UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Un percorso di avvicinamento ai modelli di apprendimento automatico basati su transformer

Relatore:

Carlo Fantozzi

Laureando:

Luca Lavezzi, 1230952

ANNO ACCADEMICO 2023 – 2024
Data di laurea 14/03/2024

Abstract

Con l'avvento di algoritmi e modelli sempre più avanzati, si è vista una rapida evoluzione nell'ambito dell'intelligenza artificiale: processori sempre più potenti, GPU specializzate e architetture di calcolo distribuito hanno rivoluzionato il campo delle reti neurali. In questa tesi vengono analizzati tre concetti chiave che hanno essenzialmente ridefinito la progettazione dei modelli di apprendimento automatico nel campo AI. Più precisamente, vengono esplorati il modello Encoder/Decoder, il meccanismo dell'attenzione e l'architettura Transformer, illustrando il loro funzionamento, mettendo in evidenza come siano correlati fra di loro e fornendo dei casi pratici, allo scopo di condurre un'analisi approfondita delle loro prestazioni.

Introduzione

Negli ultimi anni, nell'ambito dell'intelligenza artificiale, e in particolare delle reti neurali, si è visto un incremento di nuove architetture atte a compiti come la traduzione automatica, la generazione di testi e l'interpretazione di immagini. Precursori di tali architetture furono ideati già a metà degli anni 90, ma ricevettero scarse attenzioni e altrettanti successi per via delle limitazioni poste da diversi fattori di quell'epoca. Uno di questi fattori era sicuramente l'hardware, non capace di soddisfare i requisiti posti da tali modelli per via del loro costo elevato. Un altro problema consisteva nei tempi di addestramento di questi modelli, richiedendo giorni per produrre output mediocri. Sebbene fosse rivoluzionario il concetto di una macchina pensante, il grado della tecnologia dei tempi passati non permise una ricerca più estensiva di tale argomento. Infatti, altri tipi di architetture vennero usati per l'avanzamento tecnologico, lasciando in disparte il concetto di intelligenza artificiale per un modesto periodo. Avanzando di alcuni anni, una volta raggiunto un grado di capacità computazionale adeguato, si ripresero le ricerche per via del rapido sviluppo di modelli idonei a varie mansioni, superando di gran lunga le performance gli archetipi precedentemente sviluppati. L'obiettivo di questa tesi è quello di elencare dei modelli significativi, esplicitando in particolare il loro funzionamento ed evoluzione, mettendo a luce perché essi abbiano ricevuto particolare successo, e perché sia stato necessario apportarvi delle modifiche al fine di mantenere le loro prestazioni al pari degli standard odierni. Viene inizialmente introdotto il modello Encoder/Decoder da Cho et al. (2014) [20]. Questo nuovo approccio ridefinisce il modo in cui le reti neurali affrontano i compiti sequenziali, sfruttando la località e la temporalità dei dati di input per creare una rappresentazione coerente che possa essere facilmente interpretabile. Successivamente, si presenterà il meccanismo dell'attenzione di Bahdanau et al. (2014) [12], progettato sulle fondamenta del modello Encoder/Decoder, che mira a risolvere problemi creati dalla sequenzialità stessa sulla quale esso si basa. L'attenzione fa più uso del parallelismo dei dati, un concetto che verrà espanso con i modelli futuri, grazie anche all'avanzamento dell'hardware. Un degno successore dell'attenzione consiste nel modello Transformer, inizialmente ideato da Vaswani et al. (2017) [5], riportato in "Attention Is All You Need". L'architettura Transformer scarta il principio della sequenzialità dei dati, massimizzando il parallelismo ove possibile, sebbene anch'esso prenda riferimento dai suoi predecessori, facente uso di parti assimilabili a un modello Encoder/Decoder e di attenzione nelle sue sfumature. È importante notare che esistono approcci diversi a questi tipi di architetture e i tre argomenti verranno spiegati facendo riferimento ai modelli elencati precedentemente: il modello Encoder/Decoder di Cho et al. (2014) [20], il meccanismo dell'attenzione di Bahdanau et al. (2014) [12] e il modello Transformer di Vaswani et al. (2017) [5].

1 Encoder/Decoder

I modelli Encoder/Decoder, anche chiamati sequence to sequence network, trovano applicazione in vari campi dell'intelligenza artificiale, come la traduzione automatica, il riconoscimento vocale e la creazione di didascalie video. In particolare, sono stati utilizzati ampliamente per compiti di NLP (Natural Language Processing), un campo che combina la linguistica con modelli statistici di machine learning e deep learning. Il compito di questo capitolo è di introdurre i concetti fondamentali legati alle basi della architettura Encoder/Decoder, seguito poi da una introduzione all'architettura stessa. Successivamente, quest'ultimo verrà esplorato più in dettaglio e verranno discusse le sue limitazioni. Nel successivo capitolo sul meccanismo dell'attenzione, il modello basato su Encoder/Decoder introdotto da Bahdanau et al. (2014) verrà messo a confronto con un modello mirato a risolvere le sue lacune, in modo da misurare la differenza nelle loro prestazioni.

1.1 Neuroni, reti neurali e funzioni di attivazione

Una rete neurale artificiale (NN o ANN) è un gruppo interconnesso di nodi, ispirato a una semplificazione dei neuroni del cervello umano. In questa rappresentazione, ogni nodo rappresenta un neurone artificiale e una freccia rappresenta una connessione dall'output di un neurone all'input di un altro. Una NN fa uso di collezioni di neuroni che si connettono fra di loro, come le sinapsi di un cervello biologico, in modo che possano essere passati dei segnali. Il segnale in una connessione è un numero reale, e l'output di ciascun neurone è calcolato da una funzione non lineare della somma dei suoi input. Le connessioni vengono chiamate archi; i neuroni e gli archi possiedono tipicamente un peso regolabile durante il processo di apprendimento. Tipicamente, i neuroni sono aggregati in strati, o "layer", che possono eseguire trasformazioni diverse sui loro input [4]. Esistono anche strati diversi aventi funzioni specifiche, come gli strati convoluzionali, usati nelle reti convoluzionali per catturare pattern nelle rappresentazioni di input, particolarmente usato quando questi input sono immagini.

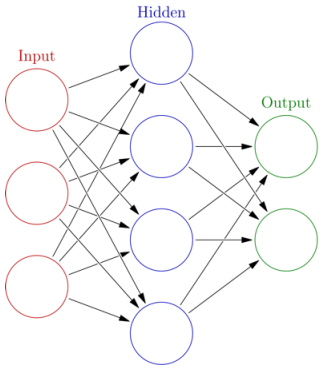


Figura 1.1 – Immagine di una rete neurale [4]

Un neurone riceve uno o più input $(x_1, \dots, x_n)$, che sono segnali provenienti da altri neuroni o dall'ingresso del modello. Questi input, quindi, vengono moltiplicati per dei pesi (w_1, w_n), che servono a rappresentare l'importanza di ogni input. Tutti gli input pesati vengono poi sommati insieme, creando un nuovo valore $z = \Sigma_i \, w_i x_i$, per poi essere passati attraverso una

funzione di attivazione f, tale che $f(z) = a$, con a corrispondente all'output del neurone, al fine di introdurre non linearità [36]. Opzionalmente si può sommare un "bias", che consiste nella somma con un numero per rappresentare una traslazione in una direzione. In questo modo z viene calcolato come $z = \Sigma_i (w_i x_i) + b$.

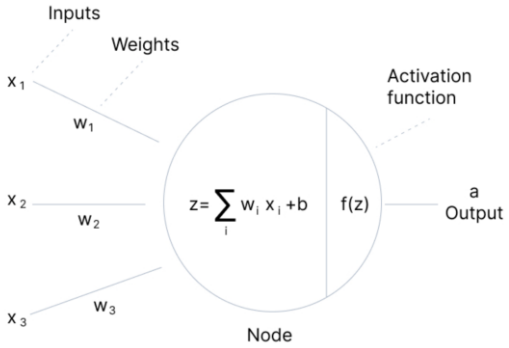


Figura 1.2 – Immagine di un neurone artificiale [28]

Dopo questi passaggi, l'output del neurone viene passato al prossimo strato, oppure rappresenta l'output finale del modello. Ogni strato ha un certo numero di neuroni: maggiore è il loro numero, maggiore è la capacità della rete di apprendere e catturare relazioni complesse tra i dati. Tuttavia, l'aggiunta di neuroni non è sempre vantaggiosa: come con gli strati, un numero eccessivo può portare all'overfitting, un fenomeno che si verifica quando il modello si adatta troppo ai dati di addestramento, non riuscendo a generalizzare abbastanza con i dati successivi. Un numero ridotto di neuroni invece potrebbe portare all'underfitting, che occorre quando il modello è troppo semplice e non è capace di catturare i pattern nei dati.
Gli strati nelle reti neurali possono essere suddivisi in tre categorie principali: input, nascosto (hidden) e output. Lo strato di input, come suggerisce il nome, rappresenta l'ingresso della rete neurale; l'input viene fornito a questo strato, che viene quindi passato al prossimo strato dopo aver subito una trasformazione. Gli stati nascosti rappresentano la parte intermedia del modello, trovandosi nel mezzo tra lo strato di input e quello di output. Vengono chiamati nascosti perché le loro attività non sono direttamente osservabili dall'esterno della rete (non possono essere generalmente monitorate come l'input fornito o il context vector). Ogni strato nascosto è connesso allo strato precedente e a quello successivo, che si tratti di uno altro strato nascosto, o di uno strato di input/output. Gli strati nascosti sono usati per apprendere le rappresentazioni complesse dei dati di input, ed il numero di essi da utilizzare varia da applicazione ad applicazione, nonostante sia difficile determinare il numero esatto [33]. Citando Ian Goodfellow [14, pag. 201]:

"Empirically, greater depth does seem to result in better generalization for a wide variety of tasks […] These results suggest that using deep architectures does indeed express a useful prior over the space of functions the model learns."

Un modo per trovare la quantità ideale di strati (anche detta profondità del modello) consiste nel condurre esperimenti ai quali possa essere assegnato un punteggio oggettivo, per misurare la qualità del modello facendo variare il numero di layer.
Lo strato di output ha la funzione di restituire il risultato finale della rete, trasportato in precedenza dagli strati nascosti. Uno strato spesso usato e di grande importanza è lo strato di

attivazione, usato per introdurre non linearità, essenziale per catturare pattern e relazioni non lineari nei dati, che sono spesso presenti nei problemi del mondo reale [28].
Esistono varie funzioni di attivazione, ognuna utilizzata per la peculiarità delle sue caratteristiche intorno a un certo intervallo di valori. Queste funzioni sono in grado di introdurre fenomeni come la non linearità e la possibilità di modificare l'output in un intervallo ben definito, come la saturazione. Una funzione di attivazione in una rete neurale determina come la somma pesata dell'input sia trasformata in un output di un nodo in un certo strato [16]. Alcune funzioni di attivazione vengono mostrate in figura 1.3:

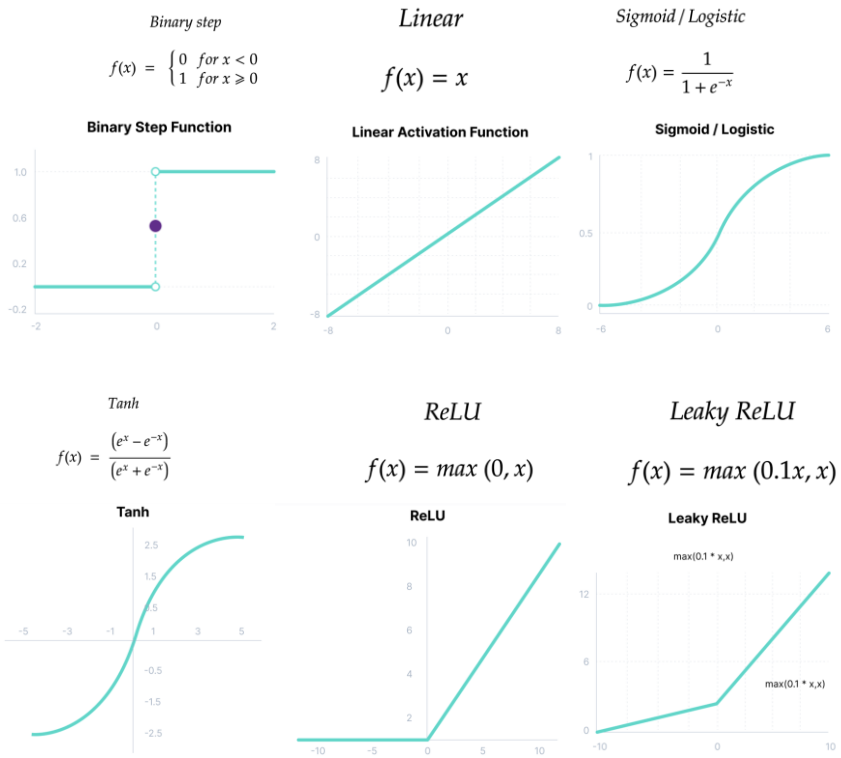


Figura 1.3 – Funzioni di attivazione e la loro rappresentazione grafica [28]

1.2 Introduzione al modello Encoder/Decoder

Il modello Encoder/Decoder è un'architettura usata nell'apprendimento automatico composta da due componenti: un Encoder e un Decoder. Questo modello, proposto inizialmente da Cho et al. (2014) [20] e poco dopo da Sutskever et al. (2014) [15], mira a risolvere il problema della gestione di sequenze di dati di lunghezze diverse [1].
Tradizionalmente, nelle reti neurali, la lunghezza degli input e degli output doveva essere fissa, ma nei problemi di linguaggio naturale, per esempio, la lunghezza delle frasi può variare notevolmente. Inoltre, questo modello è stato sviluppato per affrontare il problema della rappresentazione delle sequenze e la generazione di output coerenti e significativi.
L'architettura si divide in due componenti, Encoder e Decoder. L'Encoder prende in input una sequenza, che può essere variabile in lunghezza, e la converte in una rappresentazione a lunghezza fissa, producendo uno stato finale (solitamente chiamato context vector), che viene dato come input al Decoder. Il Decoder riceve questa rappresentazione, che serve da riassunto della informazione della sequenza di input dell'Encoder, e quindi genera una sequenza di output (anche di lunghezza diversa), basata sugli output restituiti precedentemente (essendo

autoregressivo) e sul context vector [10]. Questa architettura, essendo stata progettata per gestire sequenze di lunghezze diverse, sia per l'input che per l'output, consente una maggiore flessibilità nel trattare dati di natura sequenziale. Sia l'Encoder che il Decoder possono essere rappresentati con diverse architetture interne (RNN, LSTM, GRU sono i principali) [32].

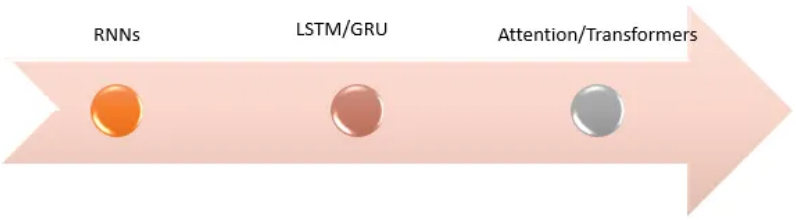


Figura 1.4 – Progressione temporale delle architetture utilizzate [30]

Le due parti vengono addestrate congiuntamente con lo scopo di migliorare la precisione del modello, sia usando le tecniche dell'apprendimento supervisionato, quando si dispone di coppie output/label (etichette), che dell'apprendimento non supervisionato, dove i dati non hanno etichette associate e il modello impara a trovare rappresentazioni significative senza supervisione esplicita. Il modello dell'Encoder/Decoder è composto da uno strato di input, uno o più strati nascosti e uno strato di output.

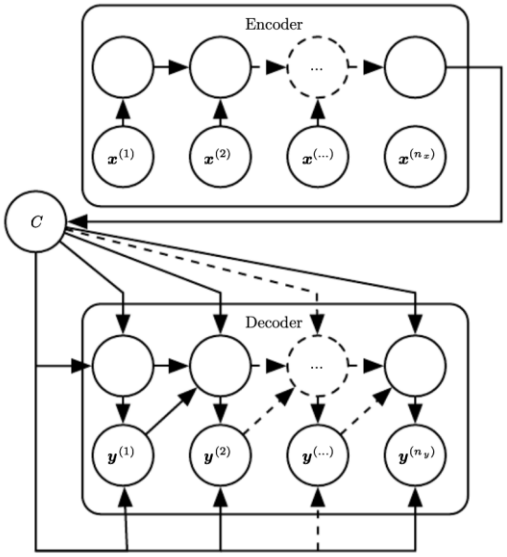


Figura 1.5 – Struttura di un modello Encoder/Decoder [14]

1.3 Funzionamento in dettaglio dell'Encoder

L'Encoder ha il ruolo di processare l'input fornito al modello in una rappresentazione compatta e determinata; questo avviene passandolo attraverso uno o più strati neurali, con lo scopo di estrarre informazioni con diverse scale di astrazione. Lo stato finale dell'Encoder corrisponde ad un vettore che verrà dato in input al Decoder, contenente informazioni su tutta la sequenza di input dell'Encoder, che si tratti di una sequenza di parole, immagini, campioni audio, ecc. Considerato l'input $(x_1, x_2, \ldots, x_n)$, viene fornito un token x_t all'Encoder per ogni

step temporale t alla volta. Ad ogni passo t, lo stato nascosto $h_{\langle t \rangle}$ dell'Encoder viene aggiornato nel modo seguente:
$h_{\langle t \rangle} = f(h_{\langle t-1 \rangle}, x_t, \theta)$, con $h_{\langle t-1 \rangle}$ lo stato nascosto al tempo $t-1$, x_t la sequenza di dati passata al tempo t e θ rappresentate i parametri del modello (quindi inclusi i pesi e i bias). La funzione f può consistere in una semplice funzione di attivazione non lineare, ma può anche essere complessa come una unità LSTM.

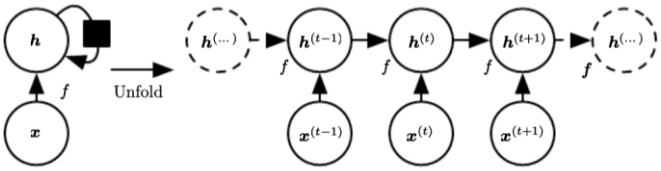


Figura 1.6 – Schema di un modello RNN (Recurrent Neural Network) [14]

Lo strato di output può essere considerato come l'ultimo strato nascosto h_n, usato come rappresentazione C della sequenza di input che viene fornita al Decoder. Il vettore C rappresenta un'informazione compatta di tutta la sequenza $(x_1, x_2, \dots, x_n)$ di input, che sarà utile al Decoder per generare la sequenza $(y_1, y_2, \dots, y_m)$ di output, e verrà utilizzato in ogni strato per fornire il contesto necessario alla generazione di un output coerente [14].

1.4 Funzionamento in dettaglio del Decoder

Il Decoder ha il compito di fornire l'output sotto forma di sequenza $(y_1, y_2, \dots, y_m)$, passo dopo passo, facendo utilizzo del vettore di contesto C. Oltre ad usare il vettore di contesto C, ogni passo al tempo t userà lo stato nascosto del decoder al tempo $t-1$ insieme all'output del tempo $t-1$: $h_{\langle t \rangle} = g\left(\boldsymbol{h}_{\langle t-1 \rangle}, \boldsymbol{y_{t-1}}, C, \boldsymbol{\gamma}\right)$, vengono utilizzati la funzione g e i parametri γ per evidenziare che sono diversi dalle funzioni e parametri dell'Encoder, nonostante si comportino in maniera analoga, mentre y_{t-1} corrisponde all'output del Decoder nel periodo $t-1$. Il processo del modello Decoder è auto regressivo, essendo la predizione y_t dipendente dalla predizione precedente y_{t-1}: ciò significa che la predizione y_t al tempo t è dipendente da tutte le predizioni precedenti $(y_1, \dots, y_{t-1})$ [34].

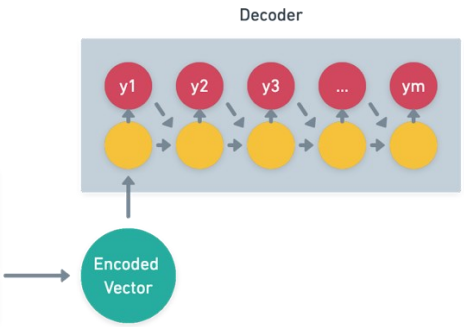


Figura 1.7 – Immagine di un Decoder [32]

1.5 Interazione tra Encoder e Decoder

Durante il processo di inferenza, finito l'addestramento, l'Encoder e il Decoder comunicano attraverso il context vector, che racchiude al suo interno un riassunto di tutte le informazioni passate dall'input. Dal punto di vista probabilistico, il modello rappresenta un metodo generale per apprendere la distribuzione condizionale su una sequenza di lunghezza variabile, data un'altra sequenza di lunghezza variabile, per esempio $p(y_1, \dots, y_{T'}|x_1, \dots, x_T)$, facendo notare che le lunghezze di input e output T' e T possono differire [10].
Durante la fase di addestramento, tuttavia, l'Encoder e il Decoder interagiscono in modo diverso. Si può dividere il processo in due parti: forward propagation e backward propagation. Durante la fase di forward propagation, i dati vengono passati attraverso la rete neurale dall'input all'output, come già descritto, con il fine di produrre un output coerente. La backward propagation è la fase in cui l'errore tra l'output predetto e l'output reale viene calcolato e propagato all'indietro attraverso la rete. Utilizzando l'errore, calcolato tipicamente attraverso una funzione di "loss", si calcolano i gradienti rispetto ai pesi della rete, trovando quanto ogni peso ha contribuito all'errore. I pesi e i bias, inizialmente impostati con valori casuali, vengono regolati in modo da ridurre l'errore generato dall'output del Decoder secondo il confronto del valore ottenuto con il valore desiderato (ground truth) [3]. Esistono diverse funzioni di loss, progettate per misurare in modi differenti la differenza tra l'output predetto da un modello e la verità dei dati di addestramento. Questa varietà consente di adattare il processo di addestramento per risolvere problemi specifici, come la regressione, la classificazione o la generazione di sequenze, ponendo l'accento su aspetti differenti dell'errore del modello. Alcune tra le funzioni più usate sono le seguenti:

Cross-Entropy Loss: è comunemente usata per problemi di classificazione e prevede la misura della discrepanza tra la distribuzione di probabilità predetta e la distribuzione reali dei dati. $\mathcal{L} = -\Sigma_i \hat{y}_i \log(y_i)$, dove $\hat{y}$ corrisponde ai valori corretti e y corrisponde ai valori predetti dal modello [3].

Negative Log-Likelihood Loss: simile alla cross entropy loss e spesso utilizzata nelle sequenze, misura la discrepanza tra le distribuzioni di probabilità previste e quelle reali. $\mathcal{L} = -\sum_{i=1}^{n} log\big(p(y_i|x_i)\big)$, dove n è il numero totale dei campioni nel set di dati, $p(y_i|x_i)$ rappresenta la probabilità prevista dal modello per la il valore corretto y_i data l'istanza x_i [22].

Mean Squared Error (MSE) Loss: è una funzione di loss comunemente usata per i problemi di regressione; misura la media delle differenze quadratiche tra i valori predetti e quelli effettivi. $\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$, dove n è il numero totale di osservazioni nel dataset, y_i è il valore effettivo dell'osservazione i e $\hat{y}_i$ è il valore predetto per l'osservazione i [3].

Al fine di ottimizzare i parametri del modello per ottenere un output oggettivamente migliore, si vuole minimizzare la funzione di loss, e per questo esistono degli algoritmi appositi come la discesa del gradiente (Gradient Descent GD) o la discesa stocastica del gradiente (Stochastic Gradient Descent SGD).

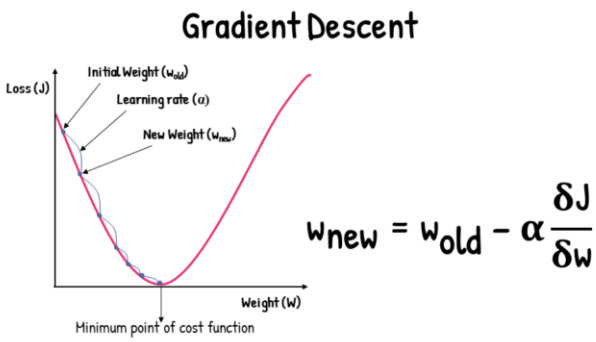


Figura 1.8 – Calcolo dei nuovi parametri utilizzando la discesa del gradiente [24]

Discesa del gradiente:
L'obiettivo che si pone la discesa del gradiente è quello di trovare i valori dei parametri del modello che minimizzano la funzione di loss. Il gradiente rappresenta la pendenza della funzione di costo in un certo punto ed indica la direzione in cui la funzione di loss cresce più rapidamente. La discesa del gradiente itera attraverso i parametri del modello in modo da spostarsi nella direzione opposta al gradiente, riducendo progressivamente l'errore del modello, con un passo di apprendimento (learning step) determinato dal tasso di apprendimento (learning rate). Questo processo continua finché non viene raggiunta una certa soglia dove la variazione nella funzione di loss è sufficientemente piccola o quando viene raggiunto un numero prefissato di iterazioni. Uno dei parametri modificabili nella discesa del gradiente è il tasso di apprendimento, che solitamente viene fatto diminuire nel corso delle varie epoche; un'epoca nella discesa del gradiente batch (a insiemi) rappresenta una singola iterazione attraverso l'intero set di dati di addestramento. Il tasso di apprendimento viene ridotto per rendere l'ottimizzazione più stabile: all'inizio dell'addestramento il modello può esplorare più rapidamente lo spazio dei parametri, mentre una riduzione successiva diminuisce il rischio di oscillazioni, in modo da convergere ad un minimo globale o locale della funzione di loss. La discesa del gradiente può essere rappresentata nel modo seguente:
$\theta_{t+1} = \theta_t - \alpha\nabla_\theta \mathcal{L}(\theta)$, dove θ_t rappresenta i parametri nell'iterazione t, α rappresenta il tasso di apprendimento e $\mathcal{L}(\theta)$ è il gradiente della funzione di loss rispetto ai parametri [24].

Discesa del gradiente stocastica:
Risolve il problema della discesa del gradiente, ovvero il costo computazionale di calcolare il gradiente di tutti i parametri interessati, che per i modelli più grandi possono diventare milioni o anche miliardi. Nella discesa del gradiente stocastica, al posto di calcolare il gradiente rispetto a tutti i parametri, si prende solo un sotto-set (batch), in modo da incrementare la velocità di convergenza. Per ogni iterazione o epoca di addestramento, si seleziona casualmente un sotto-set di dati, si calcola il gradiente della funzione di loss rispetto a quei dati e si aggiornano di conseguenza i pesi del modello lungo la direzione opposta del gradiente. L'uso di batch di dati più piccoli introduce una componente di casualità, o "rumore stocastico", che può deviare il percorso della funzione di loss verso un minimo.
Contro intuitivamente, rimbalzando, il valore potrebbe uscire fuori da minimi locali e quindi dirigersi verso valori di minimo migliori nel caso la funzione di loss sia non convessa [27].

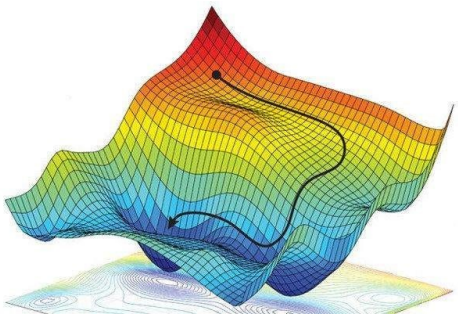


Figura 1.9 – Piano a tre dimensioni rappresentante un possibile scenario della funzione di loss e la direzione che potrebbe assumere un algoritmo di minimizzazione [27]

1.6 Limitazioni del modello Encoder/Decoder

Uno dei problemi principali che affligge i modelli Encoder/Decoder, in particolare quelli basati su RNN, LSTM e GRU, è la difficoltà nel trattare e catturare informazioni da sequenze troppo lunghe, il che può portare al problema del vanishing/exploding gradient, dove i gradienti calcolati diventano estremamente piccoli/grandi rispettivamente. Questo rende difficile l'addestramento del modello su sequenze lunghe, e può limitare la capacità del modello di catturare relazioni a lungo termine, causando problemi di accuratezza e coerenza nelle predizioni. Dal lato pratico, questo problema avviene durante la backpropagation, quando i gradienti diventano troppo piccoli/grandi, influenzando in modo trascurabile i pesi degli strati inferiori/superiori; questo è spesso dovuto alla moltiplicazione continua di valori minori/maggiori di 1 per l'utilizzo della regola della catena (chain rule) su un numero eccessivo di strati [29].

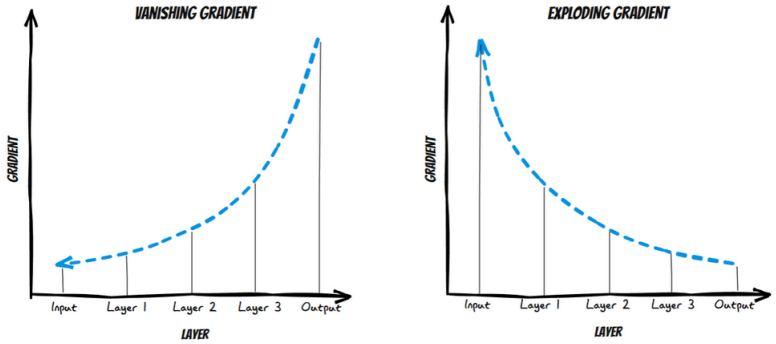


Figura 1.10 – Figura rappresentante l'effetto del vanishing/exploding gradient con l'aumentare degli strati nel modello [29]

Esistono più metodi per mitigare il problema del vanishing/exploding gradient; di seguito ne vengono elencati alcuni.
Vanishing gradient:

1) Utilizzo di funzioni di attivazione diverse: alcune funzioni di attivazione risolvono in parte il problema del vanishing gradient; presa per esempio la funzione ReLU, definita come: $f(x) = max(0, x)$, quando l'input è positivo, la derivata vale 1, evitando la diminuzione del gradiente durante la backpropagation. Tuttavia, la funzione ReLU potrebbe portare al problema della "dying ReLU", quando l'output è sempre zero, impedendo la propagazione dei gradienti [29].

2) Usare nuovi schemi di inizializzazione dei pesi: la scelta iniziale dei pesi della rete influenza la propagazione dei gradienti attraverso la rete durante la fase di backpropagation; alcuni schemi di inizializzazione dei pesi mirano a mantenere la varianza costante durante il passaggio attraverso i vari strati della rete [29].

3) Usare nuovi ottimizzatori e tassi di apprendimento: alcuni ottimizzatori come l'Adagrad o Adadelta possono gestire meglio il problema del vanishing gradient, regolando i tassi di apprendimento in base alla storia dei gradienti passati, consentendo un aggiornamento più accurato dei pesi [29].

Exploding gradient:

1) Utilizzo del gradient clipping: viene controllato il gradiente durante l'addestramento e se supera una determinata soglia viene limitato ad un valore prestabilito [29].

2) Usare nuovi schemi di inizializzazione dei pesi: in analogo al problema del vanishing gradient, usare degli schemi di inizializzazione dei pesi consoni può contribuire a ridurre la probabilità di avere gradienti divergenti a valori elevati [29].

Un'altra tecnica usata per risolvere il problema del vanishing/exploding gradient è il meccanismo dell'attenzione, che permette al modello di focalizzarsi su parti specifiche dell'input durante il processo di apprendimento, riducendo la dipendenza da collegamenti distanti all'interno della sequenza.

2 Meccanismo dell'attenzione

Il meccanismo dell'attenzione viene costruito sulle basi del modello Encoder/Decoder per migliorarne le prestazioni. Esso si basa su meccanismi che possono essere ricondotti all'attenzione cognitiva con l'obiettivo di costruire una gerarchia di priorità mentre si sta creando una sequenza di output. In questo capitolo verrà presentato il funzionamento del meccanismo in maniera completa, e in particolare verrà posta attenzione sulla metodologia di calcolo dei pesi che regolano l'importanza di ogni stato dell'Encoder per il Decoder. Verrà successivamente mostrata l'analisi delle prestazioni di un modello facente utilizzo dell'attenzione rispetto a un modello classico Encoder/Decoder. Nella parte finale del capitolo, verranno elencate alcune limitazioni dell'architettura che non permettono al modello di sfruttare il parallelismo che viene spesso fornito dai moderni sistemi hardware.

2.1 Introduzione al meccanismo dell'attenzione

Il meccanismo dell'attenzione è stato introdotto da Bahdanau et al. (2014) [12] con lo scopo di migliorare le prestazioni del modello Encoder/Decoder. Nel campo della psicologia, l'attenzione è il processo cognitivo di concentrarsi selettivamente su una o poche cose, ignorando le altre, e con questa architettura si cerca di dare al modello Encoder/Decoder una rappresentazione analoga [6]. L'attenzione mira a risolvere il problema principale che affligge il modello Encoder/Decoder: il collo di bottiglia creato dall'utilizzo di un vettore di encoding di dimensione fissa, limitando l'accesso del Decoder alle informazioni dell'input, in particolare per sequenze di lunghe dimensioni. Ad ogni passo temporale, nel Decoder vengono create delle connessioni dirette con l'Encoder per concentrarsi su una parte specifica della sequenza di input, facendo utilizzo di pesi ottenuti mettendo in relazione lo stato del Decoder con i vari stati dell'Encoder. Questi pesi vengono assegnati ai vari stati dell'Encoder; le parti più importanti degli stati dell'Encoder riceveranno i pesi più grandi. Mentre il modello Encoder/Decoder favorisce la parte più recente della sequenza per via della struttura della sua architettura, il meccanismo dell'attenzione dà la stessa quantità di considerazione ad ogni elemento della sequenza, facendo uso della computazione in parallelo. Al posto di calcolare lo stato interno del Decoder usando solamente lo stato e l'output precedenti, viene calcolato un vettore di attenzione utilizzando lo stato nascosto del Decoder nel passo temporale corrente, contenente la somma pesata degli stati interni dell'Encoder, che racchiude le informazioni sulle quali è stata posta attenzione. Il vettore di attenzione viene poi combinato con lo stato nascosto del Decoder per produrre un output che in questo modo sarà più coerente con la sequenza di input, anche per sequenze molto lunghe. Questo processo risulta nel passaggio dall'Encoder al Decoder di una quantità maggiore di informazione. Al posto di passare semplicemente il context vector, come nel modello Encoder/Decoder verrà passata anche la rappresentazione di tutti gli stati dell'Encoder, scalati in base ai loro pesi assegnati per ogni passo temporale.
È importante osservare che l'attenzione non viene utilizzata soltanto nel contesto NLP, un'altra applicazione, per esempio, rientra nel campo di "computer vision" (Xu et al. (2015)) [18], dove aiuta a riconoscere e concentrarsi su specifiche parti di un'immagine quando si cerca di generare una descrizione testuale. Focalizzandosi su oggetti specifici o regioni dell'immagine, il modello

genera descrizioni più dettagliate e coerenti, considerando anche dettagli di minore rilevanza e affrontando scenari complessi con maggiore precisione [2].

Figura 2.1 – Figura che dimostra l'utilizzo del meccanismo dell'attenzione su una immagine, le regioni sfumate in bianco indicano dove il modello pone attenzione [18]

L'attenzione viene rappresentata sotto molteplici forme; le categorie principali vengono elencate qui di seguito:

1) Soft attention: assegna i pesi di attenzione a ogni parte dell'input in modo continuo, creando una distribuzione ponderata in ogni punto. Questa attenzione consente al modello di porre attenzione in maniera graduale e su più aspetti dell'input contemporaneamente, consentendo una flessibilità maggiore nella rappresentazione delle relazioni tra le varie parti dell'input, a scapito di una computazione costosa quando l'input assume grandi dimensioni. Corrisponde essenzialmente al tipo di attenzione utilizzato da Bahdanau et al. (2014) [12].

2) Hard attention: a differenza della soft attention, la hard attention esegue selezioni più rigide e discrete delle parti dell'input, concentrandosi solo su una parte specifica. Questo approccio implica una scelta diretta e deterministica delle regioni dell'input su cui porre l'attenzione, escludendo le parti rimanenti, in modo da utilizzare meno risorse durante l'inferenza, con il difetto che il modello diventa non differenziabile (e quindi non può più fare uso di algoritmi di ottimizzazione che richiedono il calcolo del gradiente, come la discesa del gradiente) e richiede tecniche più complesse come la riduzione della varianza (variance reduction) o reinforcement learning per addestrarlo (Luong et al. (2015) [26]).

3) Self-attention: anche conosciuta come intra-attention, è un meccanismo che mette in relazione varie parti di una sequenza di input contemporaneamente, ciascun elemento dell'input contribuisce ai pesi di attenzione di ogni altro elemento, consentendo al modello di acquisire informazioni da ogni parte dell'input durante la sua elaborazione.

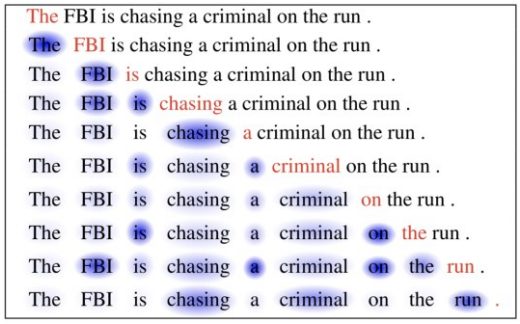


Figura 2.2 - Un esempio di self-attention. La parola dell'istante temporale corrente è evidenziata in rosso, in blu viene evidenziato il livello di attivazione delle parole rimanenti: più la tonalità diventa scura, maggiore è la correlazione [21].

In pratica, esistono diversi metodi per calcolare i pesi legati all'attenzione: un esempio è la funzione del punteggio di allineamento (alignment score function) utilizzata da Luong et al. (2015) [26], che viene calcolato come: $score(s_t, h_i) = s_t^T h_i$, dove si utilizza il prodotto scalare tra lo stato nascosto al tempo t del Decoder e lo stato nascosto al tempo i dell'Encoder. Un altro esempio è il punteggio di allineamento utilizzato da Vaswani et al. (2017) [5], chiamato scaled dot-product attention: $score(s_t, h_i) = \frac{s_t^T h_i}{\sqrt{n}}$, con n rappresentante la dimensione dello stato nascosto di input; quando l'input diventa troppo grande, la funzione di softmax può avere gradienti estremamente piccoli, inefficiente per l'apprendimento. Un altro importante tipo di attenzione è la multi-head attention, utilizzata da Vaswani et al. (2017) [5]: questa architettura computa l'attenzione più volte in parallelo, una per ogni testa (head, che consiste di un layer per il calcolo dell'attenzione) per catturare informazioni diverse per ogni testa utilizzata. È importante notare che le varie teste non sono correlate fra di loro mentre calcolano l'attenzione: operano in maniera disgiunta l'una dall'altra. Tale caratteristica sottolinea il punto di forza architetturale di questa struttura: la sua capacità di agire in modo indipendente e in parallelo. I vari punteggi di attenzione vengono poi concatenati e scalati opportunamente per rientrare nelle restrizioni poste dal modello, il che verrà successivamente spiegato in dettaglio nel capitolo 3. In generale, il meccanismo dell'attenzione fa uso di tre componenti: i queries Q, i keys K e i values V. Facendo riferimento a Bahdanau et al. (2014) [12], Q corrisponde allo stato nascosto del Decoder s_t, mentre V e K vengono associati allo stato nascosto dell'Encoder h_i. Ogni query viene associata ad una serie di keys per calcolare un punteggio: questa operazione fa riferimento al calcolo di $score$ precedentemente elencato. I punteggi vengono poi passati attraverso uno strato softmax per generare i pesi. L'attenzione viene infine calcolata facendo una somma pesata dei vettori in base al punteggio assegnatogli dal layer precedente. Un esempio per comprendere meglio il concetto di key/query/value è analogo alla ricerca dei video su Youtube: il motore di ricerca mapperà la query (quello che è stato digitato nella barra di ricerca) con un set di keys (titolo del video, descrizione, tags, ecc.) associati con i video candidati nel database, per poi presentare i video che hanno ottenuto la corrispondenza più alta (values) [19].

2.2 Funzionamento in dettaglio dell'attenzione

Il meccanismo dell'attenzione durante l'inferenza tipicamente consiste di questi passi:

1) Codifica dell'input: la sequenza dei dati in input viene passata attraverso i vari strati dell'Encoder, trasformandolo in un formato che può essere processato dal meccanismo dell'attenzione. Nell'Encoder non ci sono cambiamenti sostanziali: il processo dell'attenzione avviene nella parte del Decoder, sebbene questo possa essere considerato come una fase in più, interposto fra le due parti che compongono il modello Encoder/Decoder.

2) Generazione di una query: un vettore query viene selezionato in base allo stato corrente del Decoder s_t. Questo vettore rappresenta l'informazione sulla quale modello si concentrerà. Il vettore query funge da guida per la ricerca di dettagli o punti chiave all'interno dei dati di input.

3) Creazione di coppie key-value: questo passaggio coinvolge la suddivisione delle rappresentazioni dell'input in coppie chiave-valore. Sempre facendo riferimento a Bahdanau et al. (2014) [12], queste coppie corrispondono ai vari stati nascosti dell'Encoder h_i nell'istanza i-esima.

4) Calcolo del punteggio di similarità: rappresenta il processo in cui si calcola la similitudine tra il vettore di query e ciascuna chiave per misurare la loro compatibilità attraverso una rete feedforward. In questo modo si determina quanto una determinata chiave è rilevante rispetto alla query corrente. Possono essere utilizzate diverse metriche di similarità, come il prodotto scalare, la similarità del coseno o il prodotto scalato. Un esempio può essere quello di utilizzare il prodotto scalare tra la chiave corrispondente ai vari stati dell'Encoder h e la query corrispondente allo stato nascosto del Decoder s_t tale che $score(s_t, h) = s_t h$, dove h corrisponde a tutti gli stati nascosti dell'Encoder. Se si volessero prendere in considerazione gli elementi individuali allora si avrebbe: $score(s_t, h_i) = s_t h_i$. Più in generale, se q_t corrisponde al vettore query nell'istanza t e k_i al vettore chiave i-esimo, allora si può calcolare il punteggio di similarità corrispondente dei due come $e_{q_t,k_i} = g(q_t, k_i)$, dove g può essere una qualsiasi funzione, e nel caso del prodotto scalare si può scrivere come: $e_{q_t,k_i} = q_t k_i$.

5) Calcolo dei pesi di attenzione: dopo aver calcolato i punteggi di similarità tra la query e le chiavi, questi vengono processati attraverso una funzione softmax; questa funzione converte i punteggi in pesi di attenzione, fornendo una distribuzione di probabilità su quale chiave (e di conseguenza quale valore) è più rilevante la query in esame. Espandendo il concetto di funzione di softmax, quest'ultima è una funzione che converte un vettore di K numeri in una probabilità di distribuzione con K possibili esiti. Questa funzione viene spesso usata come l'ultima funzione di attivazione di una rete neurale per normalizzare l'output in una probabilità di distribuzione, in questo modo tutte le probabilità dei vari esiti si sommano a 1. Matematicamente, si definisce la funzione softmax come: $softmax(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_i}}$, dove $softmax(z)_i$ è il i-esimo elemento del vettore di output normalizzato, e è il numero di Nepero e z_i è l'elemento i-esimo del vettore z.

La sommatoria rappresenta la somma esponenziale di tutti gli elementi del vettore z. Si può quindi esprimere il peso associato alla chiave i-esima nello step t del Decoder come $E_{t,i} = \frac{e^{e_{q_t,k_i}}}{\sum_{i=1}^{n} e^{e_{q_t,k_i}}}$ anche chiamato attention weight (peso di attenzione) della chiave i-esima k_i, con n il numero di chiavi totali.

6) Somma pesata: una volta calcolati i pesi di attenzione attraverso la funzione softmax, questi pesi vengono applicati ai valori corrispondenti, creando una somma pesata. Questo passaggio permette di aggregare le informazioni rilevanti dell'input in base alla loro importanza determinata nella fase di attenzione. Immaginando di avere un insieme di valori associati a chiavi specifiche che rappresentano informazioni diverse all'interno dell'input, questi valori possono consistere in parole, rappresentazioni vettoriali o caratteristiche di un'immagine. I pesi di attenzione attribuiti a queste chiavi indicano quanto rilevanti tali valori sono rispetto alla query attuale. La somma pesata si ottiene moltiplicando i valori per i loro rispettivi pesi di attenzione e sommando il risultato di questi prodotti. Calcolando quindi il context vector per il passo temporale t del Decoder si ha: $C_t = \sum_{i=1}^{n} E_{t,i} V_i$, dove V_i rappresenta il valore (value) i-esimo, corrispondente allo stato nascosto h_i e n al numero di chiavi/valori totali.

7) Integrazione con il modello: il context vector viene integrato con lo stato nascosto attuale del Decoder, il che consente al modello di arricchire la sua comprensione corrente con le informazioni rilevanti catturate nel context vector. Una semplice operazione per combinare i due vettori può consistere nella concatenazione. Avendo il context vector $C_t \in \mathbb{R}^{d_1}$ e lo stato nascosto dell'Encoder $s_t \in \mathbb{R}^{d_2}$, si possono concatenare ottenendo $[C_t, s_t] \in \mathbb{R}^{d_1+d_2}$.

I passaggi dal numero due al numero sette vengono ripetuti per ogni iterazione del modello, permettendo al meccanismo dell'attenzione di porre attenzione dinamicamente su diverse parti della sequenza di input [19]. Nel processo di calcolare l'attenzione con il prodotto vettoriale si è assunto di avere la stessa dimensione per gli stati nascosti dell'Encoder e del Decoder, tale che $d_1 = d_2$.
Si può allora considerare un tipo di attenzione chiamata multiplicative attention, definita come segue: $e_{t,i} = s_t^T W h_i$, dove t rappresenta l'istanza del Decoder e i lo stato i-esimo dell'Encoder, mentre $W \in \mathbb{R}^{d_2 x d_1}$ è una matrice di parametri addestrabile, che permette al modello di imparare a quali parti dello stato s_t e h_t porre attenzione mentre si calcolano gli attention weigths. Tuttavia, la matrice W può risultare troppo grande, essendo di una certa complessità computazionale allora si può pensare di cambiare approccio, e utilizzarne un altro chiamato reduced rank multiplicative attention. Ciò consente di ridurre la complessità computazionale e i requisiti di memoria, sempre mantenendo una buona capacità di apprendimento delle relazioni tra query e chiavi. L'elemento $e_{t,i}$, utilizzando la reduced rank multiplicative attention, si può calcolare come segue: $e_{t,i} = s_t^T (U^T V) h_i = (U s_t)^T (V h_i)$, dove la differenza rispetto alla multiplicative attention consiste nel suddividere la matrice W in due matrici più piccole U e V, dove $U \in \mathbb{R}^{k x d_2}$ e $V \in \mathbb{R}^{k x d_1}$, $k \ll d_1, d_2$ [9].

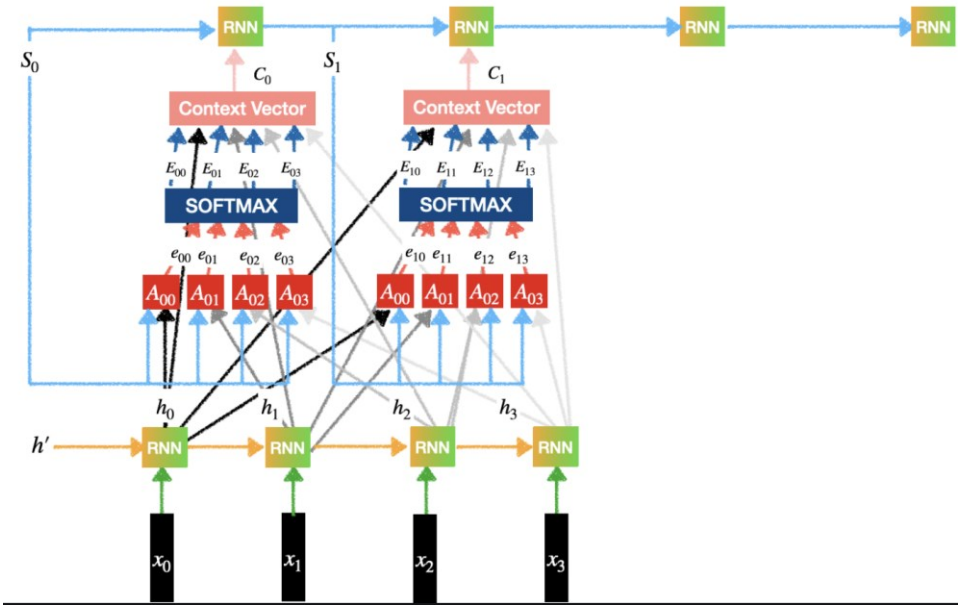


Figura 2.3 – Immagine di un modello Encoder/Decoder con attenzione [19]

2.3 Analisi delle prestazioni

Prima di fare un confronto con le prestazioni tra un modello dotato di attenzione e un classico modello Encoder/Decoder, si introduce BLEU (bilingual evaluation understudy) [7], un algoritmo utilizzato per valutare la qualità del testo tradotto da una lingua a un'altra mediante un sistema di traduzione automatica. Questo algoritmo valuta la qualità in base alla corrispondenza tra l'output del sistema di traduzione automatica e quello di una traduzione umana professionale. I punteggi vengono calcolati per singoli segmenti, generalmente frasi, confrontandoli con un insieme di traduzioni di riferimento di buona qualità. Questi punteggi vengono quindi aggregati insieme per ottenere una stima complessiva della qualità della traduzione. Il risultato ottenuto con BLEU è sempre un numero compreso tra zero e uno. Questo valore indica quanto il testo candidato sia simile ai testi di riferimento, con valori più vicini a 1 che rappresentano testi più simili.

I modelli di cui verranno ora confrontate le prestazioni sono il modello Encoder/Decoder originale di Cho et al. (2014) [20] (RNNencdec) e il modello dotato di attenzione di Bahdanau et al. (2014) [12] (RNNsearch). Nel confronto tra i due modelli sono state eseguite una serie di traduzioni da inglese a francese; le procedure di addestramento sono le stesse e viene utilizzato lo stesso dataset per entrambi i modelli. Ogni modello viene addestrato due volte: la prima con frasi di lunghezza massima pari a 30 parole (RNNencdec-30, RNNsearch-30) e la seconda con frasi di lunghezza massima pari a 50 parole (RNNencdec-50, RnNsearch-50). L'Encoder e il Decoder hanno mille unità nascoste ciascuno. Viene utilizzato l'algoritmo minibatch stochastic gradient descent (SGD) per addestrare i modelli. Ogni aggiornamento di SGD è calcolato usando un minibatch di 80 frasi. Una volta addestrato il modello, si cerca una traduzione che massimizza la probabilità condizionale [12].

2.3.1 Risultati

La figura 2.4 mostra i rapporti delle prestazioni dei vari modelli precedentemente elencati.

Model	All	No UNK°
RNNencdec-30	13.93	24.19
RNNsearch-30	21.50	31.44
RNNencdec-50	17.82	26.71
RNNsearch-50	26.75	34.16
RNNsearch-50*	28.45	36.15
Moses	33.30	35.63

Figura 2.4 – Punteggi BLEU dei vari modelli [12]

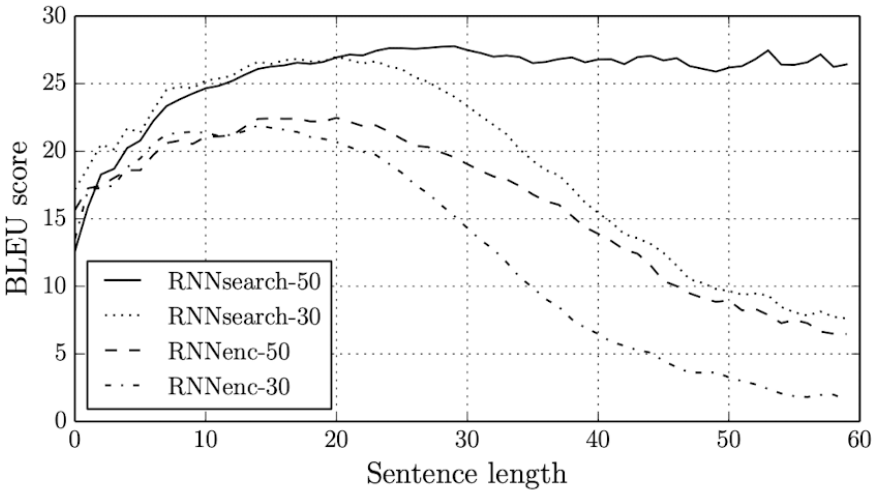


Figura 2.5 – Grafico del punteggio BLEU, facendo variare la lunghezza della frase [12]

Nella figura sono visualizzati i punteggi BLEU dei modelli addestrati sul set di test. La seconda e terza colonna mostrano rispettivamente i punteggi su tutte le frasi con e senza parole sconosciute (segnate come UNK). Moses è un sistema utilizzato specificatamente per la traduzione, e quindi viene utilizzato come punto di riferimento per gli altri modelli; si può osservare come la performance di RNNsearch ci si avvicini sostanzialmente di più rispetto al tradizionale RNNencdec quando si considera il caso dove non vengono utilizzate parole sconosciute. Dalla figura 2.5 si può osservare come RNNsearch-30 e in particolare RNNsearch-50 siano più robusti una volta che si utilizzano frasi più lunghe. In particolare, RNNsearch-50 non mostra diminuzioni di performance anche con l'utilizzo di frasi di lunghezza maggiore o uguale a 50 parole. RNNencdec, costretto all'utilizzo di un vettore di contesto di lunghezza fissa, cala rapidamente di prestazioni aumentando la lunghezza nelle frasi di test.

2.3.2 Tabella degli allineamenti

La tabella degli allineamenti è un'entità che visualizza l'associazione tra le parole in testi tradotti. Indica quale parola (o serie di parole) nella lingua di partenza corrisponde alla parola (o serie di parole) nella lingua di traduzione. Questa tabella viene generata automaticamente semplicemente visualizzando i pesi di attenzione, in tal modo è possibile capire quali posizioni nella frase di origine sono state considerate più importanti nella generazione della parola target. Si può osservare come la maggior parte degli allineamenti tra le parole inglesi e quelle francesi sia monotono, ovvero le parole nella frase di origine sono tradotte in modo sequenziale nella frase tradotta, con pesi significativi lungo la diagonale di ciascuna matrice. Tuttavia, alcuni allineamenti risultano non banali e non monotoni, ad esempio aggettivi e sostantivi sono tipicamente ordinati diversamente fra le due lingue, come si può notare dalla figura 2.6 (a). Il tipo di allineamento è soft, cioè i pesi vengono attribuiti a ciascuna parola della sequenza di input in base alla sua rilevanza rispetto alla sequenza di output, mentre l'allineamento hard farebbe una mappatura rigida tra ogni parola di input e output, non tenendo traccia del contesto della traduzione. Un altro beneficio degli allineamenti soft è la capacità di gestire frasi di origine e destinazione di lunghezze diverse.

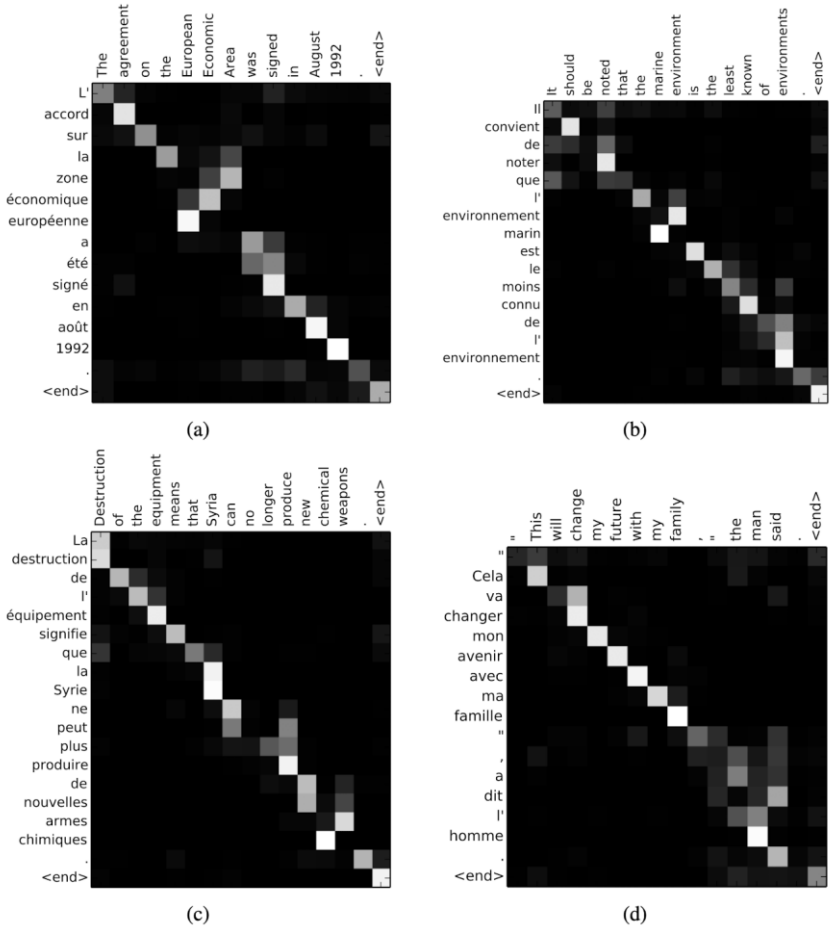


Figura 2.6 – L'asse x e y di ogni figura corrispondono alle parole della lingua sorgente e destinazione rispettivamente. Ogni pixel mostra i pesi α_{ij} della parola j-esima inglese rispetto alla parola i-esima francese, con la scala dove il pixel nero corrisponde a zero (corrispondenza minima) e il bianco a uno (corrispondenza massima) [12]

2.4 Limitazioni del meccanismo dell'attenzione

Fino a questo momento, sono stati utilizzati modelli che fanno uso di architetture come RNN, LSTM, GRU, ecc., che processano i dati in maniera sequenziale, ma come è stato possibile osservare dal meccanismo dell'attenzione, l'ordine lineare dei dati ostacola l'apprendimento per sequenze lunghe. Se si volesse, per ipotesi, mettere a confronto un'istanza di stato dell'Encoder h_i con un'altra istanza h_{i+100}, bisognerebbe "srotolare" il modello 100 volte, e una volta superata una certa soglia di lunghezza, il context vector non è più in grado di ricordare che relazione esiste tra lo stato h_i con lo stato h_{i+100} come si ricorderebbe la relazione tra lo stato h_i e uno più vicino, come h_{i+1}. In generale, due stati distanti d per interagire richiedono $O(d)$ passi.
Da qui nasce l'idea che si possa procedere in maniera analoga al meccanismo dell'attenzione, ma al posto di saltare dal Decoder all'Encoder per calcolare i pesi di attenzione, si calcolano questi pesi direttamente sull'Encoder stesso (e nello stesso modo si può operare sul Decoder) [9]. Questo meccanismo prende il nome di self-attention, ed è stato introdotto per la prima volta da Vaswani et al. (2017) [5]. La self-attention serve per catturare le relazioni tra i dati all'interno della stessa sequenza: questo processo consente al modello di comprendere le dipendenze e le interazioni in essi in maniera più efficiente rispetto ai modelli sequenziali come le reti RNN. Questo tipo di operazione è parallelizzabile, non dovendo dipendere dagli stati precedenti e successivi: lo stato h_i può calcolare la self-attention su sé stesso senza dover attendere che essa sia stata computata dagli stati precedenti e successivi; la distanza di interazione massima diventa quindi $O(1)$. Inoltre, la self-attention può essere riproposta attraverso più layer, dove ogni elemento di un layer dovrà avere accesso a tutti gli elementi del layer precedente prima di poter calcolare la self-attention.

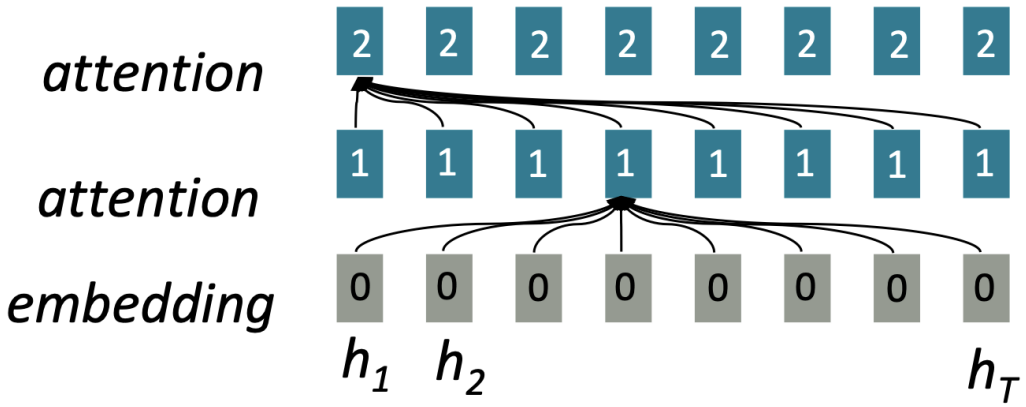


Figura 2.7 – Immagine raffigurante più strati di self-attention sovrapposti [9]

Il meccanismo dell'attenzione opera utilizzando queries, keys e values. Nella self-attention queste tre vengono prese dalla stessa sorgente: se l'output del layer precedente è x_i, allora si può considerare $v_i = k_i = q_i = x_i$. Si calcola un punteggio in maniera analoga all'attenzione, facendo un'operazione che coinvolge la query q_i con il key k_j tale che $e_{ij} = g(q_i, k_j)$. Per esempio, nel caso del prodotto scalare si ottiene: $e_{ij} = q_i^T k_j$. Successivamente si calcolano i

pesi, sempre in analogo con l'attenzione attraverso la funzione softmax: $\alpha_{ij} = \frac{e^{e_{ij}}}{\sum_j e^{e_{ij}}}$ e infine viene computata la media pesata per ogni query i: $output_i = \sum_j \alpha_{ij}\, v_j$.
Dalla self-attention però, emergono nuove problematiche legate al concetto di posizione, linearità e utilizzo all'interno del Decoder. Nel capitolo successivo dedicato ai Transformer, tali problematiche verranno analizzate più a fondo e verrà presentata una soluzione per affrontare questi punti critici [9].

3 Transformer

Il modello Transformer viene costruito sulla base dei modelli Encoder/Decoder con attenzione. Esso rappresenta un salto sostanziale rispetto alle architetture precedenti e i modelli successori ne faranno grande utilizzo come blocco fondamentale.
I Transformer sono utilizzati per vari compiti, come NLP e computer vision, ed è la base di modelli come GPT (Generative Pre-trained Transformer) e BERT (Bidirectional Encoder Representation from Transformers). Il capitolo inizierà con una introduzione ai Transformer, spiegando più a fondo perché il modello ha visto un ampio utilizzo e successo, per poi introdurre dei concetti fondamentali come lo strato di embedding e la codifica posizionale, usati per processare le informazioni che vengono date in input al Transformer. La parte seguente mostrerà in dettaglio come vengono gestiti i vettori di input passati come output della parte precedente, e come l'Encoder comunichi con il Decoder. La parte finale è dedicata all'analisi delle prestazioni di un modello facente uso di questa architettura, confrontato con i migliori modelli introdotti prima di quest'ultimo.

3.1 Introduzione ai Transformer

I Transformer sono un modello basato sull'architettura Encoder/Decoder che utilizza il meccanismo dell'attenzione e abbandona il concetto della ricorrenza. Ideati da Vaswani et al. (2017) [5], i Transformer rappresentano una nuova direzione nel campo dei modelli neurali, poiché consentono di considerare simultaneamente l'intera sequenza di input durante l'elaborazione, a differenza degli approcci precedenti che si basavano sull'approccio delle reti ricorrenti. La loro architettura è composta da due componenti principali: l'Encoder e il Decoder, a loro volta composti da diverse unità di attenzione e reti feedforward che verranno poi spiegate in dettaglio. Grazie all'utilizzo della self-attention, i Transformer possono catturare le relazioni a lungo raggio nelle sequenze, garantendo una elevata capacità di modellazione delle dipendenze tra le parti della sequenza stessa. Inoltre, le unità del modello fanno grande uso del parallelismo per elaborare l'input e generare l'output, garantendo una elevata scalabilità.
In maniera superficiale, il funzionamento del Transformer inizia dall'Encoder, a cui viene passata una sequenza di dati in input; questa sequenza però viene processata parallelamente e non sequenzialmente, a differenza delle architetture precedenti [23]. Nella fase successiva, la sequenza di input viene convertita in una serie di vettori di numeri attraverso lo strato di embedding, con lo scopo di catturare informazioni importanti per i singoli elementi di input. Nel contesto NLP, questi elementi sarebbero rappresentabili da parole, e l'embedding avrebbe lo scopo di rappresentare nel vettore associato le informazioni semantiche e contestuali in base al contesto [31]. Lavorando parallelamente, viene perso il concetto di posizione degli elementi di input, intrinseco ai modelli sequenziali, e per questo motivo lo strato successivo a quello di embedding aggiunge una codifica posizionale ai vettori risultanti. In pratica, vengono sommati dei vettori posizionali ai vettori di embedding, fornendo al modello informazioni sulla sequenza e sulla posizione dei token [25]. Lo strato successivo applica la self-attention sui vettori, con la differenza che vengono utilizzate più teste o "heads", dove ogni testa rappresenta uno strato di self-attention con pesi diversi dagli altri strati, focalizzandosi su diverse parti dell'input in modo da catturare relazioni diverse nello stesso vettore. Segue uno strato di addizione e

normalizzazione (add & norm): l'addizione consente di aggiungere l'output del livello precedente (per esempio il self-attention layer) all'input originale della rete per quella data fase, questo meccanismo è noto come "skip connection", e aiuta ad evitare il problema del vanishing gradient. Il "norm" serve a normalizzare il risultato dell'addizione residuale, in modo da ridimensionale l'output per avere una distribuzione stabile negli output. In generale, lo strato add & norm viene dopo ogni strato principale all'interno del modello. L'ultima parte importante dell'Encoder è un feedforward network, che agisce come uno strato di elaborazione per eseguire trasformazioni lineari seguite da funzioni di attivazione; la sua funzione è quella di introdurre una maggiore complessità e capacità di apprendimento. Come appena specificato, questo strato viene seguito a sua volta da uno di add & norm. Da qui viene prodotto l'output dell'Encoder, che conserva informazioni importanti sull'input e che verrà utilizzato nel Decoder per iniziare il processo di generazione di una sequenza di output. Il Decoder processa l'input nella stessa maniera dell'Encoder: attraverso un embedding layer con l'aggiunta di codifica posizionale. Viene successivamente applicata una variante della self-attention, chiamata masked self-attention, usata per garantire che ciascun elemento nella sequenza abbia accesso solo agli elementi precedenti. Questo vincolo previene che il modello "guardi" gli elementi futuri durante il processo di generazione. Dopo aver passato l'output attraverso un add & norm layer, si entra in uno strato di attenzione, che fa uso dell'output dell'Encoder per calcolare l'attenzione pesata rispetto alle rappresentazioni di input, consentendo al Decoder di concentrarsi su parti specifiche della sequenza iniziale durante la generazione di un output. Gli strati successivi consistono in un add & norm, un feedforward network e un altro add & norm, con lo scopo di introdurre non linearità e normalizzare gli output. Il penultimo strato è un linear layer, con il solo scopo di convertire l'output ottenuto in un formato adatto alla predizione, mentre l'ultimo strato è uno di softmax, atto alla computazione delle probabilità dei vari output possibili presenti nel modello [9].

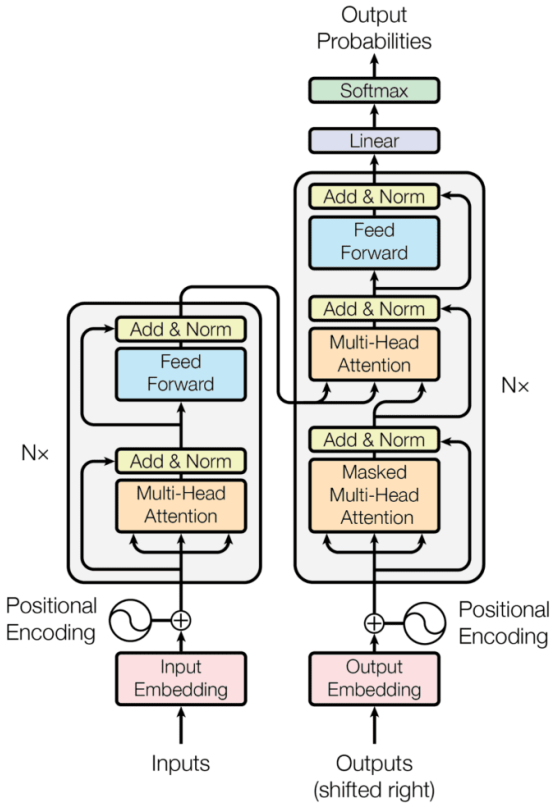


Figura 3.1 – Immagine del modello Transformer [5]

3.2 Embedding layer

Lo strato di embedding viene usato per convertire le rappresentazioni numeriche degli input in vettori densi di dimensioni fisse, chiamati embedding. Questi permettono al modello di catturare le relazioni e le similarità tra le diverse categorie e classi per rappresentare le relazioni nei dati di input in uno spazio vettoriale. Nel contesto NLP, per esempio, gli embedding rappresentano le relazioni semantiche e sintattiche tra le parole o i token. Le rappresentazioni vengono imparate dal modello attraverso un processo iterativo di addestramento, dove i vettori di embedding vengono modificati in base al compito da eseguire, come massimizzare l'accuratezza di una predizione. I vettori di embedding vengono inizializzati con numeri casuali e vengono aggiornati durante l'addestramento. Oltre al vantaggio di utilizzare vettori di grandezza fissata, questo strato riduce la dimensionalità dei dati di input, facilitando i compiti come la classificazione o la regressione, poiché viene semplificato il processo decisionale [11].

Per visualizzarne meglio il significato, viene fornito un esempio: si utilizza un metodo come l'hot-one encoding, una tecnica usata per rappresentare variabili categoriche come vettori binari. Ogni categoria viene rappresentata da un vettore di lunghezza pari al numero totale di categorie, con tutti gli elementi del vettore impostati a zero tranne uno, corrispondente alla categoria specifica. Se si prendessero 37.000 libri da Wikipedia, rappresentarli richiederebbe un vettore di 37.000 elementi per ognuno di questi libri, dove 36.999 di questi elementi corrisponderebbero a 0 e uno solo a 1, indicando il libro scelto. Questo creerebbe una matrice sparsa di elementi, rendendo l'addestramento di un qualsiasi modello sostanzialmente più lento [8]. Inoltre, non è possibile in questo modo rappresentare le similarità di un libro con un altro, per esempio il genere letterario o il numero di pagine. Utilizzando un embedding, questi vettori vengono rimappati in vettori dimensionalmente più piccoli e densi, dove ogni elemento rappresenta una categoria più o meno associabile al libro scelto. Si può pensare come un punteggio grande rappresenti una maggiore similarità del libro con la categoria e viceversa per un punteggio basso. Si risolve così il problema della rappresentazione delle similarità tra i vari elementi di input, in modo che elementi simili abbiano punteggi simili per la categoria i-esima. Inoltre, esiste la possibilità di mappare questi vettori in una dimensione inferiore (solitamente pari a due o tre), dando la possibilità di visualizzare le similitudini tra gli elementi, essendo mappati in uno spazio vettoriale comprensibile all'occhio umano. Per esempio, in figura 3.2, è possibile visualizzare il raggruppamento dei libri con una determinata categoria, come il genere letterario. I libri con lo stesso genere letterario sono rappresentati dallo stesso colore, e in quanto ci si può aspettare che siano più o meno raggruppabili in una determinata regione [35].

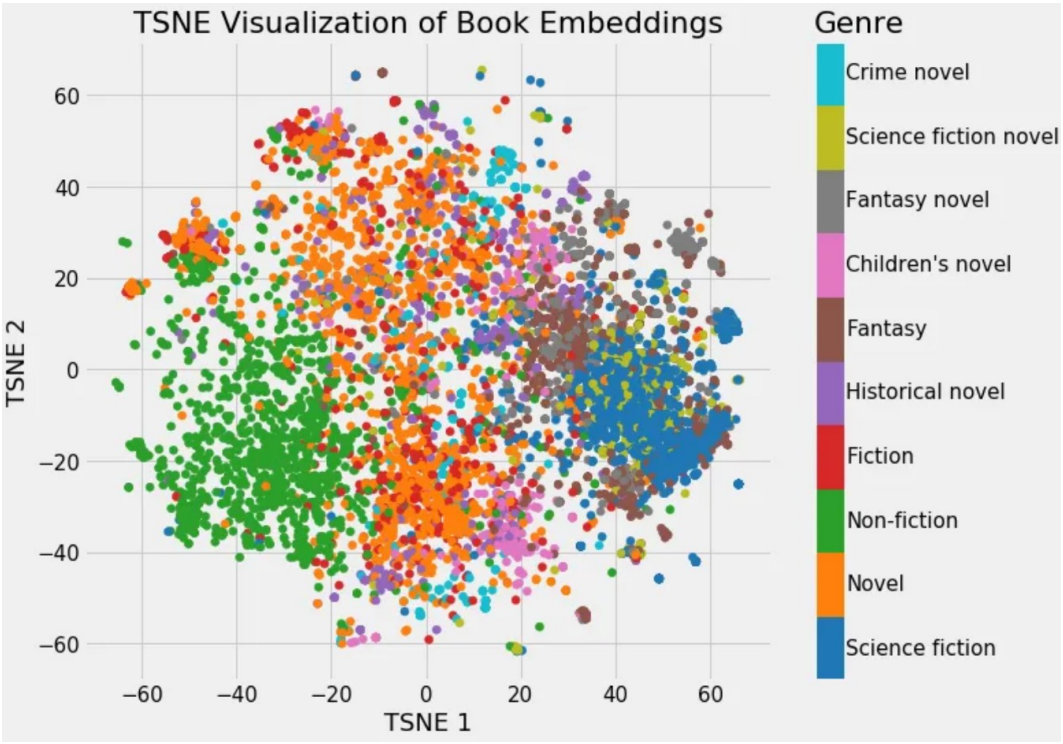


Figura 3.2 – Rappresentazione grafica degli embedding dei libri [35]

3.3 Codifica posizionale

La codifica posizionale descrive la posizione di un'entità in una sequenza, così che ad ogni posizione sia assegnata una rappresentazione unica. Nei modelli come quello del Transformer, non è presente nessuna componente ricorrente che tenga traccia dell'ordine dei dati, e in quanto è necessario introdurre informazioni sulla loro posizione direttamente dall'input. Se si rappresenta l'indice della sequenza di input come un vettore, tale che $p_i \in \mathbb{R}^d$, $i \in \{1,2,\dots,T\}$, e il vettore senza codifica posizionale in posizione i-esima è $v_i{}'$, allora il vettore con l'aggiunta della codifica posizionale si ottiene come: $v_i = v_i' + p_i, \forall i$; questo implica che la lunghezza di p_i sia uguale a quella di v_i'.
Un metodo spesso utilizzato per determinare p_i è attraverso la concatenazione di sinusoidi, che utilizza funzioni sinusoidali e cosinusoidali per codificare la posizione dei token. La formula usata per la codifica posizionale è:

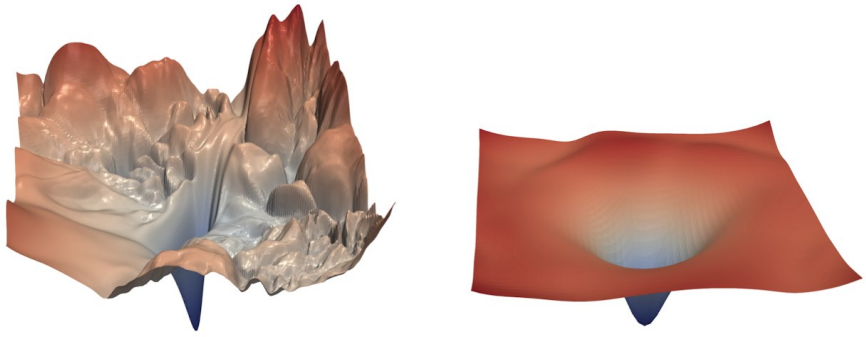
$$PE_{(pos,2i)} = sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE_{(pos,2i+1)} = cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

Dove pos rappresenta la posizione del token nella sequenza, i la posizione i-esima all'interno del token e d_{model} (chiamata anche d) la dimensione dei vettori di embedding. Quindi, per le posizioni pari verrà utilizzata la prima formula e la seconda per quelle dispari. Le codifiche posizionali vengono riutilizzate per ogni sequenza di input, di conseguenza queste rimarranno sempre le stesse per ogni determinata posizione [25].

3.4 Add & Norm layer con skip connection

Il livello di Add & Norm è parte dell'architettura delle skip connection e aiuta a mitigare il problema di vanishing/exploding gradient. Questo layer viene posizionato all'interno di ciascun blocco (quindi dopo ogni strato di attenzione e di feedforward). Le skip connection sono connessioni dirette che permettono di saltare uno o più strati all'interno di una rete neurale durante la trasmissione dei dati. In tal modo si preserva l'informazione originale dall'input direttamente all'output e la panoramica della funzione di loss diventa considerabilmente più "liscia" [13].



(a) without skip connections (b) with skip connections

Figura 3.3 – Superfici di loss che a) non fanno uso delle skip connection b) fanno uso delle skip connection [13]

Nel contesto dei Transformer, le skip connection sono implementate come connessioni residue, ovvero l'input originale di un blocco viene sommato (Add) all'output del blocco stesso. Infine, la parte di Norm si riferisce alla normalizzazione dei dati, in modo che abbiano una distribuzione dei valori più stabile e uniforme. Matematicamente, se $x \in \mathbb{R}^d$ è un vettore nel modello e $\mu = \sum_{j=1}^{d} x_j \in \mathbb{R}$ rappresenta la media, allora $\sigma = \sqrt{\frac{1}{d}\sum_{j=1}^{d}(x_j - \mu)^2} \in \mathbb{R}$ è la deviazione standard. Inoltre vengono (opzionalmente) impiegati due vettori ulteriori $\gamma \in \mathbb{R}^d$, $\beta \in \mathbb{R}^d$ chiamati rispettivamente "gain" e "bias". Da qui, la normalizzazione si calcola come $output = \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}}\,\gamma + \beta$, dove ϵ rappresenta un valore piccolo aggiunto alla varianza per evitare il rischio che il denominatore si avvicini troppo allo zero [9].

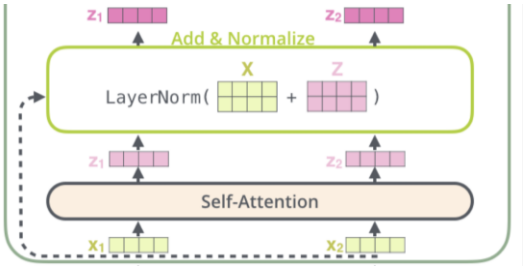


Figura 3.4 – Immagine indicante lo strato di Add & Norm con skip connection (linea tratteggiata) [17]

3.5 Parte dell'Encoder e Decoder

Dopo che l'input è stato trasformato attraverso lo strato di embedding con l'aggiunta della codifica posizionale, il modello Transformer passa per il multi-headed self-attention layer. In questa fase il modello si concentra simultaneamente su aspetti differenti della sequenza di input attraverso varie teste di attenzione parallele, ognuna delle quali crea e gestisce un'attività di attenzione indipendente dalle altre. Ogni testa opera eseguendo tre trasformazioni separate dell'output dello strato di embedding con codifica posizionale. Se tale output è rappresentabile da $x_1, \dots, x_T$; $x_i \in \mathbb{R}^d$, allora vengono creati tre vettori analogamente al meccanismo dell'attenzione chiamati keys, queries e values:

$$k_i = Kx_i, K \in \mathbb{R}^{dxd}$$
$$q_i = Qx_i, Q \in \mathbb{R}^{dxd}$$
$$v_i = Vx_i, V \in \mathbb{R}^{dxd}$$

Dove K, Q e V sono rispettivamente le matrici key, query e value, che permettono di utilizzare diverse caratteristiche o aspetti dei vettori di input x in ciascuno dei tre ruoli chiave. Questo processo consente di analizzare varie proprietà dei vettori di input in modi specifici per la generazione dell'output. La seconda fase consiste nel calcolo di un punteggio tra i vettori k_i e i vettori q_j: questo in generale avviene attraverso l'utilizzo di una funzione f tale che: $score(k_i, q_j) = f(k_i, q_j)$. Nell'articolo di Vaswani et al. (2017) [5] viene utilizzato il prodotto scalare scalato: $score(k_i, q_j) = \frac{k_i q_j}{\sqrt{d}}$; questo viene ripetuto $\forall i$, $\forall j$. Ad esempio, se si sta calcolando la self-attention per il primo valore della sequenza, allora il primo $score$ viene calcolato usando k_1 e q_1, mentre il secondo utilizzando k_2 e q_1, ecc. La divisione per $\sqrt{d}$ serve per ottenere gradienti più stabili, e mentre esistono altri parametri per raggiungere lo stesso scopo, viene utilizzato questo metodo. Se $X = [x_1, \dots x_T] \in \mathbb{R}^{Txd}$, allora $XK, XQ, XV \in \mathbb{R}^{Txd}$ rappresentano le matrici corrispondenti all'unione di tutti i vettori calcolati precedentemente. Nel passo successivo, viene calcolato il softmax del prodotto tra la matrice XK e XQ, in modo da normalizzare i punteggi, così che siano tutti positivi e la loro somma sia uno: $softmax\left(\frac{XK\ XQ}{\sqrt{d}}\right)$. Questo punteggio determina quanto ogni valore sia rilevante per ogni posizione; chiaramente l'elemento avrà un punteggio maggiore per sé stesso come si è già potuto osservare dalla matrice degli allineamenti, ma spesso sono presenti altri valori sufficientemente rilevanti da tenere in considerazione. L'ultimo passo consiste nel moltiplicare ogni vettore value per questo punteggio: $softmax\left(\frac{XK\ XQ}{\sqrt{d}}\right)XV = output \in \mathbb{R}^{Txd}$. Questi passaggi vengono ripetuti parallelamente per ogni testa del layer, ognuna avente matrici K, Q e V diverse in modo tale da porre attenzione a parti diverse dei dati di input. Se h rappresenta il numero totale di teste del layer, allora $Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{dx\frac{d}{h}}, \ell \in [1, \dots, h]$ sono le varie matrici query, key e value per la testa ℓ, e ogni testa calcola l'attenzione come:

$output_\ell = softmax\left(\frac{XQ_\ell \; XK_\ell}{\sqrt{\frac{d}{h}}}\right) XV_\ell \in \mathbb{R}^{\frac{d}{h}}$. Infine, gli output vengono combinati: $output = Y[output_1, \ldots, output_h]$, $Y \in \mathbb{R}^{dxd}$, dove Y rappresenta una matrice di parametri addestrata con lo scopo di unificare i vari output [9].

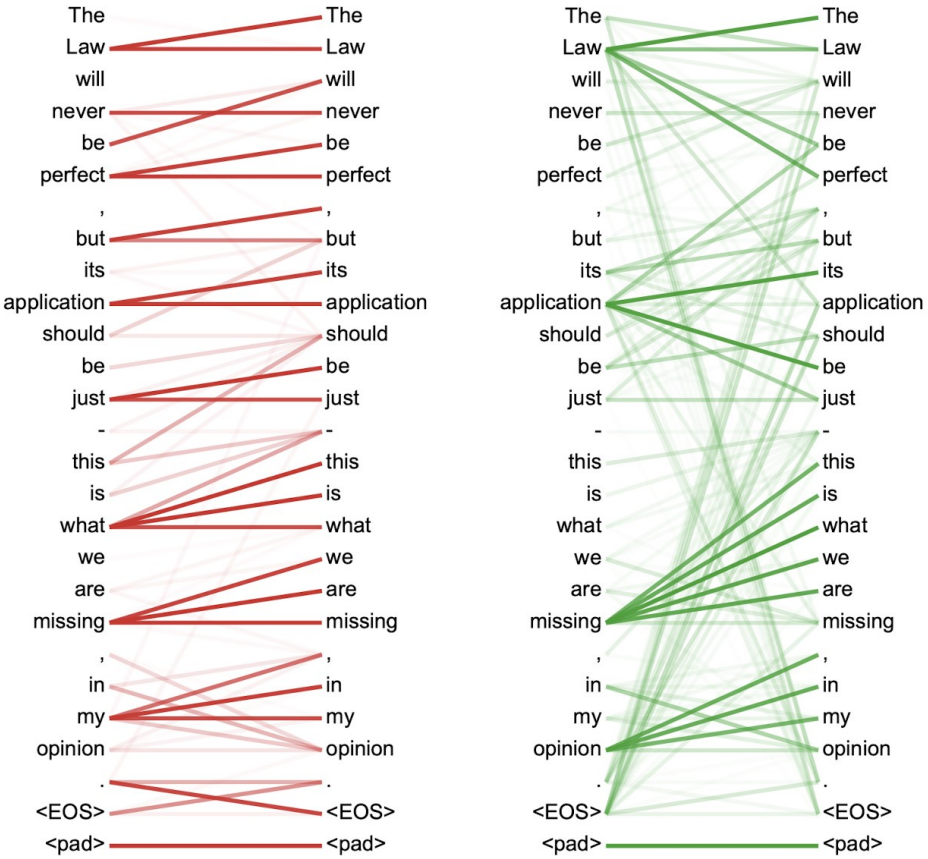


Figura 3.5 – Comportamento di due teste diverse (colore rosso a sinistra e verde a destra) dell'Encoder. Per ogni testa si ha la colonna di sinistra che rappresenta le correlazioni con quella di destra, l'intensità del colore corrisponde alla quantità di importanza assegnata (più intenso corrisponde a una correlazione più significativa). Si può osservare come le due teste abbiano imparato compiti diversi, avendo creato associazioni differenti [5]

Senza considerare lo strato Add & Norm, il livello successivo dell'Encoder è un feedforward layer: questo strato, pur essendo cruciale per l'architettura del Transformer, può essere considerato meno complesso da descrivere in quanto non coinvolge operazioni di attenzione o pesatura sulle relazioni tra i vettori di input. Come già argomentato in precedenza, questo è composto da una combinazione di strati connessi e da funzioni di attivazione per introdurre non linearità. L'output dell'Encoder è poi trasformato in un set di vettori di attenzione key K e value V, che verranno successivamente usati nel Decoder dallo strato di cross-attention, che aiuta il Decoder a porre attenzione su parti appropriate della sequenza di input.

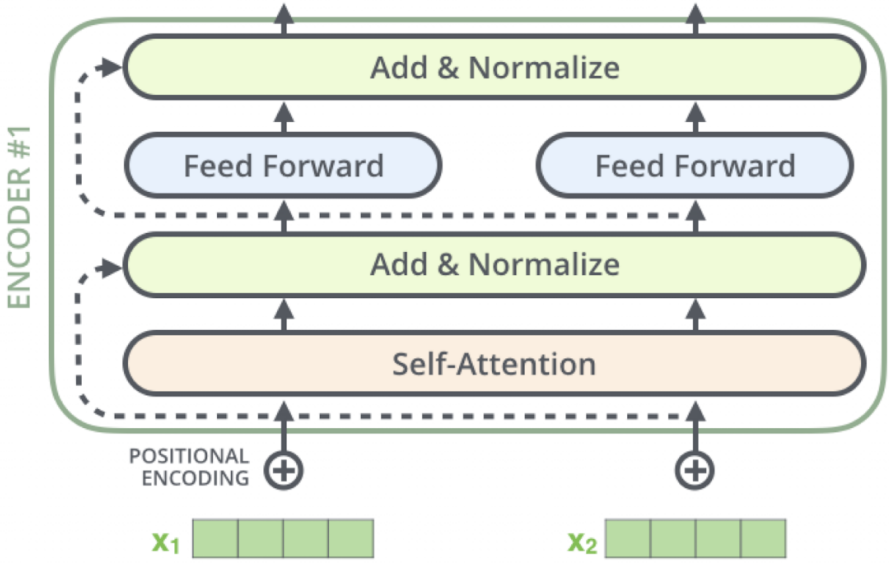


Figura 3.6 – Parte dell'Encoder nel modello Transformer, delimitata dalla linea verde [17]

La parte di input del Decoder ha un funzionamento analogo a quello dell'Encoder, in quanto si ha uno strato di embedding della sequenza dei dati seguito da una codifica posizionale per aggiungere il concetto di posizione nei vari vettori. L'unica differenza che si può far notare consiste nell'input, che viene "spostato a destra" di una posizione perché il primo token che verrà fornito in ingresso del Decoder consiste in un token speciale, solitamente chiamato <EOS> (End Of Sentence) oppure <SOS> (Start Of Sentence), usato come segnalatore di inizio generazione. Il blocco successivo è lo strato di masked multi-head self-attention, che impiega lo stesso comportamento dello strato di multi-head self-attention dell'Encoder, con una differenza che risiede nella gestione delle informazioni temporali durante la fase di decodifica. Viene applicata una maschera che impedisce al modello di vedere le informazioni successive a quella corrente: questo è essenziale durante la generazione di sequenze, poiché garantisce che il modello non possa "guardare avanti" mentre predice il prossimo token. Questa maschera viene applicata sotto forma dei pesi di attenzione come segue:

$e_{ij} = \begin{cases} f(q_i k_j) & ,j < i \\ -\infty & ,\mathrm{j} \geq \mathrm{i} \end{cases}$, mentre la self-attention dell'Encoder calcolava i pesi di attenzione usando solamente $f(q_i k_j), \forall i, j$. Si può visualizzare questo procedimento usando un esempio nel caso NLP, dove il Decoder ha a disposizione solamente le parole già dedotte precedentemente per trovare quella nell'istante temporale corrente, e non quelle future. Ponendo i pesi al di sopra della diagonale della matrice degli allineamenti pari a $-\infty$, si può prevenire che il modello provi a guardare in avanti durante l'addestramento e l'inferenza, sapendo che lo strato di softmax porrà questi pesi pari a zero durante il calcolo delle probabilità.

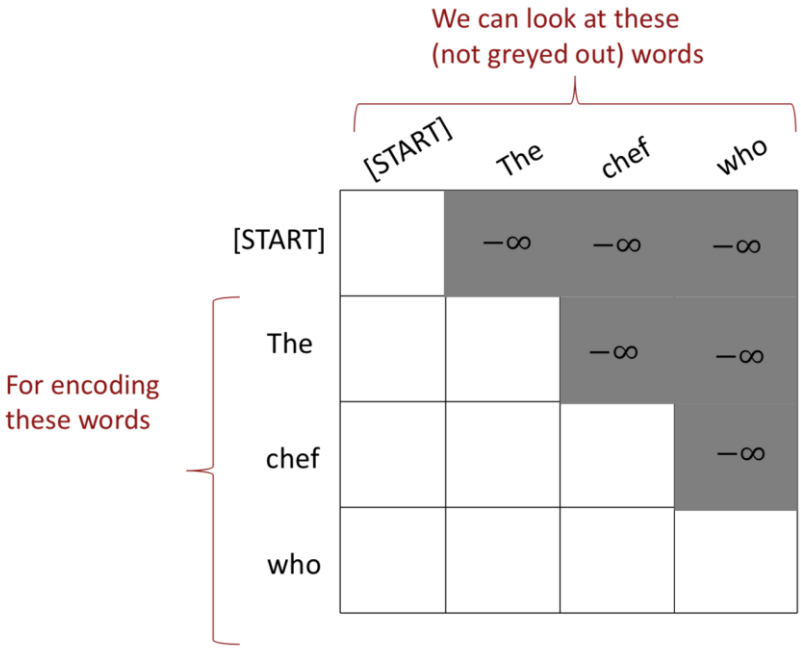


Figura 3.7 – Maschera applicata dallo strato di masked self-attention per prevenire che il Decoder del Transformer guardi le parole future [9]

Il blocco che segue è lo strato di cross-attention, ovvero uno strato che permette al Decoder di comunicare con l'Encoder attraverso l'output finale di quest'ultimo, che viene trasformato con due matrici keys K e values V. I vettori queries sono invece presi dall'output del blocco precedente del Decoder. Se $h_1, \dots, h_T$ sono i vettori di output dell'Encoder, $h_i \in \mathbb{R}^d$, e $z_1, \dots, z_T$ sono i vettori di input del Decoder, $z_i \in \mathbb{R}^d$, allora il vettore key e value i-esimi sono calcolati come: $k_i = Kh_i$, $v_i = Vh_i$, mentre il vettore query generico come: $q_i = Qz_i$, dove K, V, Q sono matrici di parametri. I pesi di attenzione sono calcolati in analogo al meccanismo dell'attenzione, prendendo $H = [h_1, \dots, h_T] \in \mathbb{R}^{Txd}$ come concatenazione dei vettori dell'Encoder e $Z = [z_1, \dots, z_T] \in \mathbb{R}^{Txd}$ come concatenazione dei vettori del Decoder, i punteggi sono definiti come: $output = softmax\big(ZQ(HK)\big)HV \in \mathbb{R}^{Txd}$, dove è stato usato il prodotto scalare tra i vettori ZQ e HK per semplicità, in quanto non è l'unica opzione tra le funzioni disponibili da utilizzare. Inoltre, è importante notare che l'output dell'Encoder non viene ricalcolato per ogni passo temporale del Decoder, essendo la sequenza di input sempre la stessa e non dovendo ricevere modifiche [9].

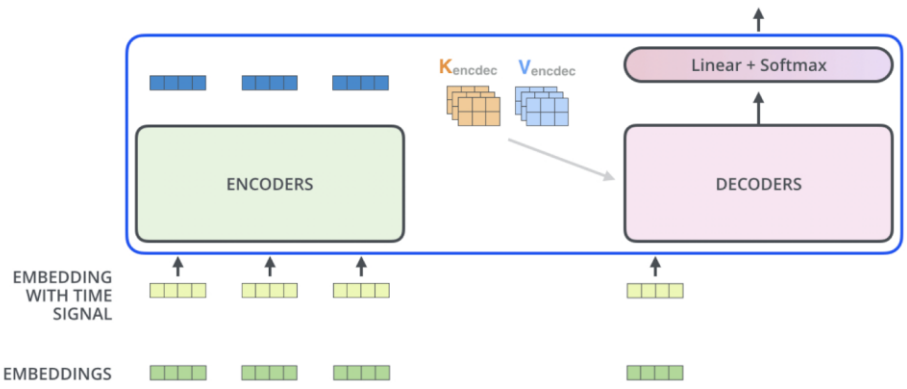


Figura 3.8 – Immagine rappresentante l'intero modello Transformer, dove si mettono in evidenza le matrici K e V usate nello strato di cross-attention del Decoder [17]

L’ultimo strato principale appartenente al blocco del Decoder è uno di feedforward, e avendo lo stesso comportamento di quello presente nell’Encoder, si evita di fornire una spiegazione con lo scopo di evitare la ripetizione.

3.6 Linear e softmax layer

Il linear layer ha la funzione principale di ridimensionare l’output delle operazioni del blocco precedente in uno spazio dimensionale coerente con l’output desiderato. Questo strato è costituito da una rete neurale completamente connessa che proietta i vettori del Decoder in un vettore chiamato “logits”. Per avere un esempio concreto, si pensi come in un modello atto alla generazione di testo il Decoder possegga un vocabolario con 10.000 parole (output vocabulary), questo renderebbe il vettore logits lungo 10.000 celle, con ogni cella corrispondente al punteggio di una parola. Lo scopo dello strato di softmax in questo punto è di convertire i punteggi del vettore logits in modo che siano tutti positivi e si sommino a uno. In seguito, viene selezionato il token con la probabilità più alta (esistono più algoritmi per la selezione, come beam search [17]), questo token sarà quindi restituito in output, per poi essere utilizzato come input del Decoder allo step successivo. Questo avviene indefinitamente finché il Decoder non genera un token speciale come <EOS> che segnala al modello di interrompere il processo di generazione di output [17].

3.7 Analisi delle prestazioni

Per analizzare le prestazioni, è stato eseguito un benchmark usando un dataset inglese-tedesco WMT 2014, contenente circa 4.5 milioni di coppie di frasi. Le frasi sono state codificate usando la tecnica di encoding byte-pair, la quale genera un vocabolario di circa 37.000 token tra la lingua sorgente e quella destinazione. Nel secondo, per l’inglese-francese, è stato utilizzato un dataset significativamente più grande, sempre dal WMT 2014, contenente 36 milioni di frasi. In questo caso, i token sono stati suddivisi in un vocabolario di circa 32.000 parole. Le coppie di frasi sono state organizzate in batch considerando approssimativamente la lunghezza della sequenza. Ogni batch di addestramento conteneva coppie di frasi che, complessivamente, racchiudevano circa 25.000 token di sorgente e 25.000 token di destinazione. I modelli sono stati addestrati su una singola macchina dotata di 8 GPU NVIDIA P100. Per i modelli base, usando gli iperparametri menzionati nel documento, ogni passo dell’addestramento ha richiesto circa 0,4 secondi. Questi modelli base sono stati addestrati per un totale di 100.000 passi, equivalente a 12 ore di tempo complessivo di addestramento. Per i modelli più grandi, il tempo richiesto per ogni addestramento è stato di 1 secondo, per poi essere addestrati per un totale di 300.000 passi, che corrispondono a circa 3,5 giorni di tempo.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

Figura 3.9 – Immagine raffigurante i punteggi BLEU e i costi di addestramento dei vari modelli per le lingue EN-DE ed EN-FR [5]

Le due sotto-colonne di BLEU corrispondono ai punteggi ottenuti usando le lingue inglese-tedesco e inglese-francese, mentre le due sotto-colonne più a destra mostrano il costo di addestramento in FLOPs (floating point operations), ovvero il numero di operazioni in virgola mobile eseguite. Nel compito di traduzione dall'inglese al tedesco, il modello Transformer grande (big) ha superato i migliori modelli precedentemente progettati di oltre 2 punti BLEU, stabilendo un nuovo punteggio di riferimento pari a 28,4. Anche il modello base ha superato i risultati di tutti i modelli precedentemente pubblicati, pur avendo un costo di addestramento notevolmente inferiore. Nel compito di traduzione dall'inglese al francese il modello Transformer (big) ha raggiunto un punteggio BLEU pari a 41,8, superando i punteggi di tutti i modelli precedentemente pubblicati, ma a una frazione del costo di addestramento.
Al fine di valutare l'importanza dei parametri nel modello, la figura 3.10 mostra le variazioni del punteggio BLEU, della perplexity PPL (indica la coerenza e accuratezza nelle previsioni del modello, un punteggio basso è migliore) e del numero di parametri totali modificando i valori dei parametri ad esso associati. I cambiamenti vengono misurati utilizzando un modello atto alla traduzione inglese/tedesco sul set di sviluppo newtest2013. Nella riga denominata "base" vengono riportati i parametri del modello; nelle righe successive i parametri che non subiscono modifiche non hanno un valore associato. Nelle righe (A) si variano il numero di teste e le dimensioni dei vettori key e value, facendo rimanere invariata la quantità di computazione. Sebbene l'attenzione a testa singola abbia un punteggio di 0.9 BLEU più bassa dell'impostazione ottimale, si può osservare come il punteggio peggiori anche aumentando eccessivamente il numero di teste. Nelle righe (B) viene diminuita la dimensione del vettore di attenzione key d_k, peggiorando i punteggi BLEU e PPL come ci si poteva aspettare, mentre il numero di parametri cala di conseguenza. Le righe (C) dimostrano come, aumentando le dimensioni dei parametri del modello, anche i punteggi migliorino a scapito del costo di computazione. In particolare, la quinta riga della sezione (C) dimostra che procedere con un approccio a forza bruta, aumentando di molto il numero dei parametri totali (più del doppio del modello ottimale) non porti necessariamente a un aumento significativo dei punteggi (differenza di +0.2 BLEU rispetto al modello ottimale). Le righe (D) invece modificano le percentuali di dropout (si disattivano casualmente dei neuroni durante l'addestramento) e di label smoothing (sostituisce le etichette one-hot in distribuzioni di probabilità più suddivise), aumentandole e rimuovendole completamente; impiegare il dropout aiuta il modello a essere

più sicuro nelle risposte, aumentando anche il punteggio BLEU, mentre il label smoothing può peggiorare il punteggio PPL, siccome il modello diventa meno sicuro, ma migliorare il BLEU. La riga (E) utilizza un modello che sostituisce la codifica posizionale sinusoidale con una codifica di posizione differente; tuttavia, i risultati rimangono praticamente identici. L'ultima riga mostra un ulteriore modello ottimale, ma significativamente più grande rispetto al modello base: l'aumento dei parametri complessivi porta a un incremento della qualità delle risposte.

	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps	PPL (dev)	BLEU (dev)	params $\times 10^6$
base	6	512	2048	8	64	64	0.1	0.1	100K	4.92	25.8	65
(A)				1	512	512				5.29	24.9	
				4	128	128				5.00	25.5	
				16	32	32				4.91	25.8	
				32	16	16				5.01	25.4	
(B)					16					5.16	25.1	58
					32					5.01	25.4	60
(C)	2									6.11	23.7	36
	4									5.19	25.3	50
	8									4.88	25.5	80
		256			32	32				5.75	24.5	28
		1024			128	128				4.66	26.0	168
			1024							5.12	25.4	53
			4096							4.75	26.2	90
(D)							0.0			5.77	24.6	
							0.2			4.95	25.5	
								0.0		4.67	25.3	
								0.2		5.47	25.7	
(E)	positional embedding instead of sinusoids									4.92	25.7	
big	6	1024	4096	16			0.3		300K	**4.33**	**26.4**	213

Figura 3.10 – Variazioni dei punteggi PPL, BLEU e del numero di parametri in base alle caratteristiche del modello utilizzato [5]

4 Conclusioni

Il modello Encoder/Decoder si è rivelato particolarmente efficace per controllare le sequenze dei dati, in modo da poter gestire in maniera indipendente le lunghezze di input rispetto a quelle di output. Entro certi limiti di lunghezza, l'Encoder è capace di raccogliere una buona parte delle informazioni della sequenza in un solo vettore, chiamato context vector, in modo da avere una rappresentazione compatta e facilmente utilizzabile di tutto l'input. Lo scopo del Decoder è quello di utilizzare questa rappresentazione compatta per generare una sequenza di output, di lunghezza indipendente dall'input. Mentre l'encoder processa la sequenza di dati, il decoder rimane non attivo e viceversa. Invece, durante l'addestramento sia l'Encoder che il Decoder sono funzionanti e vengono addestrati contemporaneamente. Durante l'addestramento, i parametri del modello vengono aggiornati opportunamente utilizzando degli algoritmi al fine di ridurre la funzione di loss, in modo da minimizzare la probabilità di errore durante la predizione. Man mano che la dimensione del modello cresce, diventa più difficile controllare tali parametri per via del problema dell'exploding/vanishing gradient. Un'altra limitazione del modello è proprio il context vector, che essendo limitato a una grandezza predefinita non può raccogliere informazioni in maniera estremamente flessibile quando la sequenza dei dati di input raggiunge una determinata lunghezza. Da qui, infatti, viene introdotto il meccanismo dell'attenzione, che mira a risolvere il problema del collo di bottiglia del modello Encoder/Decoder, creando un accesso diretto tra i due in modo che l'informazione che essi si comunicano non passi esclusivamente attraverso lo stesso vettore di dimensioni fisse. Questo risolve non solo le limitazioni introdotte dalla dimensione del context vector, ma anche quelle create dalla spazialità della struttura, poiché ogni elemento del Decoder ha la possibilità di comunicare con qualsiasi elemento dell'Encoder. Per il calcolo dell'attenzione, viene fatto uso di pesi trovati attraverso la funzione di alignment score, che può assumere diverse forme, a seconda delle necessità del modello. Passando poi per uno strato di softmax, questi punteggi vengono convertiti in percentuali con lo scopo di indicare quanto sia correlato un certo elemento del Decoder con uno qualsiasi dell'Encoder. In tale modo, si fa uso della computazione in parallelo che non trova particolare posto in un modello sequenziale come quello Encoder/Decoder. Questo aiuta a ridurre i tempi di predizione e aumenta l'efficienza complessiva del modello, sebbene sia richiesto un hardware capace di tali operazioni parallele. Quello che il meccanismo dell'attenzione non risolve risiede nella struttura di base del modello Encoder/Decoder, ovvero la natura sequenziale sulla quale esso si basa. Processando sequenze di dati in input e output, questi devono essere passati attraverso uno strato alla volta: diventa quindi impossibile parallelizzare tale processo mantenendo la stessa architettura. Per questo motivo è stato successivamente proposto un nuovo modello che fa un estensivo utilizzo del parallelismo attraverso ogni strato richiedente il processing delle sequenze di dati. Il modello prende il nome di Transformer, con lo scopo di rielaborare come vengono processati i dati, senza distogliersi troppo dal modello Encoder/Decoder. Infatti, il modello fa utilizzo di due blocchi principali chiamati anch'essi Encoder e Decoder, aventi la stessa funzione di creare una rappresentazione dell'input e di generare un output utilizzando quest'ultimo, rispettivamente. La differenza principale risiede nella gestione delle sequenze: il meccanismo dell'attenzione viene calcolato in parallelo per ogni elemento. Ciò significa che i vari elementi non sono più legati fra di loro per località, il che introduce un nuovo problema inesistente fino a prima per i modelli precedenti: la perdita del concetto di posizione. Non potendo contare sulla sequenzialità

dei dati, un elemento del Transformer non ha il concetto di posizione insito, e per questo si introduce il concetto di codifica posizionale, in modo da inserire artificialmente una posizione nei vettori appartenenti alla sequenza. Il modello Transformer fa utilizzo dell'attenzione già menzionata, ma anche della self-attention, un meccanismo usato per mettere in relazione gli elementi della sequenza stessa. La parte del Decoder, inoltre, utilizza una versione adattata della self-attention, chiamata masked self-attention, in modo da non tenere in considerazione gli elementi futuri mentre calcola i punteggi, non potendo fare affidamento alle predizioni che deve ancora generare. Questa architettura fa un utilizzo sostanzialmente più estensivo del parallelismo rispetto al modello Encoder/Decoder con attenzione, avendo eliminato la necessità di processare i dati in maniera sequenziale. Nonostante questo, rimane ancora una parte di sequenzialità che non può essere completamente rimossa, dovendo comunque processare tutte le varie attenzioni di uno strato prima di poter passare al prossimo. Inoltre, il parallelismo introdotto può diventare il nuovo collo di bottiglia, siccome il calcolo delle attenzioni deve essere svolto tra tutti gli elementi con tutti gli elementi. Questo problema è stato però in parte risolto da ricerche successive, non mettendo più in relazione tutti gli elementi fra di loro, ma solo in parte, che possa essere locale o casuale [9].

Ringraziamenti

Vorrei ringraziare coloro che hanno reso tutto questo possibile: i miei genitori e la mia famiglia, che mi hanno supportato durante il mio percorso e dato una spinta quando serviva; il mio relatore, che ha dimostrato grande interesse nel mio lavoro seguendomi e consigliandomi durante la stesura; i miei amici, tra cui Maruf, con il quale ho affrontato gli esami più difficili; e Emily, che mi è rimasta sempre accanto e ha letto con pazienza la mia tesi, aiutandomi a riformulare quando necessario.

Bibliografia:

[1] Ajitesh Kumar. *Demystifying Encoder Decoder Architecture & Neural Network*(2023). URL: https://vitalflux.com/encoder-decoder-architecture-neural-network/

[2] american-express. *A Comprehensive Guide to Attention Mechanism in Deep Learning for Everyone* (2023). URL: https://www.analyticsvidhya.com/blog/2019/11/comprehensive-guide-attention-mechanism-deep-learning/

[3] Artem Opperman. *How Loss Function Work in Neural Networks and Deep Learnings* (2022). URL: https://builtin.com/machine-learning/loss-functions#

[4] *Artificial neural network* (2023). URL: https://en.wikipedia.org/wiki/Artificial_neural_network

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. *Attention Is All You Need* (2023). URL: https://arxiv.org/abs/1706.03762

[6] *Attention (machine learning)* (2023). URL: https://en.wikipedia.org/wiki/Attention_(machine_learning)

[7] *BLEU* (2023). URL: https://en.wikipedia.org/wiki/BLEU

[8] Chamanth mvs. *Working of Encoder-Decoder model* (2023). URL: https://medium.com/mlearning-ai/working-of-encoder-decoder-model-db922c2027a6

[9] CS224N: *Natural Language Processing with Deep Learning* (2021). URL: https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1214/

[10] Daniel Jurafsky, James H. Martin. *Machine Translation and Encoder-Decoder Models* (2021). URL: https://web.stanford.edu/~jurafsky/slp3/old_dec21/10.pdf

[11] dremio. *Embedding Layer*. URL: https://www.dremio.com/wiki/embedding-layer/

[12] Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate* (2014). URL: https://arxiv.org/abs/1409.0473

[13] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, Tom Goldstein. *Visualizing the Loss Landscape of Neural Nets* (2018). URL: https://arxiv.org/abs/1712.09913

[14] Ian Goodfellow, Yoshua Bengio, Aaron Courville. *Deep Learning* (2016). URL: https://www.deeplearningbook.org

[15] lya Sutskever, Oriol Vinyals, Quoc V. Le. *Sequence to Sequence Learning with Neural Networks* (2014). URL: https://papers.nips.cc/paper_files/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html

[16] Jason Brownlee. *How to Choose an Activation Function for Deep Learning* (2021). URL: https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/

[17] Jay Alammar. *The Illustrated Transformer* (2018). URL: http://jalammar.github.io/illustrated-transformer/

[18] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, Yoshua Bengio. *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention* (2016). URL: https://arxiv.org/abs/1502.03044

[19] Keshav Balachandar. *ML – Attention mechanism* (2023). URL: https://www.geeksforgeeks.org/ml-attention-mechanism/

[20] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, Yoshua Bengio. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation* (2014). URL: https://arxiv.org/abs/1406.1078

[21] Lilian Weng. *Attention? Attention!* (2018). URL: https://lilianweng.github.io/posts/2018-06-24-attention/

[22] LJ Miranda. *Understanding softmax and the negative log-likelihood* (2017). URL: https://ljvmiranda921.github.io/notebook/2017/08/13/softmax-and-the-negative-log-likelihood/

[23] Maxime. *What is a Transformer?* (2019). URL: https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04

[24] Mbali Kalirane. *Gradient Descent vs. Backpropagation: What's the Difference?* (2023). URL: https://www.analyticsvidhya.com/blog/2023/01/gradient-descent-vs-backpropagation-whats-the-difference/

[25] Mehreen Saeed. *A Gentle Introduction to Positional Encoding in Transformer Models, Part 1* (2023). URL: https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/

[26] Minh-Thang Luong, Hieu Pham, Christopher D. Manning. *Effective Approaches to Attention-based Neural Machine Translation* (2015). URL: https://arxiv.org/abs/1508.04025

[27] Mohit Mishra. *Stochastic Gradient Descent: A Basic Explanation* (2023). URL: https://mohitmishra786687.medium.com/stochastic-gradient-descent-a-basic-explanation-cbddc63f08e0

[28] Pragati Baheti. *Activation Functions in Neural Networks [12 Types & Use Cases]* (2021). URL: https://www.v7labs.com/blog/neural-networks-activation-functions

[29] Rahul Pandey. *How to deal with vanishing and exploding gradients* (2022). URL: https://medium.com/geekculture/how-to-deal-with-vanishing-and-exploding-gradients-in-neural-networks-24eb00c80e84

[30] Ramjee Rajasekaran. *Quick Look at RNN, LSTM, GRU and Attention* (2020). URL: https://rramjee.medium.com/quick-look-at-rnn-lstm-gru-and-attention-58ad79e2528b

[31] Rutger Ruizendall. *Deep Learning #4: Why You Need to Start Using Embedding Layers* (2017). URL: https://towardsdatascience.com/deep-learning-4-embedding-layers-f9a02d55ac12

[32] Samhita Alla. *Introduction to Encoder-Decoder Sequence-to-Sequence Model (Seq2Seq)* (2020). URL: https://blog.paperspace.com/introduction-to-seq2seq-models/

[33] Sandhya Krishnan. *How do determine the number of layers and neurons in the hidden layer?* (2021). URL: https://medium.com/geekculture/introduction-to-neural-network-2f8b8221fbd3

[34] Simeon Kostadinov. *Understanding Encoder-Decoder Sequence to Sequence Model* (2019). URL: https://towardsdatascience.com/understanding-encoder-decoder-sequence-to-sequence-model-679e04af4346

[35] Will Koehrsen. *Neural Network Embeddings Explained* (2018). URL: https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526

[36] *What are neurons in neural networks / how do they work?* (2016). URL: https://stats.stackexchange.com/questions/241888/what-are-neurons-in-neural-networks-how-do-they-work