Transformers: a journey through the model

Gianluca Peri

29 ottobre 2024

Attualmente i transformers sono utilizzati come modello fondazionale per una vasta serie di applicazioni. In questi appunti ci concentreremo unicamente sul più celebre caso d'uso:

Un transformer riceve in input un paragrafo incompleto e restituisce in output la più probabile prossima parola.

Il nostro obbiettivo è descrivere il viaggio del paragrafo attraverso il transformer.
Prendiamo ad esempio il seguente testo:

Mikhail Tal was a bad smoker but a good

Per analizzare questo paragrafo il primo passaggio è la tokenizzazione.

1 TOKENIZZAZIONE

La tokenizzazione è il processo di suddivisione di un testo in unità più piccole chiamate token. Esiste molta letteratura sul tema, ma possiamo permetterci di pensare ad un token come ad una parola o ad un elemento di punteggiatura (questo non è sempre vero in pratica, anche solo perché il sistema non sarebbe resiliente ad errori di battitura).
Possiamo dunque notare come il nostro paragrafo possa essere pensato come composto da 9 token:

Mikhail, Tal, was, a, bad, smoker, but, a, good

La suddivisione del testo in elementi *"atomici"* conclude il processo di tokenizzazione. Possiamo facilmente intuire tuttavia che il transformer non possa operare direttamente su questi, ma abbia bisogno di una rappresentazione numerica. Questo ci porta al prossimo step: l'embedding.

2 EMBEDDING

Per fornire ai token una rappresentazione vettoriale la prima cosa da fare è stabilire un dizionario contenente ogni possibile token, questo dizionario avrà dimensione K. Arrivati a questo punto esistono molti modi per procedere: il più comune è partire rappresentando ogni token tramite *one-hot encoding*.

$$\text{abaco} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^K$$

$$\text{baco} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^K$$

$$\text{cane} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^K$$

$$\vdots$$

Questa rappresentazione tuttavia è solo temporanea: viene modificata dal vero e proprio step di *embedding*, che consiste semplicemente nel mappare i token in uno spazio a minore dimensionalità tramite una *matrice di embedding* $E \in \mathbb{R}^{D \times K}$, dove D è la dimensione dello spazio di embedding. Questo produrrà una rappresentazione $v_n \in \mathbb{R}^D$ per ogni token $x_n \in \mathbb{R}^K$

$$v_n = E \cdot x_n \tag{1}$$

Gli elementi della matrice di embedding sono appresi durante il training del modello, e rappresentano la conoscenza che il modello ha acquisito riguardo i token. Nel seguito, per semplicità, continueremo a riferirci ai token con x_n, anche se in realtà ci riferiamo alla loro rappresentazione vettoriale v_n nello spazio di embedding. Risulta importante porre l'accento anche sul fatto che la dimensione dello spazio di embedding D è un iperparametro del modello, e che la scelta di questo valore è cruciale per il buon funzionamento dello stesso, solitamente questa dimensione viene scelta elevata, nell'ordine delle *decine di migliaia*.
Risulta fondamentale notare la presenza di evidenze sperimentali che mostrano come il processo di embedding riesca a mappare i token in uno spazio i cui assi rappresentano concetti significativi. Questo significa che il prodotto scalare tra due vettori di embedding può essere interpretato come una misura di similarità, e che gli angoli tra i vettori rappresentano relazioni concettuali significative.
Ad esempio: la distanza angolare fra i token *king* e *queen* nello spazio di embedding è simile a quella fra uncle e *aunt*.

3 POSITIONAL ENCODING

La rappresentazione dei token nello spazio di embedding per adesso non contiene informazioni sulla posizione dei token all'interno del paragrafo, questo è un problema in quanto non pianifichiamo di dare al transformer una struttura regressiva, e dunque il sistema tratterebbe allo stesso modo la frase

Mikhail Tal was a bad smoker but a good

e la frase

Mikhail Tal was a good smoker but a bad

Questo non è ideale, in quanto si capisce come le due frasi ammettano continuazioni ben diverse.
Per ovviare a questo problema si introduce il *positional encoding*, che consiste nell'aggiungere ai vettori di embedding una rappresentazione della posizione del token all'interno del paragrafo. Questo può essere fatto in molti modi, ma il più comune è quello di aggiungere ai vettori di embedding una rappresentazione sinusoidale della posizione del token.
Nello specifico, per ogni token x_n si calcola il suo vettore di positional encoding $r_n \in \mathbb{R}^D$ come segue:

$$[r_n]_i = \begin{cases} \sin\left(\frac{n}{L^{i/D}}\right) & \text{se } i \text{ è pari} \\ \cos\left(\frac{n}{L^{(i-1)/D}}\right) & \text{se } i \text{ è dispari} \end{cases} \tag{2}$$

Questo crea una codifica simile ad una codifica di posizione binaria, con la differenza che la codifica sinusoidale è continua e periodica, il che sembra aiutare. La codifica posizionale viene poi semplicemente *sommata* al vettore di embedding del token:

$$[x_n]_{new} = \cdot x_n + r_n \tag{3}$$

Si verifica come questo non corrompa l'informazione contenuta nel token, probabilmente a causa dell'elevata dimensionalità.

4 SELF-ATTENTION

Il cuore del transformer è il meccanismo di *self-attention*. Questo nasce per risolvere un problema specifico: lo stesso token può avere significati completamente diversi a seconda del contesto. Per fare un esempio riferito alla nostra frase: il termine *smoker* può riferirsi ad un essere umano con il vizio del fumo, oppure può anche riferirsi ad un macchinario da cucina per affumicare carni. Serve mettere in relazione il token con il più grande contesto della frase per comprenderne il significato.
Il meccanismo di self-attention permette al modello di considerare ogni token in relazione a tutti gli altri token, e di dare più peso a quelli che sono più rilevanti per il contesto, e dunque dovrebbe risolvere il nostro problema.
L'idea alla base del meccanismo è semplice: vogliamo misurare la similarità (rilevanza reciproca) tra tutte le possibili coppie di tokens nel paragrafo, ed usare poi questi scalari come una maschera per far parlare diversi token fra loro, modificando di conseguenza la loro rappresentazione nello spazio di embedding (e dunque sperabilmente il loro significato).
Per fare questo definiamo la matrice X, che ha come *righe* i vettori di embedding dei token:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \in \mathbb{R}^{N \times D} \tag{4}$$

dove N in questo caso vuole indicare la lunghezza del paragrafo in esame in termini del suo numero di token (la massima dimensione possibile per il paragrafo è un iperparametro del modello, ed è nota come *context window*, nel seguito per semplicità supporremo che la frase in esame abbia lunghezza esattamente pari alla *context window* del modello).
Per calcolare la similarità tra due token x_i e x_j possiamo usare il prodotto scalare tra i loro vettori di embedding:

$$XX^T \tag{5}$$

Gli elementi di questa matrice potranno avere qualunque segno e qualunque modulo, questo non è ideale per una misura di similarità/rilevanza fra coppie di token; per questo motivo si applica una funzione di *softmax* alla matrice XX^T, agente su tutte le righe della matrice, ottenendo la matrice di self-attention A:

$$A = \text{softmax}_{rows}(XX^T) \tag{6}$$

Questa matrice viene poi utilizzata come matrice di proiezione per i vettori di embedding, ottenendo così i vettori di embedding aggiornati dal meccanismo di attenzione:

$$Y = AX = \text{softmax}_{rows}(XX^T)X \tag{7}$$

Questa è l'essenza del meccanismo di self-attention, tuttavia se ci fermassimo qui la self-attention non sarebbe *trainable*, non avrebbe parametri da modificare per apprendere come rivolgere l'attenzione ai token più rilevanti.
Potremmo pensare di aggiungere dei parametri semplicemente ridefinendo:

$$X \to XU \tag{8}$$

con U matrice di parametri, tuttavia così facendo avremmo:

$$Y = \text{softmax}_{rows}(XUU^TX^T)XU \tag{9}$$

dove dunque XUU^TX^T è chiaramente una matrice simmetrica; questo si capisce sia un grosso problema in quanto il meccanismo di attenzione darebbe necessariamente lo stesso peso alla connessione

$$\text{martello} \to \text{utensile} \tag{10}$$

e alla connessione

$$\text{utensile} \to \text{martello} \tag{11}$$

e questo chiaramente non va bene, anche semplicemente perché tutti i martelli sono utensili ma non tutti gli utensili sono martelli.
Per ovviare a questo problema notiamo che le tre matrici X in (7) hanno tre funzioni diverse: la prima la possiamo pensare (riferendoci alla teoria sui database), come la matrice *query*, la seconda come matrice *key*, e la terza ed ultima a destra come matrice *value*. Il fatto che tutte queste siano in realtà la stessa matrice è quello che motiva il termine ***self**-attention*. Dato che concettualmente sono tre matrici diverse diamogli un diverso set di parametri:

$$Q = XW^{(Q)} \tag{12}$$

$$K = XW^{(K)} \tag{13}$$

$$V = XW^{(V)} \tag{14}$$

Notiamo che le matrici W possono essere matrici $D \times D$, tuttavia alle volte si preferisce associare la matrice K ad una riduzione della dimensionalità del dataset:

$$W^{(K)}_{D \times D_K} \tag{15}$$

e dunque, per avere all'interno della softmax una matrice quadrata:

$$W^{(Q)}_{D \times D_K} \tag{16}$$

Con questa modifica la matrice dentro la softmax non sarà più necessariamente simmetrica, ed aggiungendo un comodo termine di normalizzazione possiamo scrivere quindi:

$$Y = \text{softmax}_{rows}\left(\frac{QK^T}{\sqrt{D_K}}\right)V \tag{17}$$

la (17) è proprio la formula di self-attention che si trova nel paper *Attention is All You Need*. SI noti che nulla vieta di aggiungere anche un termine di *bias trainable* a queste trasformazioni.

4.1 *Multi-Head Attention*

Arrivati a questo punto potremmo procedere senza menzionare altro, tuttavia per rendere un quadro completo non possiamo non fare attenzione al fatto che quasi sempre una sola attuazione del meccanismo di attenzione non basta per far parlare i token fra loro per bene. Servono più passate di attenzione, *multiple attention heads*, come si dice; e queste vengo fatte operare *in parallelo*.
Nello specifico si fanno operare multiple attenzioni, con parametri diversi, sullo stesso set di tokens: i risultati di questo processo vengono solitamente chiamati H_h, dove h enumera il numero di *attention heads*. Dunque abbiamo:

$$H_h = \text{softmax}_{rows}\left(\frac{QW_h^{(Q)}(KW_h^{(K)})^T}{\sqrt{D_K}}\right)VW_h^{(V)} \tag{18}$$

Tutte questi risultati, vengono poi combinati in un singolo output attraverso una combinazione lineare codificata da una matrice *trainable* $W^{(O)}$ delle giuste dimensioni:

$$Y = \text{concat}(H_1, H_2, \ldots, H_h)W^{(O)} \tag{19}$$

5 FEED-FORWARD RESIDUAL NETWORKS

Quanto descritto non può bastare per portare a termine il compito specificato all'inizio di questi appunti: per essere in grado di capire che la prossima parola nella frase è probabilmente *chess* (e quella dopo ancora *player*) il modello deve essere in grado di custodire al suo interno informazioni sul mondo, deve aver appreso, durante l'addestramento, che Tal era un ottimo giocatore di scacchi.
In parole povere dunque dobbiamo dare al modello una maggiore capacità di elaborazione e mantenimento dati, e questo può essere fatto aggiungendo degli strati computazionali MLP *Multi-Layer Perceptron*. Ci sono due cose importanti da notare a questo riguardo:

- Questi strati operano in parallelo su tutti i token: i token non parlano fra di loro durante questa elaborazione.
- Questi strati sono *residuali*: il loro output viene sommato al vettore di input.

Questo secondo punto è importante in quanto permette al modello di apprendere incrementi rispetto al vettore di input, e non di sovrascriverlo completamente, inoltre mitigano il problema del *vanishing gradient*.
In coda notiamo anche che una connessione residuale può essere anche posta sopra il blocco di *self-attention*.

6 LAYER NORMALIZATION

Per evitare problemi di ***internal*** *covariate shift* si applica una normalizzazione ai vettori di embedding, e ai vettori di output dei *feed-forward residual networks*. Questa normalizzazione è detta *layer normalization*, e consiste nel calcolare la media e la deviazione standard di ogni vettore di output, e normalizzarlo di conseguenza.

7 GOOD JOB! DO IT AGAIN.

Il blocco di *self-attention* ed il blocco MLP correttamente normalizzato sono i due blocchi fondamentali del transformer, e vengono ripetuti per un certo numero di volte (di solito 6, 12 o 24) per dare al modello la capacità di elaborare informazioni complesse.

8 OUTPUT LAYER

Dopo la sequenza di blocchi di attenzione e MLP l'output del modello è una matrice di dimensione $N \times D$, tuttavia quello che vogliamo in output è chiaramente una distribuzione di probabilità discreta sul vocabolario di dimensione K. Per ottenere questo si applica una finale trasformazione lineare, seguita da una softmax. Dato questo setup l'addestramento si svolge solitamente trattando il problema come un *task* di classificazione sul vocabolario, e minimizzando dunque la *cross-entropy loss*.
Predizione di più di una singola parola può essere svolta iterando il processo di predizione, aggiungendo ogni volta la parola predetta alla fine del paragrafo, e ripetendo il processo.
Infine notiamo che modelli specifici per una funzione (*e.g.* chatbots) possono essere creati tramite processi di *fine-tuning* e aggiunte di *prompt di contesto nascosti all'utente*. Questo apre a possibili attacchi (vedi *prompt injection attacks*).