
    angle_index = i - start + layer * (end - start)
    angle = -theta[angle_index]
    ry.ctrl(angle, control, qubits[i])
```

Se l'ancilla qubit è nello stato $|1\rangle$ si applica l'operazione unitary relativa a key.

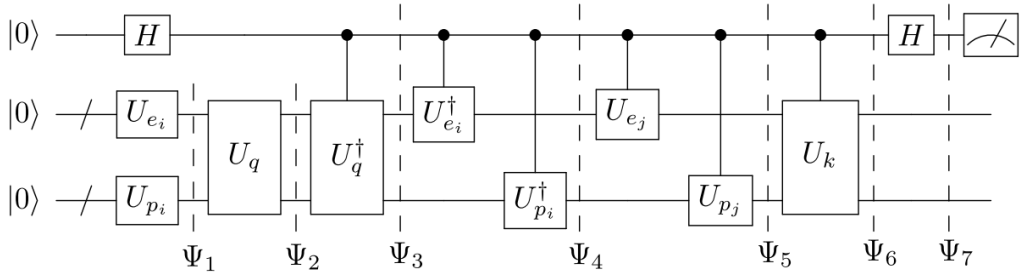
Queste tre operazioni preparano lo stato quantistico:

$$\Psi_6 = \frac{|0\rangle |q_i\rangle + |1\rangle |k_j\rangle}{\sqrt{2}}$$



- h(ancilla): si applica un secondo gate di Hadamard all'ancilla ottenendo lo stato

$$\Psi_7 = \frac{\Big(|0\rangle (|q_i\rangle + |k_j\rangle) + |1\rangle (|q_i\rangle - |k_j\rangle)\Big)}{\sqrt{2}}$$



4.0.5 Misura del valore atteso

Per misurare il valore atteso della variabile casuale definita grazie al test di Hadamard, ottenendo cosi $Re(|q_i\rangle\,|k_i\rangle)$, si utilizza il Pauli-Z gate (hamiltonian) applicato in questo modo:

```
self.hamiltonian = spin.z(0)

asyncresults = []
for i in range(self.qpu_count):
    for j in range(token_1[i].shape[0]):
        asyncresults.append(
            cudaq.observe_async(
                self.build_circuit,
                self.hamiltonian,
                token_1[i][j, :],
                position_1[i][j, :],
                query_angles[i][j, :],
                token_2[i][j, :],
                position_2[i][j, :],
                key_angles[i][j, :],
                ansatz_layers[i][j],
                num_working_qubits[i][j, :],
                qpu_id=i,
            )
        )

expectations = torch.tensor(
    [r.get().expectation() for r in asyncresults], device=device
)
```

I parametri necessari per eseguire i circuiti vengono suddivisi tra le QPU disponibili in modo da parallelizzare le esecuzioni.

```
query_angles = np.array_split(
    np.concatenate((query_token_register, query_pos_register), axis=1),
    self.qpu_count,
)
key_angles = np.array_split(
    np.concatenate((key_token_register, key_pos_register), axis=1),
    self.qpu_count,
)
```

Si itera su ogni QPU e su ogni set di parametri assegnato a quella QPU chiamando l'operazione `cudaq.observe_async(...)`. Questa misura il valore atteso dell'Hermitian spin operator, che nel nostro caso è `hamiltonian = spin.z(0)` (ovvero l'operatore Pauli-Z applicato al qubit

ancilla), alla fine dell'esecuzione del circuito, restituendo il prodotto interno tra gli stati di query e chiavi. Ogni cudaq.observe_async, restituisce un handle che viene aggiunto alla lista `asyncresults`. Per ogni handle si chiama il metodo `.get().expectation()` per ottenere le expectations di ogni circuito. Convertiamo questa lista di numeri in tensore Pytorch.

4.0.6 Matrice di attenzione

Ora che abbiamo i valori attesi calcolati usando i parametri unici, si ricostruisce il tensore di dimensione `batch_size * seq_len * seq_len` servendosi di unique_index_mapping definita in precedenza. Questo rappresenta $Q^T K$. Si applica un fattore di scala, si applica la softmax e si moltiplica per la matrice dei valori ottenendo la matrice di attenzione. Per ottenere l'output del layer di attenzione quantistica si utilizza un layer lineare chiamato `projection`.

```
attn_weight = (
    AttentionQuantumFunction.apply(
        circuit_parameters,
        self.shift,
        self.quantum_circuit,
        B,
        T,
        self.quantum_gradient_method,
        self.epsilon,
    )
    * scale_factor
)

attn_weight = torch.softmax(attn_weight, dim=-1)
attn_matrix = torch.matmul(attn_weight, v)
y = self.projection(attn_matrix)
```

---

5 Quantum Gradient Calculation

Tecniche di calcolo del gradiente in presenza di operazioni quantistiche

---

PyTorch calcola automaticamente i gradienti per le operazioni classiche (nn.Linear, nn.LayerNorm, ecc.) usando la backpropagation. Tuttavia, non può "vedere" all'interno dell'esecuzione di un circuito quantistico (che sia su un simulatore o su una QPU reale) per calcolare i gradienti in modo automatico. Il paper propone due metodi per calcolare il gradiente della funzione `forward` rispetto all'input `parameters` contentente tutti i set di angoli per ogni coppia di token (i, j): la Parameter-Shift Rule e l'SPSA.

5.1 Parameter-Shift Rule

Il valore atteso di un osservabile $\hat{O}$ (ad esempio Pauli operator) è dato da:

$$f(\theta) = \langle 0|U^\dagger(\theta)\hat{O}U(\theta)|0\rangle \tag{1}$$

e la parameter shift rule calcola la sua derivata rispetto a θ in questo modo:

$$\frac{\partial f(\theta)}{\partial \theta} = \frac{1}{2}\left[f\left(\theta + \frac{\pi}{2}\right) - f\left(\theta - \frac{\pi}{2}\right)\right] \tag{2}$$

Si valuta $f(\theta)$ per valori dei parametri $\theta \pm \frac{\pi}{2}$, riuscendo a calcolare il gradiente in modo esatto ma dovendo utilizzare due circuiti.

```
shift_right_tensors = []
shift_left_tensors = []

for i in range(groups):
    for j in range(param_count):
        shift_right = cleaned_circuit_parameters.clone()
        shift_right[:, i, j] += shift
        shift_left = cleaned_circuit_parameters.clone()
        shift_left[:, i, j] -= shift
        shift_right_tensors.append(shift_right)
        shift_left_tensors.append(shift_left)

all_shift_right = torch.stack(shift_right_tensors).reshape(-1, groups,
    param_count)
all_shift_left = torch.stack(shift_left_tensors).reshape(-1, groups, param_
    count)

with torch.no_grad():
    all_grad_expectation_right = quantum_circuit.run(all_shift_right).
        reshape(groups * param_count, -1)
    all_grad_expectation_left = quantum_circuit.run(all_shift_left).reshape
        (groups * param_count, -1)

gradient_estimates = (all_grad_expectation_right - all_grad_expectation_
    left) / (2 * shift)
```

I loop annidati scorrono ogni singolo parametro del circuito quantistico per il quale si vuole calcolare il gradiente. In shift_right si memorizzano i parametri perturbati di pi/2, in shift_left i parametri perturbati di -pi/2. Si eseguono i circuiti quantistici due volte per ogni singolo parametro e si calcola il gradiente esatto usando la formula descritta sopra.

5.2 Simultaneous perturbation stochastic approximation (SPSA)

Calcolare il gradiente esatto di un circuito quantistico può essere costoso. La Parameter-Shift rule, come abbiamo visto, richiede due esecuzioni del circuito per ogni parametro. L'algoritmo Simultaneous Perturbation Stochastic Approximation (SPSA) offre un'alternativa più efficiente in termini di numero di esecuzioni. L'idea è di perturbare tutti i parametri $\mathbf{x}$ contemporaneamente di una piccola quantità ϵ lungo una direzione casuale Δ per poi calcolare il valore della funzione solo in due punti: $\mathbf{x} + \epsilon\Delta$ e $\mathbf{x} - \epsilon\Delta$.
In particolare, si cerca un vettore di parametri $\mathbf{x} \in \mathbb{R}^m$ che minimizzi

$$\min_{\mathbf{x}} f(\mathbf{x}) = \min_{\mathbf{x}} \mathbb{E}_{\xi}\left[F(\mathbf{x}, \xi)\right], \tag{3}$$

dove $F : \mathbb{R}^m \to \mathbb{R}$ dipende dal vettore di parametri $\mathbf{x} \in \mathbb{R}^m$ e da un termine di rumore ξ.

Il gradiente $\nabla f(\mathbf{x})$ viene approssimato in questo modo:

$$\nabla f(\mathbf{x}) \approx \frac{f(\mathbf{x} + \epsilon \cdot \boldsymbol{\Delta}) - f(\mathbf{x} - \epsilon \cdot \boldsymbol{\Delta})}{2\epsilon\boldsymbol{\Delta}}, \tag{4}$$

dove $\boldsymbol{\Delta}$ è un vettore m-dimensionale con elementi scelti casualmente tra ± 1, e ϵ è il passo di perturbazione, nel nostro caso fissato a 0,01.

```
delta = (torch.randint(0, 2, cleaned_circuit_parameters.shape, device=
    device).float() * 2 - 1) * epsilon
params_plus, params_minus = cleaned_circuit_parameters + delta, cleaned_
    circuit_parameters - delta
with torch.no_grad():
    exp_concat = quantum_circuit.run(torch.cat((params_plus, params_minus),
        dim=0))
    num_circuits = params_plus.size(0)
    exp_plus = exp_concat[:num_circuits]
    exp_minus = exp_concat[num_circuits:]
exp_diff = (exp_plus - exp_minus).unsqueeze(-1).unsqueeze(-1)
spsa_gradient_unique = exp_diff.to(device) / (2 * delta)
```

Indipendentemente dal numero di parametri, SPSA richiede due sole esecuzioni del circuito per stimare l'intero gradiente. Questo lo rende molto più veloce rispetto al Parameter-Shift. Tuttavia è una stima rumorosa a causa della perturbazione casuale Δ che può rendere l'addestramento meno stabile.

---

6 Inferno GPT

Applicazione del transformer quantistico per generare token testuali

---

Una volta implementata l'architettura quantistica proposta dal paper, che nasce per la generazione di molecole in formato SMILE, abbiamo voluto provare la stessa architettura su un dominio differenze e realizzare un trasformer quantistico GPT-2 like. Di seguito le modifiche apportate che ci hanno permesso di ottenere un modello capace di generare testo usando lo stesso stile linguistico di Dante Alighieri addestrando il transformer sulle terzine di tutti i canti dell'inferno. Non avendo più molecole in formato SMILE come input, ma semplice testo, è stato necessario modificare il vocabolario, che ora deve contenere i caratteri propri della lingua italiana e il tokenizer, che considera ogni carattere come token. Di seguito il loop di generazione:

```
for _ in range(max_len - len(prompt_tokens)):
    idx_cond = idx if idx.size(1) <= block_size else idx[:, -block_size:]

    logits = model(idx_cond)

    if temperature == 0.0:
        idx_next = torch.argmax(logits, dim=-1, keepdim=True) # Shape: (1,
            1)
    else:
        logits = logits / temperature

        if top_k is not None and top_k > 0:
            v, _ = torch.topk(logits, min(top_k, logits.size(-1)))
            logits[logits < v[:, [-1]]] = -float('Inf')

        probs = F.softmax(logits, dim=-1) # Shape: (1, vocab_size)

        idx_next = torch.multinomial(probs, num_samples=1) # Shape: (1, 1)

    idx = torch.cat((idx, idx_next), dim=1) # Shape: (1, T_current + 1)

```

```
21      if idx_next.item() == eos_idx:
22          break
```

E' stato volutamente utilizzato un tokenizer a livello di carattere a titolo esemplificativo e non siamo riusciti a portare a termine un training soddisfacente a causa di scarsità di risorse computazionali. Ciò nonostante il modello funziona correttamente.

7 Conclusioni

Conclusioni

In questo lavoro abbiamo esplorato l'integrazione di circuiti quantistici all'interno dei transformers, focalizzandoci su un modello di *Quantum Transformer*. A differenza dei Transformer classici, in cui la similarità tra token viene calcolata tramite prodotti scalari, il modello proposto stima tale similarità attraverso circuiti quantistici parametrizzati (PQC), sfruttando il *Hadamard test* per calcolare i coefficienti di attenzione.

Dopo un'attenta analisi del paper originale, abbiamo implementato una versione semplificata dell'architettura proposta, escludendo il condizionamento sulle proprietà molecolari per ridurre la complessità computazionale. In fase di inferenza, abbiamo utilizzato direttamente i pesi rilasciati dagli autori, senza eseguire un training completo.

Per riprodurre fedelmente l'architettura anche nella fase di training, pur senza eseguire l'ottimizzazione dei parametri, è stato necessario gestire con attenzione la struttura dei circuiti e la loro esecuzione parallela. In particolare, è stato impiegato il modulo `cudaq.observe_async` per distribuire il calcolo dei PQC su più QPU, riducendo il costo computazionale. Inoltre, sono state considerate due tecniche di ottimizzazione per i parametri quantistici: la *Parameter-Shift Rule* e l'algoritmo *SPSA* (Simultaneous Perturbation Stochastic Approximation), evidenziandone vantaggi e limitazioni.

Infine, abbiamo condotto un esperimento di *domain shift* addestrando il modello su un corpus poetico tratto dall'*Inferno* di Dante, dimostrando che l'architettura ibrida è in grado di adattarsi anche a compiti di generazione testuale fuori dominio.

Nonostante le attuali limitazioni dell'hardware quantistico, questo studio conferma la fattibilità e il potenziale dell'integrazione quantistica in modelli complessi.

References

[1] Dante Alighieri. Inferno, la divina commedia. *https://archive.org/stream/ladivinacommedia00997gut/1ddcd09.txt*, 1314.

[2] et al. Biamonte, Jacob. Quantum machine learning. *Nature 549.7671 (2017): 195-202.*, 2017.

[3] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2013.

[4] Raghunathan Ramakrishnan, Pavlo O Dral, Matthias Rupp, and O Anatole von Lilienfeld. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific Data*, 1:140022, 2014.

[5] Anthony M. Smaldone, Yu Shee, Gregory W. Kyro, Marwa H. Farag, Zohim Chandani, Elica Kyoseva, and Victor S. Batista. A hybrid transformer architecture with a quantized self-attention mechanism applied to molecular generation. *arXiv preprint arXiv:2502.19214*, 2025.

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.

[7] David Weininger. Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of chemical information and computer sciences*, 28(1):31–36, 1988.