UNIVERSITÀ DI PISA

DIPARTIMENTO DI MATEMATICA

Laurea Triennale in Matematica

Input di Lunghezza Arbitraria e Misure Gaussiane nei Layer di Attention di Architetture Transformer

Relatore:

Prof. Dario Trevisan

Candidato:

Edoardo Ponti

ANNO ACCADEMICO 2024/2025

Ringraziamenti

Al Professor Dario Trevisan, per avermi proposto un argomento molto interessante seppur lontano dalla matematica tradizionale, e per avermi aiutato nella realizzazione della tesi.
Un ringraziamento speciale va ai miei genitori, ai miei nonni, ai miei amici, per avermi accompagnato e affiancato lungo l'intero percorso universitario.
Infine, il mio pensiero va alla mia fidanzata, per avermi supportato, e quando necessario sopportato, fin dal primo giorno in cui ci siamo incontrati.

Indice

Introduzione

Negli ultimi anni, i modelli di reti neurali hanno ottenuto un enorme successo in numerose applicazioni, tra cui l'elaborazione automatica del linguaggio naturale, delle immagini e dei dati audio. Questi modelli fanno parte di un campo più ampio conosciuto come Intelligenza Artificiale (AI), che mira a sviluppare tecniche computazionali in grado di risolvere problemi complessi, tra cui quelli che solitamente richiedono capacità cognitive tipiche del ragionamento umano. L'Intelligenza Artificiale è ormai alla base di molte applicazioni quotidiane, come assistenti virtuali, traduzione automatica e riconoscimento vocale.

Nel contesto dell'Elaborazione del Linguaggio Naturale (Natural Language Processing, abbreviato come NLP), l'introduzione di nuovi modelli di reti neurali ha notevolmente migliorato la qualità dei risultati, consentendo approcci molto più efficienti e precisi, come ad esempio i modelli di ChatGPT sviluppati da *OpenAI*.

La sfida principale dell'NLP è quella di comprendere il linguaggio umano, e, al contempo, di essere in grado di generarlo, al fine di mantenere una conversazione. Per raggiungere questo obiettivo, i modelli di Intelligenza Artificiale utilizzano le parole generate fino a quel momento come input, per poi calcolare una distribuzione di probabilità per le parole successive ancora da generare. La chiave di questo processo è comprendere come le varie parole si influenzano tra di loro, e capire il significato delle parole nel contesto più ampio della conversazione.

É evidente che esprimere matematicamente le proprietà del linguaggio sia complesso. Nel campo dell'Intelligenza Artificiale, il tentativo per affrontare questa difficoltà è quello di rappresentare le parole come vettori di alta dimensione, tramite un processo chiamato *embedding*. Questi vettori devono catturare le informazioni più significative relative a ciascuna parola, ma il loro funzionamento sarà spiegato più nel dettaglio in seguito.

Inoltre, vengono utilizzate matrici parametriche di grandi dimensioni, al fine di consentire al modello di poter affrontare svariate situazioni. Riguardo ciò, vi è una fase preliminare di *training*, durante la quale il sistema viene addestrato su un insieme di input già noti, al fine di ottimizzare i parametri delle matrici utilizzate. Se il sistema non genera output coerenti, i parametri delle matrici che hanno avuto maggiore influenza sull'output errato vengono modificati per migliorare le prestazioni del modello. Questo processo viene ripetuto per molte iterazioni, e alla fine del ciclo di addestramento ci si aspetta che il sistema possa operare efficacemente su qualsiasi input.

Una delle architetture più recenti e innovative è quella dei *Transformers*, introdotta per la prima volta da Vaswani et al. nell'Articolo [2]. Essa si distingue dai modelli precedenti grazie all'adozione di un meccanismo chiamato *Attention*. Il funzionamento specifico di questa architettura è trattato nel primo capitolo, dove vengono presentate le sue componenti e spiegato nel dettaglio il funzionamento di ciascuna di esse.

Invece, nel secondo capitolo viene analizzato un caso più teorico, basato sull'Articolo [1] di Peyré et al., nel quale viene discusso come trattare la situazione in cui la dimensione dell'input, ad esempio del numero di parole, sia arbitrario, e quindi anche infinito.

In questo contesto, vengono descritti i cambiamenti necessari al modello di Vaswani, il quale si basa solamente su matrici. Sarà infatti necessario introdurre una distribuzione di probabilità per descrivere come le parole sono disposte, e si lavorerà direttamente con questa misura, ad esempio tramite un *push-forward*.

Successivamente, in questo lavoro citeremo il risultato principale ottenuto da Peyré nell'Articolo [1], il quale riguarda l'*universalità* dei Transformers in questa particolare configurazione, ovvero la capacità di approssimare arbitrariamente una funzione continua da $\mathcal{P}(\Omega) \times \Omega$ a $\mathbb{R}^{d'}$, dove la particolarità di questo risultato è la validità anche per un input di dimensione arbitraria.

Verrà quindi presentata una versione alternativa a questo teorema, basando la dimostrazione su quanto svolto nell'Articolo [3]. Sebbene questo risultato sia più debole rispetto ad altri già noti, la dimostrazione utilizza solamente nozioni matematiche elementari, che rende quindi il risultato interessante.

Infine, nell'ultimo capitolo eseguiremo uno studio numerico mirato su matrici simmetriche con elementi estratti da distribuzioni uniformi o gaussiane — strutture che ricorrono di frequente durante il processo di training dei modelli.

Dimostreremo innanzitutto che, se l'input è distribuito secondo una legge gaussiana, anche la misura di probabilità trasformata resta gaussiana, sfruttando l'importante proprietà di riproducibilità, introdotta nell'Appendice A.2. Dopodiché, andremo a verificare sperimentalmente le formule trovate per la media e la nuova matrice di covarianza: inizializziamo campioni gaussiani, e convalideremo i risultati teorici. Successivamente, itereremo la trasformazione per studiarne la dinamica a lungo termine, e vedremo che, inizializzando i parametri in maniera standard, si avrà instabilità numerica e divergenza dello spettro. Invece, scegliendo accuratamente le condizioni iniziali, sarà possibile ottenere una qualche convergenza dello spettro di queste matrici.

Infine, simuleremo la fase di *training* di un Transformer: forniremo all'architettura un insieme di esempi (input,output) e, mediante discesa del gradiente, aggiorneremo iterativamente i suoi parametri. Al termine dell'addestramento, nella fase di *validation* mostreremo che la rete produce output accurati anche su dati mai visti, a conferma dell'apprendimento corretto del comportamento desiderato.

Capitolo 1

Transformers

Il modello Transformer, introdotto da Vaswani et al. nell'Articolo [2], rappresenta un approccio innovativo per la modellazione di sequenze. A differenza delle reti neurali ricorrenti (RNN), dove è necessaria un'analisi sequenziale dell'input, qui il meccanismo di *Self-Attention* consente un'esaminazione simultanea della sequenza in entrata, permettendo una maggiore parallelizzazione delle operazioni e quindi un tempo ridotto nel processo di training. Infatti, le reti neurali utilizzano matrici parametriche per calcolare l'output, che devono essere precedentemente *allenate* al fine di performare al meglio su un input qualsiasi.

Il modello Transformer è strutturato attorno a due componenti principali: l'*encoder* e il *decoder*, che sono stati introdotti per la prima volta da Cho et al. nell'Articolo [6]. L'encoder ha il compito di mappare una sequenza di input $(x_1, \ldots, x_n)$, che può essere ad esempio una frase o un'immagine, in un insieme di vettori di rappresentazione continui $(z_1, \ldots, z_n)$, a partire dai quali il decoder genera una sequenza di simboli in output $(y_1, \ldots, y_n)$, un elemento alla volta.

Dal punto di vista strutturale, un Transformer è composto da L[1] *layer*, dove ciascuno è a sua volta costituito da diversi moduli, di numero differente a seconda che si tratti dell'encoder o del decoder, rispettivamente a sinistra e a destra della Figura 1.1.

[1] pari a 96 in ChatGPT3 [7].

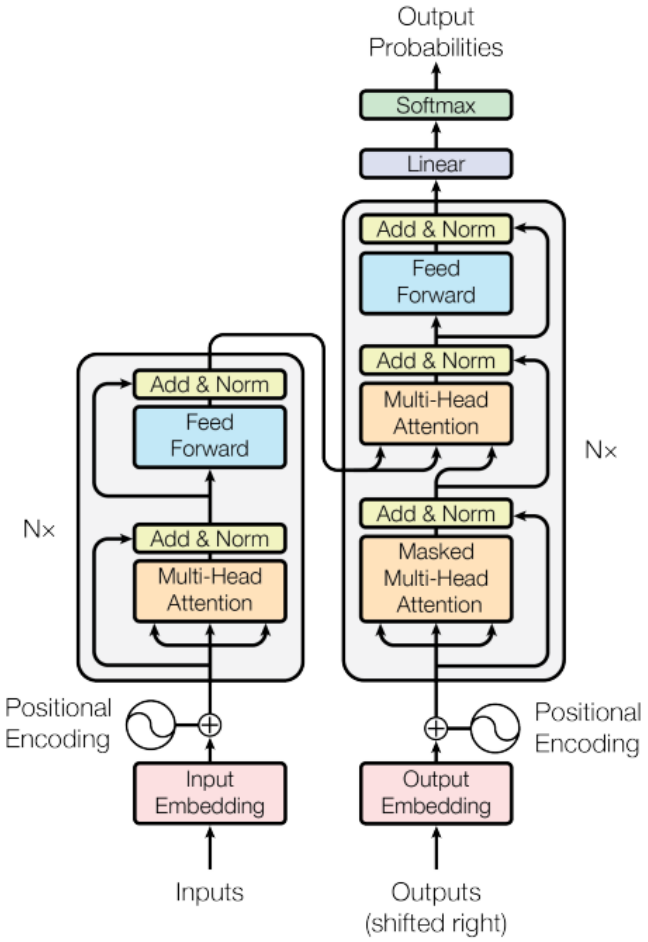


Figura 1.1: L'architettura di un Transformer.

Il compito di un'architettura Transformer è predire l'elemento successivo in una sequenza, considerando tutte le informazioni fornite dall'input. In output, il modello genera una distribuzione di probabilità sull'insieme di tutti i possibili elementi, che corrisponde ad *Output Probabilities* nella Figura 1.1, dalla quale viene selezionato l'elemento successivo da generare. Ad esempio, nel *Natural Language Processing*, l'obiettivo è quello di prevedere la parola successiva in una frase, partendo da tutte quelle generate in precedenza.

Per fare ciò, nelle prossime sezioni vedremo come il modello è in grado di imparare relazioni più o meno profonde tra le parole, al fine di comprendere il significato della frase ed effettuare una predizione che sia coerente con il contesto fornito dalle altre parole.

1.1 Embedding

Sia nell'encoder che nel decoder il primo step è *l'embedding*, che consiste nel dividere l'input in n porzioni, chiamati *token*, che vengono successivamente trasformati tramite una matrice parametrica in *vettori di embedding* di dimensione d_{in}[2]. Quest'ultimi vengono organizzati in una matrice X di dimensione $d_{in} \times n$, dove ciascun vettore di embedding è inserito in una colonna della matrice.

Questi vettori densi racchiudono tutte le informazioni necessarie per caratterizzare ogni singolo token. Ossia, nello spazio d_{in}-dimensionale ciascuna combinazione di coordinate è in grado di rappresentare un determinato concetto: ad esempio, aggettivi corrispondenti o sinonimi saranno mappati in vettori simili.

Inoltre, vi è la possibilità di proiettare questi vettori in uno spazio di dimensione inferiore, solitamente pari a 2 o 3, al fine di visualizzare le similitudini tra i diversi embedding. Una rappresentazione di questo tipo è mostrata nella Figura 1.2, tratta dall'Articolo [8], in cui viene affrontato il problema di generare un'animazione 3D del viso umano, mimando le espressioni facciali corrispondenti ad una nota vocale passata come input. In tale contesto, il segnale audio viene suddiviso in segmenti, che vengono poi trasformati in vettori di embedding, che codificano le informazioni emotive intrinseche alla componente linguistica. Infine, questa rappresentazione compatta consente di generare un'animazione coerente del volto, dove le espressioni facciali riflettono le emozioni estratte dal parlato.

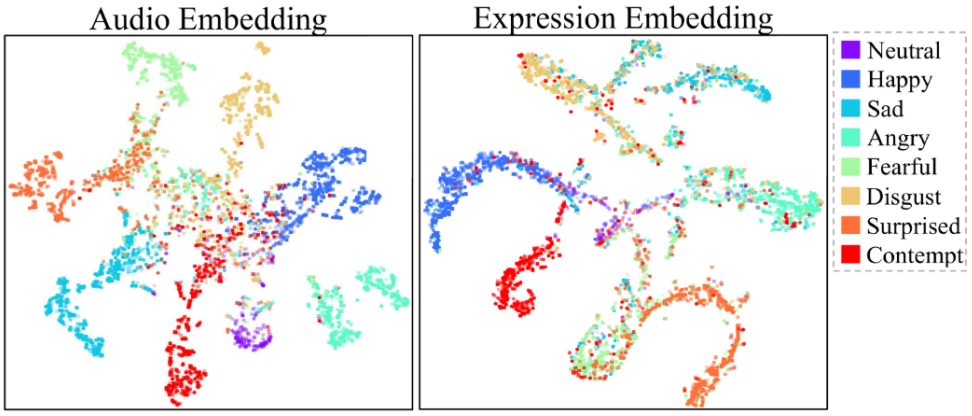


Figura 1.2: Raggruppamenti dei token di audio (sinistra) e dell'espressione del viso (destra), in base all'emozione espressa da ciascuno di essi.

[2]in ChatGPT3 la dimensione di embedding è pari a 12288 [7].

Inoltre, nell'Articolo [11] è riportata la Figura 1.3, la quale mostra come la differenza tra il vettore per la parola ***woman*** e quello per la parola ***man***, se sommato ad un nome maschile, lo trasforma nel corrispondente nome femminile. Viene infatti citato il famoso esempio, introdotto per la prima volta da Mikolov nell'Articolo [13], di
king - ***man*** + ***woman*** $\approx$ ***queen***. Similmente invece, il vettore differenza tra i vettori di un particolare nome femminile ed il suo corrispettivo maschile è molto simile al vettore ***woman-man***, suggerendo quindi una struttura matematica sottostante agli spazi semantici.

Viene sempre utilizzata la proiezione in 2 dimensioni per facilitare l'interpretazione geometrica tra queste somiglianze.

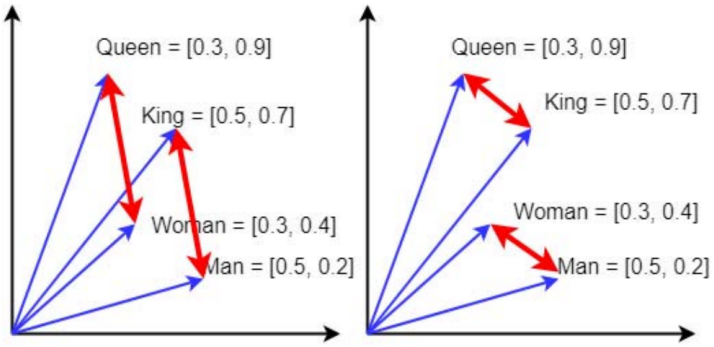


Figura 1.3: Rappresentazioni di parole che danno luogo a similarità.

Questo dà un'ulteriore esempio di come i vettori di embedding siano in grado di rappresentare parole e concetti diversi, e di come in $\mathbb{R}^{d_{in}}$ ogni direzione possa racchiudere un significato.

Il passo successivo nella preparazione dell'input è il *positional encoding*, il quale crea un vettore contenente informazioni sulla posizione relativa e assoluta dei token in una sequenza; quest'operazione è necessaria dal momento che il modello non ha accesso all'ordine della sequenza, dato che non vi è utilizzo di ricorrenza. Il vettore di positional encoding viene aggiunto ai singoli vettori di embedding di input all'inizio dell'encoder e del decoder, ed è progettato per avere la stessa dimensione degli embedding per consentirne la somma; a questo scopo, nei Transformers si utilizzano funzioni seno e coseno con frequenze diverse, per garantire una caratterizzazione unica di ciascun token.

Successivamente, nell'encoder, si trovano due *sub-layer*: uno di *Multi-Head Attention* e uno di *Feed Forward*, chiamato anche *Multi-Layer Perceptron*, abbreviato come *MLP*. In aggiunta, nel decoder è presente un terzo *sub-layer*, chiamato *Masked Multi-Head Attention*. Ciascuno di questi moduli ha il compito di arricchire i vettori di embedding, equipaggiandoli di informazioni più dettagliate, provenienti sia dagli altri token sia da quanto il modello ha imparato durante il processo di training.

A questo fine, in ciascuno di questi viene generato un vettore di dimensione d_{in} che viene sommato al vettore di input, ed infine viene applicata una normalizzazione.

Invece, la differenza principale tra l'operazione di *Attention* e quella di *MLP* risiede nel fatto che, nel primo caso, il vettore risultante dipende dall'intera sequenza di embedding, permettendo al modello di catturare le relazioni tra i token. Al contrario, nell'*MLP* le operazioni vengono eseguite indipendentemente su ogni singolo token, senza alcuna interazione diretta tra quest'ultimi. Questi concetti verranno esplorati più nel dettaglio e trattati ulteriormente nelle Sezioni 1.2 e 1.3.

1.2 Attention

L'Attention, introdotta per la prima volta da Bahdanau nell'Articolo [5], è un modello che effettua delle predizioni dipendenti dal contesto, ovvero modifica gli embedding in base ad un insieme di token visti precedentemente. Ossia, "muove informazioni da un embedding ad un altro", caratterizzando ciascun token con il contesto fornitogli da tutti gli altri.

Un ulteriore aspetto fondamentale è che l'operazione di mappatura viene eseguita in un'unica fase, tenendo in considerazione di tutte le possibili relazioni tra i token indipendentemente dalla loro distanza. Questo approccio differisce dai modelli precedenti, la cui struttura sequenziale non consentiva al sistema di imparare relazioni tra token distanti, poiché le informazioni degli elementi più vecchi tendevano a perdersi man mano che la rete si concentrava sugli ultimi token della sequenza. É importante sottolineare che la vicinanza tra le parole non è necessariamente un segno di relazione, ma è il significato contestuale che implica le correlazioni tra le parole.

Questa mappatura viene eseguita utilizzando delle *query* e delle *key*, ovvero dei vettori di dimensione d_k[3], che rappresentano rispettivamente delle richieste e delle risposte. Ad esempio, una possibile *query* può essere la richiesta da parte di un nome di "cercare" aggettivi che si riferiscono a lui, e la corrispondente *key* la risposta degli eventuali aggettivi a questa *query*. Altri esempi simili possono essere relazioni grammaticali come verbo-oggetto, nome-pronome, verbo-avverbio, o anche relazioni semantiche, come ad esempio causa-effetto, domanda-risposta o oggetto-utilizzo; in ogni caso, ogni token tramite le query guarda il contesto attorno a lui e cerca di capire che relazioni sussistono tra le parole.

Riguardo ciò, ad ogni singolo token corrisponde un vettore di query e uno di key, ottenuti moltiplicando a sinistra l'embedding per delle matrici parametriche Q e K, di dimensione $d_k \times d_{in}$. La moltiplicazione per Q e K per tutti i token può essere riscritta utilizzando le due matrici QX e KX, di dimensione $d_k \times n$: queste ultime hanno per colonne i singoli vettori di query e key, per ciascuno degli n token.

[3] in ChatGPT3 è di 128 [7].

1.2.1 Scaled dot-product Attention

Per studiare le relazioni tra i diversi embedding nell'Articolo [2] viene usata la *scaled dot-product Attention*: l'allineamento tra un vettore di key e uno di query in $\mathbb{R}^{d_k}$ significa che il contenuto della key è coerente con la richiesta effettuata dalla query: conseguentemente, il loro prodotto scalare è positivo e di modulo grande.

Al seguito del training delle matrici parametriche Q e K per ottenere i vettori di key e query, lo studio delle relazioni tra i vari token si riduce quindi al calcolo di prodotti scalari: a questo proposito si costruisce la matrice $(QX)^\top(KX)$, chiamata *Attention-Pattern*, dove nella riga i vi sono i prodotti scalari tra la query i e tutte le key da 1 a n. Ciascuno di questi valori indica quanto la key j, per $j \in \{1, \dots n\}$, è "rilevante" per la richiesta effettuata nella query i, ovvero quanto sono allineati i vettori. Inoltre, per evitare di ottenere entrate della matrice troppo grandi, ogni elemento viene diviso per $\sqrt{d_k}$; proprio per questo viene definita *scaled-Attention*.

A questo punto, si vuole ottenere una distribuzione di probabilità sulle key per ogni query, quindi si applica la funzione *softmax* alle righe dell'Attention-Pattern, che definiamo qui di seguito.

Definizione 1.1. Dati $(x_1, \dots, x_n) \in \mathbb{R}^n$, la funzione *softmax* è definita come:

$$\text{softmax}(x_1, \dots, x_n) = \left(\frac{e^{x_1}}{\sum_{j=1}^n e^{x_j}}, \dots, \frac{e^{x_n}}{\sum_{j=1}^n e^{x_j}} \right) \in [0,1]^n.$$

Applicando questa funzione alle righe della matrice Attention-Pattern si ottiene una distribuzione di probabilità su tutte le key, per ciascuna query. Si ottiene quindi che ciascun termine della matrice diventa non negativo, e la somma degli elementi di ciascuna riga è 1.

Quindi, questa funzione trasforma un insieme di n valori reali in una distribuzione di probabilità, mediante una trasformazione esponenziale: viene amplificata esponenzialmente la differenza tra gli x_i, ossia viene data molta più importanza ai termini di valore più grande. Quest'operazione è simile a considerare il massimo, ma viene fatto tramite l'esponenziale, che è C^∞.

Al fine di controllare come viene distribuita la probabilità, peò essere introdotto un parametro $T > 0$ di *temperatura* nella funzione *softmax*. Quindi, vale:

$$\text{softmax}_T(x_1, \dots, x_n) = \left(\frac{e^{x_1/T}}{\sum_{j=1}^n e^{x_j/T}}, \dots, \frac{e^{x_n/T}}{\sum_{j=1}^n e^{x_j/T}} \right) \in [0,1]^n.$$

In questa maniera, se T assume valori più grandi di 1, la distribuzione sarà più uniforme, mentre per $T < 1$ è concentrata maggiormente sui valori x_i più alti.

Vi è poi un'ulteriore matrice parametrica $V \in \mathbb{R}^{d_v \times d_{in}}$, dove d_v viene spesso scelto uguale a d_{in}. La sua funzione principale è quella di trasformare le informazioni contenute nei token attraverso un'operazione lineare, con l'obiettivo di arricchire gli embedding

con tutto contesto, al fine di produrre una rappresentazione utile per il calcolo finale. La matrice V è necessaria quindi affinché il prodotto VX contenga i dati effettivi da sommare al vettore di input, e può essere visto come una sintesi delle interazioni tra i token.

Tutti questi dati vengono poi ponderati in base alle probabilità ottenute tramite l'applicazione della funzione softmax all'Attention-Pattern.

Riassumendo tutte queste operazioni in una singola equazione si ottiene:

$$Attention(X) \coloneqq VX\ softmax\Big(\frac{X^\top Q^\top KX}{\sqrt{d_k}}\Big),$$

dove $X \in \mathbb{R}^{d_{in} \times n}$, mentre $Attention(X) \in \mathbb{R}^{d_v \times n}$.

1.2.2 Multi-Head Attention

Al fine di catturare le relazioni tra i token da diverse prospettive, questa operazione viene svolta in contemporanea in H[4] *heads* diverse, utilizzando matrici di query, key e value differenti.

Infine, i vettori di output di queste H operazioni vengono mappati e pesati tramite delle matrici $W_h \in \mathbb{R}^{d_{in} \times d_v}$, per $h \in \{1, \dots, H\}$, chiamate appunto matrici dei pesi, le quali sono anche necessarie a rendere $Attention(X)$ di dimensione $d_{in} \times n$. Dopodiché, le risultanti matrici vengono sommate ai token di partenza X, ovvero:

$$MAtt_\theta(X) \coloneqq X + \sum_{h=1}^{H} W_h \Big(V_\theta X\ softmax(\frac{X^\top Q_\theta^\top K_\theta X}{\sqrt{d_k}})\Big), \tag{1.2.1}$$

dove θ indica che le matrici parametriche sono differenti per ogni head. Infatti, le matrici Q_θ, K_θ e V_θ sono allenate separatamente e differentemente per ciascun θ, al fine di garantire che il modello sia in grado di considerare informazioni riguardo i token di diversa natura, cosa che non è possibile fare con una singola head di Attention.

[4] in ChatGPT3 è pari a 96 [7].

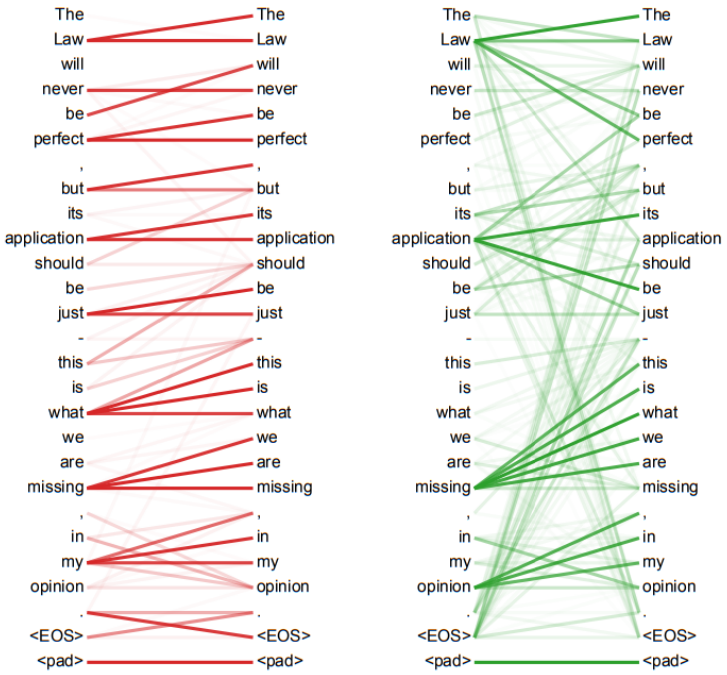


Figura 1.4: Esempio di relazioni individuate da due heads di Attention differenti [2].

Nella Figura 1.4 è rappresentata la correlazione tra diversi token all'interno di una frase, dove l'intensità del colore riflette l'importanza del legame tra ciascuna coppia di parole. É evidente come le due heads, contenute nell'Encoder, abbiano imparato ad eseguire task differenti e a prestare attenzione a dettagli diversi: nel primo caso la query di ogni token attribuisce un'importanza significativa alle key dei token immediatamente precedenti a quello in esame, mentre nel secondo caso questa dipendenza ravvicinata non è così esplicita.

Sono inoltre rappresentati dei token speciali, che indicano al modello la fine della frase: *EOS* vuol dire esattamente "End of sentence".

Esempio 1.1. Per illustrare il ruolo del meccanismo di Attention nella modifica contestuale degli embedding, si consideri come esempio la parola polisemica "**campo**". Se il contesto suggerisce che si sta parlando di una struttura algebrica dotata di due operazioni, allora l'embedding di **campo** verrà mappato in prossimità di vettori come quello di $\mathbb{R}$ o $\mathbb{Q}$, o di vettori associati a concetti come *anello* o *spazio vettoriale*.

Invece, in presenza di termini come *gradiente* o *flusso*, l'embedding verrrà modificato dall'Attention e mappato in un vettore simile a quello di *campo vettoriale*, o simile all'embedding di concetti geometrici come *funzione* o di *spazio euclideo*.

O ancora, se il modulo di Attention suggerisce che la parola **campo** si riferisce ad un contesto agricolo, l'embedding verrà mappato in un vettore in prossimità di quelli di *coltivazione* o di *campagna*.

1.2.3 Masked Attention

All'interno del decoder, il primo modulo è chiamato *Masked Multi-Head Attention*, come visto nella Figura iniziale 1.1, relativa all'architettura di un Transformer. Il termine *Masked* indica che, durante processo di training, ciascuna query può "prestare attenzione" soltanto alle key associate a token precedenti. Ciò è necessario per preservare la proprietà di auto-regressività, ovvero la caratteristica per cui ogni predizione si basa esclusivamente sui dati passati, senza accedere a informazioni future. Nell'Articolo [2], tale meccanismo viene realizzato *mascherando* ciascun valore al di sopra della diagonale nell'Attention-Pattern, ponendoli pari a $-\infty$. In questo modo, quando viene applicata la funzione $softmax$, tutte le posizioni (i, j) tale che $i < j$ vengono poste a 0, così che le keys successive non possano influenzare la query corrente.

Questa operazione è nota anche come *Masking causale*, in quanto impedisce a un token di essere influenzato da posizioni future. In tal modo, il modello mantiene la corretta direzionalità temporale, preservando l'auto-regressività e assicurando che l'informazione non fluisca dai token successivi a quelli precedenti.

1.3 FeedForward

A seguito del layer di Attention vi è l'operazione di *Multi-Layer Perceptron*, solitamente abbreviato come *MLP*. Esso consiste in un network chiamato *FeedForward*, spesso utilizzato in Machine Learning. Come già brevemente introdotto la principale differenza con la procedura di Attention è che il layer di *MLP* agisce simultaneamente ed indipendentemente su ogni singolo token, in maniera separata; significa che ogni token viene trasformato tramite lo stesso processo in un vettore di output, senza alcuna dipendenza dal contesto fornito dagli altri token.

Definizione 1.2 (FeedForward)**.** Un *FeedForward Network* è una funzione definita come la composizione di una serie di L trasformazioni lineari e non lineari, e caratterizzata dall'assenza di retroazioni (feedback). Più formalmente, dato un vettore di input $x \in \mathbb{R}^n$, la rete implementa

$$f(x) = f_L \circ f_{L-1} \circ \cdots \circ f_1(x),$$

dove per ogni layer si ha l'applicazione di due trasformazioni lineari, e di una funzione di attivazione nel mezzo, introdotta nella Definizione 1.3. L'assenza di cicli nella struttura implica che l'informazione fluisce unicamente in avanti, dall'input all'output.

Ogni layer è composto da diversi nodi, chiamati *neuroni*, e ognuno di questi è connesso ai neuroni dello strato successivo tramite un *arco pesato*. Il segnale in ciascun arco è un numero reale, e l'output di ciascun neurone è calcolato da una funzione non

lineare della somma *pesata* di tutti i suoi input, ovvero di tutti i segnali provenienti dal layer precedente.

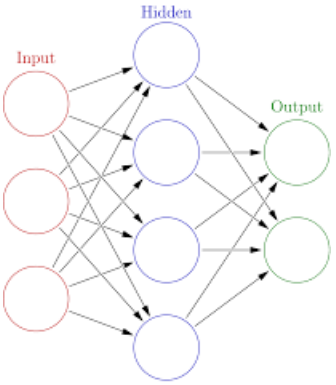


Figura 1.5: Struttura di una rete FeedForward

Formalmente, un neurone riceve in input il vettore x da un neurone del layer precedente, che viene moltiplicato per la matrice dei pesi $W_1 \in \mathbb{R}^{d_{in} \times d_{hid}}$, dove d_{hid} viene spesso scelto pari a $4d_{in}$, che è una matrice parametrica ed allenabile.

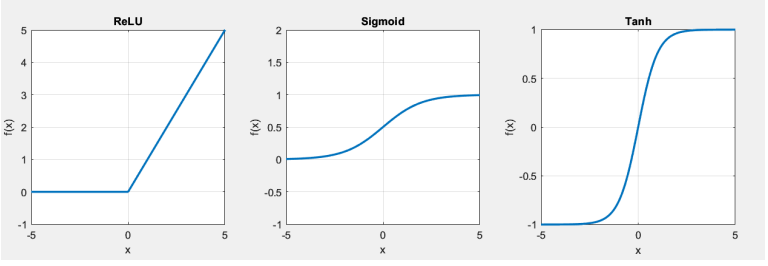
Inoltre, si somma un vettore $b_1 \in \mathbb{R}^{d_{hid}}$, chiamato *bias*, che rappresenta una traslazione in una determinata direzione.

Definizione 1.3 (Funzione di Attivazione)**.** Una funzione di attivazione è una funzione non lineare applicata all'output del neurone dopo la somma pesata degli input e l'aggiunta del bias.

Questa operazione è fondamentale perché, introducendo la non-linearità, consente alla rete di apprendere relazioni complesse e di approssimare funzioni non lineari.

Vi sono diversi esempi di funzioni di attivazione:

- $ReLU$, definita come $ReLU(x) = max(0, x)$, che è la più utilizzata;
- la funzione sigmoidale, definita come $\sigma(x) = \frac{1}{1+e^{-x}}$;
- la tangente iperbolica, definita come $tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$.



Successivamente, $f(xW_1 + b_1)$ viene mappato in $\mathbb{R}^{d_v}$ moltiplicandolo a sinistra per $W_2 \in \mathbb{R}^{d_{hid} \times d_v}$, a cui sommo un secondo bias $b_2 \in \mathbb{R}^{d_v}$. Queste operazioni possono essere riassunte in

$$MLP(x) = max(0, xW_1 + b_1)W_2 + b_2,$$

dove viene usata come funzione di attivazione $ReLU$.

Nel contesto dei Tranformers, i layer di FeedForward sono necessari ad arricchire ciascun embedding con informazioni già note alla rete neurale, che sono state apprese durante il processo di training.

Mentre l'Attention effettua principalmente operazioni lineari, la funzione di attivazione in questo contesto introduce non-linearità, permettendo quindi alla rete l'approssimazione di funzioni complesse. Inoltre, l'espansione dimensionale ($d_\text{v} \rightarrow 4d_\text{v}$) permette di modellare più concetti e significati dei singoli token.

Esempio 1.2. In generale, la rete FeedForward è in grado di arricchire gli embedding con informazioni aggiuntive. Di seguito riportiamo l'esempio del nome **Kolmogorov**, simile a quanto descritto nella fonte [12].

Il *Multi-Layer Perceptron*, attraverso l'apprendimento di una vasta gamma di concetti, è in grado di integrare nell'embedding del nome **Kolmogorov** componenti legate alla teoria della probabilità, come ad esempio gli *assiomi di Kolmogorov* o la *legge 0-1*. In aggiunta, la rete arricchisce l'embedding **Kolmogorov** con informazioni che codificano l'Università di Mosca, l'anno di nascita 1903, e la vincita del Premio Wolf.

Di conseguenza, l'output dell'MLP risulta essere un vettore arricchito di informazioni che la rete è stata in grado di incorporare grazie al processo di allenamento.

1.4 Relazione tra encoder e decoder

Al seguito del passaggio attraverso gli L layer nell'encoder (Figura 1.1), gli embedding di input vengono trasformati ed equipaggiati del contesto fornitogli dagli altri token attraverso il processo di Attention e di FeedForward.

Il decoder, utilizza questa memoria attraverso il meccanismo di *Cross-Attention*. Ovvero, in ogni suo layer fa uso delle informazioni provenienti dall'encoder al fine di arricchire i vettori di embedding con informazioni aggiuntive.

Utilizzando queste informazioni e riprocessandole attraverso ulteriori layer di Attention e di *MLP*, il decoder genera successivamente una distribuzione di probabilità a partire dalla quale viene selezionato il token successivo da aggiungere ai token in input.

Capitolo 2

Fondamenti Matematici dei Transformers e Token Arbitrari

Un'estensione naturale degli argomenti finora trattati consiste nell'esaminare il caso in cui il numero di token in input sia arbitrario. Tale configurazione è stata analizzata in numerosi studi, ad esempio nell'Articolo [1] di Gabriel Peyré, dove si indaga la capacità dell'architettura Transformer di gestire simultaneamente un numero potenzialmente infinito di token.

Per fare ciò nell'Articolo [1] viene utilizzato un Transformer *profondo*, ovvero formato da molti layer, costituito da un numero di heads H finito, e che utilizza una dimensione di embedding d_{in} anch'essa finita.

É evidente che la struttura matematica finora descritta, composta prevalentemente da matrici, risulta inadeguata per affrontare questo caso limite; infatti, ad esempio, la matrice *Attention-Pattern* ha dimensione $n \times n$. In questo capitolo verranno descritti i cambiamenti necessari a estendere la trattazione al caso continuo.

2.1 Riformulazione della funzione di Attention

Abbiamo definito l'Attention come un modello che effettua predizioni basate sul contesto, ossia una mappatura da token a token, che dipende dall'insieme di token precedentemente osservati: per questa ragione viene anche chiamata *in context mapping*. Come sopra descritto la lunghezza del contesto può essere arbitrariamente lunga, il che rappresenta il focus principale di questo capitolo.

Al fine di effettuare un'analisi rigorosa di tale situazione e caratterizzare il comportamento medio di un insieme di token, vedremo come risulterà conveniente considerare l'Attention come un operatore che agisce direttamente sulle distribuzioni di probabilità associate ai token, rispetto che sui token stessi.

Per arrivare a ciò, in primo luogo riscriviamo la funzione di Attention $A_\theta(X, \cdot)$ applicata ad un singolo token x_i, dove, come nel capitolo precedente, X denota l'insieme

di tutti i token, e dunque il contesto. Formalmente, per ogni x_i definiamo

$$A_\theta(X,x) \;:=\; x \;+\; \sum_{h=1}^{H} W_h \sum_{j=1}^{n} \frac{\exp\Big(\frac{1}{\sqrt{d_k}}\,\langle Q_h\,x,\; K_h\,x_j\rangle\Big)}{\sum_{\ell=1}^{n}\exp\Big(\frac{1}{\sqrt{d_k}}\,\langle Q_h\,x,\; K_h\,x_\ell\rangle\Big)}\,V_h\,x_j, \tag{2.1.1}$$

per il quale vale

$$MAtt_\theta(X) = (A_\theta(X,x_i))_{i=1}^n,$$

dove $MAtt_\theta(X)$ è dato dalla formula (1.2.1) per il calcolo dell'output dell'Attention, definita nel Capitolo 1. Questa formulazione è esattamente la stessa descritta precedentemente, ma riscritta per un singolo vettore x, al fine di mettere in evidenza come ogni token venga trasformato in funzione dell'intero contesto.

Invece, nel caso *masked*, i token non sono invarianti per permutazione, dato che deve essere garantita l'auto-regressività. Pertanto, è necessario tenere traccia anche dell'indice i associato al token x, poiché si intende che soltanto i token precedenti a quello in esame possano influenzarlo.

In tal senso, definiamo la funzione di Attention come

$$A_\theta(X,x,i) \;:=\; x \;+\; \sum_{h=1}^{H} W_h \sum_{j=1}^{i} \frac{\exp\Big(\frac{1}{\sqrt{d_k}}\,\langle Q_h\,x,\; K_h\,x_j\rangle\Big)}{\sum_{\ell=1}^{i}\exp\Big(\frac{1}{\sqrt{d_k}}\,\langle Q_h\,x,\; K_h\,x_\ell\rangle\Big)}\,V_h\,x_j. \tag{2.1.2}$$

La sommatoria è limitata all'indice i, il che equivale a considerare esclusivamente i termini al di sotto della diagonale, come descritto nella Sezione riguardante la Masked Attention 1.2.3. Anche in questo caso vale:

$$MAtt_\theta(X) = (A_\theta(X,x_i,i))_{i=1}^n.$$

2.1.1 Composizione di mappe

Abbiamo descritto un Transformer come la composizione di L strati, ciascuno dei quali è costituito da un modulo di Attention e da un Multi-layer Perceptron, ovvero:

$$MLP_{\xi_L} \circ MAtt_{\theta_L} \circ \ldots \circ MLP_{\xi_1} \circ MAtt_{\theta_1}, \tag{2.1.3}$$

dove MLP_ξ opera su ciascun token in modo separato e indipendente, ossia, usando la notazione introdotta sopra, è indipendente dal contesto X:

$$MLP_\xi(X,x) := F_\xi(x), \quad \text{per ogni } x \in \mathbb{R}^{d_{in}}.$$

Invece, la composizione di mappe nella forma espressa in (2.1.1) segue la regola seguente, che denotiamo con $\diamond$:

$$(A_2 \diamond A_1)(X,x) := A_2(X_1, A_1(X,x)) \quad \text{dove} \quad X_1 := (A_1(X,x_i))_{i=1}^n. \tag{2.1.4}$$

Infatti, X_1 è il contesto che entra in ingresso in A_2, modificato dalla trasformazione A_1, mentre $A_1(X, x)$ indica il vettore x trasformato dal modulo A_1.

Similmente vale nel caso masked:

$$(A_2 \diamond A_1)(X, x, i) := A_2(X_1, A_1(X, x, i), i) \quad \text{dove} \quad X_1 := (A_1(X, x_i, i))_{i=1}^n. \tag{2.1.5}$$

Questa regola è applicabile sia che $A_1(X, \cdot)$ o $A_2(X, \cdot)$ dipendano o meno dal contesto X; quindi, la definizione riportata in (2.1.3) per la composizione di mappe diventa

$$F_{\xi_L} \diamond A_{\theta_L} \diamond \ldots \diamond F_{\xi_1} \diamond A_{\theta_1}. \tag{2.1.6}$$

2.1.2 Passaggio alla rappresentazione probabilistica

É importante notare come la formula per l'Attention del capitolo precedente (1.2.1) perda significato per $n \to \infty$, poiché diverse matrici utilizzate sono di taglia n. Per superare tale problema, si utilizza la formula dell'Attention applicata ad un singolo token x modificato dal contesto X (2.1.1), sostituendo all'insieme dei token X la distribuzione di probabilità μ su $\mathbb{R}^{d_{in}}$, che indica come sono distribuiti i token in $\mathbb{R}^{d_{in}}$.

In questo modo la mappa diventa, per ogni $(\mu, x) \in \mathcal{P}(\mathbb{R}^{d_{in}}) \times \mathbb{R}^{d_{in}}$,

$$\Gamma_\theta(\mu, x) := x + \sum_{h=1}^{H} W_h \int \frac{\exp\left(\frac{1}{\sqrt{d_k}} \langle Q_h x, K_h y \rangle\right)}{\int \exp\left(\frac{1}{\sqrt{d_k}} \langle Q_h x, K_h z \rangle\right) d\mu(z)} V_h y \, d\mu(y), \tag{2.1.7}$$

dove il numero H di heads e la dimensione di embedding d_{in} sono fissati, e l'integrale appare come limite della sommatoria per $n \to \infty$.

Il caso discreto è comunque contenuto in questa definizione più generale, considerando la misura empirica discreta:

$$\mu_n = \frac{1}{n} \sum_{i=1}^{n} \delta_{x_i} \in \mathcal{P}(\mathbb{R}^{d_{in}}). \tag{2.1.8}$$

Da qui in avanti si utilizzerà la notazione $\Gamma(\mu)(x) := \Gamma(\mu, x)$, per indicare la mappa da $\mathbb{R}^{d_{in}}$ a $\mathbb{R}^{d_v}$. In questa scrittura μ viene considerata come un parametro, e la mappa $\Gamma_\theta(\mu)$ diventa una mappa tra spazi euclidei.

Tramite questa nuova notazione, la mappa di Attention può essere interpretata come uno spostamento delle posizioni dei token, che corrisponde ad applicare un *push-forward*, definito nell'Appendice A.1.1, alla misura μ:

$$\mu \in \mathcal{P}\big(\mathbb{R}^{d_{\text{in}}}\big) \longmapsto \Gamma_\theta(\mu)_{\#}\,\mu \in \mathcal{P}\big(\mathbb{R}^{d_{\text{v}}}\big).$$

Analogamente, la composizione di mappe definita in (2.1.4) per il caso discreto si estende al contesto delle misure (nella configurazione unmasked) come

$$(\Gamma_2 \diamond \Gamma_1)(\mu, x) := \Gamma_2(\mu_1, \Gamma_1(\mu, x)), \quad \text{dove} \quad \mu_1 := \Gamma_1(\mu)_\sharp \mu. \tag{2.1.9}$$

Sostituendo quindi questa nuova definizione dei layer di Attention, la formula (2.1.6) per la composizione con $\diamond$ diventa

$$F_{\xi_L} \diamond \Gamma_{\theta_L} \diamond \ldots \diamond F_{\xi_1} \diamond \Gamma_{\theta_1}.$$

In questa scrittura, le mappe F_ξ sono indipendenti, oltre che dal contesto, anche dalla misura μ.

Nella configurazione masked, per il passaggio alle misure è necessario interpretare il corrispondente caso continuo della condizione sulla matrice di Attention-Pattern $M_{i,j} = 0$ se $j > i$: a questo scopo si utilizza la funzione indicatrice $1_{j \leq i}$, che vale 1 soltanto se $j \leq i$, e 0 altrimenti.

Inoltre, al fine di codificare in maniera esplicita l'ordine dei token all'interno del contesto e mantenere l'auto-regressività, si introduce una componente temporale t_i: ad esempio, se è noto un upper bound L del numero di token n, si può porre $t_i := \frac{i}{L}$. In maniera più generale, l'input assume la forma $\{x_i, t_i\}_{i=1}^n$, dove i t_i sono scelti in maniera arbitraria in $[0,1]$, permettendo così di trattare anche il caso di un numero infinito di token.

Di conseguenza, il contesto è codificato mediante una misura *spazio-temporale* $\mu \in \mathcal{P}(\mathbb{R}^{d_{in}} \times [0,1])$. Analogamente a quanto fatto nella formula dell'Attention per misure di probabilità (2.1.7), introduciamo la mappatura in-context, definita per ogni $(x,t) \in \mathbb{R}^{d_{in}} \times [0,1]$ come

$$\Gamma_\theta(\mu, x, t) := x + \sum_{h=1}^{H} W_h \int \frac{\exp\Big(\frac{1}{\sqrt{d_k}} \langle Q_h x, K_h y \rangle\Big) 1_{[0,t]}(r)}{\int \exp\Big(\frac{1}{\sqrt{d_k}} \langle Q_h x, K_h z \rangle\Big) 1_{[0,t]}(s) d\mu(z,s)} V_h y \, d\mu(y,r), \quad (2.1.10)$$

dove l'auto-regressività è mantenuta dal fatto che viene integrata la funzione indicatrice fino a t, che è passato in input, che corrisponde ad avere come estremi di integrazione 0 e t.

Il caso discreto è comunque contenuto in questa formulazione generale, adottando la misura empirica discreta mostrata in (2.1.8).

La composizione di mappe per misure viene eseguita similmente al caso non mascherato (2.1.9) appena descritto, ad eccezione del fatto che la variabile temporale rimane invariata, mentre il *push-forward* viene applicato soltanto sullo spazio, ossia

$$(\Gamma_2 \diamond \Gamma_1)(\mu, x, t) \;:=\; \Gamma_2\big(\mu_1,\; \Gamma_1(\mu, x, t),\; t\big), \quad \text{dove} \quad \mu_1 \;:=\; \big(\Gamma_1(\mu),\; \mathrm{Id}_{\mathbb{R}}\big)_{\#}\, \mu. \quad (2.1.11)$$

In questa definizione, $(\Gamma_1(\mu),\ \mathrm{Id}_{\mathbb{R}}) : (x,t) \in \mathbb{R}^{d_{in}+1} \to \Big(\Gamma_1(\mu)(x,t), t\Big) \in \mathbb{R}^{d_{in}+1}$, ovvero il push-forward viene applicato solamente sulle prime d_{in} componenti, nonché il tempo rimane invariato.

2.2 Teoremi di Universalità

Nell'Articolo [1] il risultato principale è espresso dal seguente teorema, che sintetizza il concetto di *universalità*, ovvero la capacità dei Transformers di approssimare qualsiasi funzione continua definita su un dominio compatto.

Teorema 2.1 (Universalità)**.** *Sia $\Omega \subset \mathbb{R}^d$ un insieme compatto e sia $\Lambda^* : \mathcal{P}(\Omega) \times \Omega \to \mathbb{R}^{d'}$ una funzione continua, dove $\mathcal{P}(\Omega)$ è dotato della topologia debole*[1]*. Allora, per ogni $\varepsilon > 0$, esistono un intero L e dei parametri $(\theta_\ell, \xi_\ell)_{\ell=1}^{L}$ tali che*

$$\forall (\mu, x) \in \mathcal{P}(\Omega) \times \Omega, \quad \left| \left(F_{\xi_L} \diamond \Gamma_{\theta_L} \diamond \cdots \diamond F_{\xi_1} \diamond \Gamma_{\theta_1}(\mu, x) \right) - \Lambda^*(\mu, x) \right| \leq \varepsilon,$$

con $d_{\text{in}}(\theta_\ell) \leq d + 3\,d'$, $\quad d_k(\theta_\ell) = d_{\text{v}}(\theta_\ell) = 1$, $\quad H(\theta_\ell) \leq d'$.

Si osservi innanzitutto che, nel Teorema 2.1, per ciascun layer di Attention $\ell \in \{1, \ldots, L\}$ valgono le seguenti restrizioni sui parametri θ_ℓ:

- La dimensione dell'embedding in ingresso $d_{\text{in}}(\theta_\ell)$ è controllata superiormente da $d + 3\,d'$.
- Il numero di teste $H(\theta_\ell)$ è limitato da d'.
- Le dimensioni delle key e delle query, nonché il numero di righe della matrice dei value, sono fissate a 1: $d_k(\theta_\ell) = d_v(\theta_\ell) = 1$.

  Ciò implica che, in ciascuna head, query e key sono scalari e l'output abbia dimensione unitaria.

Il punto chiave di questo risultato è che l'approssimazione fornita dall'architettura è indipendente da n, ed anzi, funziona anche per un numero di token infinito.

Invece, una possibile debolezza di questo risultato è che non vi è alcun controllo sul numero di layer L, che anzi cresce proporzionalmente con la dimensione di output d'.

In aggiunta, non vi è alcuna dipendenza esplicita tra questi bound e il valore di ϵ, ossia, non sono specificate le dimensioni necessarie di una rete neurale per ottenere una certa precisione nell'approssimazione.

2.2.1 Universalità per una Rete FeedForward

In questa sezione verranno presentati formalmente tre lemmi preparatori ed una versione alternativa del Teorema 2.1, basandosi su quanto dimostrato nella breve nota [3].

Il risultato esposto in questa sezione riguarda una rete FeedForward composta da quattro layer totali, di cui tre *hidden* e l'ultimo di output. Il teorema che verrà dimostrato differisce dal teorema di universalità di Peyré appena enunciato, in quanto

[1] Topologia definita a partire dalla nozione di convergenza definita in A.2.

la struttura presentata non comprende i layer di Attention. Piuttosto, si tratta di un risultato più debole di quelli già ottenuti negli articoli [9] e [10], dove è stata dimostrata l'universalità per una rete composta da due layer totali, uno *hidden* ed un altro di output. Ciononostante, la semplicità delle nozioni matematiche utilizzate rende questo risultato interessante, dato che si avvale soltanto di strumenti di analisi matematica e topologia di base.

Nelle seguente sezione si vuole dimostrare che l'insieme di queste reti neurali è denso nello spazio $C(K)$, ossia l'insieme delle funzioni continue a valori reali da un insieme compatto $K \subset \mathbb{R}^n$, rispetto alla *norma uniforme*, la quale sarà introdotta successivamente nella Definizione 2.5.

Come funzione di attivazione, già brevemente introdotta nella Definizione 1.3, viene utilizzata una funzione continua crescente e limitata, presentata nella seguente definizione.

Definizione 2.1. Una funzione σ si dice *di compressione* se è crescente, continua e $\lim_{x\to-\infty} \sigma(x) = 0$ e $\lim_{x\to\infty} \sigma(x) = 1$.

Ad esempio, la funzione sigmoidale definita nel Capitolo precedente, è una funzione di compressione. Infatti, da qui in avanti i termini *di compressione* e *sigmoidale* saranno usati come sinonimi.

Definizione 2.2. Sia $K \subset \mathbb{R}^n$ un sottoinsieme. Si definisce $C(K)$ l'insieme delle funzioni continue definite su K e a valori in $\mathbb{R}$.

Da qui in avanti per K si intenderà sempre un insieme *compatto*.

Definizione 2.3. Sia $n \in \mathbb{N}$, *l'insieme delle funzioni affini* di $x_1, \dots, x_n \in \mathbb{R}$ è:

$$\mathcal{N}_1 = \{f \in C(K) \ : \ f(x_1, \dots, x_n) = a_0 + a_1x_1 + \cdots + a_nx_n, \text{ per certi } a_0, \dots, a_n \in \mathbb{R}\},$$

mentre $\mathcal{N}_1^\sigma = \{F \in C(K) \ : \ F = \sigma \circ f, \quad \text{per } f \in \mathcal{N}_1\}$ rappresenta *l'insieme delle possibili funzioni di output dopo un Layer di FeedForward.*

Più in generale, dato $k \geq 1$ sia:

$$\mathcal{N}_{k+1} = \{g \in C(K) \ : \ g = a_0 + a_1F_1 + \cdots + a_mF_m, \text{ con } F_1, \dots, F_m \in \mathcal{N}_k^\sigma, \ a_0, \dots, a_m \in \mathbb{R}\},$$

grazie al quale definisco:

Definizione 2.4. L'insieme dei possibili output nel Layer $k+1$ è

$$\mathcal{N}_{k+1}^\sigma = \{G \in C(K) \ : \ G = \sigma \circ g, \quad \text{per qualche } g \in \mathcal{N}_{k+1}\},$$

che rappresenta *l'insieme delle possibili funzioni di output dopo* $k+1$ *Layer di FeedForward*, di cui k *hidden* e l'ultimo di *output.*

Valgono poi i seguenti:

Lemma 2.1. *Dato $k \in \mathbb{N}$, vale $\mathcal{N}_k^\sigma \subset \mathcal{N}_{k+1}$.*

Dimostrazione. Sia $F \in \mathcal{N}_k^\sigma$; allora $F = 0 + 1 \cdot F$ è una combinazione lineare di elementi in $\mathcal{N}_k^\sigma$ a coefficienti reali, ovvero $F \in \mathcal{N}_{k+1}$. □

Lemma 2.2. *Dato $k \in \mathbb{N}$ e $g_1, g_2 \in \mathcal{N}_k$, allora $a_0 + a_1 g_1 + a_2 g_2 \in \mathcal{N}_k$, per ogni $a_0, a_1, a_2 \in \mathbb{R}$.*

Dimostrazione. Siano $g_1, g_2 \in \mathcal{N}_k$. Per la definizione di $\mathcal{N}_k$, sappiamo che g_1 e g_2 sono combinazioni lineari di elementi in $\mathcal{N}_k^\sigma$, ovvero

$$g_1 = \sum_{i=0}^{m_1} b_i F_i^{(1)} \quad \text{e} \quad g_2 = \sum_{j=0}^{m_2} c_j F_j^{(2)},$$

dove $F_i^{(1)}, F_j^{(2)} \in \mathcal{N}_k^\sigma$ e $b_i, c_j \in \mathbb{R}$.

Consideriamo ora una combinazione lineare di g_1 e g_2:

$$a_0 + a_1 g_1 + a_2 g_2 = a_0 + a_1 \left(\sum_{i=0}^{m_1} b_i F_i^{(1)} \right) + a_2 \left(\sum_{j=0}^{m_2} c_j F_j^{(2)} \right).$$

Distribuendo i coefficienti reali a_1 e a_2, otteniamo:

$$a_0 + a_1 g_1 + a_2 g_2 = a_0 + \sum_{i=0}^{m_1} a_1 b_i F_i^{(1)} + \sum_{j=0}^{m_2} a_2 c_j F_j^{(2)}.$$

Poiché $F_i^{(1)}, F_j^{(2)} \in \mathcal{N}_k^\sigma$, la combinazione lineare di questi elementi è ancora un elemento di $\mathcal{N}_k^\sigma$. Ciò implica che

$$a_0 + a_1 g_1 + a_2 g_2 \in \mathcal{N}_k,$$

che è la tesi. □

Ora, vogliamo inizialmente mostrare che σ separa punti in $\mathbb{R}^n$ in senso forte, poi che $\mathcal{N}_2$ separa punti da insiemi chiusi in una maniera simile, e successivamente che $\mathcal{N}_3$ separa chiusi da chiusi. Infine, verrà data una dimostrazione per il teorema di universalità usando l'ultimo risultato su $\mathcal{N}_3$.

Lemma 2.3. *Siano x_0 e x_1 numeri reali distinti. Per ogni $\epsilon > 0$ esistono $s, t \in \mathbb{R}$ tali che $\sigma(s + t x_0) < \epsilon$ e $\sigma(s + t x_1) > 1 - \epsilon$.*

Inoltre, se $x_0 < x_1$ e $\epsilon < 1/2$, allora $\sigma(s + tx) < \epsilon$ nell'intervallo $(-\infty, x_0]$ e $\sigma(s + tx) > 1 - \epsilon$ nell'intervallo $[x_1, \infty)$.

Dimostrazione. Senza perdita di generalità, possiamo assumere $\epsilon < 1$. Per definizione di funzione sigmoidale, esistono $y_0, y_1 \in \mathbb{R}$ tali che $\sigma(y_0) = \epsilon/2$ e $\sigma(y_1) = 1 - \epsilon/2$. Dato che $x_1 - x_0 \neq 0$, il sistema lineare

$$\begin{pmatrix} 1 & x_0 \\ 1 & x_1 \end{pmatrix} \begin{pmatrix} s \\ t \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \end{pmatrix},$$

ha soluzione, che indico con $(s_0, t_0)^T$. Quindi,

$$\sigma(s_0 + t_0 x_0) = \sigma(y_0) = \epsilon/2 < \epsilon, \quad \text{e}$$

$$\sigma(s_0 + t_0 x_1) = \sigma(y_1) = 1 - \epsilon/2 > 1 - \epsilon.$$

Si supponga inoltre che $x_0 < x_1$ e $\epsilon < 1/2$. Poiché $\sigma(s + tx)$ è monotona e $\sigma(s + tx_0) < \epsilon < 1/2 < 1 - \epsilon < \sigma(s + tx_1)$, segue che $\sigma(s + tx)$ è crescente, che mostra la seconda parte del lemma. □

Lemma 2.4. *Sia $B \subset K$ un insieme chiuso contenuto in un compatto, e $x_0 \in K \setminus B$. Per ogni $\epsilon > 0$ esiste $g \in \mathcal{N}_2$ tale che $g > 1 - \epsilon$ su B e $g(x_0) < \epsilon$.*

Dimostrazione. Senza perdita di generalità si può assumere che $0 < \epsilon < 1/3$. Sia $b \in B$. Poiché $x_0 \in K \setminus B$, allora $b \neq x_0$, quindi per il Lemma 2.3 esiste una funzione $f_b \in \mathcal{N}_1$ tale che $f_b(x_0) < \epsilon/2$ e $f_b(b) > 1 - \epsilon/2$. Sia

$$U_b = \{x \in K \ : \ f_b(x) > 1 - \epsilon\}.$$

Poiché f_b è continua, U_b è un insieme aperto. Inoltre, $b \in U_b$, quindi $\{U_b\}_{b \in B}$ è un ricoprimento aperto di B, che è compatto in quanto insieme chiuso contenuto in un compatto, dimostrato nell'Appendice A.1. Per definizione di compattezza esistono $b_1, \ldots, b_N \in B$ tali che $\{U_{b_1}, \ldots, U_{b_N}\}$ ricoprono B[2].

Sempre per il Lemma 2.3, esistono $s, t \in \mathbb{R}$ tali che $\sigma(s + tx) < \epsilon/N$ su $(-\infty, \epsilon)$ e $\sigma(s + tx) > 1 - \epsilon$ su $(1 - \epsilon, \infty)$. Per ogni $j \in \{1, \ldots, N\}$, definiamo la funzione $F_j = \sigma(s + t f_{b_j})$.

Poiché $f_{b_j}(x_0) < \epsilon/2$ per costruzione, abbiamo $F_j(x_0) < \epsilon/N$, dato che $F_j(x_0) = \sigma(s + t f_{b_j}(x_0)) < \sigma(s + t\epsilon/2) < \epsilon/N$.

Similmente, $F_j \geq 1 - \epsilon$ su U_{b_j}, poiché $f_{b_j}(x) > \epsilon/2$ per $x \in U_{b_j}$, e quindi $F_j(x) \geq 1 - \epsilon$. Inoltre, per costruzione, $F_j \in \mathcal{N}_1^\sigma$.

Definiamo infine $g = \sum_{j=1}^N F_j$: vale innanzitutto che $g \in \mathcal{N}_2$, poi che $g(x_0) < \epsilon$, poiché ciascuna $F_j(x)$ è minore di ϵ/N. Per concludere mostriamo che $g \geq 1 - \epsilon$ su B: dato che $\{U_{b_1}, \ldots, U_{b_N}\}$ è un ricoprimento di B, per ogni $x \in B$ esiste $j \in \{1, \ldots, N\}$ tale che $x \in U_{b_j}$. In tal caso, $F_j(x) \geq 1 - \epsilon$. Inoltre, per ogni $i \neq j$, $F_i(x) \geq 0$ per positività di σ. Pertanto, $g(x) \geq 1 - \epsilon$ per ogni $x \in B$. □

Lemma 2.5. *Siano A e B due sottoinsiemi chiusi disgiunti di K. Allora, per ogni $\epsilon > 0$,*

(i) esiste $h \in \mathcal{N}_3$ tale che $h < \epsilon$ su B e $h > 1 - \epsilon$ su A,

(ii) esiste $H \in \mathcal{N}_3^\sigma$ tale che $0 \leq H < \epsilon$ su B e $1 - \epsilon < H \leq 1$ su A.

[2] Ovvero per ogni $x \in B$ esiste $i \in \{1, \ldots, N\}$ tale che $x \in U_{b_i}$.

Dimostrazione. Come sopra, senza perdita di generalità, assumiamo che $\epsilon \in (0, 1/3)$. Per ogni $a \in A$, per il Lemma 2.4, esiste $\widetilde{g}_a \in \mathcal{N}_2$ tale che $\widetilde{g}_a > 1 - \epsilon/2$ su B e $\widetilde{g}_a(a) < \epsilon/2$. Sia $g_a = 1 - \widetilde{g}_a$, così vale che $g_a < \epsilon/2$ su B, e $g_a(a) > 1 - \epsilon/2$. Sia

$$U_a = \{x \in K \ : \ g_a(x) > 1 - \epsilon\}.$$

Come nella dimostrazione precedente, per continuità di g_a ogni U_a è aperto. E poiché $a \in U_a$, segue che $\{U_a\}_{a \in A}$ è un ricoprimento aperto dell'insieme compatto A. Siano $a_1, \ldots, a_N \in A$ tale che $\{U_{a_1}, \ldots, U_{a_N}\}$ ricoprono A.

Per il Lemma 2.3 esistono $s, t \in \mathbb{R}$ tali che $\sigma(s+tx) < \epsilon/N$ su $(-\infty, \epsilon)$ e $\sigma(s+tx) > 1 - \epsilon$ su $(1 - \epsilon, \infty)$.

Definisco $h = \sum_{j=1}^{N} \sigma(s + t g_{a_j})$, notiamo subito che $h \in \mathcal{N}_3$. In aggiunta, se $a \in A$, allora $a \in U_{a_k}$ per un certo k, segue quindi che $g_{a_k}(a) > 1 - \epsilon$ e sempre per positività di σ vale che $h(a) > 1 - \epsilon$. Invece, se $b \in B$ vale per costruzione che $g_{a_j}(b) < \epsilon/2$ per ogni j, ovvero $\sigma(s + g_{a_j}(b)) < \epsilon/N$ per ogni j e quindi $h(b) < \epsilon$. Questo dimostra il punto (i).

Per dimostrare il secondo punto, dato che σ è crescente, per il Lemma 2.3 esistono $s, t \in \mathbb{R}$ tali che $\sigma(s + tx) < \epsilon$ sull'intervallo $(-\infty, \epsilon)$ e $\sigma(s + tx) > 1 - \epsilon$ sull'intervallo $(1 - \epsilon, \infty)$. Allora $H = \sigma(s + th)$ ha le proprietà richieste. □

Definizione 2.5. Sia K un sottoinsieme compatto di $\mathbb{R}^n$ e $f : K \longrightarrow \mathbb{R}$ una funzione continua. La *norma uniforme* di f, denotata $\|f\|_u$, è definita come

$$\|f\|_u = \sup\{|f(x)| : x \in K\}.$$

Teorema 2.2 (Universalità)**.** *Sia σ una funzione di compressione, e $\mathcal{N}_k$, $\mathcal{N}_k^\sigma$ definiti in 2.4. Sia $T : K \longrightarrow \mathbb{R}$ una funzione continua. Allora per ogni $\epsilon > 0$ esiste $f \in \mathcal{N}_4$ tale che $\|f - T\|_u < \epsilon$; ovvero, $\mathcal{N}_4$ è denso in $C(K)$ rispetto alla* norma uniforme*.*

Dimostrazione. Suppongo per assurdo che esista una funzione continua $T : K \longrightarrow \mathbb{R}$ tale che

$$\inf_{f \in \mathcal{N}_4} \|f - T\|_u = \alpha > 0.$$

Sia $\widehat{f} \in \mathcal{N}_4$ tale che $\alpha \leq \|\widehat{f} - T\|_u < 4\alpha/3$. Definiamo

$$U^+ = \left\{ x \in K \ : \ \frac{\alpha}{3} \leq (\widehat{f} - T)(x) \leq \frac{4\alpha}{3} \right\},$$

$$U^- = \left\{ x \in K \ : \ -\frac{4\alpha}{3} \leq (\widehat{f} - T)(x) \leq -\frac{\alpha}{3} \right\}.$$

Ora, per continuità di $\widehat{f} - T$ segue che U^+ e U^- sono chiusi, inoltre sono disgiunti, quindi per il Lemma 2.5 esiste $H \in \mathcal{N}_3^\sigma$ tale che

$$0 \leq H < \frac{1}{6}, \text{ ovvero scegliamo } \epsilon = \frac{1}{6}, \quad \text{su } U^-, \quad \text{e } \frac{5}{6} < H \leq 1 \quad \text{su } U^+.$$

Consideriamo ora $f = \widehat{f} - \alpha H + \frac{\alpha}{2}$, che $\in \mathcal{N}_4$ dato che $\widehat{f} \in \mathcal{N}_4$, mentre $-\alpha H + \frac{\alpha}{2}$ è combinazione lineare di elementi in $\mathcal{N}_3^\sigma$, che quindi appartiene a $\mathcal{N}_4$. In 2.2 abbiamo mostrato che $\mathcal{N}_4$ è chiuso per somma. Vogliamo mostrare che $\|f - T\|_u < \alpha$, che risulterebbe in un assurdo.

Si presentano tre casi: se $x \in U^+$, allora

$$(f - T)(x) = (\widehat{f} - T)(x) - \alpha H(x) + \frac{\alpha}{2} < \frac{4\alpha}{3} - \frac{5\alpha}{6} + \frac{\alpha}{2} = \alpha.$$

Vale inoltre che

$$(f - T)(x) = (\widehat{f} - T)(x) - \alpha H(x) + \frac{\alpha}{2} \geq \frac{\alpha}{3} - \alpha + \frac{\alpha}{2} = -\frac{\alpha}{6} > -\alpha,$$

e quindi $|f - T| < \alpha$ su U^+. se $x \in U^-$, allora

$$(f - T)(x) = (\widehat{f} - T)(x) - \alpha H(x) + \frac{\alpha}{2} \leq -\frac{\alpha}{3} + \frac{\alpha}{2} = \frac{\alpha}{6} < \alpha,$$

dove vi sono due soli termini nella maggiorazione dato che $-\alpha H(x)$ è compreso tra $-\frac{\alpha}{6}$ e 0, quindi è maggiorato da 0. Vale poi che

$$(f - T)(x) = (\widehat{f} - T)(x) - \alpha H(x) + \frac{\alpha}{2} > -\frac{4\alpha}{3} - \frac{\alpha}{6} + \frac{\alpha}{2} = -\alpha,$$

ovvero che $|f - T| < \alpha$ su U^-. infine, studio il caso $x \in K - (U^- \cup U^+)$. Poiché $-\frac{4\alpha}{3} < (\widehat{f} - T)(y) < \frac{4\alpha}{3}$ per ogni $y \in K$, vale che $-\frac{\alpha}{3} < (\widehat{f} - T)(x) < \frac{\alpha}{3}$, per definizione degli insiemi U^- e U^+. Grazie a questo e a che $0 \leq H \leq 1$ vale che

$$(f - T)(x) = (\widehat{f} - T)(x) - \alpha H(x) + \frac{\alpha}{2} < \frac{\alpha}{3} + \frac{\alpha}{2} = \frac{5\alpha}{6} < \alpha,$$

e

$$(f - T)(x) = (\widehat{f} - T)(x) - \alpha H(x) + \frac{\alpha}{2} > -\frac{\alpha}{3} - \alpha + \frac{\alpha}{2} = -\frac{5\alpha}{6} > -\alpha.$$

Quindi $\|f - T\|_u < \alpha$ che dà un assurdo. □

Capitolo 3

Sperimentazione

3.1 Matrici per il Training e Sperimentazione

Come visto nella Sezione 1.3, i layer di FeedForward sono necessari ad arricchire ciascun embedding con informazioni già note al modello, che sono state apprese durante il processo di training.

Infatti, le matrici Q, K, V, e W sono parametriche, e una rete neurale deve apprendere quali valori utilizzare come entrate di queste matrici al fine di essere in grado di generare degli output coerenti.

Un aspetto cruciale dell'addestramento riguarda *l'inizializzazione* di tali matrici, che solitamente vengono scelte simmetriche, ad entrate uniformi in $[0, 1]$ oppure distribuite come una normale standard. Nel proseguo presentiamo un'analisi degli autovalori e dei valori singolari di queste matrici inizializzate come i due schemi, al fine di valutarne l'impatto sul comportamento del modello; tali indicatori offrono infatti una misura diretta della stabilità delle operazioni di Attention ed aiutano a interpretare le differenti dinamiche che emergono durante le prime fasi di training.

3.1.1 Legge del semicircolo di Wigner

All'interno della teoria delle matrici aleatorie la *legge del semicerchio di Wigner*, introdotta nell'Articolo [14], descrive il profilo limite del *bulk* spettrale di matrici simmetriche con entrate indipendenti. Per *bulk* si intende l'insieme degli autovalori che, nel limite di grandi dimensioni, si addensano in un intervallo compatto di $\mathbb{R}$.

Teorema 3.1 (Semicerchio di Wigner)**.** *Sia* $\{W_{ij}\}_{1\le i\le j\le n}$ *una famiglia di variabili reali indipendenti con*

$$W_{ij} = W_{ji}, \qquad \mathbb{E}[W_{ij}] = 0, \qquad \mathrm{Var}(W_{ij}) = \sigma^2 < \infty.$$

Definiamo

$$X_n := \frac{W_n}{\sqrt{n}}, \qquad \lambda_1, \dots, \lambda_n = \mathrm{spec}(X_n), \qquad \mu_{X_n} := \frac{1}{n}\sum_{j=1}^{n} \delta_{\lambda_j}.$$

Allora, per $n \to \infty$, la misura empirica μ_{X_n} converge in probabilità alla misura μ_{sc} a densità

$$\rho_{\text{sc}}(x) = \frac{1}{2\pi\sigma^2}\sqrt{(2\sigma)^2 - x^2}\,\mathbf{1}_{\{|x|\le 2\sigma\}}(x),$$

nota come semicerchio di Wigner.

Di conseguenza

$$\mu_{X_n} \xrightarrow[n\to\infty]{\mathbb{P}} \mu_{\text{sc}} \qquad \textit{in senso debole,}$$

con la nozione di convergenza debole richiamata nella Definizione A.2 dell'Appendice.

Le prove numeriche svolte in MATLAB mostrano però che due ipotesi, se non verificate, possono portare grandi cambiamenti nello spettro:

(i) **centratura:** $\mathbb{E}[W_{ij}] = 0$;

(ii) **riscalamento:** degli autovalori con il fattore $\sqrt{n}$.

Violando (i) compare un autovalore isolato, chiamato *outlier*, dato che la matrice W può essere riscritta come $\mu E + W'$, dove $E = ones(n)$ e W' matrice che rispetta la legge del semicircolo. Ora, la matrice E ha come autovettore destro $e = ones(n, 1)$, per l'autovalore n. Quindi, la matrice μE avrà un solo autovalore non nullo, in quanto matrice di rango 1, pari a μn, che comparirà come outlier. Invece, gli altri $n - 1$ autovalori saranno distribuiti secondo la legge del semicircolo di Wigner.

In aggiunta, omettendo (ii), il bulk si dilata, e vedremo secondo che fattore, aumentando quindi il raggio spettrale della matrice.

3.1.2 Conseguenze numeriche e simulazioni

Nell'appendice, nella sezione dedicata ai Codici A.2, viene riportato un esempio di codice per simulare una matrice ad entrate gaussiane e per mostrare come viene studiato il suo spettro e i suoi valori principali.

Di seguito riportiamo diversi risultati ottenuti su MATLAB:

- **Entrate gaussiane** $\mathcal{N}(0, 1)$: Se le entrate vengono riscalate per $\sqrt{n}$, valgono $m = 0$ e $\sigma^2 = \frac{1}{n}$, quindi non vi è quindi la necessità di eliminare l'autovalore outlier; gli autovalori della matrice riempiono l'intervallo $[-2, 2]$ e gli istogrammi ottenuti in MATLAB coincidono con la curva teorica.

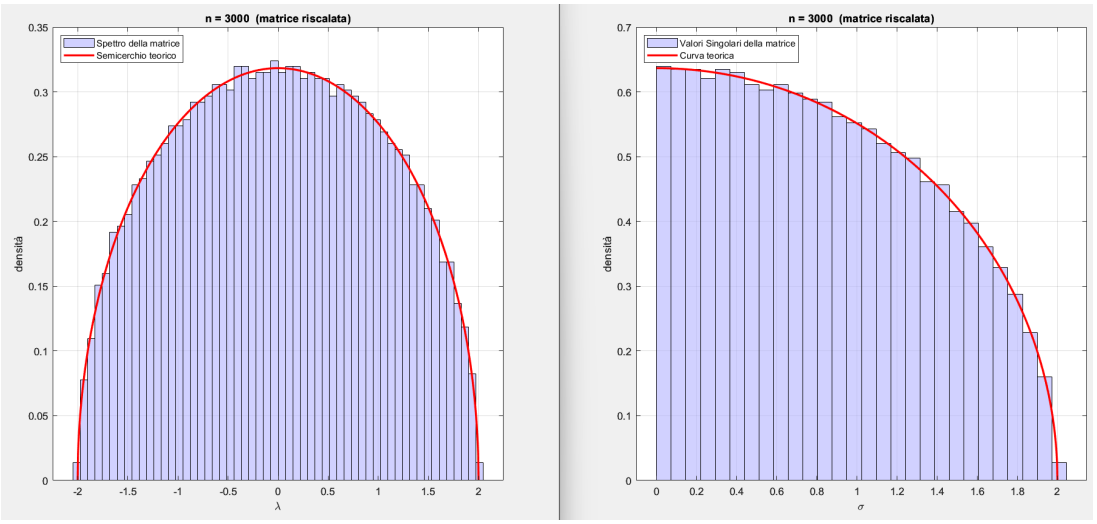


Figura 3.1: Sulla sinistra la distribuzione degli autovalori, mentre sulla destra quella dei valori singolari.

Invece, nel caso in cui le entrate della matrice non siano normalizzate, la varianza σ^2 è pari a 1, e quindi il prefattore, ovvero l'altezza della curva, diventa $pref = \frac{1}{2*pi*\sigma^2*n} = 5.3e-5$, ed il raggio diventa $R = 2*\sqrt{\sigma^2*n} = 109.5$, per $n = 3000$.

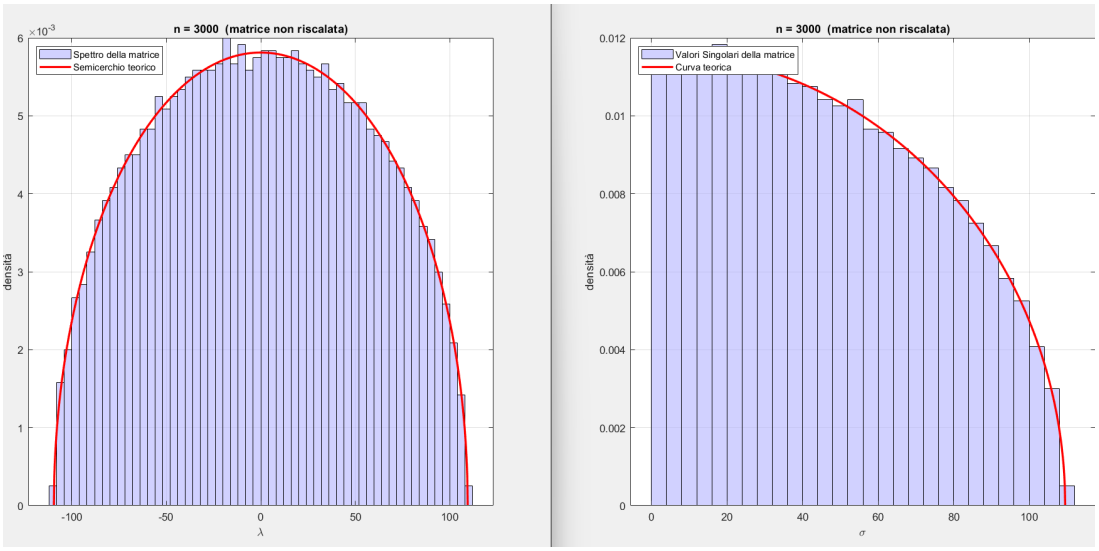


Figura 3.2: Distribuzioni per una matrice ad entrate gaussiane senza applicare il riscalamento.

- **Entrate uniformi** $U(0,1)$. Qui $\mu = \frac{1}{2}$ introduce un autovalore pari a $n\mu$. È quindi necessario rimuovere l'outlier, che è molto lontano dal bulk, come si vede nella Figura 3.3.

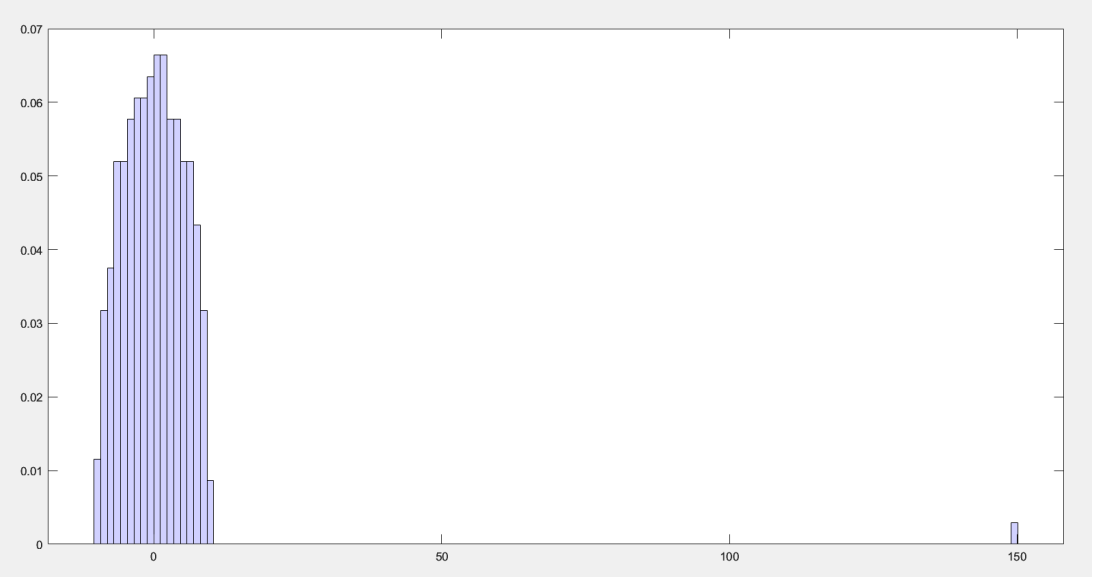


Figura 3.3: Plot per una matrice quadrata ad entrate uniformi di taglia 300, compare quindi l'outlier, pari a 150.

A questo scopo, si inserisce il comando $xlim([-2 * R \;\; 2 * R])$ per visualizzare il plot soltanto entro un certo range, ed escludere quindi l'outlier, che non permetteva di visualizzare l'allineamento tra la distribuzione dello spettro e la curva del fit. Il plot è illustrato nella Figura 3.4.

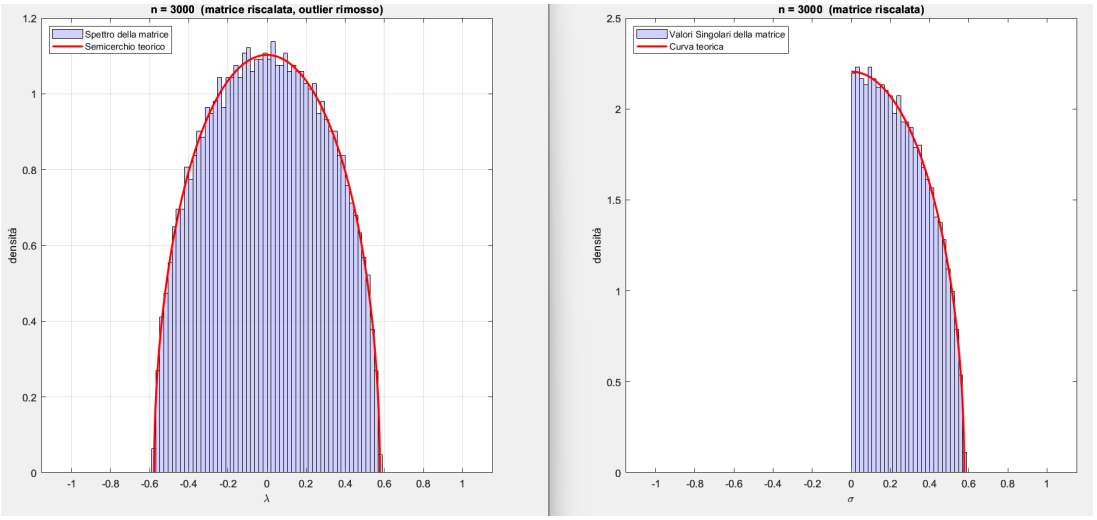


Figura 3.4: Distribuzioni per una matrice ad entrate uniformi.

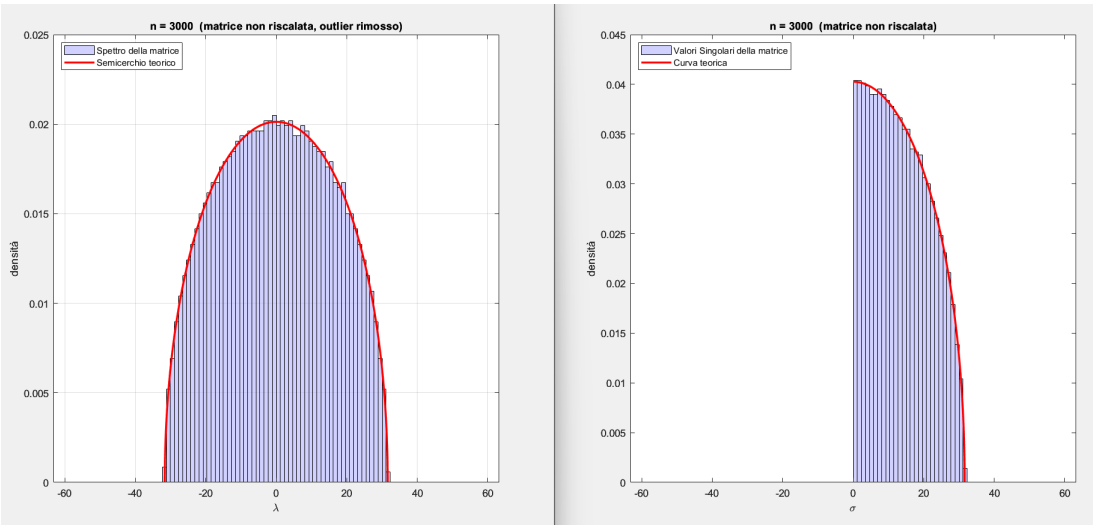
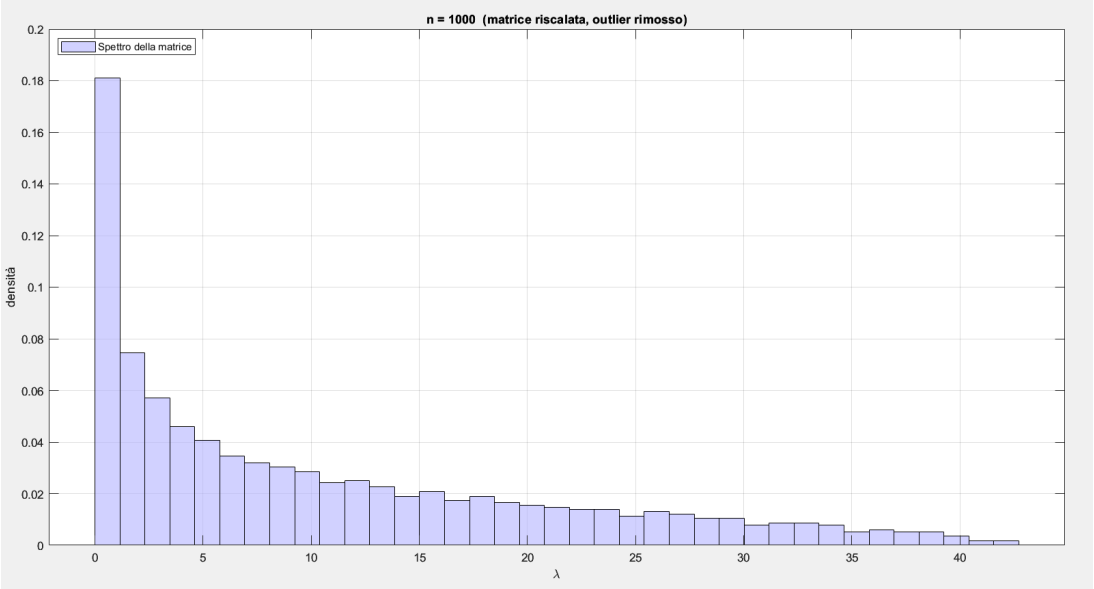


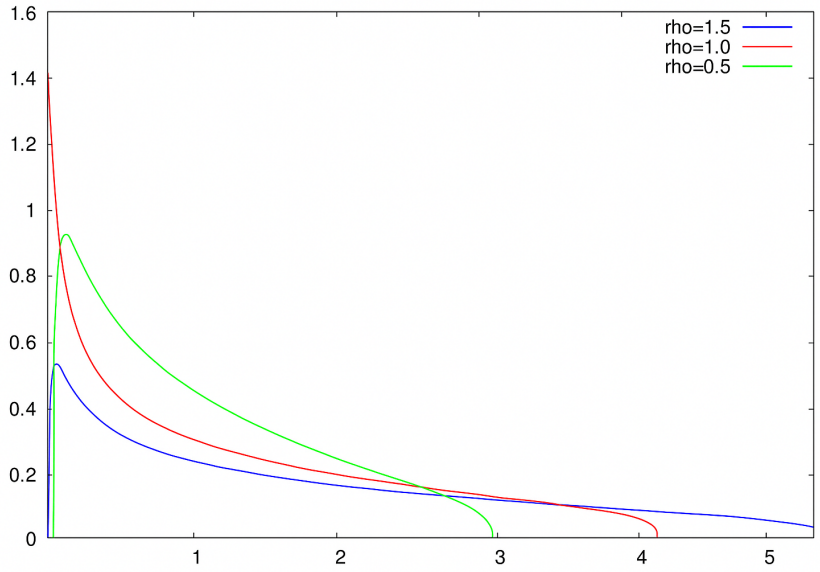
Figura 3.5: Distribuzioni per una matrice ad entrate uniformi senza applicare il riscalamento.

- **Entrate uniformi a media** 0 **e matrice definita positiva**. È possibile inizializzare una matrice ad entrate in $[-1, 1]$ tramite il comando $W = 2 * (rand(n)) - 1$, dopodiché si può simmetrizzare la matrice tramite $W = W * W'/sqrt(n)$: se calcoliamo la distribuzione degli autovalori si ottiene il seguente plot:



Dove, in questo caso non includiamo lo studio dei valori singolari dato che la matrice è definita positiva, ed essi coincidono con gli autovalori della matrice W.

La distribuzione visualizzata è una *Marchenko–Pastur* [15], che ritornerà anche nelle analisi quando andremo ad iterare la trasformazione di Attention per una misura gaussiana.



3.2 Caso Gaussiano

In questa Sezione si vuole mostrare che se la misura di partenza μ è gaussiana allora dopo la trasformazione descritta per le misure in (2.1.7) si ottiene ancora una misura gaussiana, di cui vogliamo calcolare la nuova media μ_T e la nuova varianza σ_T^2.

È importante notare però che questo caso non ricade in quanto descritto dal Teorema di Universalità 2.1 dimostrato da Peyré, in quanto è richiesto che Ω sia compatto. Qui, lavoreremo su $\mathbb{R}^d$, che non è limitato, dunque non è compatto.

Questo primo risultato sarà fondamentale nel seguito, dove vorremo provare ad iterare la trasformazione più volte, partendo da una distribuzione gaussiana.

Consideriamo il particolare caso in cui il numero di teste H è pari ad 1, ma il caso più generale seguirebbe per *riproducibilità* delle variabili gaussiane, concetto introdotto nell'Appendice A.2.

Partendo da un caso più facile, vogliamo dimostrare che, data X variabile aleatoria gaussiana reale di media μ e varianza σ^2, la trasformazione

$$X \to X + \int \frac{e^{xy} y}{\int e^{xu} d\mu(u)} d\mu(y) := \Gamma(X)$$

è affine, e seguirebbe che $\Gamma(X)$ è una gaussiana, sempre per la proprietà di riproducibilità. Nell'integrale y e u sono variabili di integrazione distribuite come la variabile aleatoria X, mentre x coincide con la variabile aleatoria X con cui stiamo operando.

Per fare ciò, inizio a calcolare l'integrale al denominatore, che è infatti uguale a:

$$E[e^{xU}]\Big|_X = e^{\mu X + \frac{1}{2}\sigma^2 X^2},$$

usando la funzione generatrice dei momenti di $U \sim \mathcal{N}(\mu, \sigma^2)$, definita nell'Appendice A.3.

Invece, il numeratore può essere calcolato come $E[e^{XY} Y | X = x] = E[e^{xY} Y]\big|_X$; scritta quindi $Y := \mu + \sigma Z$, con $Z \sim \mathcal{N}(0,1)$, la sostituisco e ottengo:

$$E[e^{x(\mu+\sigma Z)}(\mu + \sigma Z)]\Big|_X = \mu\, E[e^{x(\mu+\sigma)Z}]\Big|_X + \sigma\, E[e^{x(\mu+\sigma)Z} Z]\Big|_X \overset{(*)}{=}$$

$$\overset{(*)}{=} \mu\, E[e^{x(\mu+\sigma)Z}]\Big|_X + \sigma^2 x\, E[e^{xY}]\Big|_X = (\mu + \sigma^2 X) e^{\mu X + \frac{1}{2}\sigma^2 X^2}.$$

Dove l'uguaglianza $\overset{(*)}{=}$ segue dal cambio di variabile gaussiano, riportato nell'Appendice A.3, che è applicabile dato che $e^{x(\mu+\sigma)}$ cresce meno di quanto la densità gaussiana descresca a 0 per $x \to \pm\infty$.

Ora, rimettendo tutto assieme si ottiene:

$$X \to X + \frac{(\mu + \sigma^2 X) e^{\mu X + \frac{1}{2}\sigma^2 X^2}}{e^{\mu X + \frac{1}{2}\sigma^2 X^2}} = X + \mu + \sigma^2 X,$$

che è una trasformazione affine, di cui possiamo calcolare la nuova media e la nuova varianza, che valgono rispettivamente $\mu_T = (2+\sigma^2)\mu$ e $\sigma_T^2 = (1+\sigma^2)^2\sigma^2$.

Similmente, e in maniera più generale, se X è un vettore gaussiano di media $m \in \mathbb{R}^n$ e matrice di covarianza Σ, considerando ora le matrici Q, K, V, W di quey, key, value e weight e il riscalamento per la dimensione d_k, la trasformazione diventa:

$$X \to X + WV \int \frac{e^{\frac{\langle Qx, Ky\rangle}{\sqrt{d_k}}} y}{\int e^{\frac{\langle Qx, Ku\rangle}{\sqrt{d_k}}} d\mu(u)} d\mu(y).$$

Con calcoli analoghi a prima, il denominatore vale

$$e^{\frac{1}{\sqrt{d_k}} X^\top Q^\top K m + \frac{1}{2d_k} X^\top Q^\top K \Sigma K^\top Q X},$$

mentre il numeratore

$$(m + \frac{1}{\sqrt{d_k}} \Sigma K^\top Q X) e^{\frac{1}{\sqrt{d_k}} X^\top Q^\top K m + \frac{1}{\sqrt{d_k}} X^\top Q^\top K \Sigma K^\top Q X}.$$

Mettendo assieme e semplificando come nel caso precedente ottengo che la trasformazione è equivalente a

$$X \to X + WV(m + \frac{1}{\sqrt{d_k}} \Sigma K^\top Q X),$$

dove il nuovo vettore delle medie è

$$m_T = (I + WV + \frac{1}{\sqrt{d_k}} WV \Sigma K^\top Q))m,$$

e la nuova matrice di covarianza è

$$\Sigma_T = (I + \frac{1}{\sqrt{d_k}} WV \Sigma K^\top Q) \Sigma (I + \frac{1}{\sqrt{d_k}} WV \Sigma K^\top Q)^\top.$$

Riguardo quest'ultimo, sviluppando il prodotto e tenendo in considerazione soltanto i termini fino a $\frac{1}{\sqrt{d_k}}$ si ottiene:

$$\Sigma_T = \Sigma + \frac{1}{\sqrt{d_k}} WV \Sigma K^\top Q \Sigma + \frac{1}{\sqrt{d_k}} \Sigma Q^\top K \Sigma V^\top W^\top + o(\frac{1}{d_k}).$$

Si noti che entrambi i risultati sono coerenti con quanto presentato nella sezione relativa al caso continuo della presentazione di Peyré [4], semplicemente completi dell'utilizzo di W e del riscalamento per d_k.

Vogliamo in primo luogo verificare la correttezza di queste formule trovate per il vettore delle medie e per la matrice delle covarianze, e per farlo inizializziamo le matrici Q, K, V e W ad entrate gaussiane riscalate, in quanto abbiamo visto che in questa maniera gli autovalori rimangono contenuti in $[-2, 2]$ e non vi è l'outlier che avrebbe potuto portare instabilità numerica.
Per $d_{in} = d_v = 300$, $d_k = 128$ e $Nsamp = 20000$ gli errori relativi sono pari a $2.49e-02$ per la media e a $5.32e-02$ per la matrice di covarianza.

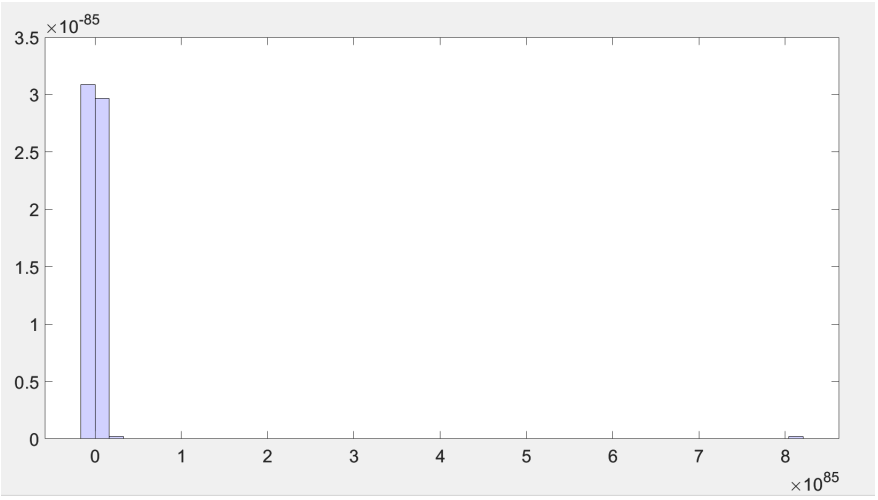
3.2.1 Primo Tentativo di Iterazione

Ora, l'interesse della sperimentazione si rivolge verso l'iterazione ripetuta di questa trasformazione su una misura gaussiana, al fine di studiarne una qualche convergenza o la divergenza dello spettro.

È quindi necessario ricalcolare ogni volta la matrice di covarianza, per riapplicare la trasformazione successiva. L'iterazione di applicazione della trasformazione è racchiusa nel seguente ciclo for:

```
for i = 1:it
    S     = cov(X.');
    B     = (1/sqrt(d_k)) * W * V * S * K.' * Q;
    Xnew  = X + W * V * mu + B * X;
    if norm(X - Xnew) < tol
        fprintf('Convergenza raggiunta\n');
        break
    end
    X = Xnew;
    fprintf('%d\n', i);
end
```

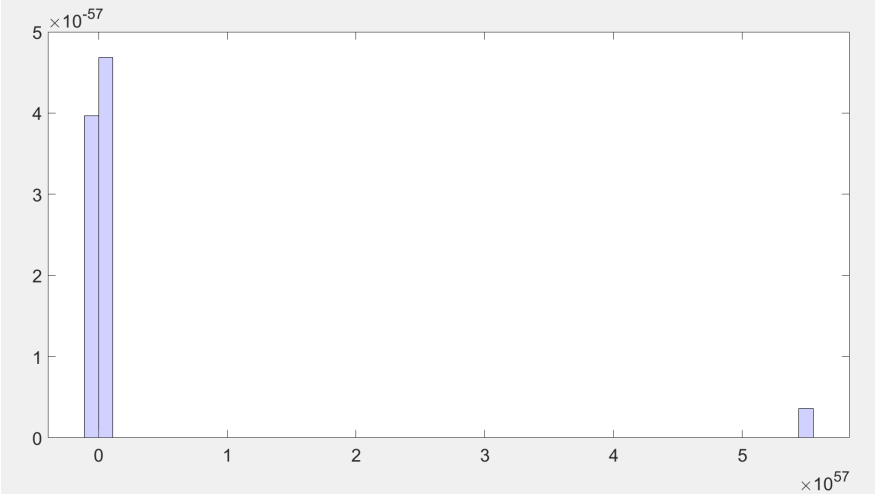
In questo sistema vi sono moltissimi parametri che possono essere modificati, ma cominciamo da un'inizializzazione standard. Scegliamo $d_{in} = d_v = 300$, $d_k = 128$ e $Nsamp = 20000$, e proviamo ad eseguire soltanto 5 iterazioni. Per questo plot abbiamo inizializzato le matrici Q, K, W e V come $randn(d_k, d_{in})/sqrt(d_{in})$, affinché abbiano autovalori contenuti.



Si vede come, anche con soltanto 5 iterazioni, lo spettro della matrice di covarianza esploda, infatti scegliendo le stesse dimensioni dell'input bastano 7 iterazioni per far sì che gli autovalori diventino troppo grandi e MATLAB non sia più in grado di calcolarli.

Dobbiamo quindi modificare qualche parte del codice, ed una prima idea, che può venire in mente è aumentare d_k e diminuire d_{in} e d_v, dato che abbiamo il riscalamento per $\frac{1}{\sqrt{d_k}}$.

Diminuendo $d_{in} = d_v = 30$ e aumentando $d_k = 1280$, e iterando sempre 5 volte il risultato è molto simile, soltanto con circa 30 ordini di grandezza in meno:



È evidente che questo tentativo non è in alcun modo sufficiente, riduce solamente la scala del problema, non il fenomeno. Inoltre, non rappresenta nemmeno una situazione realistica: come descritto da OpenAI nell'Articolo [7], la dimensione dei vettori di embedding è pari a 12288 nel modello ChatGPT3, mentre la dimensione delle query e delle keys è soltanto 128, ovvero $\frac{1}{96}$ di d_{in}.

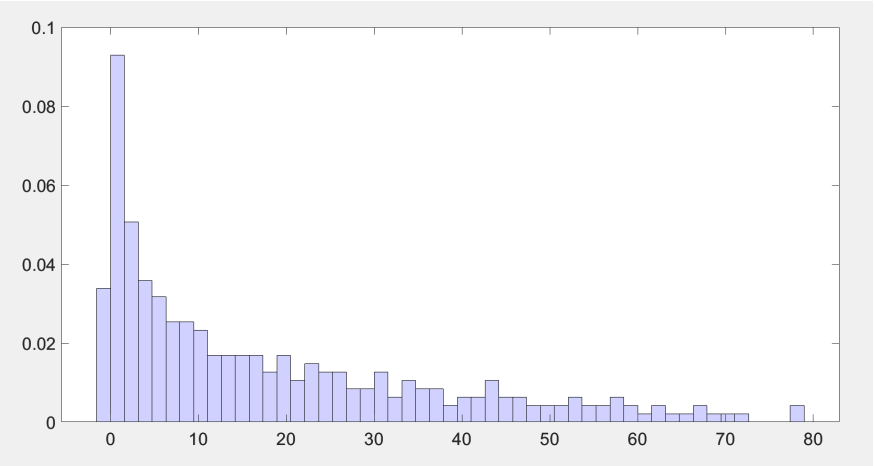
3.2.2 Interventi di Stabilità

Innanzitutto, per rendere i conti più trasparenti e isolare l'effetto della trasformazione, abbiamo riscritto il codice in modo da applicarla direttamente alla matrice di covarianza Σ invece di iterare sui campioni gaussiani X. In questo schema "teorico" non c'è più rumore statistico: la matrice è ottenuta in forma chiusa a partire dalla matrice del passo precedente. Tuttavia, la stabilità (o instabilità) della dinamica non cambia: tutti i casi in cui l'iterazione empirica faceva divergere la covarianza mostrano la stessa divergenza anche in questa versione. L'esplosione, quindi, non dipende dal campionamento precedentemente eseguito, ma è una proprietà intrinseca della trasformazione.

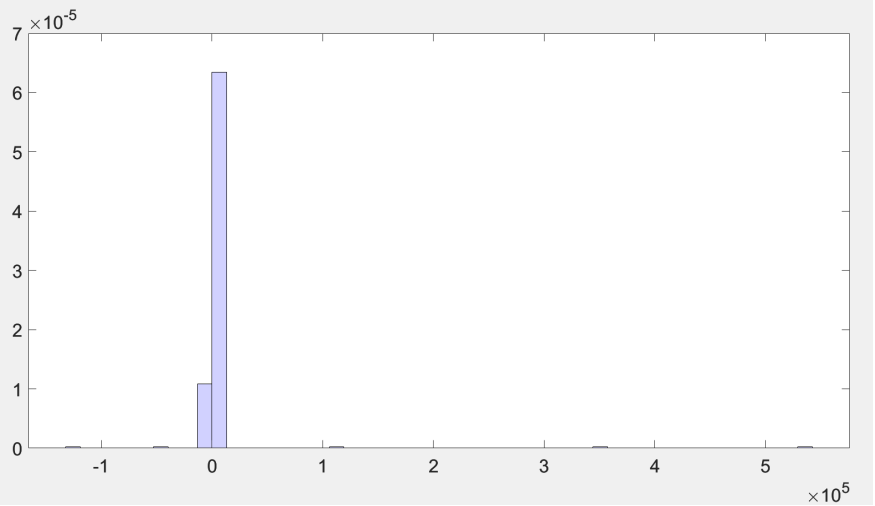
Inoltre, può essere utile introdurre un parametro ϵ per simulare il passaggio a tempi continui, per provare ad emulare il lavoro svolto nella presentazione di Peyré [4], che permette di fare dei passi più piccoli, proprio dell'ordine di ϵ.

Questo nuovo parametro andrà in maniera molto semplice a moltiplicare ogni iterazione, come è mostrato nel codice nella Sezione A.2.

Proviamo quindi ad utilizzare i valori della precedente iterazione e a scegliere $\epsilon = 1e-2$, e facendo sempre 5 iterazioni si ottiene il seguente risultato:



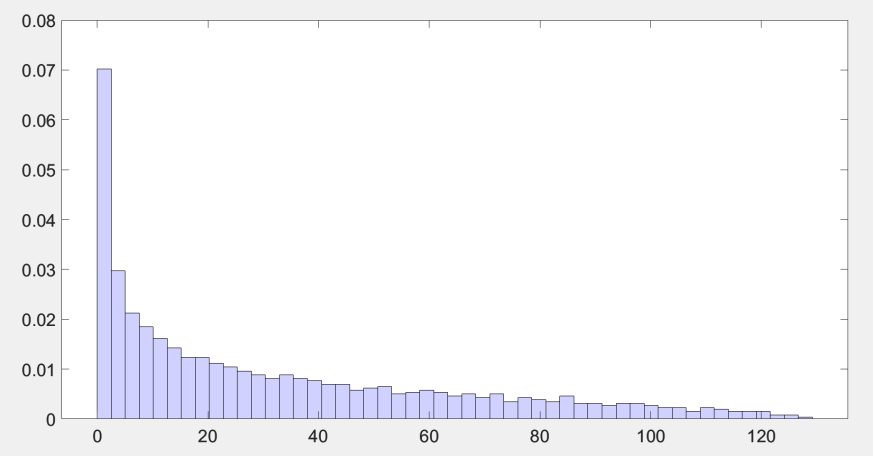
È sicuramente più convincente, ma vorremmo comunque provare ad iterare un numero maggiore di volte, e con questo valore di ϵ gli autovalori non rimangono contenuti per troppo, infatti ad esempio per $it = 25$ si ottiene come plot:



Tuttavia, in questo tentativo compaiono degli autovalori negativi. Nel caso continuo, la matrice di covarianza dovrebbe rimanere sempre definita positiva. In questa versione discreta, l'uso di passi di ampiezza ϵ potrebbe creare la possibilità di muoversi troppo e oltrepassare i vincoli della definitezza positiva, generando quindi una matrice aventi alcuni autovalori negativi. Una volta che un autovalore diventa negativo, poi può crescere tanto quanto quelli positivi, infatti alcuni sono pari a $-1e\ 5$.

Al fine di evitare questo problema e l'esplosione dello spettro si può abbassare ancora il valore di ϵ per simulare in maniera ancora più precisa i tempi continui. Ad

esempio, scegliendo il valore di $1e-5$, ed abbassando conseguentemente il valore della tolleranza a $1e-3$, per evitare di uscire dopo poche iterazioni dal ciclo for, si possono alzare i valori di d_{in} e d_v ponendoli pari a 1024, e iterare anche 1000 volte, ottenendo il seguente plot:



In quest'ultimo tentativo la distribuzione sembra efettivamente una *Marchenko–Pastur* [15], come visto per lo spettro di una matrice definita positiva ad entrate uniformi e centrate, nella Figura 3.1.2. Lo stesso comportamento di convergenza può essere ottenuto anche usando il Codice A.2.1, ovvero usando la raccolta di campioni gaussiani X e calcolando da qui la sua matrice di covarianza.

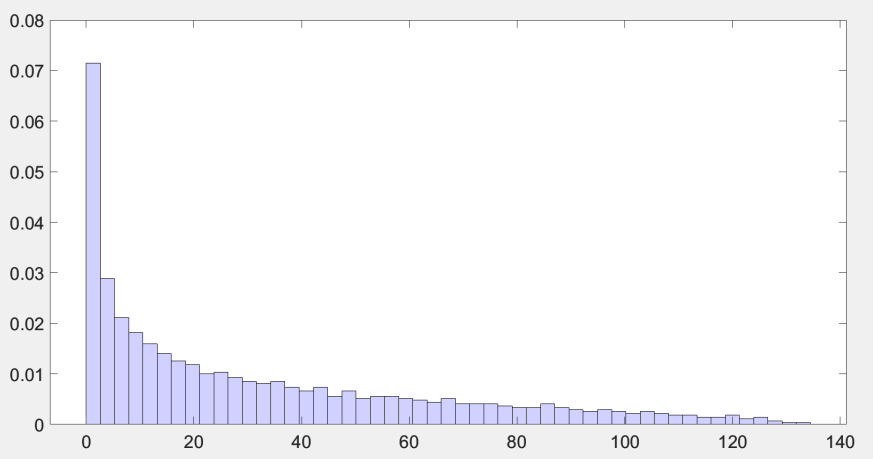


Figura 3.6: Plot inserito per validare la corrispondenza tra il comportamento dei due approcci.

3.3 Training

In questa Sezione, oltre che presentare i risultati numerici, vogliamo spiegare come funziona il processo *training* di una rete neurale nel contesto *Teacher-Student*, che utilizzeremo da qui in avanti.

Abbiamo visto che un Transformer è un sistema composto da numerose matrici parametriche. Nel nostro schema distinguiamo:

- il *Teacher*, dotato di parametri congelati e mai aggiornati durante la simulazione;
- lo *Student*, inizializzato in maniera casuale e aggiornato ad ogni iterazione per imitare il Teacher.

Per ogni matrice di input simmetrica definita positiva Σ_{in} calcoliamo, tramite il Teacher, l'uscita di riferimento Σ_{out}. L'insieme $\{(\Sigma_{\text{in}}^{(i)}, \Sigma_{\text{out}}^{(i)})\}_{i=1}^{n}$ costituisce il *training–set*. Lo Student riceve la stessa $\Sigma_{\text{in}}^{(i)}$ e produce invece $\widehat{\Sigma}^{(i)}$.

Il meccanismo di apprendimento si basa sulla definizione di una funzione obiettivo, chiamata anche *loss* nel seguito e nelle funzioni MATLAB, che misura la distanza fra l'output prodotto dal modello *Teacher* e quello *Student*, a partire dalla quale si modificano iterativamente i parametri per ridurre tale distanza. A questo scopo abbiamo scelto di utilizzare la *Mean Squared Error* sulla norma Frobenius:

$$L(W,V,Q,K) \;=\; \frac{1}{n\, d_{in}^2} \sum_{i=1}^{n} \big\| \,\widehat{\Sigma}^{(i)} - \Sigma_{\text{out}}^{(i)} \big\|_F^2 \;.$$

Una loss pari a zero significa che lo *Student* riproduce esattamente il comportamento del *Teacher*.

Al fine di ridurre L si impiega il metodo di *Discesa Gradiente*: calcoliamo il gradiente della loss rispetto a ciascun parametro e ci muoviamo nella direzione opposta, cioè verso la *direzione più ripida in discesa*. In forma compatta:

$$\theta_{k+1} \;= \theta_k \;-\; \eta\, \nabla_{\theta_k} L, \qquad \text{dove } \theta = \{W,V,Q,K\}, \tag{3.3.1}$$

dove $\eta > 0$ è il *learning rate*, ovvero un parametro che gestisce quanto muoversi nella direzione stabilita. Invece, il pendice k indica che l'aggiornamento viene ripetuto per diverse epoche, finché la loss decresce ed i parametri dello *Student* si stabilizzano in prossimità di un minimo della funzione obiettivo.

Per monitorare l'apprendimento e per ottenere un valore rappresentativo della capacità della rete di generalizzare a qualsiasi input si tiene da parte una frazione del dataset (25% nel nostro caso), composto da esempi mai visti dalla rete, su cui testare la correttezza dei valori parametrici appena modificati a seguito del training. Questa fase è chiamata *validation*.

A questo punto, vogliamo provare a simulare il processo di training, e vogliamo verificare l'efficacia del procedimento su tre scenari di complessità crescente.

Per ciascun scenario presentiamo le curve di convergenza della funzione obiettivo; per i dettagli implementativi delle funzioni MATLAB rimandiamo invece alla Sezione A.2.

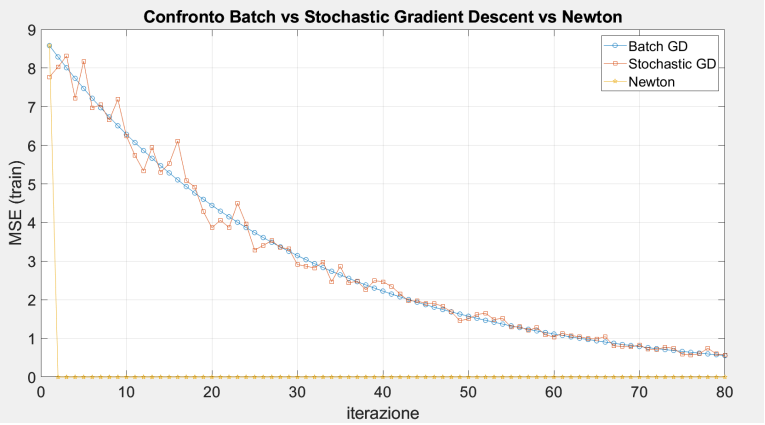
3.3.1 Caso β-scalare

Lo *Student* deve apprendere un singolo parametro β, cercando di ricostruire il valore β^* utilizzato dal *Teacher*. Questo scalare modula il contributo aggiuntivo applicato alla matrice di covarianza a ogni iterazione, per cui la trasformazione diventa

$$\Sigma_T = (I + \frac{\beta^* \epsilon}{\sqrt{d_k}} WV\Sigma K^\top Q)\Sigma(I + \frac{\beta^* \epsilon}{\sqrt{d_k}} WV\Sigma K^\top Q)^\top. \tag{3.3.2}$$

Per apprendere β mettiamo a confronto tre algoritmi di ottimizzazione: la *Discesa Gradiente*, la *Discesa Gradiente Stocastica* e il metodo di Newton.

Cominciamo dunque dal primo caso: inizializziamo $d_{in} = 164, d_k = 66$, costruiamo 300 esempi di matrici simmetriche definite positive, ed effettuiamo 80 iterazioni per ciascuno dei tre esempi; otteniamo come output il seguente grafico:



Innanzitutto, è evidente che il metodo di Discesa Gradiente e quello Stocastico presentano la stessa pendenza di discesa, ma nel grafico di SGD, riportato in rosso, compaiono oscillazioni più pronunciate. Infatti, il gradiente è stimato su un singolo campione estratto in maniera casuale, invece che sull'intero dataset, introducendo rumore statistico nella direzione di aggiornamento. Dunque, la direzione scelta potrebbe non essere ottimale.

Il vantaggio però è evidente in termini di costo computazionale: il primo codice ha un tempo di esecuzione di 30.704 s, mentre il secondo di 0.177 s. È comunque importante notare che dopo circa 40 iterazioni le due traiettorie si sovrappongono e forniscono la stessa stima, $\beta = 0.750$ in 80 iterazioni.

Aumentando le iterazioni a 250 si ottiene per entrambi i metodi $\beta = 0.987$, che è molto vicino al valore $\beta^* = 1$ che stiamo cercando di approssimare; anche qui il rapporto fra i tempi resta nettamente a favore di SGD (52.024 s contro 0.256 s).

Invece, il Metodo di Newton necessita esattamente un'iterazione per trovare il valore corretto, dato che il gradiente rispetto a β è una funzione quadratica in β. Ciononostante, il costo per l'inversione dell'Hessiana fa salire il tempo totale a 35.765 s, non paragonabile alla velocità della *Discesa Gradiente Stocastica*.

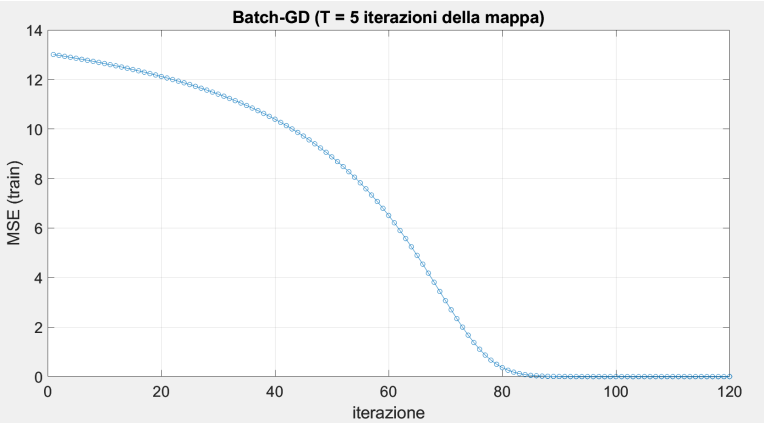
3.3.2 Iterazioni multiple della trasformazione

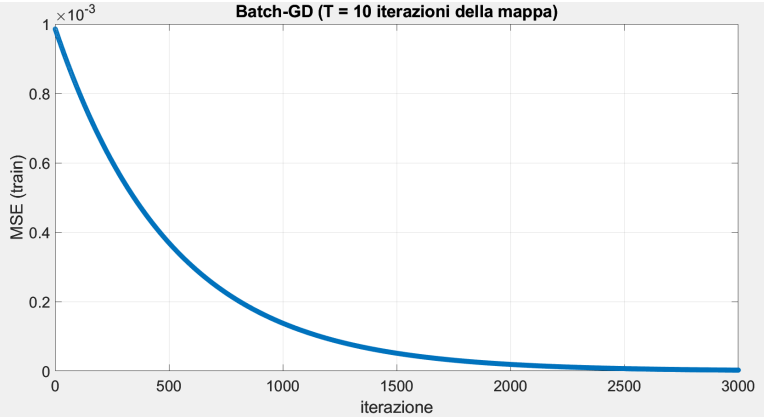
Estendiamo l'analisi al caso in cui la mappa di attenzione venga applicata T volte di seguito, per valutare gli effetti cumulativi sull'accuratezza e sulla stabilità numerica.

Vogliamo infatti provare a simulare il training quando le matrici definite positive di esempio sono trasformate attraverso diversi layer di Attention, simile a quanto visto nella Sezione precedente 3.2.1.

In questa sperimentazione, è necessario scegliere ϵ piccolo al fine di poter lavorare con matrici con entrate modeste, evitando che i valori esplodano in norma.

Il codice è simile a quanto già fatto in precedenza, soltanto vi è la necessità di calcolare il gradiente rispetto a β quando la trasformazione viene applicata T volte:





Nei due grafici analizziamo l'effetto dell'applicare la trasformazione (3.3.2) un numero diverso di volte, mantenendo $d_{\text{in}} = 44$ e $d_k = 16$. Nel primo caso si simulano 5 layer di Attention, e scegliamo dunque $\epsilon = 1e-1$. Invece, il secondo caso vuole rappresentare una rete più profonda, infatti la trasformazione (3.3.2) viene applicata 10 volte: abbiamo dovuto necessariamente scegliere $\epsilon = 1e-3$; inoltre, abbiamo abbassato

il valore del *learning rate* a $5e-1$, al fine di iterare un numero maggiore di volte, in questo caso 3000.

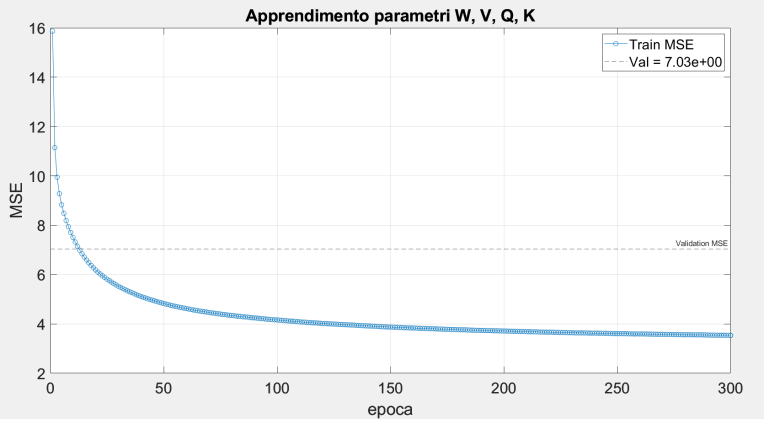
In ciascun esempio è comunque evidente come lo *Student* sia in grado di replicare il *Teacher*, anche in questo contesto più complicato.

3.3.3 Apprendimento delle matrici complete

Infine consideriamo il regime più realistico, in cui lo *Student* stima l'intero insieme di matrici (W, V, Q, K). Mostriamo come il *mini-batching* acceleri la convergenza, pur introducendo rumore stocastico nei gradienti.

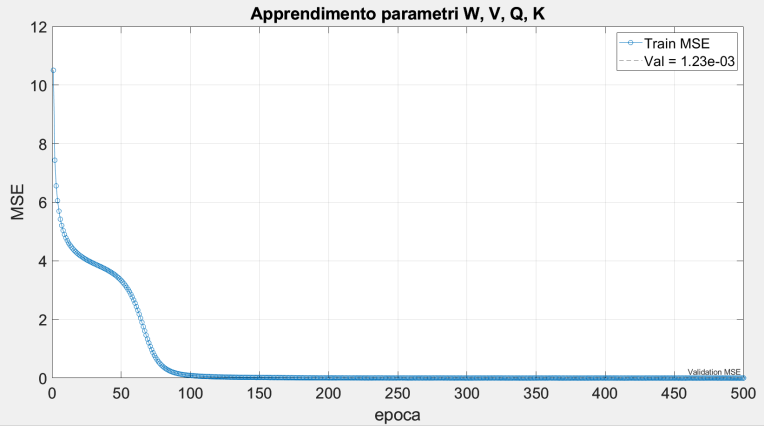
I due modelli Teacher e Student vengono inizializzati in modo indipendente e l'ottimizzazione procede applicando la discesa del gradiente *stocastica* alle matrici, in quanto abbiamo visto essere molto meno costoso come metodo; la funzione obiettivo resta la MSE in norma di Frobenius, ora calcolata su un *mini–batch* di $B = 32$ esempi: infatti, calcolare ad ogni iterazione la loss e farne la media su tutto il *training set* può risultare dispendioso, ed è invece sufficiente farlo su una porzione ridotta di esso, cosa che può introdurre del rumore nella discesa del gradiente.

Ad ogni passo calcoliamo quindi il gradiente medio del batch rispetto a ciascuna matrice (W, V, Q, K) e lo utilizziamo per aggiornare i parametri secondo l'iterazione di aggiornamento del parametro θ (3.3.1) vista precedentemente, con learning rate fissato.

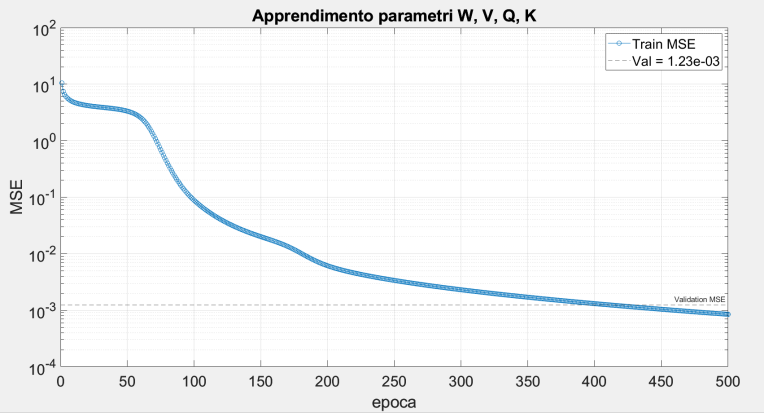


Nel plot precedente 3.3.3 abbiamo usato matrici di dimensione $d_{in} = 64, d_k = 36$, ed è stato necessario aumentare il valore del *learning rate* da $1e-3$ a $1e-1$, dato che altrimenti l'*errore quadratico medio* decresceva troppo lentamente.

La dimensione del *batch* è 32, ed effettivamente fa sì che il rumore statistico non sia visibile; quindi, ripetiamo l'esperimento con $B = 1$ nella Figura 3.3.3, ed un numero di epoche esteso a 500. In questo caso la MSE decresce con andamento più irregolare, tipico della SGD priva di mini–batch, ma converge comunque allo stesso valore finale, confermando la correttezza dell'algoritmo anche in presenza di gradiente altamente rumoroso.



Poiché nella parte finale del grafico la curva tende ad appiattirsi, abbiamo tracciato la stessa evoluzione in scala logaritmica (Figura 3.3.3): il plot `semilogy` mostra che la loss continua a diminuire in modo regolare anche oltre la centesima epoca.



In quest'ultimo grafico 3.3.3 è evidente come l'iterazione mostri andamenti diversi, a seconda delle varie epoche. Questo comportamento potrebbe essere dovuto al fatto che la funzione di Loss presenta dei minimi locali, dove la dinamica rimane *intrappolata*, rimanendo fissa per diverse iterazioni, come ad esempio tra la trentesima e la cinquantesima.

Proprio a questo scopo è utile che la discesa gradiente sia *stocastica* e che si utilizzi una dimensione di batch: il rumore statistico creato può risultare utile ad uscire da una situazione simile, dato che la direzione scelta non è necessariamente la migliore, ma ha una componente *aleatoria*. In questa maniera, se si riesce ad allontanarsi a sufficienza dal minimo locale, l'iterazione può proseguire diminuendo ulteriormente la loss, cercando dunque il minimo globale.

Appendice

A.1 Risultati utilizzati

In quest'ultimo capitolo sono racchiuse diverse definizioni e dimostrazioni già trattate durante vari insegnamenti della triennale, le quali si sono rivelate utili in diversi passaggi dei capitoli precedenti.

Definizione A.1 (Push-forward)**.** Sia $\Omega \subset \mathbb{R}^d$ un insieme compatto e sia $T : \Omega \to \Omega' \subset \mathbb{R}^{d'}$ una funzione misurabile. Sia inoltre $\mu \in \mathcal{P}(\Omega)$. Il *push-forward* di μ tramite T è la misura $T_\sharp\mu \in \mathcal{P}(\Omega')$ definita, per ogni $A \in \mathcal{B}(\Omega')$, da

$$T_\sharp\mu(A) = \mu\big(T^{-1}(A)\big).$$

Ad esempio, l'operatore push-forward $T_\sharp$ modifica misure discrete semplicemente cambiando il loro supporto, come

$$T_\sharp\Big(\tfrac{1}{n}\sum_{i=1}^{n}\delta_{x_i}\Big) := \tfrac{1}{n}\sum_{i=1}^{n}\delta_{T_\sharp(x_i)}.$$

Invece, per una misura generale $\nu := \mu$, $T_\sharp\mu$ è definita come un cambio di variabili nell'integrale, cioè per ogni $g \in C(\Omega)$ si ha

$$\int_{\Omega'} g(y)\, d\nu(y) \;=\; \int_{\Omega} g\big(T(x)\big)\, d\mu(x). \tag{A.1.1}$$

Definizione A.2 (Convergenza debole)**.** Sia $\Omega \subset \mathbb{R}^d$ un insieme compatto. Diciamo che una successione $(\mu_k)_{k\in\mathbb{N}} \subset \mathcal{P}(\Omega)$ *converge debolmente* a $\mu \in \mathcal{P}(\Omega)$ (si scrive $\mu_k \rightharpoonup^* \mu$) se e solo se, per ogni funzione continua $f \in C(\Omega)$, vale

$$\int f(x)\, d\mu_k(x) \;\longrightarrow\; \int f(x)\, d\mu(x).$$

Nel caso particolare di misure discrete, per un numero finito di punti n, questa convergenza corrisponde alla convergenza usuale in dimensione finita:

$$\frac{1}{n}\sum_{i=1}^{n}\delta_{x_{i,k}} \implies \frac{1}{n}\sum_{i=1}^{n}\delta_{x_i} \iff X_k = \big(x_{i,k}\big)_{i=1}^{n} \in \mathbb{R}^{d\times n} \longrightarrow X = \big(x_i\big)_{i=1}^{n} \in \mathbb{R}^{d\times n}.$$

Lemma A.1. *Sia X uno spazio topologico compatto e sia $C \subset X$ un sottoinsieme chiuso. Allora C è compatto.*

Dimostrazione. Per mostrare che F è compatto, dobbiamo provare che ogni suo ricoprimento aperto ammette un sottoricoprimento finito. Sia dunque $\{U_\alpha\}_{\alpha\in\Lambda}$ un ricoprimento aperto di F, ossia:

$$F \subset \bigcup_{\alpha\in\Lambda} U_\alpha.$$

Poiché F è chiuso in X, $X \setminus F$ è aperto in X. Consideriamo allora la famiglia di insiemi aperti in X:

$$\mathcal{U} = \{U_\alpha\}_{\alpha\in\Lambda} \cup \{X \setminus F\}.$$

Questa è chiaramente un ricoprimento aperto di X, in quanto:

$$X = F \cup (X \setminus F) \subset \left(\bigcup_{\alpha\in\Lambda} U_\alpha\right) \cup (X \setminus F).$$

Poiché X è compatto, esiste un sottoricoprimento finito $\{U_{\alpha_1}, \ldots, U_{\alpha_n}, X\setminus F\}$ che copre X. Tuttavia, $X \setminus F$ non è necessario per coprire F, dunque l'insieme $\{U_{\alpha_1}, \ldots, U_{\alpha_n}\}$ fornisce un sottoricoprimento finito di F.

Dunque, ogni ricoprimento aperto di F ammette un sottoricoprimento finito, il che dimostra che F è compatto. □

Lemma A.2 (Riproducibilità)**.** *Se $X \sim \mathcal{N}(\mu_X, \sigma_X^2)$ e $Y \sim \mathcal{N}(\mu_Y, \sigma_Y^2)$ sono variabili aleatorie* indipendenti*, allora*

$$X + Y \;\sim\; \mathcal{N}\big(\mu_X + \mu_Y,\; \sigma_X^2 + \sigma_Y^2\big).$$

In particolare, somma di variabili aleatorie gaussiane è gaussiana.

Dimostrazione. Ricordiamo che la funzione caratteristica di una variabile aleatoria gaussiana Z è definita come

$$\phi_Z(t) = \exp\Big(i\,\mu\,t \;-\; \tfrac{1}{2}\,\sigma^2\,t^2\Big).$$

Inoltre, se X e Y sono *indipendenti*, la funzione caratteristica della loro somma è il prodotto delle funzioni caratteristiche:

$$\phi_{X+Y}(t) = \mathbb{E}\big[e^{\,i\,t\,(X+Y)}\big] = \mathbb{E}\big[e^{\,i\,t\,X}\big]\;\mathbb{E}\big[e^{\,i\,t\,Y}\big] = \phi_X(t)\,\phi_Y(t).$$

Poiché

$$\phi_X(t) = \exp\Big(i\,\mu_X\,t \;-\; \tfrac{1}{2}\,\sigma_X^2\,t^2\Big), \quad \phi_Y(t) = \exp\Big(i\,\mu_Y\,t \;-\; \tfrac{1}{2}\,\sigma_Y^2\,t^2\Big),$$

allora

$$\phi_{X+Y}(t) = \exp\Big(i\,(\mu_X + \mu_Y)\,t \;-\; \tfrac{1}{2}\,(\sigma_X^2 + \sigma_Y^2)\,t^2\Big).$$

Questa è la funzione caratteristica di una variabile aleatoria gaussiana con media $\mu_X + \mu_Y$ e varianza $\sigma_X^2 + \sigma_Y^2$. Dunque $X+Y$ è anch'essa gaussiana con i parametri desiderati. □

Definizione A.3 (Funzione Generatrice dei Momenti)**.** Sia X una variabile aleatoria reale. La *funzione generatrice dei momenti* (MGF) di X è definita come

$$M_X(t) = \mathbb{E}\left[e^{tX}\right],$$

per ogni $t \in \mathbb{R}$ per cui esiste il valore atteso.

Ad esempio, per una variabile gaussiana $X \sim \mathcal{N}(\mu, \sigma^2)$, la funzione generatrice dei momenti vale

$$M_X(t) = e^{\mu t + \frac{1}{2}\sigma^2 t^2}$$

Infatti, dobbiamo risolvere l'integrale

$$\int_{-\infty}^{\infty} e^{tx} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2} + tx} dx,$$

dove possiamo riscrivere l'esponente, completando il quadrato:

$$-\frac{(x-\mu)^2}{2\sigma^2} + tx = -\frac{1}{2\sigma^2}\left[(x-\mu)^2 - 2\sigma^2 tx\right] = -\frac{1}{2\sigma^2}\left[(x-\mu-\sigma^2 t)^2 - \sigma^4 t^2 - 2\sigma^2\mu t\right].$$

Pertanto, l'integrale diventa:

$$M_X(t) = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu-\sigma^2 t)^2}{2\sigma^2}} e^{\mu t} e^{\frac{\sigma^2 t^2}{2}} dx = e^{\mu t + \frac{1}{2}\sigma^2 t^2} \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu-\sigma^2 t)^2}{2\sigma^2}} dx$$

Ora, possiamo effettuare il cambio di variabile $y = x - \mu - \sigma^2 t$, che implica $dy = dx$, ottenendo:

$$M_X(t) = e^{\mu t + \frac{1}{2}\sigma^2 t^2} \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{y^2}{2\sigma^2}} dy$$

L'integrale è ora la normale standard, il cui valore è 1, quindi, la funzione generatrice dei momenti per una variabile casuale gaussiana di media μ e varianza σ^2 è:

$$M_X(t) = e^{\mu t + \frac{1}{2}\sigma^2 t^2}$$

Lemma A.3. *Sia X una variabile aleatoria standard, ossia tale che $X \sim \mathcal{N}(0,1)$, sia p la sua densità, e sia $f : \mathbb{R} \to \mathbb{R}$ una funzione differenziabile tale che $\lim_{x\to\pm\infty} f(x)p(x) = 0$. Allora*

$$\mathbb{E}[Xf(X)] = \mathbb{E}[f'(X)].$$

Dimostrazione. Sappiamo che la densità di X è data da

$$p(x) = \frac{1}{\sqrt{2\pi}} \exp\Big(-\frac{x^2}{2}\Big).$$

Osserviamo che differenziando $p(x)$ otteniamo:

$$p'(x) = -x\, p(x).$$

Il valore atteso $\mathbb{E}[X f(X)]$ si scrive come

$$\mathbb{E}[X f(X)] = \int_{-\infty}^{+\infty} x\, f(x)\, p(x)\, dx = -\int_{-\infty}^{+\infty} f(x)\, p'(x)\, dx.$$

Applichiamo ora l'integrazione per parti ponendo

$$u = f(x) \quad \text{e} \quad dv = p'(x)\, dx.$$

Si ha dunque:

$$du = f'(x)\, dx, \quad v = p(x).$$

Pertanto,

$$-\int_{-\infty}^{+\infty} f(x)\, p'(x)\, dx = -\Big[f(x)p(x)\Big]_{-\infty}^{+\infty} + \int_{-\infty}^{+\infty} f'(x)\, p(x)\, dx.$$

Poiché f cresce meno della densità gaussiana $p(x)$ per $x \to \pm\infty$, il termine $\Big[f(x)p(x)\Big]_{-\infty}^{+\infty}$ è nullo. Quindi,

$$-\int_{-\infty}^{+\infty} f(x)\, p'(x)\, dx = \mathbb{E}[f'(X)],$$

che era quanto si voleva mostrare. □

A.2 Codici MATLAB

A.2.1 Codici per l'Iterazione

In questa sezione sono contenuti i codici MATLAB utilizzati per la parte di sperimentazione, trattata nel Capitolo 3.

Iniziamo dal codice necessario per plottare la distribuzione degli autovalori e dei valori singolari per matrici aventi entrate che seguono una distribuzione precisa (qui è riportato il caso gaussiano riscalato):

```
function gaussian_random_matrix(n)

W = tril (randn(n));
W = (W + W' - diag(diag(W))) / sqrt(n);

sigma2 = 1 / n;
R =  2 * sqrt(sigma2) * sqrt(n);
pref = 1 / ( 2 * pi * sigma2 * n);

[~, eigenvalues_W] = eig(W);
eigenvalues_W = diag(eigenvalues_W);

figure;
  histogram(eigenvalues_W, 'Normalization', 'pdf', 'BinWidth', 2*
   R / sqrt(n));
  hold on;
x = linspace(-R, R, 1000);
rho =  pref * sqrt(R^2 - x.^2);
plot(x, rho, 'r', 'LineWidth', 2);

xlabel('\lambda'); ylabel('densita'');
legend('Spettro', 'Semicerchio teorico');
title(sprintf('n = %d (matrice riscalata)', n));
grid on;

s = svd(W);
figure;
  histogram(s, 'Normalization','pdf', 'BinWidth', 2*R / sqrt(n));
  hold on;
x = linspace(0, R, 1000);
rho = 2*pref * sqrt(R^2 - x.^2);
plot(x, rho, 'r', 'LineWidth', 2);

xlabel('\sigma'); ylabel('densita'');
legend('Valori singolari', 'Curva teorica');
title(sprintf('n = %d (matrice riscalata)', n));
grid on;
end
```

In questo codice, alla linea 14, 'Normalization' e 'pdf' sono necessari per normalizzare la somma delle aree delle colonne dell'istogramma pari a 1, e alla riga 30 il prefattore viene moltiplicato per 2: infatti, la legge del semicircolo è simmetrica rispetto all'asse y ed i valori singolari sono tutti positivi, quindi ciascuna colonna dell'istogramma avrà altezza doppia rispetto agli autovalori.

Invece, nel secondo codice viene verificata la correttezza dei valori teorici della media e della varianza della guassiana trasformata dal layer di Attention, effettuando un campionamento:

```
function verifica_valori_gaussian(d_in, d_v, d_k, Nsamp)
    mu    = randn(d_in,1);
    Sigma = randn(d_in);
    Sigma = Sigma*Sigma'/sqrt(d_in);

    L = chol(Sigma,'lower');
    Z = randn(d_in, Nsamp);
    X = mu + L*Z;

    Q = randn(d_k,d_in)/sqrt(d_in);
    K = randn(d_k,d_in)/sqrt(d_in);
    V = randn(d_v,d_in)/sqrt(d_in);
    W = randn(d_in,d_v)/sqrt(d_v);

    B = (1/sqrt(d_k)) *  W * V * Sigma * K.' * Q;

    I_d      = eye(d_in);
    mu_th    = (I_d +  W * V + B) * mu;
    Sigma_th = (I_d + B) * Sigma * (I_d + B).';

    Xnew = X + W * V * mu + B * X;

    mu_emp    = mean(Xnew, 2);
    Sigma_emp = cov(Xnew.', 1);

    err_mu = norm(mu_emp - mu_th) / norm(mu_th);
    err_S  = norm(Sigma_emp - Sigma_th, 'fro') / norm(Sigma_th, '
   fro');

    fprintf('Errore rel. media : %.2e\n', err_mu);
    fprintf('Errore rel. cov   : %.2e\n', err_S);
end
```

In questa funzione viene innanzitutto usata la fattorizzazione di Cholesky, vista al corso di Calcolo Scientifico, per usare una radice della matrice di covarianza.

Per poter fattorizzare tramite Choleski è necessario che la matrice Sigma sia definita positiva, che è verificato con probabilità 1 dato che l'evento di non avere rango massimo

per una matrice ad entrate casuali ha probabilità 0. Inoltre la fattorizzazione è più economica di altre alternative, in quanto ha costo pari a $\frac{1}{3}n^3$.

Il terzo codice rappresenta l'iterazione semplificata della trasformazione, andando a modificare direttamente la matrice di covarianza:

```
function gaussian_transform_matrix(d_in, d_v, d_k, it, tol,
    epsilon)
    Sigma = randn(d_in);
    Sigma = Sigma * Sigma' / sqrt(d_in);

    W = randn(d_in,d_v)/sqrt(d_v);
    V = randn(d_v,d_in)/sqrt(d_in);
    Q = randn(d_k,d_in)/sqrt(d_in);
    K = randn(d_k,d_in)/sqrt(d_in);
    A = W * V;

    for i = 1:it
        Sigma_new = Sigma + (epsilon * 1/sqrt(d_k)) * (A*Sigma*K
    '*Q*Sigma + Sigma*Q'*K*Sigma*A');
        if norm(Sigma - Sigma_new) < tol
           Sigma = Sigma_new;
           fprintf('Convergenza raggiunta in %d iterazioni\n',i);
           break
        end
        Sigma = Sigma_new;
    end

    [~, eigenvalues_Sigma] = eig(Sigma);
    eigenvalues_Sigma = diag(eigenvalues_Sigma);

    vmin = min(eigenvalues_Sigma);
    vmax = max(eigenvalues_Sigma);
    binwidth = (vmax - vmin) / 50;

    figure;
    histogram(eigenvalues_Sigma, 'Normalization', 'pdf', '
    BinWidth', binwidth, 'FaceColor', [0.7 0.7 1]);
end
```

A.2.2 Codici per il Training

Di seguito verranno riportate e brevemente descritte le funzioni per la parte di simulazione del processo di training, dove ciascuna è risultata utile in un setup differente: la prima, *TrainingBeta*, racchiude il primo tentativo, maggiormente semplificato, di far imparare al modello studente un singolo parametro passato in input β^*, che moltiplica nell'iterazione le matrici come prefattore:

```
function training_beta(d_in, d_k, eps, N, maxIt, lr, beta_star)

alpha = eps / sqrt(d_k);
trainRatio = 0.75;

rng(1);
A = randn(d_in)/sqrt(d_in) * randn(d_in)/sqrt(d_in);
Q = randn(d_k,d_in)/sqrt(d_in);
K = randn(d_k,d_in)/sqrt(d_in);

teachT = @(S) S + alpha*beta_star*(A*S*K'*Q*S + S*Q'*K*S*A');


SigIn = cell(N,1);  SigOut = cell(N,1);
for i = 1:N
    S = randn(d_in); S = S*S'/sqrt(d_in);
    SigIn{i}  = S;
    SigOut{i} = teachT(S);
end
idxTrain = 1:round(trainRatio*N);
froMSE = @(A,B) sum((A-B).^2,'all')/d_in^2;


beta = 0;
trainBatch = zeros(1,maxIt);
for it = 1:maxIt
    loss = 0; grad = 0;
    for j = idxTrain
        S = SigIn{j};  Tgt = SigOut{j};
        F = A*S*K'*Q*S + S*Q'*K*S*A';
        P = S + alpha*beta*F;
        E = P - Tgt;
        loss = loss + froMSE(P,Tgt);
        grad = grad + 2*alpha*sum(E.*F,'all')/d_in^2;
    end
    loss = loss/numel(idxTrain);
    grad = grad/numel(idxTrain);

    beta = beta - lr*grad;
    trainBatch(it) = loss;
end
fprintf('Batch-GD terminato: beta = %.3f\n',beta);

beta = 0;
trainStoch = zeros(1,maxIt);
for it = 1:maxIt
    j = idxTrain(randi(numel(idxTrain)));
    S = SigIn{j};
```

```
    Tgt = SigOut{j};
    F = A*S*K'*Q*S + S*Q'*K*S*A';
    P = S + alpha*beta*F;
    E = P - Tgt;

    loss = froMSE(P,Tgt);
    grad = 2*alpha*sum(E.*F,'all')/d_in^2;

    beta = beta - lr*grad;
    trainStoch(it) = loss;
end
fprintf('Stochastic-GD terminato: beta = %.3f\n',beta);

beta = 0;
trainNewton = zeros(maxIt,1);
for it = 1:maxIt
    grad = 0;  hess = 0; loss= 0;
    for j = idxTrain
        S = SigIn{j};
        Tgt = SigOut{j};
        F = A*S*K'*Q*S + S*Q'*K*S*A';
        Pred= S + alpha*beta*F;
        E = Pred - Tgt;
        loss = loss + froMSE(Pred,Tgt);
        grad = grad + 2*alpha*sum(E.*F,'all')/d_in^2;
        hess = hess + 2*alpha^2*sum(F.^2,'all')/d_in^2;
    end
    loss = loss/numel(idxTrain);
    grad = grad/numel(idxTrain);
    hess = hess/numel(idxTrain);

    beta = beta - grad/hess;
    trainNewton(it) = loss;
end
fprintf('Newton\terminato: beta = %.3f\n',beta);

figure;
plot(1:maxIt,trainBatch,'-o','DisplayName','Batch GD'); hold on;
plot(1:maxIt,trainStoch,'-s','DisplayName','Stochastic GD');
plot(1:maxIt, trainNewton, '-p', 'DisplayName','Newton');
xlabel('iterazione'); ylabel('MSE (train)');
legend('Location','northeast'); grid on;
title('Confronto Batch vs Stochastic Gradient Descent vs Newton')
    ;
ax = gca;
ax.FontSize = 20; % ingrandisce i numeri degli assi
ax.TickLabelInterpreter = 'tex';
end
```

Dove si vuole paragonare vari metodi per provare ad approssimare il valore β^*, ovvero il metodo di *Discesa Gradiente*, di *Discesa Gradiente Stocastica*, e il *Metodo di Newton*.

Vengono dunque inizializzati diversi parametri e le matrici necessarie, dove ad esempio il valore *trainRatio* rappresenta la percentuale delle matrici simmetriche e definite positive calcolate e modificate tramite la handle *teachT*, definita alla riga 11, sono destinate al processo di *training*, mentre le rimanenti verranno utilizzate per la fase di *validation* finale.

Successivamente, viene creata una cella di matrici simmetriche e definite positive, le quali corrispondono agli esempi su cui il modello Student deve effetuare il training e modificare in base al risultato il proprio paramentro β, che viene inizializzato a 0.

Dunque, si inizia il ciclo di training, dove si inizializza l'array *trainBatch* al fine di salvare la loss ad ogni iterazione. Viene eseguito un ulteriore ciclo for interno per j che scorre tutti gli indici destinati al training, e, considerata la matrice simmetrica j-esima, si calcola a partire da essa la matrice di output applicando la trasformazione con in valore β corrente del modello Student. Dopodiché, calcolata la differenza tra quest'output e l'output calcolato dal modello Teacher con lo stesso input, viene calcolato il *Mean-Square-Error* in norma Frobenius, che viene aggiunto al parametro *loss*.

Viene dunque calcolato il gradiente per ciascun'iterazione j, che corrisponde alla derivata parziale della formula di iterazione rispetto al parametro β, che corrisponde a quanto scritto nella riga 34.

Alla fine di ciascuno di questi cicli interni viene calcolata la media della loss e del gradiente, e viene aggiornato il valore di β, alla riga 39, e ciò viene ripetuto per *maxIt* volte.

Successivamente, al seguito della reinizializzazione del valore di $\beta = 0$, inizia il ciclo di training usando il metodo di *Discesa Gradiente Stocastica*: l'iterazione è molto simile a quella appena descritta, ma la differenza cruciale è che

Infine, si applica il *Metodo di Newton*, che come descritto precedentemente converge in un'iterazione: è necessario calcolare l'Hessiano rispetto a β, che diventa sempre più dispendioso in quanto è una matrice di dimensione che cresce quadraticamente con la dimensione di input.

Nel codice *TrainingMultistep* l'approccio è lo stesso del codice precedente, ovvero si vuole far imparare al modello Student un singolo parametro β^*. Invece, differisce perché in questo caso le matrici simmetriche e definite positive di output sono calcolate applicando T volte la trasformazione, invece che soltanto una. Ciò rappresenta la situazione che abbiamo già trattato nella Sezione 3.2.1, quando abbiamo provato a descrivere una convergenza del processo iterativo.

```
1 function training_multistep(d_in,d_k,eps,N,maxIt,lr,beta_star,T)
2
3 alpha = eps/sqrt(d_k);
4 trainRatio = 0.75;
5
```

```
rng(1);
A = randn(d_in)/sqrt(d_in) * randn(d_in)/sqrt(d_in);
Q = randn(d_k,d_in)/sqrt(d_in);
K = randn(d_k,d_in)/sqrt(d_in);

F0 = @(S) alpha*( A*S*K'*Q*S + S*Q'*K*S*A' );

teachIter = @(S) S + beta_star*F0(S);
froMSE = @(X,Y) sum((X-Y).^2,'all')/d_in^2;

SigIn = cell(N,1);  SigOut = cell(N,1);
for i = 1:N
    S = randn(d_in);  S = S*S'/sqrt(d_in);
    SigIn{i} = S;
    Sout = S;
    for t = 1:T
        Sout = teachIter(Sout);
    end
    SigOut{i} = Sout;
end

idxTrain = 1:round(trainRatio*N);

beta = 0;
trainHist = zeros(1,maxIt);

for it = 1:maxIt
    loss = 0;
    grad = 0;
    for j = idxTrain
        S0 = SigIn{j};
        Sout = SigOut{j};
        S = S0;
        G = zeros(d_in);
        for t = 1:T
            F = F0(S);
            DF_G = alpha*(A*G*K'*Q*S + A*S*K'*Q*G + G*Q'*K*S*A' +
    S*Q'*K*G*A');
            S = S + beta*F;
            G = G + F + beta*DF_G;
        end

        E = S - Sout;
        loss = loss + froMSE(S,Sout);
        grad = grad + 2/d_in^2 * sum(E(:).*G(:));  %norm(E,G)_Fro
    end

    loss = loss/numel(idxTrain);
    grad = grad/numel(idxTrain);
```

```
54
55      beta = beta - lr*grad;
56      trainHist(it) = loss;
57      fprintf('it %2d:  beta = %+6.4f   train MSE = %.2e\n',it,beta
        ,loss);
58  end
59
60  figure;
61  plot(1:maxIt,trainHist,'-o');
62  xlabel('iterazione'); ylabel('MSE (train)');
63  title(sprintf('Batch-GD (T = %d iterazioni della mappa)',T));
64  grid on;
65  ax = gca;
66  ax.FontSize = 20; % ingrandisce i numeri degli assi
67  ax.TickLabelInterpreter = 'tex';
68  end
```

A livello del codice, ciò che cambia è che il gradiente rispetto a β viene calcolato iterativamente in un ciclo for interno, e come funzione viene usata soltanto la *Discesa Gradiente*.

Infine, l'ultimo codice rappresenta una situazione più reale e sofisticata, dove al posto che modificare un singolo valore β nel modello Student si prova a far imparare i valori delle matrici che definiscono l'Architettura.

```
1   function training_matrix(d_in, d_v, d_k, eps, N, maxIt, lr,
        miniBatch)
2
3   alpha = eps / sqrt(d_k);
4   trainRatio = 0.75;
5
6   rng(1);
7   teacher.W = randn(d_in, d_v) / sqrt(d_v);
8   teacher.V = randn(d_v, d_in) / sqrt(d_in);
9   teacher.Q = randn(d_k, d_in) / sqrt(d_in);
10  teacher.K = randn(d_k, d_in) / sqrt(d_in);
11
12  transform = @(S, W, V, Q, K)  S + alpha * (W*V*S*K'*Q*S + S*Q'*K*
        S*V'*W');
13
14  SigIn = cell(N,1);
15  SigOut = cell(N,1);
16  for i = 1:N
17      S = randn(d_in); S = S*S' / sqrt(d_in);
18      SigIn{i} = S;
19      SigOut{i}= transform(S, teacher.W, teacher.V, teacher.Q,
        teacher.K);
20  end
```

```
21
22 idx = randperm(N);
23 idxTrain = idx(1:round(trainRatio*N));
24 idxVal = idx(round(trainRatio*N)+1:end);
25
26 rng(2);
27 W = randn(d_in, d_v) / sqrt(d_v);
28 V = randn(d_v, d_in) / sqrt(d_in);
29 Q = randn(d_k, d_in) / sqrt(d_in);
30 K = randn(d_k, d_in) / sqrt(d_in);
31
32 trainLossHist = zeros(1, maxIt);
33
34 for it = 1:maxIt
35     totSE = 0;
36     for b = 1:miniBatch:numel(idxTrain)
37         batchIdx = idxTrain(b:min(b+miniBatch-1, numel(idxTrain))
   ); %per ogni batch, seleziono esattamente |batch| elementi su
    cui fare il training
38         gradW = zeros(size(W)); gradV = gradW;
39         gradQ = zeros(size(Q)); gradK = gradQ;
40         for jj = batchIdx
41             S_in = SigIn{jj};
42             S_out = SigOut{jj};
43
44             A = W*V;
45             X = S_in*K'*Q*S_in;
46             Y = S_in*Q'*K*S_in;
47
48             S_pred = transform(S_in, W, V, Q, K);
49
50             E = S_pred - S_out;
51             se = sum(E.^2,'all');
52             totSE = totSE + se;
53
54             G = (2/d_in^2)*E;
55             gradA = alpha*(G*X' + G*Y');
56             gradW = gradW + gradA*V';
57             gradV = gradV + W'*gradA;
58             term1Q = K*S_in*A'*G*S_in;
59             term2Q = S_in*G*A*S_in*K';
60             gradQ = gradQ + alpha*(term1Q + term2Q');
61             termK = S_in*A'*G*S_in*Q';
62             gradK = gradK + 2*alpha*termK';
63         end
64         nb = numel(batchIdx);
65         W = W - lr*(gradW/nb);
66         V = V - lr*(gradV/nb);
67         Q = Q - lr*(gradQ/nb);
```

```
68          K = K - lr*(gradK/nb);
69      end
70      trainLossHist(it) = totSE / (numel(idxTrain)*d_in^2);
71      fprintf('epoca %3d  |  Train-MSE = %.4e\n', it, trainLossHist
        (it));
72  end
73
74  valSE = 0;
75  for j = idxVal
76      S_pred = transform(SigIn{j}, W, V, Q, K);
77      valSE = valSE + sum((S_pred - SigOut{j}).^2, 'all');
78  end
79  valLoss = valSE / (numel(idxVal)*d_in^2);
80
81  fprintf('\nTrain-MSE ultima epoca : %.4e\n', trainLossHist(end));
82  fprintf('Validation-MSE: %.4e\n', valLoss);
83
84  figure;
85
86  plot(1:maxIt, trainLossHist, '-o', 'DisplayName','Train MSE');
        hold on;
87  yline(valLoss, '--', 'Validation MSE', 'DisplayName', sprintf('
        Val = %.2e', valLoss));
88  xlabel('epoca');
89  ylabel('MSE');
90  title('Apprendimento parametri W, V, Q, K');
91  legend('Location','northeast');
92  grid on;
93  ax = gca;
94  ax.FontSize = 20;espre
95  ax.TickLabelInterpreter = 'tex';
96  end
```

Lo scheletro del codice è molto simile, solamente vengono inizializzati due gruppi di matrici, rispettivamente per il Teacher e per lo Student, e poi nel ciclo di Training si usano una dimensione di *Batch* e la Discesa Gradiente Stocastica, al fine di ridurre il costo di ogni iterazione.

Vengono dunque calcolati i gradienti per ogni matrice utilizzando la *Chain-Rule* per la derivazione, e viene aggiornato ciascun parametro.

Infine, allo scopo di ottenere un grafico simile a quello riportato in 3.3.3, alla riga 86 è possibile sostituire *semilogy* a *plot*, così che l'asse y adotta una scala logaritmica, e la discesa della loss e meglio visibile anche per le ultime iterazioni.

Bibliografia

[1] Gabriel Peyré, Takashi Furuya, Maarten de Hoop, Valérie Castin, Pierre Ablin, *Transformers are Universal in Context Learners* (2024)

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jacob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, *Attention Is All You Need* (2017)

[3] Chris Monico, *An elementary proof of a universal approximation theorem* (2024)

[4] Gabriel Peyré, Takashi Furuya, Maarten de Hoop, Valérie Castin, Pierre Ablin, *Slides about 'Transformers are Universal in Context Learners'* (2024)

[5] Dzmitry Bahdanau, KyungHyun Cho, Yoshua Bengio, *Neural Machine Translation by Jointly Learning to Align and Translate* (2014)

[6] 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)

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan et al., *Language Models are Few-Shot Learners* (2020)

[8] Jisoo Kim, Jungbin Cho1, Joonho Park, Soonmin Hwang, Da Eun Kim, Geon Kim, Youngjae Yu, DEEPTalk: Dynamic Emotion Embedding for Probabilistic Speech-Driven 3D Face Animation (2024)

[9] George V. Cybenko, Approximation by superpositions of a sigmoidal function (1989)

[10] Kurt Hornik, Maxwell Stinchcombe, and Halbert White, Multilayer feedforward networks are universal approximators (1989)

[11] Peter Sutor Jr, Cornelia Fermuller, Yiannis Aloimonos, Douglas Summers-Stay, Metaconcepts: Isolating Context in Word Embeddings (2019)

[12] Grant Sanderson, Neural Networks (2017)

[13] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean, Efficient Estimation of Word Representations in Vector Space (2013)

[14] E. P. Wigner, *On the Distribution of the Roots of Certain Symmetric Matrices* (1958).

[15] V. A. Marchenko, L. A. Pastur, *Distribution of eigenvalues for some sets of random matrices* (1958).