Sentiment Analysis

Utilizzo dei transformer per la classificazione di recensioni di film

Progetto per il corso di Computational Learning
Jacopo Manetti - Matricola: [7124953]
Gennaio 2024

1 Obiettivo

Questo progetto si focalizza sull'analisi del sentiment delle recensioni di film utilizzando il dataset IMDB [1]. L'obiettivo è sviluppare un modello in grado di classificare le recensioni in categorie positive o negative.

2 Preparazione del Dataset

Il dataset IMDB contiene 50.000 recensioni, suddivise equamente tra training e test set. Ogni set include recensioni etichettate come positive o negative. Il dataset IMDB viene caricato utilizzando la libreria **torchtext**, che fornisce un accesso diretto e semplificato ai dati. In questo processo, i dati sono suddivisi in due insiemi: uno per l'addestramento (training) e uno per il test. Ogni insieme comprende sia le etichette (labels) che i testi delle recensioni.

Il testo grezzo del dataset IMDB richiede una pulizia per rimuovere elementi superflui e standardizzare il formato. La funzione **clean_text** è progettata per questo scopo e svolge le seguenti operazioni:

- **Rimozione dei Tag HTML:** Utilizzando BeautifulSoup, vengono eliminati tutti i tag HTML dal testo, lasciando solo il testo puro.

  Durante l'esecuzione dello script, è comparso un avviso riguardante il parsing di alcuni testi come file HTML anziché markup effettivo. Tuttavia, ciò non ha impedito il corretto funzionamento del processo di training del modello, il quale è stato completato con successo e ha prodotto risultati soddisfacenti.

- **Conversione in Minuscolo:** Il testo viene convertito in minuscolo per garantire uniformità e ridurre la complessità del modello.

- **Rimozione della Punteggiatura:** Vengono eliminati tutti i caratteri speciali e la punteggiatura, lasciando solo parole e numeri.

Dopo la pulizia, il testo viene tokenizzato utilizzando torchtext. In questo processo, le recensioni vengono divise in token (parole) individuali.

Il passo successivo è la costruzione di un vocabolario, che serve a mappare ogni parola unica a un indice intero. Questo processo consente di convertire il testo in una forma che può essere elaborata da modelli di machine learning. Si limita la dimensione del vocabolario a 15,000 parole per gestire la complessità e migliorare l'efficienza del modello.

Dopodichè, il testo tokenizzato viene convertito in sequenze di interi utilizzando la mappatura vocabolario-indice creata. Queste sequenze sono poi troncate o paddate per avere la stessa lunghezza, assicurando che ogni input al modello abbia dimensioni uniformi.

Infine, il test set fornito viene splittato in 2 parti uguali in modo casuale in modo da avere un validation set da usare per scegliere la configurazione del transformer e un test set per valutarne le prestazioni finali.

3 Il modello

Il modello Transformer [2], è un'architettura di punta nell'elaborazione del linguaggio naturale, noto per il suo efficace meccanismo di attenzione. Nel nostro progetto di sentiment analysis sul dataset IMDB, abbiamo utilizzato solo la parte dell'encoder del Transformer. Eccone una panoramica generale:

Il modello Transformer si basa su un'architettura che processa il testo attraverso una serie di trasformazioni. Iniziando dal basso, abbiamo **l'Input Embedding** che converte ogni parola in un vettore ad alta dimensione, catturando così le sfumature semantiche. Questi vettori vengono poi combinati con il **Positional Encoding** per mantenere il contesto sequenziale all'interno della nostra serie di dati. Salendo nell'architettura, incontriamo una serie di **blocchi encoder**, ciascuno con un'**attenzione multi-testa** e uno strato **feed-forward**. Questi blocchi, ripetuti N volte, consentono al modello di focalizzarsi su parti differenti del testo per comprendere le relazioni complesse tra le parole. Gli strati di **"Add & Norm"** integrati con ogni blocco forniscono la normalizzazione e i collegamenti residuali, che sono fondamentali per l'apprendimento stabile e l'integrazione efficace delle informazioni attraverso la rete. Questa struttura, rappresentata nell'immagine, culmina nel **layer di classificazione**, che utilizza l'output elaborato per determinare il sentiment di ogni recensione.

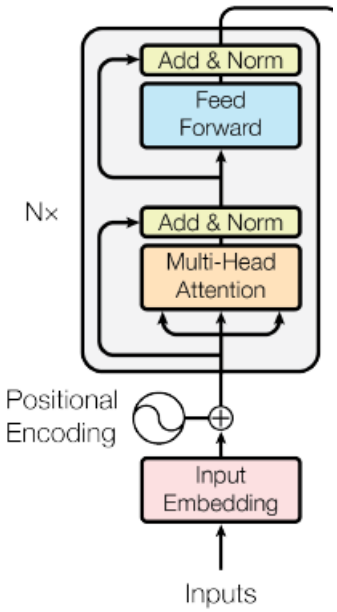


Figure 1: *Architettura encoder*

3.1 Input embedding

La classe **InputEmbedding** è responsabile di convertire l'input testuale in una rappresentazione vettoriale, che sarà poi utilizzata dal modello Transformer. Questa classe prende due parametri nel suo costruttore: *vocab_size*, che rappresenta la dimensione del vocabolario, e *d_model*, che indica la dimensione del vettore di embedding. All'interno del suo metodo **__init__**, la classe inizializza un layer di embedding con dimensioni [vocab_size × d_model]. Durante il passaggio in avanti (**forward**), l'input x, che è una sequenza di token, viene trasformato in vettori di embedding utilizzando il layer di embedding. Per preservare le scale, ogni vettore di embedding viene moltiplicato per $\sqrt{d\_model}$.

Questo è importante perché durante l'allenamento del modello, le operazioni di moltiplicazione e somma degli embedding possono portare a valori che crescono eccessivamente, causando problemi di stabilità numerica o rendendo più difficile l'ottimizzazione del modello.

Inizializzazione degli Embedding Inizialmente, i pesi del layer di embedding sono assegnati casualmente. L'ottimizzazione degli embedding non si limita solo alla loro inizializzazione ma continua attraverso l'addestramento del modello. Tuttavia questa scelta iniziale, può essere migliorata

per accelerare la convergenza o per aumentare la qualità generale dell'embedding appreso, mediante embedding pre-addrestati come GloVe.

3.2 Positional encoding

La classe **PositionalEncoding** è responsabile di aggiungere informazioni sulla posizione alle rappresentazioni degli embedding di input nel modello Transformer. Questo è fondamentale poiché i modelli Transformer non hanno una struttura intrinseca di sequenza come le reti neurali ricorrenti (RNN) o le reti neurali convoluzionali (CNN), e quindi richiedono un modo per comprendere l'ordine sequenziale dei token all'interno dell'input.

Nel suo costruttore, questa classe prende tre parametri: *d_model*, che rappresenta la dimensione del vettore di embedding, *max_len*, che indica la lunghezza massima della sequenza di input, e *dropout*, che specifica la probabilità di dropout da applicare durante il processo di encoding posizionale.

Durante l'inizializzazione (**__init__**), la classe calcola una matrice di encoding posizionale (pe) utilizzando una combinazione di funzioni sinusoidali e cosinusoidali. Questa matrice fornisce informazioni sulla posizione relativa di ciascun token all'interno della sequenza di input. La frequenza delle funzioni sinusoidali e cosinusoidali varia in base alla posizione nella sequenza e alla dimensione del vettore di embedding (d_model), garantendo che ogni posizione abbia una rappresentazione unica. In particolare, il termine sinusoidale è utilizzato per le posizioni pari, mentre il termine cosinusoidale è utilizzato per le posizioni dispari, come illustrato nelle formule del paper [2]:

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

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

dove *pos* è la posizione e i è la dimensione dell'embedding,
che va da 0 a (d_model - 1).

Nel metodo **forward**, l'input x, che rappresenta gli embedding di input, viene aggiunto all'encoding posizionale *pe*. Questo aggiornamento viene eseguito elemento per elemento, poiché ogni elemento dell'input è associato a una posizione specifica nella sequenza. La parte *.requires_grad_(False)* viene utilizzata per garantire che l'encoding posizionale non venga coinvolto nel processo di backpropagation durante l'allenamento, poiché è una parte fissa del modello. Infine, il dropout viene applicato all'output per introdurre della regolarizzazione e ridurre l'overfitting durante l'addestramento del modello.

3.3 Multi Head Attention

La classe **MultiHeadAttention** è un componente cruciale nell'architettura del Transformer, consentendo al modello di captare simultaneamente diverse relazioni tra le parole in input, arricchendo così la comprensione del contesto. Questo meccanismo si distingue per la sua capacità di parallelizzare l'attenzione su più "teste", ciascuna focalizzata su aspetti diversi dell'input.

Durante l'inizializzazione (**__init__**), la classe configura i parametri essenziali: il numero di teste di attenzione (h), la dimensione dell'embedding (*d_embed*), e una componente di dropout. Viene calcolata la dimensione di ogni testa di attenzione (*d_k*) dividendo la dimensione dell'embedding per il numero di teste, permettendo così una distribuzione equa dell'informazione tra tutte le teste.

Il fulcro dell'attenzione multi-testa si svolge nel metodo **forward**, dove l'input subisce una trasformazione mediante tre layer lineari distinti, destinati a generare le rappresentazioni di query (Q), chiavi (K), e valori (V). Queste rappresentazioni sono poi divise per ciascuna testa di attenzione, mantenendo una separazione funzionale che consente a ogni "testa" di focalizzarsi su un sottoinsieme specifico dell'input, in questo modo ogni testa avrà pesi diversi.

La computazione dell'attenzione per ogni testa segue la formula:

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

dove Q, K, e V rappresentano rispettivamente le query, le chiavi e i valori, mentre d_k è la dimensione di ciascuna testa.
Una volta calcolata l'attenzione per ciascuna testa, i risultati vengono concatenati e proiettati attraverso un ulteriore layer lineare (W^O), finalizzando l'integrazione delle diverse prospettive captate dalle teste. La maschera, se applicata, garantisce che l'attenzione venga assegnata solo a posizioni valide, escludendo token di padding o anticipando informazioni non ancora rivelate in contesti sequenziali.

3.4 Layer normalization

La classe **LayerNormalization** implementa il layer di normalizzazione del livello è posizionato dopo ciascun layer principale dell'encoder. Questo significa che segue ogni passaggio di attenzione multipla e di feedforward nell'encoder. La normalizzazione del livello è simile alla normalizzazione del batch, ma invece di normalizzare le dimensioni del batch, normalizza le dimensioni del livello.
Nel costruttore della classe (__**init**__), viene inizializzato il layer di normalizzazione del livello con un parametro , che è un valore piccolo utilizzato per evitare la divisione per zero e migliorare la stabilità numerica durante il calcolo. Viene anche inizializzato un parametro α (chiamato anche fattore di scala) e un parametro β (chiamato anche bias), entrambi come tensori di dimensione 1, utilizzati per moltiplicare e sommare rispettivamente l'output normalizzato.
Nel metodo **forward**, vengono calcolate la media e la deviazione standard lungo l'ultima dimensione del tensore di input *x*. Questo viene fatto utilizzando le funzioni di PyTorch *mean* e *std*. Successivamente, l'input viene normalizzato utilizzando la formula della normalizzazione del livello:

$$LN(x) = \alpha \cdot \frac{x - mean(x)}{std(x) + \epsilon} + \beta$$

dove α e β sono i parametri appresi dal modello. Questa normalizzazione assicura che le caratteristiche all'interno di ciascun livello siano normalizzate e centrate intorno a zero, migliorando così la stabilità del training e accelerando la convergenza del modello.

3.5 Feed forward

La classe **FeedForward** implementa il layer di feedforward all'interno dell'architettura del Transformer. Questo layer è essenziale per introdurre non linearità e complessità nel modello, consentendo di apprendere relazioni più complesse tra le caratteristiche di input.
Nel costruttore della classe (__**init**__), vengono inizializzati due layer lineari: *linear1* e *linear2*. Il primo layer (*linear1*) proietta l'input da una dimensione di *d_model* a una dimensione di *d_ff* (il numero di unità nascoste del feedforward). Questo passaggio è seguito dalla funzione di attivazione ReLU, che è implementata come $\max(0, xW_1 + b_1)$, dove W_1 rappresenta i pesi del primo layer e b_1 il bias. Il dropout viene quindi applicato per introdurre della regolarizzazione e ridurre l'overfitting durante l'addestramento.
Il secondo layer (*linear2*) proietta l'output del primo layer da una dimensione di *d_ff* a una dimensione di *d_ model*, riportandolo alla dimensione dell'input originale. Questo passaggio è rappresentato dalla formula $(xW_2 + b_2)$, dove W_2 rappresenta i pesi del secondo layer e b_2 il bias.
Nel metodo **forward**, l'input x passa attraverso il primo layer lineare seguito dalla funzione di attivazione ReLU e il dropout. Successivamente, l'output viene passato attraverso il secondo layer lineare per ottenere l'output finale del layer di feedforward.

3.6 Encoder

La classe **Encoder** rappresenta la parte di encoding dell'architettura del Transformer. Essa comprende sia l'embedding dei token che l'embedding posizionale, seguito da uno stack di blocchi dell'encoder e un layer di normalizzazione.
Nel costruttore della classe, vengono inizializzati i componenti principali dell'encoder:

- **input_embedding:** l'embedding dei token, che proietta i token di input in uno spazio vettoriale.

- **positional_encoding:** l'embedding posizionale, che aggiunge informazioni sulla posizione relativa dei token all'interno della sequenza.
- **encoder_blocks:** uno stack di blocchi dell'encoder, ognuno dei quali applica un'attenzione multi-testina seguita da un feedforward.
- **norm:** un layer di normalizzazione che normalizza l'output dell'encoder.

Nel metodo **forward**, l'input viene prima sottoposto all'embedding dei token e all'embedding posizionale. Successivamente, l'input viene passato attraverso ogni blocco dell'encoder nell'encoder_blocks. L'output dell'ultimo blocco dell'encoder viene infine normalizzato utilizzando il layer di normalizzazione *norm*.
La classe **EncoderBlock**, invece, rappresenta un singolo blocco dell'encoder. Esso consiste in un'operazione di self-attention seguita da un feedforward position-wise.
Nel metodo **forward**, l'input segue lo schema riportato nel disegno dell'encoder.

3.7 Transformer

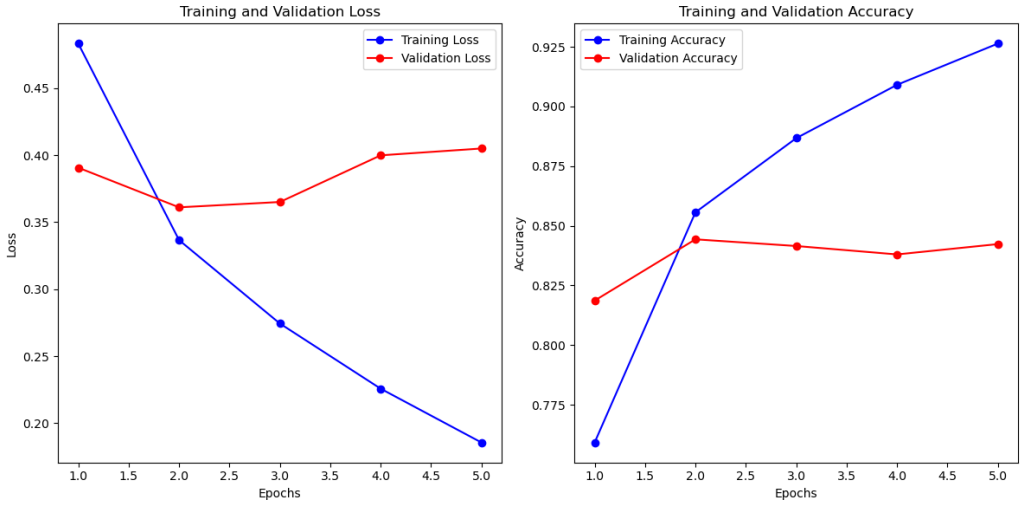
Infine, la classe **Transformer**, rappresenta l'intera architettura del Transformer, che comprende un encoder seguito da un layer lineare per la classificazione.
Nel metodo **forward**, l'input passa attraverso l'encoder per ottenere una rappresentazione codificata. Questa rappresentazione viene poi ridotta ad un singolo vettore di dimensione *num_classes* utilizzando una media lungo l'asse temporale. Infine, il vettore ottenuto viene passato attraverso il layer lineare per ottenere l'output finale del modello, che consiste nelle probabilità di appartenenza alle diverse classi di output.

4 Valutazione e testing

La parte finale del progetto implementa il processo di addestramento e valutazione del modello Transformer. Dopo aver definito la configurazione del modello viene inizializzato il modello utilizzando la funzione **make_model()**.
L'addestramento è gestito dalla funzione **train()**, che incorpora un meccanismo di early stopping per prevenire l'overfitting e ottimizzare l'uso delle risorse di calcolo. Questa funzione itera l'addestramento del modello per un numero prestabilito di epoche, monitorando contemporaneamente le prestazioni su set di addestramento e validazione utilizzando l'ottimizzatore *Adam* e la funzione di perdita **nn.CrossEntropyLoss()**. La presenza di una maschera di padding all'interno della funzione **train_epoch** garantisce che il modello gestisca efficacemente le sequenze di lunghezza variabile, preservando l'integrità dell'input e migliorando l'accuratezza delle previsioni.
Significativamente, il progetto introduce una valutazione dettagliata del modello attraverso la funzione **evaluate()**, utilizzata sia durante l'addestramento per la validazione che alla fine per testare le prestazioni complessive del modello. Questo permette di ottenere un feedback tempestivo sulle prestazioni e di regolare la strategia di addestramento se necessario.
È stata aggiunta la visualizzazione dei risultati di addestramento e validazione tramite la funzione **plot_metrics**, che fornisce un'analisi grafica dell'andamento della loss e dell'accuratezza nel corso delle epoche. Tale visualizzazione facilita l'identificazione visiva di possibili problemi come l'overfitting o l'underfitting e consente di valutare l'efficacia dell'early stopping.
Per la configurazione del modello non sono risultate necessarie particolari accortezze dato che in ogni caso si raggiungono ottimi valori di accuratezza in relativamente poche epoche. Per la costruzione dell'encoder abbiamo provato una configurazione simile a quella riportata nel paper [2] (*Modello 1*), una intermedia (*Modello 2*) e una piccola (*Modello3*), da cui abbiamo ottenuto i seguenti risultati:

Configurazione del modello 1

Parametro	Valore
Dimensione del vocabolario dell'encoder	15000
Dimensione dell'embedding	512
Dimensione del feed-forward	2048
Numero di teste di attenzione	8
Numero di encoder	6
Massima lunghezza della sequenza	200
Batch size	128
Probabilità di dropout	0.1

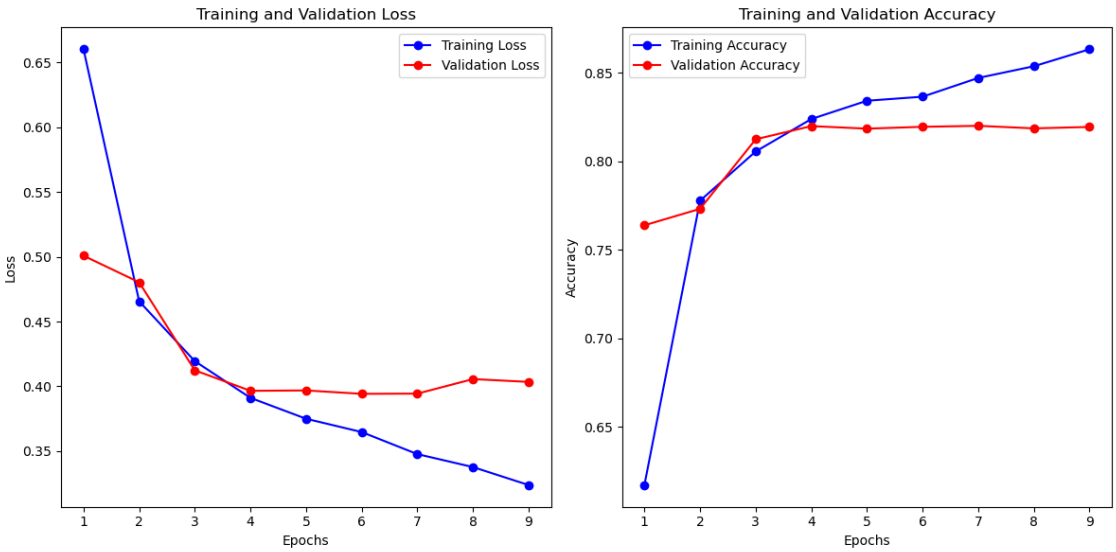


Considerazioni:
Il training loss e l'accuracy mostrano che il modello impara correttamente ma il validation resta quasi costante indicando che la configurazione è troppo complessa e porta all'overfitting.

Con questa configurazione l'early stopping ci fa interrompere la computazione alla 5 epoca.
I valori finali sul validation set sono:
Val Loss: 0.4050, Val Acc: 0.8423

Configurazione del modello 2

Parametro	Valore
Dimensione del vocabolario dell'encoder	15000
Dimensione dell'embedding	256
Dimensione del feed-forward	1024
Numero di teste di attenzione	8
Numero di encoder	4
Massima lunghezza della sequenza	200
Batch size	128
Probabilità di dropout	0.1

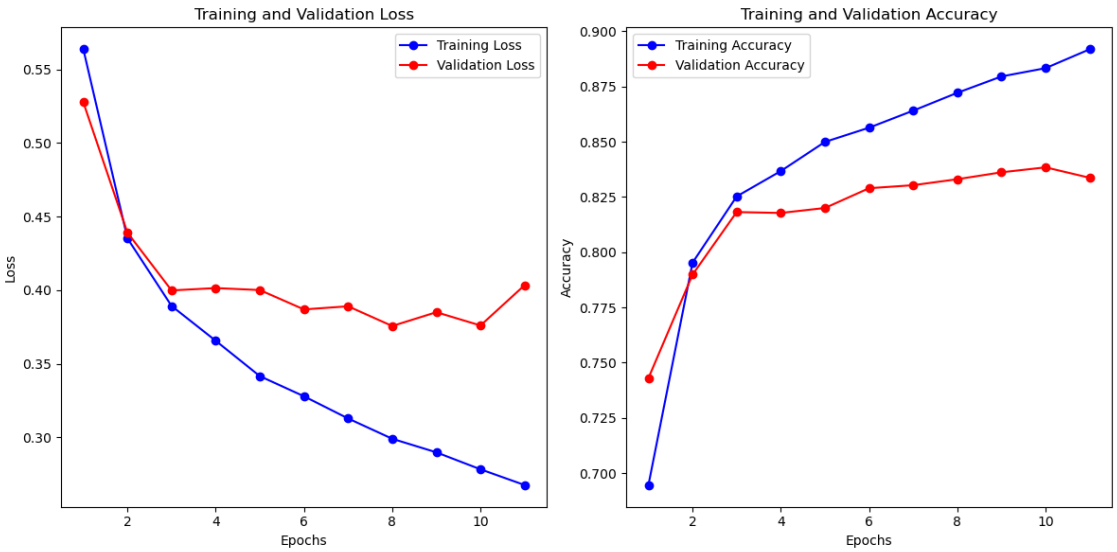


Considerazioni:
Possiamo notare un miglioramento rispetto a prima, il training continua a comportarsi bene mentre la validation mostra segni di stabilizzazione o leggero aumento verso le ultime epoche, suggerendo un possibile inizio di overfitting.

Con questa configurazione l'early stopping ci fa interrompere la computazione alla 9 epoca.
I valori finali sul validation set sono:
Val Loss: 0.4035, Val Acc: 0.8194

Configurazione del modello 3

Parametro	Valore
Dimensione del vocabolario dell'encoder	15000
Dimensione dell'embedding	128
Dimensione del feed-forward	512
Numero di teste di attenzione	4
Numero di encoder	2
Massima lunghezza della sequenza	200
Batch size	128
Probabilità di dropout	0.1



Considerazioni:
La tendenza all'overfitting rimane anche qui ottenendo prestazioni abbastanza simili a quelle del modello precedente.

Con questa configurazione l'early stopping ci fa interrompere la computazione alla 11 epoca.
I valori finali sul validation set sono:
Val Loss: 0.4035, Val Acc: 0.8337

Preferisco la terza configurazione poiché, nonostante i modelli abbiano risultati di accuratezza simili, il terzo modello è il più leggero in termini di parametri mantenedo una accuracy in validation quasi uguale al primo modello richiedendo però meno risorse computazionali durante l'addestramento e l'inferenza.

Adesso possiamo usare il nostro modello sul test set, da cui abbiamo ottenuti i seguenti risultati:
Test set: *Loss:* 0.3689, *Accuracy:* 0.8337

5 Play Ground

È stato creato e messo a disposizione un ulteriore file in cui è possibile usare il modello su una recensione a piacere per verificarne direttamente la funzionalità, è possibile usare questo script fin da subito in quanto il progetto comprende già un modello allenato da me 'model.pth' con la configurazione riportata sopra, eseguendolo verrà restituito il contenuto della recensione inserita accompagnato da una stringa che indica se la recensione è positiva o negativa.

Attenzione: Se si vuole usare il playground con un modello allenato usando una configurazione diversa è necessario cambiare la configurazione anche in questo file.

References

[1] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.