SEBASTIAN RASCHKA

Sviluppare Large Language Model

Costruire da zero LLM su misura

APOGEO

SEBASTIAN RASCHKA

Sviluppare Large Language Model

Costruire da zero LLM su misura

APOGEO

Sviluppare Large Language Model
Costruire da zero LLM su misura

Sebastian Raschka

APOGEO

© Apogeo - Giangiacomo Feltrinelli Editore S.r.l.

ISBN ebook: 9788850338832

Original English–language edition published with the title “Build a Large Language Model (From Scratch)”. Authorized translation of the English edition © 2025 Manning Publications Co. All rights reserved. This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.

Il presente file può essere usato esclusivamente per finalità di carattere personale. Tutti i contenuti sono protetti dalla Legge sul diritto d’autore.

Nomi e marchi citati nel testo sono generalmente depositati o registrati dalle rispettive case produttrici.

L’edizione cartacea è in vendita nelle migliori librerie.

~

Sito web: www.apogeonline.com

Scopri le novità di Apogeo su Facebook

Seguici su X

Collegati con noi su LinkedIn

Guarda cosa stiamo facendo su Instagram

Rimani aggiornato iscrivendoti alla nostra newsletter

Premessa

Sono sempre stato affascinato dai modelli linguistici. Più di un decennio fa, ho iniziato il mio viaggio nell'intelligenza artificiale con un corso di classificazione di pattern statistici, per il quale ho sviluppato il mio primo progetto: un modello e un'applicazione web per rilevare il mood di una canzone in base al testo.

Avanti veloce fino al 2022: con l'uscita di ChatGPT, i modelli LLM (*Large Language Model*) hanno invaso il mondo e hanno rivoluzionato il modo di lavorare di molti di noi. Questi modelli sono incredibilmente versatili e utili in attività come il controllo grammaticale, la stesura di e-mail, la sintesi di lunghi documenti e molto altro. Ciò è dovuto alla loro capacità di analizzare e generare testi, il che è importante in vari campi, dal supporto clienti alla creazione di contenuti e perfino in ambiti più tecnici come la programmazione e l'analisi dei dati.

Come suggerisce il "Large" nel nome, un segno distintivo dei modelli LLM è che sono "grandi", molto grandi, e comprendono milioni o anche miliardi di parametri. A titolo di confronto, utilizzando metodi statistici o di machine learning più tradizionali, il dataset sui fiori Iris può essere classificato con un'accuratezza superiore al 90 percento utilizzando un piccolo modello formato da due soli parametri. Tuttavia, nonostante le loro grandi dimensioni rispetto ai metodi più tradizionali, i modelli LLM non devono necessariamente essere "scatole nere".

In questo libro imparerete a realizzare un modello LLM un passo alla volta. Alla fine, avrete una solida comprensione di come funziona un modello LLM, come quelli usati in ChatGPT. Credo che per avere successo sia fondamentale sviluppare una solida conoscenza dei concetti fondamentali e del codice sottostante. Questo non solo aiuta a correggere i bug e a migliorare le prestazioni, ma consente anche di sperimentare nuove idee.

Diversi anni fa, quando iniziai a lavorare con i modelli LLM, ho dovuto imparare a implementarli nel modo più “duro”, consultando molti scritti e repository di codice incompleti. Con questo libro, spero di rendere più accessibile l’argomento, sviluppando e condividendo un tutorial di implementazione che descriva in dettaglio tutti i componenti principali e le varie fasi di sviluppo di un modello LLM.

Sono fermamente convinto che il modo migliore per comprendere i modelli LLM sia programmarne uno partendo da zero. E scoprirete che la cosa può anche essere divertente!

E allora, buona lettura e buona programmazione!

Ringraziamenti

Scrivere un libro è un’impresa importante, e vorrei esprimere la mia più sincera gratitudine a mia moglie, Liza, per la sua pazienza e il suo supporto durante tutto questo processo. Il suo amore incondizionato e il suo costante incoraggiamento sono stati assolutamente essenziali, per me.

Sono indicibilmente grato a Daniel Kleine, il cui feedback inestimabile sui capitoli e sul codice è andato ben oltre il dovuto. Col suo occhio attento per i dettagli e i suoi suggerimenti perspicaci, i contributi di Daniel hanno senza dubbio reso questo libro un’esperienza di lettura più fluida e piacevole.

Vorrei anche ringraziare il meraviglioso staff di Manning Publications, tra cui Michael Stephens, per le numerose discussioni che hanno contribuito a dare una migliore direzione di questo libro, e Dustin Archibald, il cui feedback costruttivo e la cui guida nell’aderire alle linee guida di Manning sono stati cruciali. Apprezzo anche la loro flessibilità nell’adattare i requisiti unici di questo approccio non convenzionale. Un ringraziamento speciale va ad Aleksandar Dragosavljević, Kari Lucke e Mike Beady, per il loro lavoro sui layout professionali, e a Susan Honeywell e al suo team, per aver rifinito e perfezionato la grafica.

Desidero esprimere la mia più sincera gratitudine a Robin Campbell e al suo straordinario team di marketing, per il loro prezioso supporto durante tutto il processo di scrittura.

Infine, esprimo i miei ringraziamenti ai revisori: Anandaganesh Balakrishnan, Anto Aravinth, Ayush Bihani, Bassam Ismail, Benjamin Muskalla, Bruno Sonnino, Christian Prokopp, Daniel Kleine, David Curran, Dibyendu Roy Chowdhury, Gary Pass, Georg Sommer, Giovanni Alzetta, Guillermo Alcántara, Jonathan Reeves, Kunal Ghosh, Nicolas Modrzyk, Paul Silisteanu, Raul Ciotescu, Scott Ling, Sriram Macharla, Sumit Pal, Vahid Mirjalili, Vaijanath Rao e Walter Reade per il loro feedback approfondito sulle bozze. I vostri occhi attenti e i vostri commenti sono stati essenziali per migliorare la qualità di questo libro.

Sono sinceramente grato a tutti coloro che hanno contribuito a questo viaggio. Il vostro supporto, la vostra competenza e la vostra dedizione sono stati determinanti nel portare a compimento questo libro. Grazie!

Introduzione

Questo libro è stato scritto per aiutarvi a comprendere e creare da zero un modello LLM (*Large Language Model*) simile a un GPT. Il libro si apre concentrandosi sui fondamenti dell'elaborazione di dati testuali e sulla programmazione dei meccanismi di attenzione, poi guida nell'implementazione da zero di un modello GPT. Il libro tratta poi il meccanismo di pre-addestramento e di fine-tuning per attività specifiche, come la classificazione del testo e la risposta a istruzioni. Alla fine del libro, avrete una profonda conoscenza del funzionamento dei modelli LLM e le competenze necessarie per costruire i vostri modelli. I modelli che creerete saranno di dimensioni ridotte rispetto ai grandi modelli commerciali, ma utilizzano i medesimi concetti e fungono da potenti strumenti didattici per comprendere i meccanismi e le tecniche utilizzati nella creazione dei modelli LLM oggi all'avanguardia.

A chi si rivolge questo libro

Questo libro è dedicato agli appassionati di machine learning, agli ingegneri, ai ricercatori, agli studenti e ai professionisti che desiderano acquisire una profonda comprensione del funzionamento dei modelli LLM e imparare a costruire i propri modelli. Sia i principianti sia gli sviluppatori più esperti saranno in grado di utilizzare le proprie competenze per comprendere i concetti e le tecniche utilizzate nella creazione di modelli LLM.

Ciò che distingue questo libro è la sua “copertura” completa dell’intero processo di creazione di un modello LLM, dal lavoro con i dataset all’implementazione dell’architettura del modello, al pre-addestramento su dati non etichettati e alla sua messa a punto per attività specifiche. Al momento della stesura, nessun’altra risorsa fornisce un approccio così completo e pratico alla creazione di modelli LLM partendo da zero.

Per comprendere gli esempi di codice presentati in questo libro, occorre avere una solida conoscenza nel campo della programmazione Python. Sebbene una certa familiarità con il machine learning, il deep learning e l’intelligenza artificiale possa essere utile, non è richiesta una vera formazione approfondita in questi campi. I modelli LLM sono un sottoinsieme specifico dell’intelligenza artificiale, quindi anche se siete relativamente alle prime armi in questo campo, sarete in grado di seguire le indicazioni fornite.

Se avete una certa esperienza con le reti neurali profonde, alcuni concetti vi risulteranno più familiari, poiché i modelli LLM si basano su queste architetture. Tuttavia, la competenza in PyTorch non è un prerequisito. L’Appendice A costituisce un’introduzione sintetica a PyTorch, per dotarvi delle competenze necessarie per comprendere gli esempi di codice presenti nel libro.

Mentre esploreremo il funzionamento dei modelli LLM, scoprirete che vi sarà utile ricorrere alla matematica che avete studiato alle scuole superiori, in particolare quando lavoreremo con vettori e matrici. Tuttavia, per poter comprendere i concetti e le idee chiave presentati in questo libro non è necessaria una conoscenza troppo avanzata.

Il prerequisito più importante è una solida base nella programmazione Python. Con queste conoscenze, sarete ben

equipaggiati a esplorare l’affascinante mondo dei modelli LLM e a comprendere i concetti e gli esempi di codice presenti nel libro.

Struttura del libro: una tabella di marcia

Questo libro è progettato per essere letto in sequenza, poiché ogni capitolo si basa sui concetti e sulle tecniche introdotti nei capitoli precedenti. Il libro è diviso in sette capitoli che trattano gli aspetti essenziali dei modelli LLM e della loro implementazione.

- Il Capitolo 1 è un’introduzione generale ai concetti fondamentali che sono alla base dei modelli LLM. Esplora l’architettura a transformer, che costituisce la base per i modelli LLM come quelli utilizzati sulla piattaforma ChatGPT.
- Il Capitolo 2 espone il piano per costruire un modello LLM da zero. Tratta il processo di preparazione del testo per l’addestramento del modello LLM, inclusa la suddivisione del testo in token di parole e sub-parole, l’utilizzo della codifica a coppie di byte per la tokenizzazione avanzata, il campionamento di esempi di addestramento con un approccio a finestra scorrevole e la conversione dei token in vettori che alimentano il modello LLM.
- Il Capitolo 3 si concentra sui meccanismi di attenzione utilizzati nei modelli LLM. Introduce un semplice framework di auto-attenzione e progredisce verso un meccanismo di auto-attenzione più avanzato. Il capitolo tratta anche l’implementazione di un modulo di attenzione causale, che consente ai modelli LLM di generare un token alla volta, mascherando determinati pesi di attenzione, selezionati casualmente, con un dropout, per ridurre l’overfitting e

impilando più moduli di attenzione causale in un modulo di attenzione multi-head.

- Il Capitolo 4 si concentra sulla programmazione di un modello LLM di tipo GPT che può essere addestrato per generare testo di tipo umano. Tratta tecniche come la normalizzazione dei layer per stabilizzare l'addestramento della rete neurale, l'aggiunta di connessioni scorciatoia nelle reti neurali profonde per addestrare i modelli in modo più efficace, l'implementazione di blocchi transformer per creare modelli GPT di varie dimensioni e il calcolo del numero di parametri e dei requisiti di archiviazione dei modelli GPT.
- Il Capitolo 5 implementa il processo di pre-addestramento dei modelli LLM. Tratta il calcolo dei valori loss dei set di addestramento e di validazione, per valutare la qualità del testo generato dal modello LLM, l'implementazione di una funzione di addestramento e il pre-addestramento del modello LLM, il salvataggio e il caricamento dei pesi del modello, per continuare ad addestrarlo, e il caricamento dei pesi pre-addestrati da OpenAI.
- Il Capitolo 6 introduce diversi approcci di fine-tuning di un modello LLM. Tratta la preparazione di un dataset per la classificazione del testo, la modifica di un modello LLM pre-addestrato per passare al fine-tuning e poi il fine-tuning di un modello LLM per identificare i messaggi di spam e la valutazione dell'accuratezza di un classificatore LLM dopo il fine-tuning.
- Il Capitolo 7 esplora il processo di fine-tuning dei modelli LLM per rispondere a istruzioni. Tratta la preparazione di un dataset per il fine-tuning per rispondere a istruzioni con supervisione, l'organizzazione dei dati delle istruzioni in batch di

addestramento, il caricamento di un modello LLM pre-addestrato e il suo fine-tuning per rispondere a istruzioni umane, l'estrazione delle risposte generate dal modello LLM e la valutazione di un modello LLM dopo il fine-tuning.

Il codice

Per rendere il più semplice possibile la lettura, tutti gli esempi di codice di questo libro sono disponibili sul sito web di Apogeo all'indirizzo `https://bit.ly/apo-sllm`, nonché nel formato Jupyter notebook su GitHub all'indirizzo `https://github.com/rasbt/LLMs-from-scratch`. E se vi sentite bloccati, le soluzioni a tutti gli esercizi si trovano nell'Appendice C.

Questo libro contiene molti esempi di codice in listati numerati e in linea nel testo. In entrambi i casi, il codice sorgente è formattato in un `font a larghezza fissa` per distinguerlo dal testo ordinario.

In molti casi, il codice sorgente originale è stato riformattato; abbiamo aggiunto interruzioni di riga e rielaborato l'indentazione per adattarci allo spazio disponibile nella pagina del libro. Inoltre, molti commenti sono stati rimossi dai listati quando il codice era già descritto nel testo. Le annotazioni al codice che accompagnano molti listati evidenziano i concetti più importanti.

Uno degli obiettivi principali di questo libro è l'accessibilità, quindi gli esempi di codice sono stati calibrati per funzionare in modo efficiente anche su un normale laptop, senza la necessità di dotarsi hardware speciale. Ma se avete accesso a una GPU, alcuni paragrafi forniscono suggerimenti utili per estendere di scala i dataset e i modelli in modo da sfruttare la loro potenza.

Nel corso del libro, utilizzeremo PyTorch come libreria di riferimento per i tensori e il deep learning. Se non conoscete PyTorch,

vi consiglio di iniziare con l'Appendice A, che fornisce un'introduzione approfondita, completa di consigli di configurazione.

Altre risorse online

Siete interessati alle ultime tendenze della ricerca in materia di intelligenza artificiale e modelli LLM?

- Date un'occhiata al mio blog su `https://magazine.sebastianraschka.com`, dove parlo regolarmente delle ultime ricerche sull'intelligenza artificiale, con particolare attenzione ai modelli LLM.

Avete bisogno di aiuto per iniziare a usare al meglio il deep learning e PyTorch?

- Offro diversi corsi gratuiti sul mio sito web all'indirizzo `https://sebastianraschka.com/teaching`. Queste risorse possono mettervi rapidamente al passo con le ultime tecniche.

Cercate materiali bonus correlati al libro?

- Visitate il repository GitHub del libro all'indirizzo `https://github.com/rasbt/LLMs-from-scratch` per trovare ulteriori risorse ed esempi per integrare il vostro apprendimento.

L'autore

Sebastian Raschka, PhD, lavora nel campo del machine learning e dell'intelligenza artificiale da oltre un decennio. Oltre a essere un ricercatore, Sebastian ha una grande passione per l'istruzione. È noto per i suoi bestseller sul machine learning con Python e per i suoi contributi al software open source.

Sebastian è ingegnere ricercatore presso *Lightning AI*, specializzato nell'implementazione e nell'addestramento di modelli LLM. Prima della sua esperienza in questo settore, Sebastian è stato professore associato presso il Dipartimento di Statistica dell'Università del Wisconsin-Madison, dove si è concentrato sulla ricerca relativa al deep learning. Per saperne di più su Sebastian: https://sebastianraschka.com.

Capitolo 1
I modelli LLM

I modelli LLM (*Large Language Model*), come quelli offerti da ChatGPT di OpenAI, sono modelli a reti neurali profonde, una tecnologia sviluppata in questi ultimi anni. Essi hanno inaugurato una nuova era nell'elaborazione del linguaggio naturale (NLP – *Natural Language Processing*). Prima dell'avvento dei modelli LLM, i metodi tradizionali eccellevano in attività differenti, di catalogazione, come la classificazione delle e-mail di spam e il riconoscimento di semplici pattern, che potevano essere catturati con regole specifiche o modelli più semplici. Tuttavia, in genere offrivano prestazioni inferiori nelle attività linguistiche, che richiedevano capacità più complesse di comprensione e generazione, come l'analisi di istruzioni dettagliate, l'esecuzione di analisi contestuali e la creazione di testo coerente e appropriato al contesto. Per esempio, le generazioni precedenti di modelli linguistici non erano in grado di scrivere un'e-mail in base a un elenco di parole chiave, un'attività che oggi è banale per i modelli LLM.

I modelli LLM offrono capacità notevoli nel comprendere, generare e interpretare il linguaggio umano. Tuttavia, è importante chiarire che, quando diciamo che i modelli linguistici "capiscono", intendiamo far riferimento solo alla loro capacità di elaborare e generare testo in modi che ci appaiono coerenti e contestualmente rilevanti, non al fatto che possiedano una coscienza o una comprensione dei concetti simili a quelle umane.

Grazie ai progressi nel deep learning, un sottoinsieme del machine learning e dell'intelligenza artificiale focalizzato sulle reti neurali, i modelli LLM sono stati e sono addestrati su grandi quantità di dati testuali. Questa formazione su larga scala consente loro di catturare maggiori informazioni contestuali e sottigliezze del linguaggio umano rispetto agli approcci precedenti. Di conseguenza, i modelli LLM hanno migliorato significativamente le prestazioni in un'ampia varietà di attività di elaborazione del linguaggio naturale, tra cui la traduzione di testi, l'analisi del sentiment, le risposte alle domande e molto altro.

Un'altra distinzione importante tra i precedenti modelli di elaborazione del linguaggio naturale e i nuovi modelli LLM è che in genere i primi erano progettati per compiti specifici, come la catalogazione del testo, la traduzione linguistica e così via. Mentre quei primi modelli eccellevano nelle loro specifiche applicazioni, i modelli LLM dimostrano una competenza più ampia in una gamma più generale di compiti di elaborazione del linguaggio naturale.

Il successo dei modelli LLM può essere attribuito all'architettura a transformer su cui si basano molti modelli LLM e alle grandi quantità di dati su cui tali modelli vengono formati, che consente loro di catturare un'ampia gamma di sfumature, contesti e modelli linguistici che sarebbero difficili da programmare manualmente.

Questo passo implementativo a modelli basati sull'architettura a transformer e all'utilizzo di grandi dataset di addestramento per addestrare i modelli LLM ha trasformato radicalmente l'elaborazione del linguaggio naturale, fornendo strumenti più efficaci per comprendere il linguaggio umano e interagire con esso.

La discussione di questo capitolo getta le basi per raggiungere l'obiettivo principale di questo libro: comprendere i modelli LLM

implementando passo dopo passo tramite codice un modello LLM simile a ChatGPT, basato sull'architettura a transformer.

Che cosa è un modello LLM?

Un modello LLM è una rete neurale progettata per comprendere, generare e rispondere a testi simili a quelli producibili da un essere umano. Questi modelli sono reti neurali profonde addestrate su enormi quantità di dati testuali, che a volte comprendono grandi porzioni dell'intero sapere testuale disponibile oggi al pubblico su Internet.

Il termine *Large* di *Large Language Model* si riferisce sia alle dimensioni del modello in termini di parametri, sia all'immenso dataset sui quali è addestrato. Si tratta di modelli che spesso hanno decine o addirittura centinaia di miliardi di parametri, che sono i pesi regolabili della rete, che durante l'addestramento vengono ottimizzati per prevedere la parola successiva in una sequenza. La previsione della parola successiva è sensata, perché sfrutta la natura sequenziale intrinseca del linguaggio per addestrare i modelli sulla comprensione del contesto, della struttura e delle relazioni all'interno del testo. Tuttavia, questo è un compito molto semplice, e quindi molti ricercatori si sorprendono del fatto che possa produrre modelli così efficaci. Nei prossimi capitoli, discuteremo e implementeremo la procedura di addestramento alla scelta della parola successiva, passo dopo passo.

I modelli LLM utilizzano un'architettura chiamata a *transformer*, che consente loro di prestare un'attenzione selettiva a diverse parti dell'input nel fare le loro previsioni, cosa che li rende particolarmente abili nel gestire le sfumature e le complessità del linguaggio umano.

Poiché i modelli LLM sono in grado di *generare* testo, spesso sono considerati una forma di *intelligenza artificiale generativa*, spesso abbreviata come *GenAI*. Come mostra la Figura 1.1, l'intelligenza artificiale comprende il campo più ampio della creazione di macchine in grado di eseguire attività che richiedono un'intelligenza di tipo umano, tra cui la comprensione del linguaggio, il riconoscimento di schemi e la presa di decisioni, e include sottocampi come il machine learning e il deep learning.

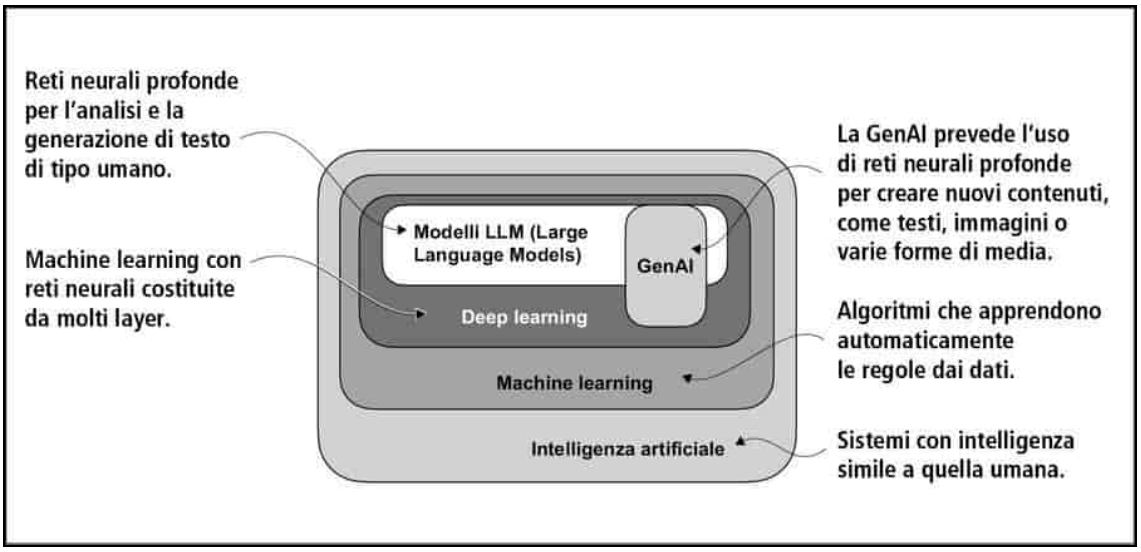


Figura 1.1 Come suggerisce questa rappresentazione gerarchica della relazione tra i diversi campi, i modelli LLM rappresentano un'applicazione specifica delle tecniche di deep learning, utilizzando la loro capacità di elaborare e generare testo di tipo umano. Il deep learning è una branca specializzata del machine learning, che si concentra sull'utilizzo di reti neurali multi-layer. Il machine learning e il deep learning sono campi volti a implementare algoritmi che consentano ai computer di apprendere dai dati e di eseguire attività che in genere richiedono un'intelligenza umana.

Gli algoritmi utilizzati per implementare l'intelligenza artificiale sono al centro del campo del machine learning. In particolare, il machine learning implica lo sviluppo di algoritmi in grado di apprendere, fare previsioni o prendere decisioni sulla base di dati ma

senza essere programmati in modo esplicito. Per illustrare ciò, immaginate un'applicazione pratica del machine learning: un filtro antispam. Invece di scrivere manualmente le regole per identificare le e-mail di spam, un algoritmo di machine learning riceve esempi di e-mail pre-etichettate come spam e legittime. Riducendo al minimo l'errore nelle sue previsioni su un dataset di addestramento, il modello impara a riconoscere gli schemi e le caratteristiche indicativi di spam, cosa che gli consente di classificare le nuove e-mail come spam o non spam.

Come illustrato nella precedente Figura 1.1, il deep learning è un sottoinsieme del machine learning che si concentra sull'utilizzo di reti neurali a tre o più layer (chiamate anche reti neurali profonde) per modellare gli schemi e le astrazioni complesse presenti nei dati. A differenza del deep learning, il machine learning tradizionale richiede l'estrazione manuale delle caratteristiche. Ciò significa che ci devono essere esperti umani che si occupano di identificare e selezionare le caratteristiche più rilevanti per il modello.

Sebbene il campo dell'intelligenza artificiale sia oggi dominato dal machine learning e dal deep learning, include anche altri approcci, per esempio l'uso di sistemi basati su regole, algoritmi genetici, sistemi esperti, logica fuzzy o ragionamento simbolico.

Tornando all'esempio della classificazione dello spam, nel machine learning tradizionale, degli esperti umani potrebbero estrarre manualmente certe caratteristiche dal testo dell'e-mail, come la frequenza di determinate parole trigger (per esempio, "premio", "vincita", "gratis"), il numero di punti esclamativi, l'uso di parole tutte in maiuscolo o la presenza di link sospetti. Questo dataset, creato in base a queste caratteristiche definite, appunto, da esperti, viene poi utilizzato per addestrare il modello. A differenza del machine learning tradizionale, il deep learning non richiede

l’estrazione manuale delle caratteristiche. Ciò significa che non è necessario che degli esperti umani identifichino e selezionino le caratteristiche più rilevanti per il modello di deep learning. Tuttavia, sia il machine learning tradizionale sia il deep learning per la classificazione dello spam richiedono comunque la raccolta di etichette, spam/non-spam, che devono essere raccolte da un esperto o dagli utenti.

Esaminiamo ora alcuni dei problemi che i modelli LLM possono risolvere oggi, le sfide che devono affrontare e l’architettura generale dei modelli LLM che implementeremo nel corso del libro.

Applicazioni dei modelli LLM

Grazie alle loro capacità avanzate di analizzare e comprendere dati testuali non strutturati, i modelli LLM hanno un’ampia gamma di applicazioni in vari domini. Oggi, sono impiegati per la traduzione automatica, la generazione di nuovi testi (vedi Figura 1.2), l’analisi del sentiment, la sintesi di testi e molti altri compiti, e sono sempre più utilizzati per la creazione di nuovi contenuti, come la scrittura di narrativa, articoli e perfino codice di programmazione.

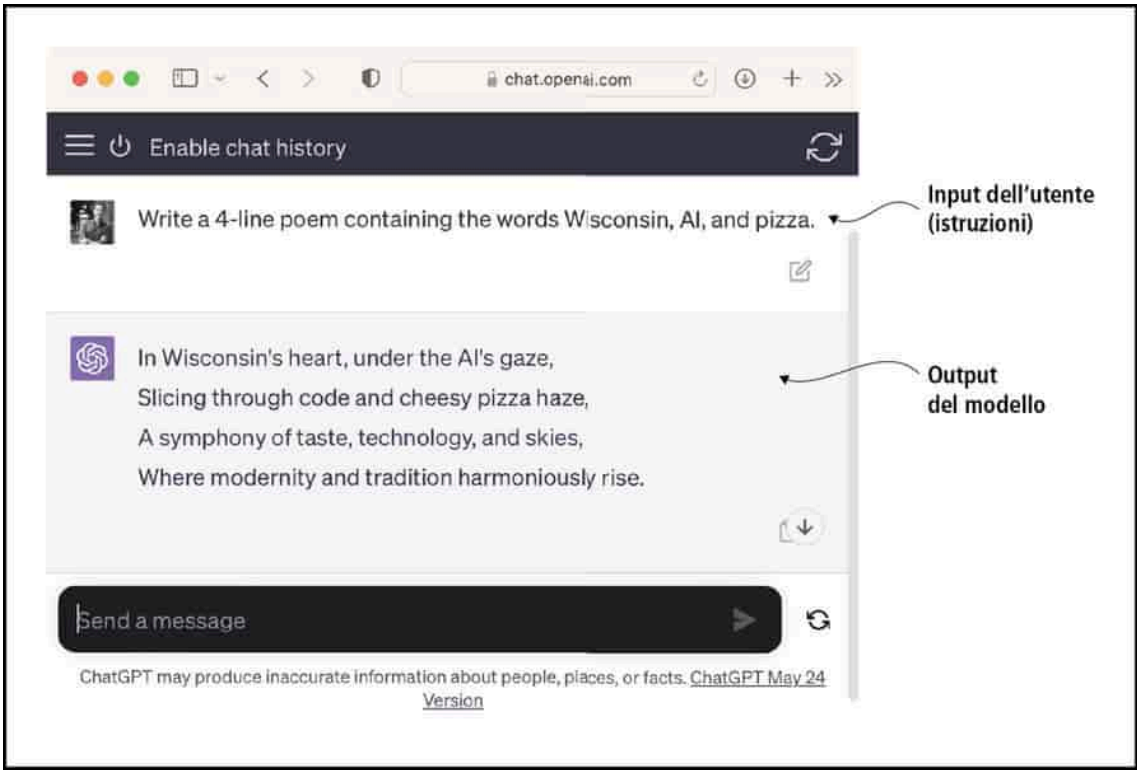


Figura 1.2 Le interfacce LLM rendono possibile la comunicazione in linguaggio naturale tra gli utenti e un sistema. Questa schermata mostra ChatGPT che scrive una poesia sulla base delle specifiche di un utente.

I modelli LLM possono anche alimentare chatbot e assistenti virtuali sofisticati, come ChatGPT di OpenAI o Gemini di Google, che possono rispondere alle query degli utenti e potenziare i motori di ricerca tradizionali come Google Search o Microsoft Bing.

Inoltre, i modelli LLM possono essere utilizzati per un recupero efficace delle conoscenze da grandi volumi di testo in aree specializzate come la medicina e il diritto. Ciò include l'analisi dei documenti, la sintesi di lunghi passaggi e la creazione di risposte a domande tecniche.

In breve, i modelli LLM sono utili per automatizzare quasi ogni compito che implichi l'analisi e la generazione di testi. Le loro applicazioni sono praticamente infinite e, mentre continuiamo a

innovare ed esplorare nuovi modi per utilizzare questi modelli, è sempre più chiaro che essi hanno le potenzialità per ridefinire il nostro rapporto con la tecnologia, rendendolo più colloquiale, intuitivo e accessibile.

Pertanto, ci concentreremo sulla comprensione del funzionamento dei modelli LLM, a partire dalle basi, programmando un modello LLM in grado di generare testi. Imparerete anche le tecniche che consentono ai modelli LLM di eseguire query, che vanno dalla risposta alle domande alla sintesi del testo, alla traduzione del testo in diverse lingue e altro ancora. In altre parole, studieremo il funzionamento di assistenti complessi come ChatGPT, costruendone uno passo dopo passo.

Fasi di costruzione e utilizzo dei modelli LLM

Per quale motivo dovremmo creare i nostri LLM? Perché programmare un modello LLM da zero è un esercizio eccellente per comprenderne i meccanismi e i limiti. Inoltre, ci fornisce le conoscenze necessarie per pre-addestrare o perfezionare le architetture LLM open source esistenti, in base ai nostri dataset o compiti specifici del nostro dominio.

> **NOTA**
> La maggior parte dei modelli LLM di oggi è implementata utilizzando la libreria di deep learning PyTorch, che è quella che useremo anche noi. I lettori possono trovare un'introduzione completa a PyTorch nell'Appendice A.

La ricerca ha dimostrato che, in termini di prestazioni di modellazione, i modelli LLM personalizzati, ovvero concepiti su misura per specifici compiti o domini, possono superare i modelli LLM generici, come quelli forniti da ChatGPT, rivolti a un'ampia gamma di applicazioni. Esempi di questi includono BloombergGPT

(specializzato per la finanza) e modelli LLM pensati su misura per rispondere a domande mediche (vedi l'Appendice B per maggiori dettagli).

L'utilizzo di modelli LLM personalizzati offre diversi vantaggi, in particolare per quanto riguarda la privacy dei dati. Per esempio, molte aziende potrebbero non voler condividere dati sensibili con fornitori terzi di modelli LLM come OpenAI, a causa di problemi di riservatezza. Inoltre, lo sviluppo di modelli LLM personalizzati e più piccoli ne consente l'implementazione direttamente sui dispositivi dei clienti, anche laptop e smartphone, un'idea che aziende come Apple stanno attualmente esplorando. Questa implementazione locale può ridurre significativamente sia la latenza sia i costi di utilizzo dei server. Inoltre, i modelli LLM personalizzati garantiscono agli sviluppatori la massima autonomia, consentendo loro di aggiornare e modificare il modello in base alle necessità.

Il processo generale di creazione di un modello LLM include una fase di pre-addestramento e fine-tuning. Il "pre" in "pre-addestramento" si riferisce alla fase iniziale in cui il modello (anche LLM) viene addestrato su un dataset ampio e diversificato per sviluppare una comprensione generale del linguaggio. Questo modello pre-addestrato fungerà poi da risorsa fondamentale che può essere ulteriormente perfezionata, un processo in cui il modello viene addestrato in modo specifico su un dataset più ristretto, specifico per determinate attività o domini. Questo approccio all'addestramento in due fasi costituito da pre-addestramento e fine-tuning è raffigurato nella Figura 1.3.

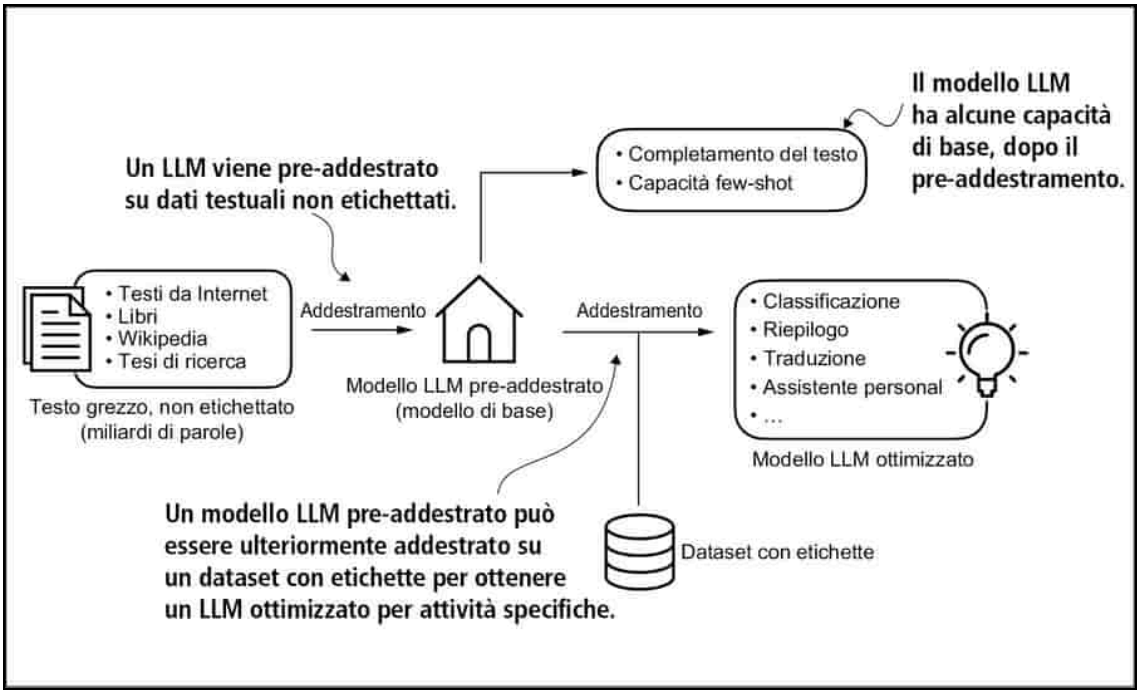


Figura 1.3 Il pre-addestramento di un modello LLM comporta la previsione della parola successiva su grandi dataset testuali. Un modello LLM pre-addestrato può poi essere perfezionato utilizzando un dataset con etichette più piccolo.

Il primo passo per creare un modello LLM consiste nell'addestrarlo su un ampio corpus di dati testuali, *testo grezzo*, dove "grezzo" si riferisce al fatto che questi dati sono solo puro testo, senza alcuna etichettatura (ma potrebbero essergli stati applicati dei filtri, come la rimozione di caratteri di formattazione o l'imposizione di usare documenti in altre lingue).

NOTA

Coloro che hanno un background relativo al machine learning potrebbero notare che in genere le informazioni di etichettatura sono necessarie per i modelli tradizionali di machine learning e per le reti neurali profonde addestrate tramite il paradigma tradizionale di apprendimento con supervisione convenzionale. Tuttavia, la fase di pre-addestramento dei modelli LLM è differente. In questa fase, i modelli LLM utilizzano l'apprendimento ad auto-supervisione, in cui il modello genera da sé le proprie etichette dai dati di input.

Questa prima fase di addestramento di un modello LLM è chiamata *pre-addestramento*, e crea un primo modello LLM pre-addestrato, chiamato *Modello di base*. Un tipico esempio di tale modello è GPT-3 (il precursore del modello offerto da ChatGPT). Questo modello è in grado di completare il testo, ovvero di finire una frase iniziata da un utente. Ha anche limitate capacità *few-shot*, ovvero può imparare a svolgere nuove attività basandosi anche solo su pochi esempi, anziché aver bisogno di dati di addestramento estese.

Dopo aver ottenuto un modello LLM pre-addestrato tramite l'addestramento su grandi dataset testuali, in cui il modello LLM viene solo addestrato a prevedere la parola successiva nel testo, possiamo addestrare ulteriormente il modello LLM su dati etichettati, operazione nota anche come *fine-tuning*.

Le due categorie più popolari di fine-tuning dei modelli LLM sono il *fine-tuning per rispondere a istruzioni* e il *fine-tuning per attività di classificazione*. Nel fine-tuning per rispondere a istruzioni, il dataset con etichette è costituito da coppie istruzione-risposta, come una query che chiede di tradurre un testo, accompagnata dal testo tradotto correttamente. Nel fine-tuning per attività di classificazione, il dataset con etichette è costituito da testi ed etichette delle classi, per esempio, e-mail associate alle etichette "spam" e "non spam".

Tratteremo le implementazioni del codice di pre-addestramento e fine-tuning di un modello LLM e approfondiremo le specifiche del fine-tuning per rispondere a istruzioni e a classificazioni dopo il pre-addestramento di un semplice modello LLM.

Introduzione all’architettura a transformer

La maggior parte dei modelli LLM di oggi si basa sull’architettura a transformer, un’architettura a rete neurale profonda introdotta nel documento del 2017 *Attention Is All You Need* (https://arxiv.org/abs/1706.03762). Per comprendere i modelli LLM, dobbiamo comprendere il funzionamento dei transformer, sviluppati per la traduzione automatica, traducendo testi dall’inglese al tedesco e al francese. La Figura 1.4 mostra una versione semplificata dell’architettura a transformer.

L’architettura a transformer è composta da due sub-moduli: un encoder e un decoder. Il modulo encoder elabora il testo di input e lo codifica in una serie di rappresentazioni numeriche o vettori che catturano le informazioni contestuali dell’input. Poi, il modulo decoder prende questi vettori codificati e genera il testo in output. In un’attività di traduzione, per esempio, l’encoder codificherebbe il testo dalla lingua di origine in vettori, e il decoder decodificherebbe questi vettori per generare il testo nella lingua di destinazione. Sia l’encoder sia il decoder sono costituiti da molti layer collegati da un meccanismo di auto-attenzione. A questo punto, potreste avere molte domande su come gli input vengono pre-elaborati e codificati. Affronteremo queste questioni nell’implementazione passo-passo che svolgeremo nei prossimi capitoli.

Un componente chiave dei transformer e dei modelli LLM è il meccanismo di auto-attenzione (non mostrato in figura), che consente al modello di valutare l’importanza delle diverse parole (o token) in una sequenza, l’una rispetto all’altra. Questo meccanismo consente al modello di catturare le dipendenze a lungo raggio e le relazioni contestuali all’interno dei dati di input, migliorando la sua capacità di

generare output coerenti e contestualmente rilevanti. Tuttavia, a causa della sua complessità, rimanderemo ulteriori spiegazioni al Capitolo 3, dove discuteremo e implementeremo passo dopo passo questo meccanismo.

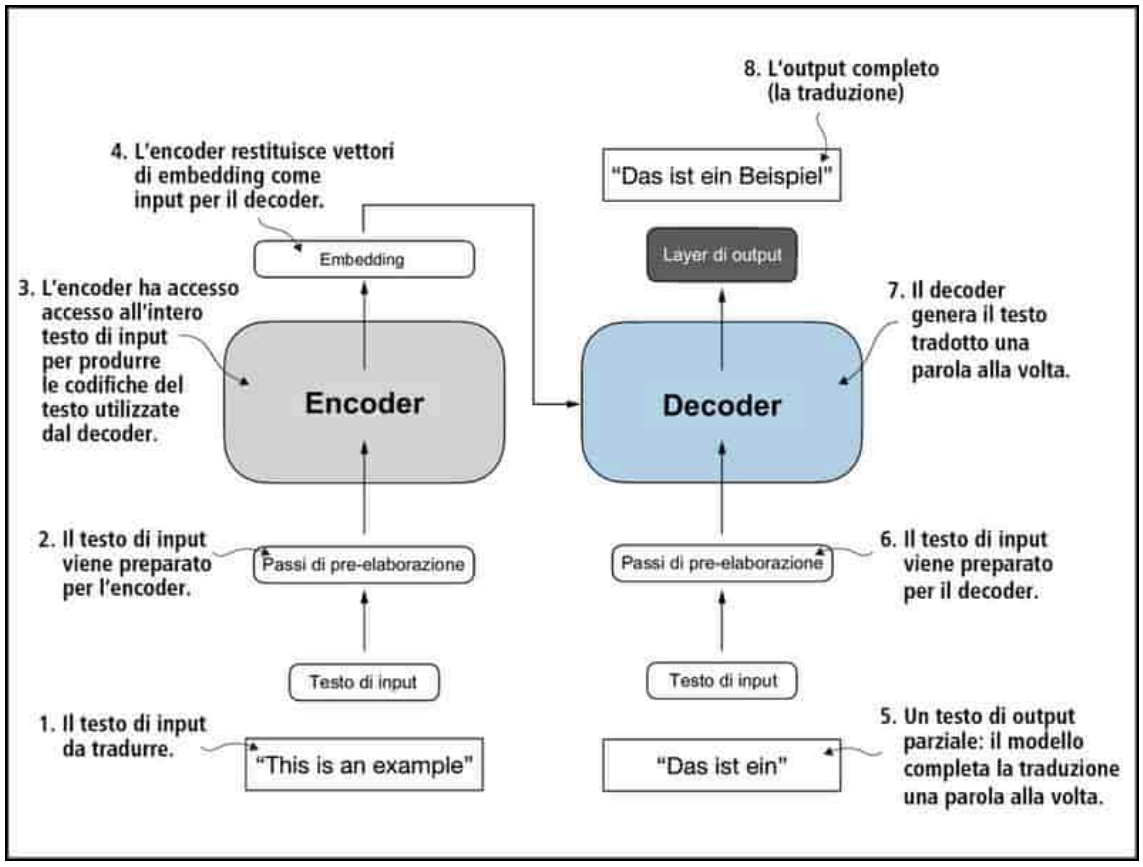


Figura 1.4 Una rappresentazione semplificata dell'architettura originale a transformer, che è un modello di deep learning per la traduzione linguistica. Il transformer è costituito da due parti: (a) un encoder che elabora il testo di input e produce una rappresentazione di embedding (una rappresentazione numerica che cattura molti fattori in varie dimensioni) del testo che il (b) decoder può utilizzare per generare il testo tradotto, una parola alla volta. Questa figura mostra la fase finale del processo di traduzione, in cui il decoder deve generare solo la parola finale ("Beispiel"), dato il testo di input originale ("This is an example") e una frase parzialmente tradotta ("Das ist ein"), per completare la traduzione.

Varianti successive dell'architettura a transformer, come BERT (*Bidirectional Encoder Representations From Transformers*) e i vari modelli GPT (*Generative Pretrained Transformer*), si sono basate su questo concetto per adattare questa architettura a diverse attività. Se siete interessati, potete fare riferimento all'Appendice B per ulteriori suggerimenti di lettura.

Un BERT, che è costruito sul sub-modulo encoder del transformer, differisce nel suo approccio all'addestramento da un GPT. Mentre un GPT è progettato per attività generative, un BERT e le sue varianti sono specializzate nella previsione di parole mascherate, dove il modello deve prevedere parole mascherate o nascoste in una data frase, come mostra la Figura 1.5. Questa speciale strategia di addestramento fornisce a BERT alcuni punti di forza nelle attività di classificazione del testo, tra cui la previsione del sentiment e la catalogazione dei documenti. Come applicazione delle sue capacità, al momento della stesura, X (ex-Twitter) utilizza un BERT per rilevare i contenuti tossici.

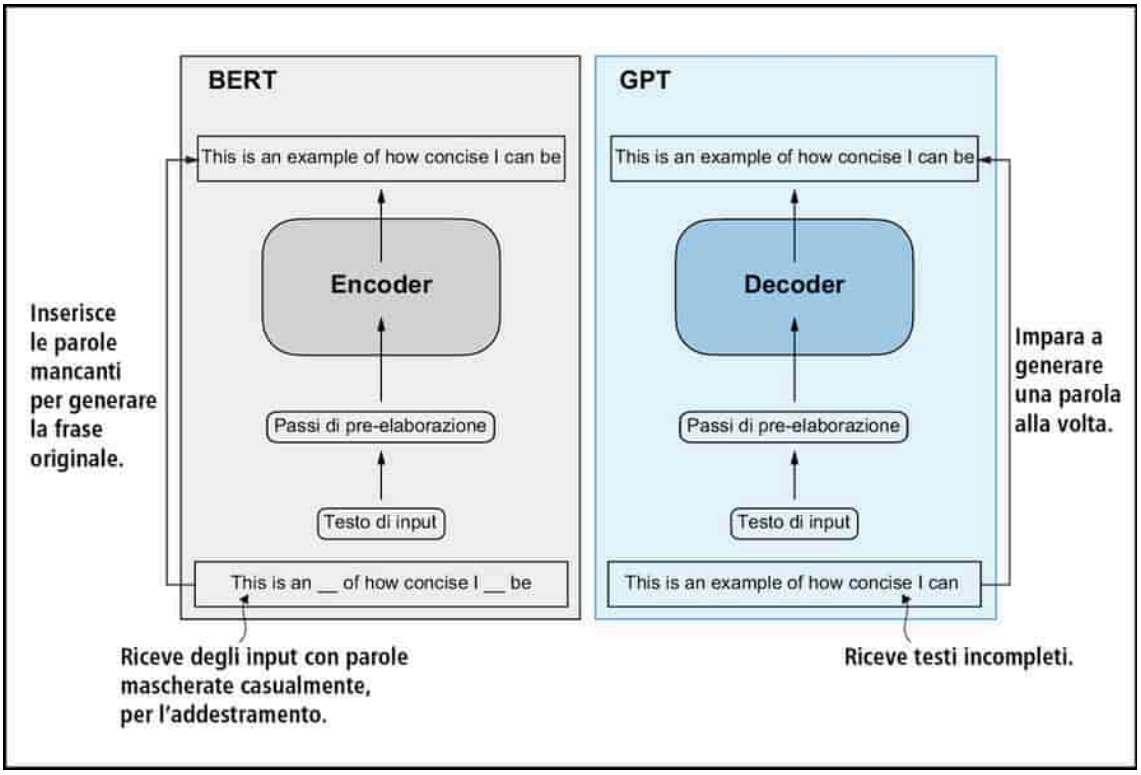


Figura 1.5 Una rappresentazione visuale dei sub-moduli encoder e decoder del transformer. A sinistra, il segmento encoder esemplifica i modelli LLM di tipo BERT, che si concentrano sulla previsione di parole mascherate e sono utilizzati principalmente per attività come la classificazione del testo. A destra, il segmento decoder mostra i modelli LLM di tipo GPT, progettati per attività generative e per produrre sequenze di testo coerenti.

Un GPT, al contrario, si concentra sulla parte decoder dell'architettura a transformer, ed è progettato per attività che richiedono la generazione di testi. Ciò include la traduzione automatica, la sintesi di testo, la scrittura di brani, la scrittura di codice per computer e altro ancora.

I modelli GPT, progettati e addestrati principalmente per eseguire attività di completamento del testo, dimostrano anche una notevole versatilità nelle loro capacità. Questi modelli sono abili nell'eseguire attività di apprendimento sia zero-shot sia few-shot. L'apprendimento

zero-shot si riferisce alla capacità di generalizzare rispetto ad attività completamente invisibili, senza alcun esempio specifico precedente. Al contrario, l'apprendimento few-shot implica l'apprendimento da un numero minimo di esempi che l'utente fornisce come input, come mostra la Figura 1.6.

Transformer vs LLM

I modelli LLM di oggi si basano sull'architettura a transformer. Quindi, transformer e modelli LLM sono termini che spesso vengono usati come sinonimi, in letteratura. Tuttavia, non tutti i transformer sono LLM, poiché i transformer possono essere utilizzati anche per la visione artificiale. Inoltre, non tutti i modelli LLM sono transformer, poiché ci sono modelli LLM basati su architetture ricorrenti e convoluzionali. La motivazione principale alla base di questi approcci alternativi è quella di migliorare l'efficienza computazionale dei modelli LLM. Resta da vedere se queste architetture LLM alternative potranno competere con le capacità dei modelli LLM basati su transformer e se saranno poi adottate nella pratica. Per semplicità, utilizzo il termine "LLM" per riferirmi ai modelli LLM basati su transformer, simili a GPT. I lettori interessati possono trovare riferimenti bibliografici che descrivono queste architetture nell'Appendice B.

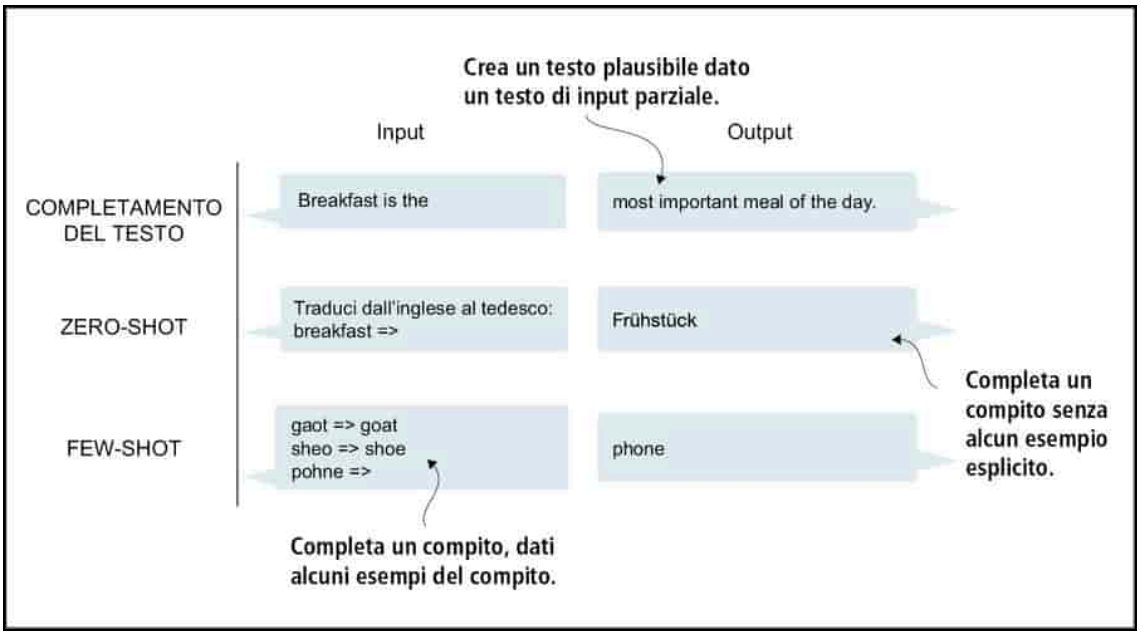


Figura 1.6 Oltre al completamento del testo, i modelli LLM di tipo GPT possono risolvere vari task in base ai loro input senza bisogno di riaddestramento, fine-tuning o modifiche

specifiche dell'architettura del modello. A volte, è utile fornire nell'input alcuni esempi del target (un'operazione few-shot). Tuttavia, i modelli LLM di tipo GPT sono in grado di svolgere i loro compiti anche senza un esempio specifico (un'operazione zero-shot).

Utilizzo di grandi dataset

I grandi dataset di addestramento dei modelli di tipo GPT e BERT rappresentano *corpora* di testo diversi e completi, che comprendono miliardi di parole e includono una vasta gamma di argomenti e linguaggi, naturali e di programmazione. Per fornire un esempio concreto, la Tabella 1.1 riassume il dataset utilizzato per il pre-addestramento di GPT-3, che è servito come pattern di base per la prima versione di ChatGPT.

La Tabella 1.1 riporta il numero di token, dove un token è un'unità di testo che un modello legge, e il numero di token di un dataset è più o meno equivalente al numero di parole e segni di punteggiatura presenti nel testo. Il Capitolo 2 affronta l'argomento della tokenizzazione: il processo di conversione del testo in token.

La conclusione è che la scala e la diversità del dataset di addestramento impiegato consentono a questi modelli di funzionare bene in diversi compiti, rispettando la sintassi, la semantica e il contesto del linguaggio, anche se in alcuni casi sono richieste altre conoscenze generali.

Tabella 1.1 Il dataset di pre-addestramento del modello LLM di GPT-3.

Nome dataset	Descrizione	Numero di token	Proporzione nei dati di addestramento
CommonCrawl (filtrato)	Scansione del Web	410 miliardi	60%

WebText2	Scansione del Web	19 miliardi	22%
Books1	Corpus di libri da Internet	12 miliardi	8%
Books2	Corpus di libri da Internet	55 miliardi	8%
Wikipedia	Testo di alta qualità	3 miliardi	3%

I dettagli del dataset GPT-3

La Tabella 1.1 menziona i dataset utilizzati per GPT-3. La colonna Proporzioni riassume la distribuzione dei dati campionati, con arrotondamenti. Sebbene i valori riportati nella colonna Numero token raggiungano in totale i 499 miliardi, il modello è stato addestrato solo su 300 miliardi di token. Gli autori del documento GPT-3 non hanno specificato perché il modello non sia stato addestrato su tutti i 499 miliardi di token.

Per contestualizzare, considerate le dimensioni del dataset CommonCrawl, che da solo è composto da 410 miliardi di token e richiede circa 570 GB di spazio di archiviazione. In confronto, le iterazioni successive di modelli di tipo GPT-3, come LLaMA di Meta, hanno ampliato il loro ambito di addestramento per includere fonti di dati aggiuntive, come i documenti di ricerca di Arxiv (92 GB) e le Q&A relative al codice di StackExchange (78 GB).

Gli autori del documento relativo a GPT-3 non hanno condiviso pubblicamente il dataset di addestramento, ma un dataset comparabile pubblicamente disponibile è *Dolma: An Open Corpus of Three Trillion Tokens for LLM Pretraining Research*, di Soldaini et al. 2024 (https://arxiv.org/abs/2402.00159). Tuttavia, la raccolta può contenere opere protette da copyright e i termini di utilizzo possono dipendere dal caso d'uso previsto e dal paese.

La natura pre-addestrata di questi modelli li rende incredibilmente versatili per un'ulteriore e successivo fine-tuning dei compiti, motivo per cui sono anche noti come modelli base. Il pre-addestramento dei modelli LLM richiede l'accesso a risorse significative ed è molto costoso. Per esempio, il costo del pre-addestramento GPT-3 è stimato

in 4,6 milioni di dollari in termini di crediti di cloud computing (https://mng.bz/VxEW).

Fortunatamente, sono disponibili molti modelli LLM pre-addestrati open source, che possono essere utilizzati come strumenti generici per scrivere, estrarre e modificare testi che non facevano parte dei dati di addestramento. Inoltre, i modelli LLM possono essere ottimizzati per attività specifiche con dataset relativamente più piccoli, riducendo le risorse computazionali necessarie e migliorando le prestazioni.

Più avanti, implementeremo il codice per il pre-addestramento e lo useremo per pre-addestrare un modello LLM a fini didattici. Tutti i calcoli sono eseguibili su normale hardware di livello consumer. Dopo aver implementato il codice di pre-addestramento, vedremo come riutilizzare i pesi disponibili pubblicamente, e come caricarli nell'architettura che implementeremo, il che ci consentirà di saltare la costosa fase di pre-addestramento quando dovremo perfezionare il modello LLM.

L'architettura GPT

La tecnologia GPT è stata presentata nel documento *Improving Language Understanding by Generative Pre-Training* (https://mng.bz/x2qg) di Radford et al. di OpenAI. GPT-3 è una versione ampliata di questo modello, con più parametri e addestrata su un dataset più ampio. Inoltre, il modello originale offerto da ChatGPT è stato creato perfezionando GPT-3 su un ampio dataset di istruzioni utilizzando un metodo descritto nel documento *InstructGPT* di OpenAI (https://arxiv.org/abs/2203.02155). Come avete avuto modo di vedere nella Figura 1.6, questi sono modelli di completamento del testo e possono svolgere altre attività, come la correzione ortografica,

la classificazione o la traduzione linguistica. Questo è davvero notevole, dato che i modelli GPT sono pre-addestrati su un compito di previsione della parola successiva relativamente semplice, come mostra la Figura 1.7.

Figura 1.7 Nel compito di pre-addestramento dei modelli GPT, il sistema impara a prevedere la parola successiva in una frase esaminando le parole che l'hanno preceduta. Questo approccio aiuta il modello a comprendere come si compongono normalmente le parole e le frasi, formando una base che può essere applicata a vari altri compiti.

Il compito di previsione della parola successiva è una forma di apprendimento ad auto-supervisione, che è una forma di auto-etichettatura. Ciò significa che non abbiamo bisogno di compilare esplicitamente le etichette per i dati di addestramento in modo esplicito, ma possiamo usare la struttura degli stessi dati: possiamo usare la parola successiva in una frase o in un documento come etichetta per ciò che il modello dovrebbe prevedere. Poiché questo compito di previsione della parola successiva ci consente di creare etichette "al volo", per addestrare i modelli LLM potete usare enormi dataset testuali non etichettati.

Rispetto all'architettura originale a transformer che abbiamo trattato in precedenza, l'architettura generale GPT è relativamente semplice. In sostanza, è solo la parte decoder senza l'encoder (Figura 1.8). Poiché i modelli a decoder come i GPT generano testo prevedendo il testo una parola alla volta, sono considerati modelli *autoregressivi*. I modelli autoregressivi incorporano i loro output precedenti come input per le previsioni future. Di conseguenza, in un

GPT, ogni nuova parola viene scelta in base alla sequenza che la precede, il che migliora la coerenza del testo prodotto.

Architetture come GPT-3 sono significativamente più grandi rispetto al modello originale a transformer. Per esempio, il transformer originale ripeteva i blocchi encoder e decoder per sei volte. GPT-3 ha 96 layer transformer e 175 miliardi di parametri in totale.

GPT-3 è stato introdotto nel 2020, il che, secondo gli standard del deep learning e dello sviluppo di modelli LLM, è considerato “molto tempo fa”. Tuttavia, architetture più recenti, come i modelli Llama di Meta, si basano ancora sugli stessi concetti di fondo, introducendo solo piccole modifiche.

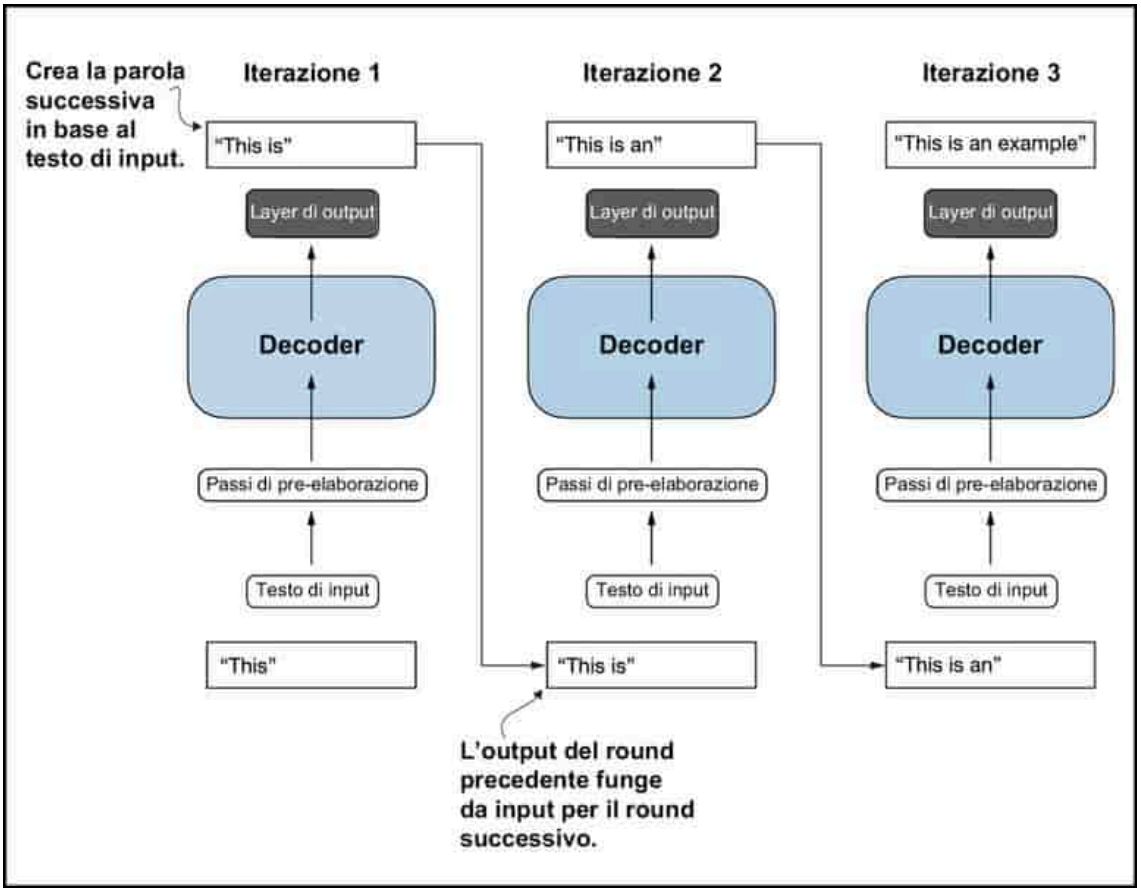


Figura 1.8 L'architettura GPT impiega solo la parte decoder del transformer originale. È progettata per l'elaborazione unidirezionale, da sinistra a destra, il che la rende adatta ad attività di generazione del testo e previsione della parola successiva, per generare testo in modo iterativo, una parola alla volta.

Di conseguenza rimane fondamentale comprendere il funzionamento di un GPT. Per questo mi concentro sull'implementazione di un'architettura GPT, fornendovi al contempo indicazioni sulle varianti impiegate da altri modelli LLM.

Sebbene il modello a transformer originale, costituito da blocchi encoder/decoder, fosse stato progettato esplicitamente per la traduzione linguistica, anche i modelli GPT, nonostante la loro architettura più ampia ma più semplice basata solo sul decoder e mirata alla previsione della parola successiva, sono in grado di

eseguire attività di traduzione. Inizialmente, questa capacità è stata un po' una sorpresa per i ricercatori, poiché è emersa da un modello addestrato principalmente su un compito di previsione della parola successiva, un'attività che non aveva come obiettivo specifico la traduzione.

La capacità di eseguire attività per cui il modello non era stato esplicitamente addestrato è un *comportamento emergente*. Questo perché tale capacità non viene insegnata esplicitamente durante l'addestramento, ma *emerge* come conseguenza naturale dell'esposizione del modello a grandi quantità di dati multilingue in vari contesti. Il fatto che i modelli GPT possano "apprendere" i modelli di traduzione tra le lingue ed eseguire attività di traduzione anche se non sono stati specificamente addestrati per farlo dimostra i vantaggi e le capacità di questi modelli linguistici generativi. Possiamo usarli per compiti differenti senza utilizzare modelli differenti.

Costruzione di un modello LLM

Ora che abbiamo gettato le basi per comprendere i modelli LLM, scriviamone uno partendo da zero. Prenderemo l'idea su cui si basa GPT e affronteremo l'impresa questo in tre fasi, come delineato nella Figura 1.9.

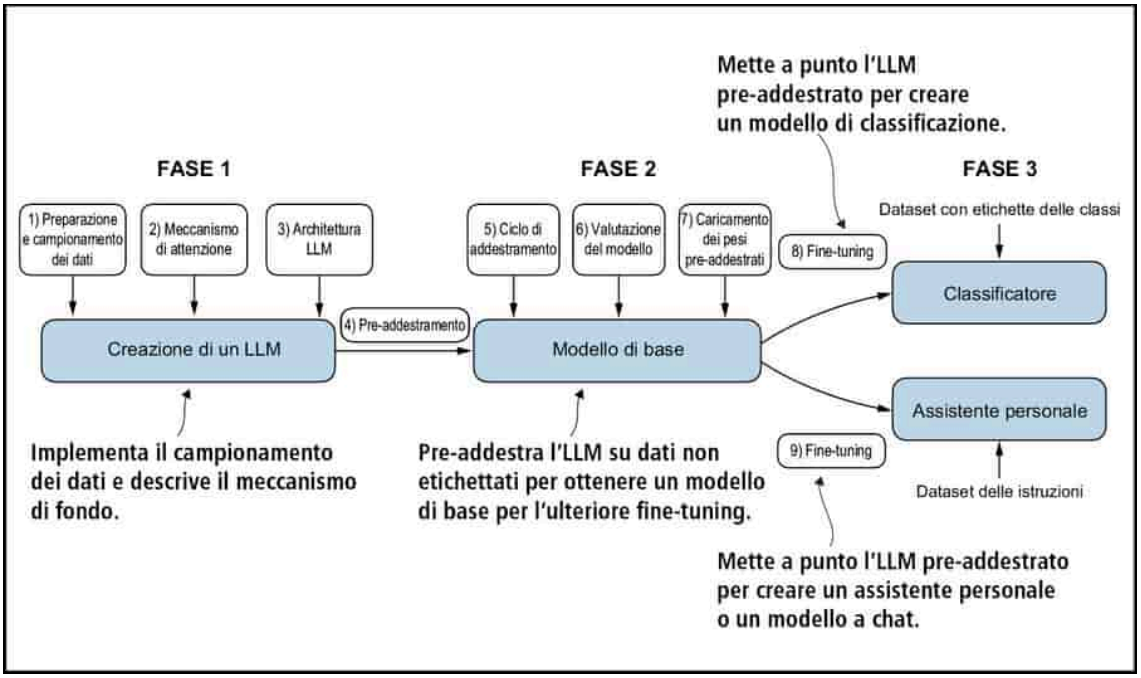


Figura 1.9 Le tre fasi della programmazione di un modello LLM sono l'implementazione dell'architettura LLM e del processo di preparazione dei dati (Fase 1), il pre-addestramento di un modello LLM per creare un pattern di base (Fase 2) e il fine-tuning del pattern di base per trasformarlo in un assistente personale o classificatore di testi (Fase 3).

Nella Fase 1, apprenderemo i passi fondamentali di pre-elaborazione dei dati e programmeremo il meccanismo di attenzione che è al cuore di ogni modello LLM. Nella Fase 2, impareremo a programmare e pre-addestrare un modello LLM di tipo GPT, in grado di generare nuovi testi. Esamineremo anche i fondamenti della valutazione dei modelli LLM, che è essenziale per sviluppare sistemi efficaci di elaborazione del linguaggio naturale.

Pre-addestrare un modello LLM da zero è un'impresa significativa, che richiede migliaia o milioni di dollari in termini di costi di elaborazione. Pertanto, la Fase 2 è rivolta principalmente all'implementazione dell'addestramento per scopi educativi,

utilizzando solo un piccolo dataset. Inoltre, fornisco anche esempi di codice per caricare pesi liberamente disponibili.

Infine, nella Fase 3, prenderemo un modello LLM pre-addestrato e lo perfezioneremo per rispondere a istruzioni come rispondere a query o classificare testi, le attività più comuni in molte applicazioni e ricerche.

Riepilogo

- I modelli LLM hanno trasformato l'elaborazione del linguaggio naturale, che in precedenza si basava principalmente su sistemi basati su regole esplicite e metodi statistici più semplici. L'avvento dei modelli LLM ha introdotto nuovi approcci basati sul deep learning, che hanno portato a progressi nella comprensione, generazione e traduzione del linguaggio umano.
- I modelli LLM di oggi prevedono principalmente due fasi di addestramento:
  - innanzitutto, vengono pre-addestrati su un ampio *corpus* di testo non etichettato, utilizzando come etichetta la previsione della parola successiva in una frase;
  - poi vengono perfezionati su un dataset target più piccolo ed etichettato, per rispondere a istruzioni o eseguire attività di classificazione.
- I modelli LLM si basano sull'architettura a transformer. L'idea chiave di tale architettura è un meccanismo di attenzione che fornisce al modello LLM un accesso selettivo all'intera sequenza di input mentre genera l'output una parola alla volta.
- L'architettura originale a transformer è composta da un encoder per l'analisi del testo e da un decoder per la generazione del testo.

- I modelli LLM per la generazione del testo e l’esecuzione di istruzioni, come GPT-3 e ChatGPT, implementano solo moduli decoder, semplificando l’architettura.
- Per il pre-addestramento dei modelli LLM sono essenziali grandi dataset composti da miliardi di parole.
- Mentre il compito generale di pre-addestramento dei modelli di tipo GPT è quello di prevedere la parola successiva in una frase, questi modelli LLM esibiscono proprietà emergenti, come la capacità di classificare, tradurre o riassumere i testi.
- Una volta pre-addestrato un modello LLM, il pattern di base risultante può essere perfezionato in modo più efficiente per diverse altre attività.
- Su attività specifiche, i modelli LLM ottimizzati su dataset personalizzati possono avere prestazioni migliori dei modelli LLM generici.

Capitolo 2
Operare su dati testuali

Finora abbiamo trattato solo la struttura generale dei modelli LLM (*Large Language Model*) e abbiamo scoperto che sono pre-addestrati su grandi quantità di testo. In particolare, ci siamo concentrati sui modelli LLM a decoder basati sull'architettura a transformer, che sta alla base dei modelli utilizzati da ChatGPT e altri noti modelli LLM simili.

Durante la fase di pre-addestramento, i modelli LLM elaborano il testo una parola alla volta. L'addestramento dei modelli LLM con milioni o miliardi di parametri utilizzando un compito di previsione della parola successiva produce modelli dotati di capacità impressionanti. Questi modelli possono poi essere ulteriormente perfezionati per rispondere a istruzioni generali o eseguire compiti specifici. Ma prima di poter implementare e addestrare i modelli LLM, dobbiamo preparare il dataset di addestramento, come illustra la Figura 2.1.

Imparerete a preparare il testo di input per l'addestramento dei modelli LLM. Ciò comporta la suddivisione del testo in token di parole e sub-parole, che possono poi essere codificati in rappresentazioni vettoriali per il modello LLM. Scoprirete anche gli schemi avanzati di tokenizzazione come la codifica a coppie di byte, utilizzata nei modelli LLM più noti, come i GPT. Infine, implementeremo una strategia di campionamento e caricamento dei

dati per produrre le coppie input-output necessarie per l'addestramento dei modelli LLM.

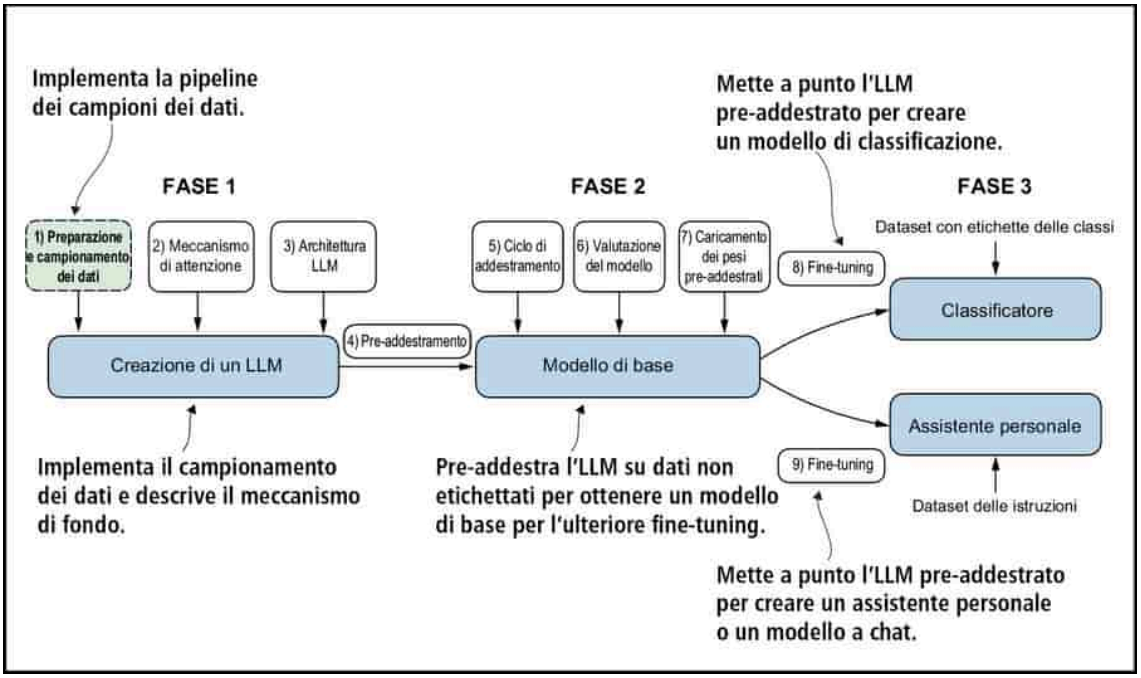


Figura 2.1 Le tre fasi principali della programmazione di un modello LLM. Questo capitolo si concentra sul Passo 1 della Fase 1: l'implementazione della pipeline dei campioni dei dati.

Embedding di parole

I modelli a reti neurali profonde, inclusi i modelli LLM, non possono elaborare direttamente il testo grezzo. Poiché il testo è categoriale, non è compatibile con le operazioni matematiche utilizzate per implementare e addestrare le reti neurali. Pertanto, abbiamo bisogno di un modo per rappresentare le parole come vettori a valori continui.

NOTA

Coloro che non hanno familiarità con i vettori e i tensori in un contesto computazionale possono approfondire l'argomento nell'Appendice A.

Il concetto che consiste nel convertire i dati in un formato vettoriale è chiamato *embedding*. Utilizzando uno specifico layer della rete neurale o un altro modello a rete neurale pre-addestrato, possiamo incorporare diversi tipi di dati, per esempio video, audio e testo, come mostra la Figura 2.2. Tuttavia, è importante notare che formati differenti dei dati richiedono modelli di embedding differenti. Per esempio, un modello di embedding progettato per il testo non sarebbe adatto all'embedding di dati audio o video.

In sostanza, un embedding crea una mappatura di oggetti discreti (parole, immagini o perfino interi documenti) in punti in uno spazio vettoriale continuo; lo scopo principale degli embedding consiste nel convertire dati non numerici in un formato che le reti neurali possano elaborare.

Sebbene gli embedding per parole siano la forma più comune di embedding di testi, esistono anche embedding per frasi, paragrafi o anche interi documenti.

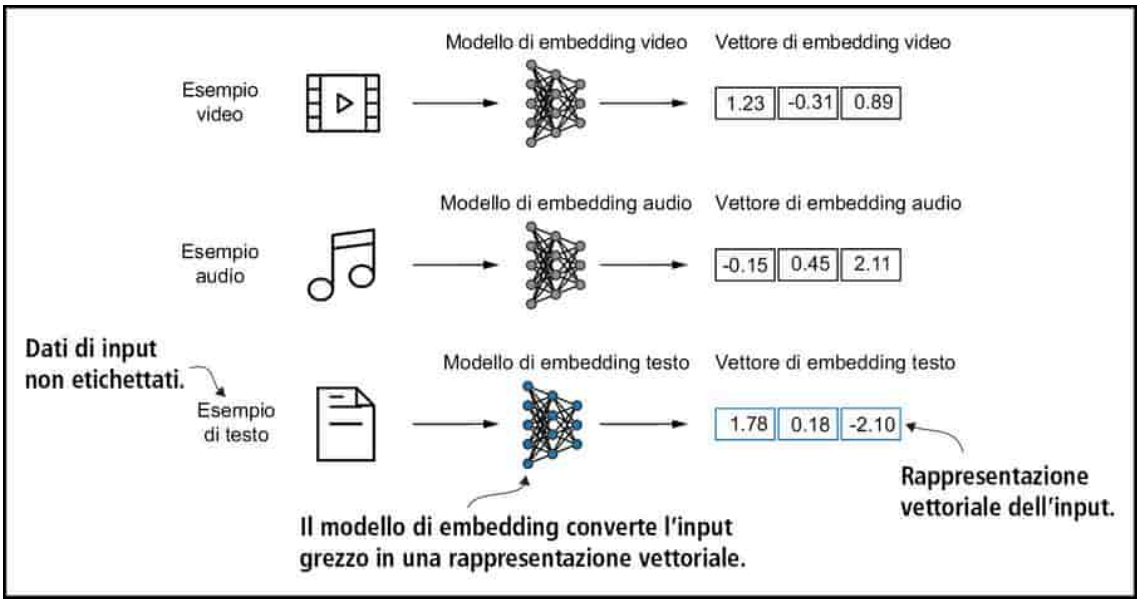


Figura 2.2 I modelli di deep learning non possono elaborare direttamente dati video, audio e testuali. Pertanto, utilizziamo un modello di embedding per trasformare questi dati grezzi

in una rappresentazione vettoriale densa, che le architetture di deep learning possano facilmente comprendere ed elaborare. In particolare, la figura illustra il processo di conversione dei dati grezzi in un vettore numerico tridimensionale.

Gli embedding per frasi o per paragrafi sono scelte comunemente usate per la *generazione aumentata dal recupero* (*retrieval-augmented generation*), che combina la generazione (la produzione di testo) col recupero (la ricerca in una base di conoscenza esterna) per estrarre informazioni rilevanti durante la generazione del testo, che è una tecnica che va oltre gli scopi di questo libro. Poiché il nostro obiettivo è quello di addestrare modelli LLM simili a GPT, che imparano a generare testo una parola alla volta, ci concentreremo sugli embedding per parole (*word embedding*).

Sono stati sviluppati diversi algoritmi e framework per generare l'embedding per parole. Uno degli esempi precedenti e più popolari è l'approccio *Word2Vec*, che ha addestrato l'architettura a rete neurale per generare l'embedding per parole con lo scopo di prevedere il contesto di una parola data la parola target o viceversa. L'idea alla base di Word2Vec è che le parole che appaiono in contesti simili tendono ad avere significati simili. Di conseguenza, quando vengono proiettati in un embedding per parole bidimensionale, per scopi di visualizzazione, i termini simili vengono raggruppati insieme, come illustra la Figura 2.3.

Gli embedding per parole possono avere dimensioni variabili, da una a migliaia. Una dimensionalità più elevata può catturare relazioni più sfumate, ma a scapito dell'efficienza computazionale.

Mentre per generare gli embedding per i modelli di machine learning possiamo usare modelli pre-addestrati, come Word2Vec, i modelli LLM producono comunemente i propri embedding, che fanno parte del layer di input e vengono aggiornati durante l'addestramento. Il vantaggio di ottimizzare gli embedding nell'ambito

dell’addestramento del modello LLM invece di usare Word2Vec è che gli embedding risultano ottimizzati per l’attività specifica e i dati disponibili. Implementeremo tali layer di embedding più avanti in questo capitolo. I modelli LLM possono creare anche embedding di output contestualizzati, come discutiamo nel Capitolo 3.

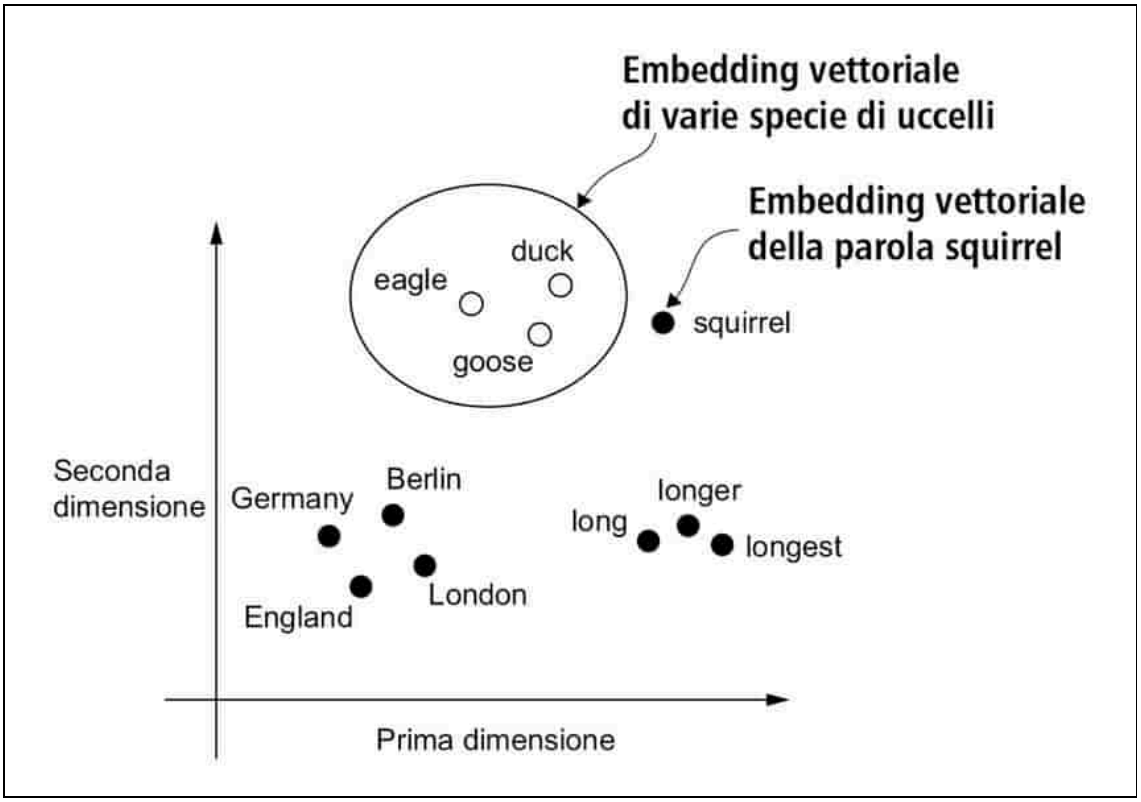


Figura 2.3 Se gli embedding per parole sono bidimensionali, possiamo visualizzarli in un grafico a dispersione bidimensionale, come mostrato qui. Quando si utilizzano tecniche di embedding per parole, come Word2Vec, spesso le parole corrispondenti a concetti simili appaiono vicine tra loro nello spazio di embedding. Per esempio, le diverse specie di uccelli appaiono più vicini tra loro nello spazio di embedding rispetto ai paesi e alle città.

Sfortunatamente, gli embedding a elevata dimensionalità rappresentano una sfida per la visualizzazione, perché la nostra percezione sensoriale e le tipiche rappresentazioni grafiche sono intrinsecamente limitate a tre dimensioni o meno, motivo per cui la

Figura 2.3 mostra embedding bidimensionali in un grafico a dispersione che è bidimensionale. Tuttavia, quando lavoriamo con i modelli LLM, in genere utilizziamo embedding aventi una dimensionalità molto più elevata. Sia GPT-2 sia GPT-3, impiegano una dimensionalità dell'embedding (spesso definita dimensionalità degli stati nascosti del modello) che è differente in base alla variante e alle dimensioni specifiche del modello, in base a un compromesso tra prestazioni ed efficienza. I modelli GPT-2 più piccoli (117 e 125 milioni di parametri) utilizzano un embedding 768-dimensionale per fornire esempi concreti. Il modello GPT-3 più grande (175 miliardi di parametri) utilizza un embedding a 12.288 dimensioni.

Ora, esamineremo i passi necessari per preparare gli embedding utilizzati da un modello LLM, che includono la suddivisione del testo in parole, la conversione delle parole in token e la trasformazione dei token in vettori di embedding.

Tokenizzazione del testo

Vediamo ora come si suddivide il testo di input in singoli token, un passo di pre-elaborazione obbligatorio per la creazione degli embedding per un modello LLM. Questi token sono singole parole o caratteri speciali, inclusi i segni di punteggiatura, come mostra la Figura 2.4.

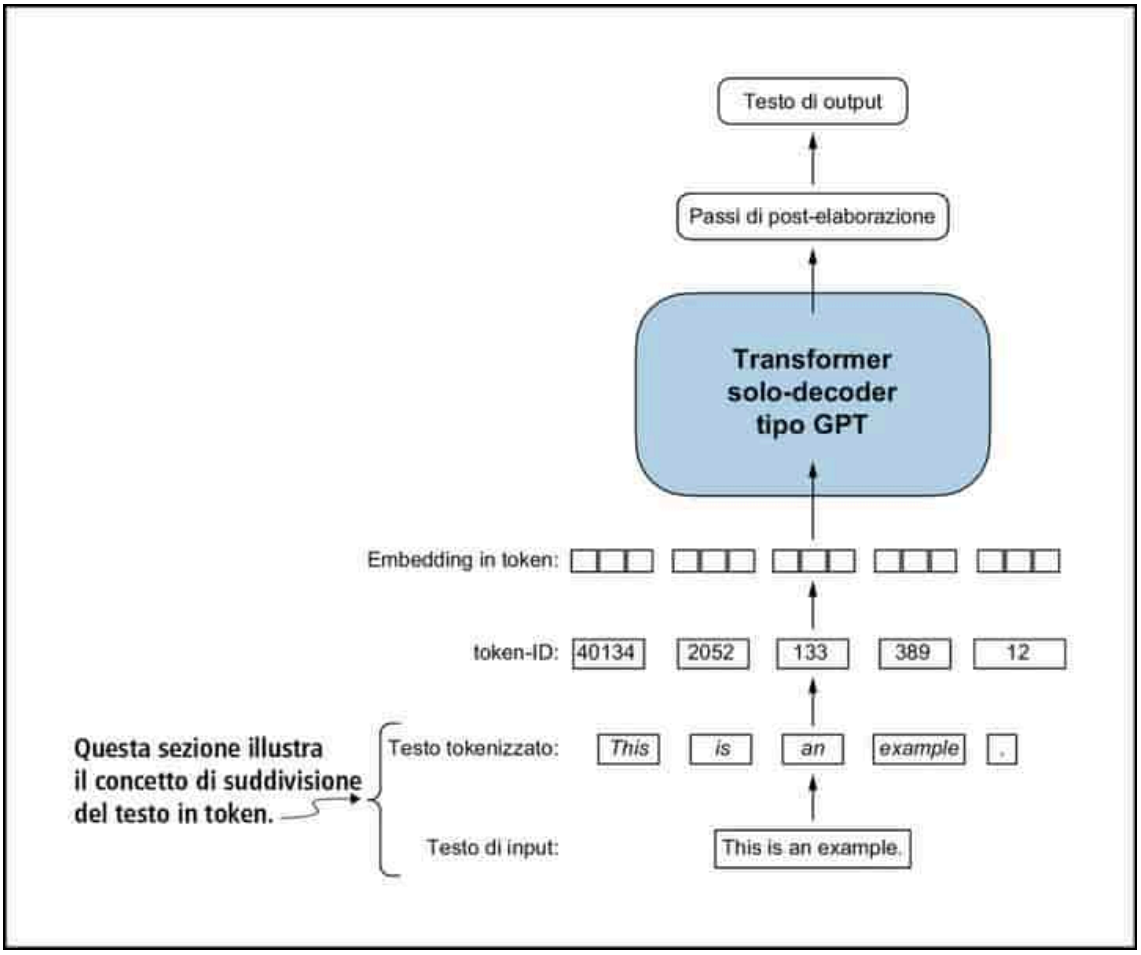


Figura 2.4 I passi di elaborazione del testo nel contesto di un modello LLM. Qui, dividiamo un testo di input in singoli token, che sono parole o caratteri speciali, come i segni di punteggiatura.

Il testo che tokenizzeremo per l'addestramento del modello LLM è *The Verdict*, un racconto breve di Edith Wharton, che è stato rilasciato come pubblico dominio e pertanto è utilizzabile per questo genere di attività. Il testo è disponibile su Wikisource in `https://en.wikisource.org/wiki/The_Verdict`, e potete copiarlo e incollarlo in un file di testo: io l'ho copiato nel file di testo `the-verdict.txt`.

In alternativa, trovate questo file, `the-verdict.txt`, nel repository GitHub di questo libro su `https://mng.bz/Adng`. Potete scaricare il file col

seguente codice Python:

```
import urllib.request
url = ("https://raw.githubusercontent.com/rasbt/"
       "LLMs-from-scratch/main/ch02/01_main-chapter-code/"
       "the-verdict.txt")
file_path = "the-verdict.txt"
urllib.request.urlretrieve(url, file_path)
```

Ora possiamo caricare il file `the-verdict.txt` utilizzando le utility standard di Python per la lettura di file.

Listato 2.1 Lettura di un racconto breve come esempio di testo in Python.

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
print("Total number of character:", len(raw_text))
print(raw_text[:99])
```

Il comando `print` stampa il numero totale di caratteri, seguito dai primi 100 caratteri di questo file, a scopo illustrativo:

```
Total number of character: 20479
I HAD always thought Jack Gisburn rather a cheap genius--though a good fellow
enough--so it was no
```

Il nostro obiettivo è quello di tokenizzare questo racconto breve di 20.479 caratteri in singole parole e caratteri speciali, che potremo poi trasformare in embedding per l'addestramento del modello LLM.

NOTA

È pratica comune, quando si lavora con i modelli LLM, elaborare milioni di articoli e centinaia di migliaia di libri, molti gigabyte di testo. Tuttavia, per scopi didattici, è sufficiente lavorare con campioni di testo molto più piccoli, come un singolo libro, per illustrare le idee principali alla base delle fasi di elaborazione del testo e per rendere possibile l'esecuzione degli esercizi in un tempo ragionevole su hardware consumer.

Come possiamo suddividere al meglio questo testo per ottenere un elenco di token? Per questo passo, a scopo illustrativo facciamo una piccola escursione e utilizziamo la libreria per espressioni regolari di Python, `re`. Non preoccupatevi: non vi chiedo di imparare o di memorizzare la sintassi delle espressioni regolari, poiché in seguito passeremo a un tokenizer pronto all'uso.

Utilizzando un semplice testo campione, possiamo usare il comando `re.split` con la seguente sintassi per dividere un testo in base ai segni di spaziatura:

```
import re
text = "Hello, world. This, is a test."
result = re.split(r'(\s)', text)
print(result)
```

Il risultato è un elenco di singole parole, spazi e segni di punteggiatura:

```
['Hello,', ' ', 'world.', ' ', 'This,', ' ', 'is', ' ', 'a', ' ', 'test.']
```

Questo semplice schema di tokenizzazione funziona principalmente per separare il testo campione in singole parole; tuttavia, alcune parole sono ancora collegate a segni di punteggiatura, che vorremmo considerare come elementi separati. Inoltre, evitiamo di rendere tutto il testo minuscolo, perché la presenza di maiuscole e minuscole aiuta i modelli LLM a distinguere tra nomi propri e nomi comuni, a comprendere la struttura della frase e a imparare a generare testo con la corretta configurazione maiuscole/minuscole.

Modifichiamo la suddivisioni basata su espressioni regolari considerando spazi vuoti (`\s`), virgole e punti (`[,.]`):

```
result = re.split(r'([,.]|\s)', text)
print(result)
```

Possiamo vedere che ora le parole e i segni di punteggiatura sono elementi distinti, proprio come volevamo:

```
['Hello', ',', '', ' ', 'world', '.', '', ' ', 'This', ',', '', ' ', 'is', ' ',
'a', ' ', 'test', '.', '']
```

Un piccolo problema rimanente è che l’elenco include ancora i segni di spaziatura. Facoltativamente, possiamo rimuovere questi spazi ridondanti in modo sicuro, come segue:

```
result = [item for item in result if item.strip()]
print(result)
```

L’output risultante, senza spazi, ha il seguente aspetto:

```
['Hello', ',', 'world', '.', 'This', ',', 'is', 'a', 'test', '.']
```

NOTA

Quando si sviluppa un semplice tokenizer, la scelta se codificare gli spazi come caratteri a sé o semplicemente rimuoverli dipende dalla nostra applicazione e dai suoi requisiti. La rimozione degli spazi riduce i requisiti di memoria e di elaborazione. Tuttavia, mantenere gli spazi può essere utile se alleniamo modelli sensibili all'esatta struttura del testo (per esempio, il codice Python è sensibile all'indentazione e alla spaziatura). Qui, per semplicità e brevità degli output tokenizzati, rimuoviamo gli spazi. In seguito, passeremo a uno schema di tokenizzazione che considera anche gli spazi.

Lo schema di tokenizzazione che abbiamo ideato qui funziona bene sul semplice testo campione. Modifichiamolo ulteriormente in modo che possa gestire anche altri tipi di punteggiatura, come punti interrogativi, virgolette e i doppi trattini che abbiamo visto in precedenza nei primi 100 caratteri del racconto di Edith Wharton, insieme ad altri caratteri speciali:

```
text = "Hello, world. Is this-- a test?"
result = re.split(r'([,.:;?_!"()\']|--|\s)', text)
result = [item.strip() for item in result if item.strip()]
print(result)
```

Il risultato in output è:

```
['Hello', ',', 'world', '.', 'Is', 'this', '--', 'a', 'test', '?']
```

Come potete vedere dai risultati riassunti nella Figura 2.5, ora il nostro schema di tokenizzazione è in grado di gestire correttamente i vari caratteri speciali presenti nel testo.

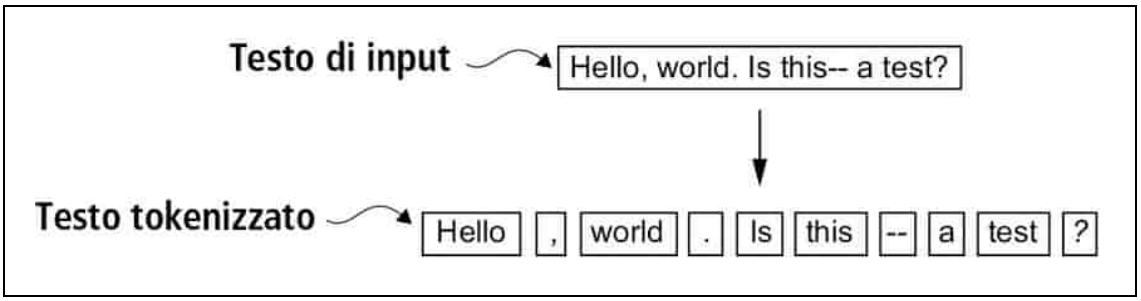


Figura 2.5 Lo schema di tokenizzazione che abbiamo implementato finora divide il testo in singole parole e segni di punteggiatura. In questo specifico esempio, il testo campione viene diviso in dieci singoli token.

Ora che abbiamo un semplice tokenizer funzionante, applichiamolo all'intero racconto di Edith Wharton:

```
preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', raw_text)
preprocessed = [item.strip() for item in preprocessed if item.strip()]
print(len(preprocessed))
```

Questa istruzione print restituisce `4690`, il numero di token presenti in questo testo (senza spazi). Mostriamo i primi trenta token per un rapido controllo visuale:

```
print(preprocessed[:30])
```

L'output risultante mostra che il tokenizer sembra gestire bene il testo, poiché separa con precisione tutte le parole e i caratteri speciali:

```
['I', 'HAD', 'always', 'thought', 'Jack', 'Gisburn', 'rather', 'a', 'cheap',
'genius', '--', 'though', 'a', 'good', 'fellow', 'enough', '--', 'so', 'it',
'was', 'no', 'great', 'surprise', 'to', 'me', 'to', 'hear', 'that', ',', 'in']
```

Conversione dei token in token-ID

Ora, convertiremo questi token da una stringa Python a una rappresentazione intera, per produrre i token-ID. Questa conversione è un passo intermedio prima di convertire i token-ID in vettori di embedding.

Per mappare i token generati in precedenza in token-ID, dobbiamo prima creare un vocabolario. Questo vocabolario definisce come mappiamo ogni parola e carattere speciale univoco su un intero univoco, come mostra la Figura 2.6.

Ora che abbiamo tokenizzato il racconto di Edith Wharton e lo abbiamo assegnato a una variabile Python chiamata `preprocessed`, creiamo un elenco di tutti i token univoci e ordiniamoli alfabeticamente per determinare le dimensioni del vocabolario:

```
all_words = sorted(set(preprocessed))
vocab_size = len(all_words)
print(vocab_size)
```

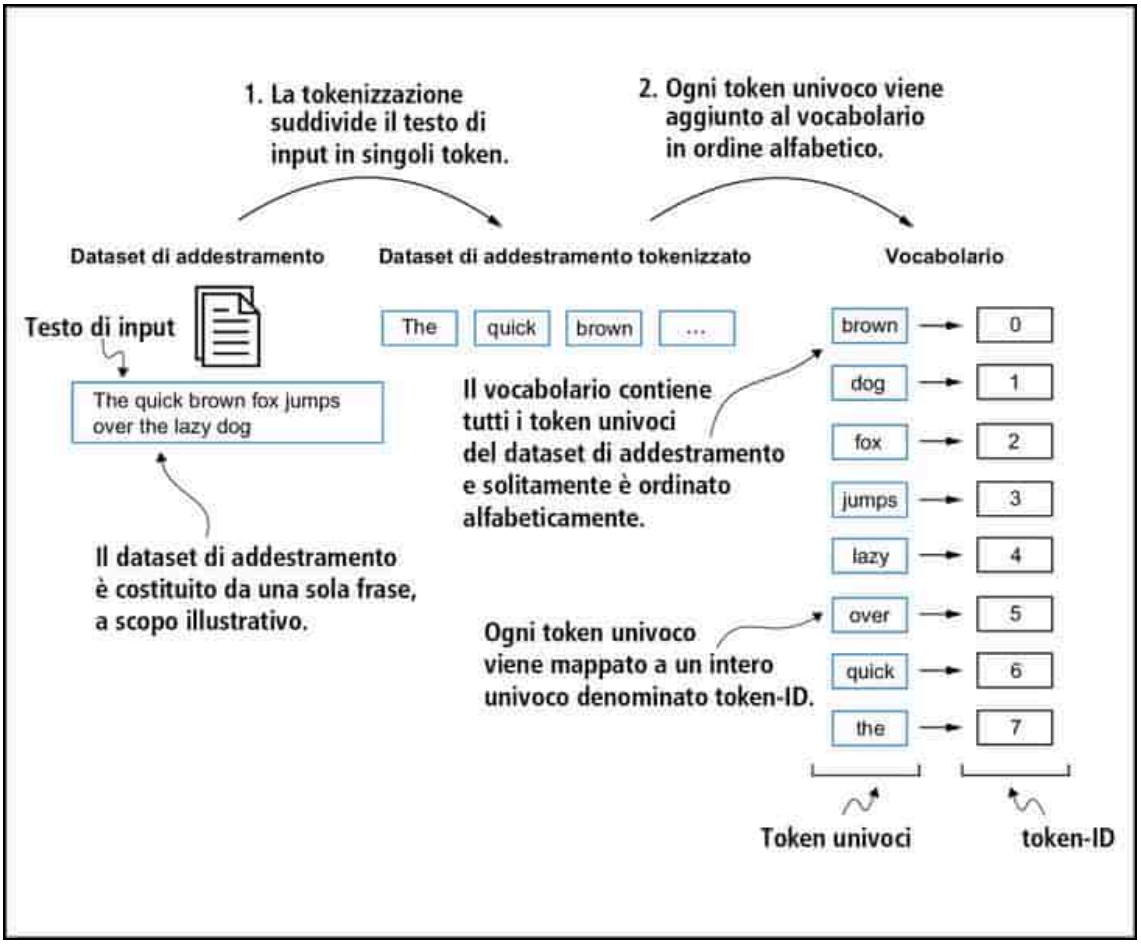


Figura 2.6 Costruiamo un vocabolario tokenizzando l'intero testo del dataset di addestramento in singoli token. Questi singoli token vengono poi ordinati alfabeticamente, e i token duplicati vengono rimossi. I token univoci vengono infine aggregati in un vocabolario che definisce una mappa da ciascun token univoco a un valore intero univoco. Il vocabolario raffigurato è volutamente piccolo e, per semplicità, non contiene segni di punteggiatura o caratteri speciali.

Dopo aver determinato tramite questo codice che le dimensioni del vocabolario sono di 1.130 token, creiamo il vocabolario e visualizziamo i primi 51 elementi, a scopo illustrativo.

Listato 2.2 Creazione di un vocabolario.

```
vocab = {token:integer for integer,token in enumerate(all_words)}
for i, item in enumerate(vocab.items()):
```

```
    print(item)
    if i >= 50:
        break
```

L’output è il seguente:

```
('!', 0)
('"', 1)
("'", 2)
...
('Her', 49)
('Hermia', 50)
```

Come potete vedere, il dizionario contiene singoli token associati a etichette intere univoche. Il nostro prossimo obiettivo è quello di applicare questo vocabolario per convertire il nuovo testo in token-ID (Figura 2.7).

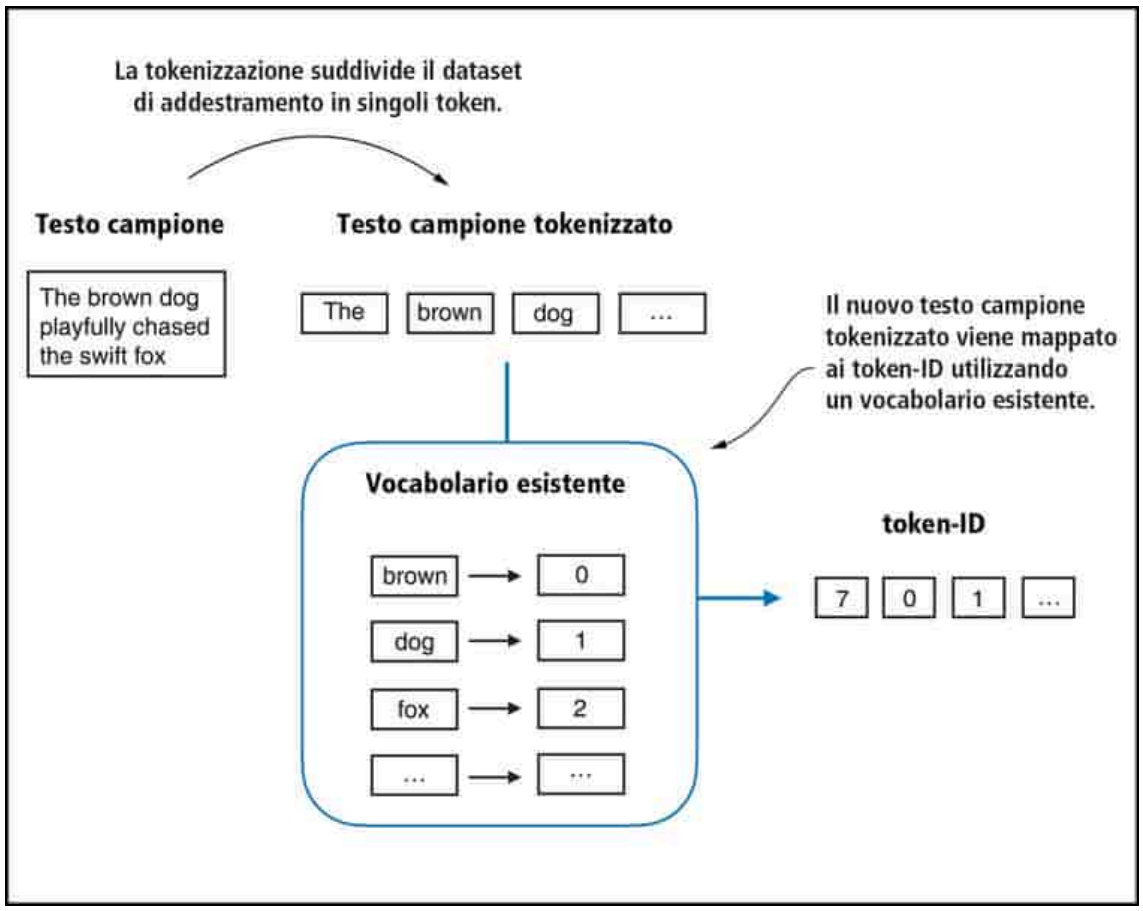


Figura 2.7 Partendo da un nuovo campione di testo, tokenizziamo il testo e utilizziamo il vocabolario per convertire i token di testo in token-ID. Il vocabolario è costruito dall’intero dataset di addestramento e può essere applicato al dataset di addestramento stesso e a

qualsiasi nuovo campione di testo. Per semplicità, il vocabolario raffigurato non contiene segni di punteggiatura o caratteri speciali.

Per convertire gli output di un modello LLM da numeri a testo, abbiamo bisogno di un modo per trasformare i token-ID di nuovo in parole. Per questo, possiamo creare una versione inversa del vocabolario, che mappa i token-ID ai token di testo corrispondenti.

Implementiamo in Python una classe tokenizer con un metodo `encode` che divide il testo in token ed esegue la mappatura stringa-intero per produrre i token-ID tramite il vocabolario. Inoltre, implementeremo un metodo `decode` che esegue la mappatura inversa, intero-stringa, per convertire i token-ID in testo. Il seguente listato mostra il codice per questa implementazione del tokenizer.

Listato 2.3 Implementazione di un semplice tokenizer di testo.

```
class SimpleTokenizerV1:
    def __init__(self, vocab):
        self.str_to_int = vocab (1)
        self.int_to_str = {i:s for s,i in vocab.items()} (2)

    def encode(self, text): (3)
        preprocessed = re.split(r'([,.?_!"()\']|--|\s)', text)
        preprocessed = [
            item.strip() for item in preprocessed if item.strip()
        ]
        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    def decode(self, ids): (4)
        text = " ".join([self.int_to_str[i] for i in ids])
        text = re.sub(r'\s+([,.?!"()\'])', r'\1', text) (5)
        return text
```

(1) Memorizza il vocabolario come attributo della classe per accedervi dai metodi `ecode` e `decode`.

(2) Crea un vocabolario inverso che mappa i token-ID ai token di testo originali.

(3) Elabora il testo di input in token-ID.

(4) Converte i token-ID di nuovo in testo.

(5) Rimuove gli spazi prima del segno di punteggiatura specificato.

Utilizzando la classe Python `SimpleTokenizerV1`, ora possiamo creare nuovi oggetti tokenizer tramite un vocabolario esistente, che possiamo poi utilizzare per codificare e decodificare il testo, come mostra la Figura 2.8.

Istanziamo un nuovo oggetto `tokenizer` dalla classe `SimpleTokenizerV1` e tokenizziamo un passo del racconto di Edith Wharton per metterlo alla prova:

```
tokenizer = SimpleTokenizerV1(vocab)
text = """It's the last he painted, you know,"Mrs. Gisburn said with pardonable
pride."""
ids = tokenizer.encode(text)
print(ids)
```

Il codice precedente visualizza i seguenti token-ID:

```
[1, 56, 2, 850, 988, 602, 533, 746, 5, 1126, 596, 5, 1, 67, 7, 38, 851, 1108, 754,
793, 7]
```

Ora vediamo se possiamo trasformare nuovamente questi token-ID in testo utilizzando il metodo `decode`:

```
print(tokenizer.decode(ids))
```

Questo produce:

```
'" It\' s the last he painted, you know," Mrs. Gisburn said with pardonable
pride.'
```

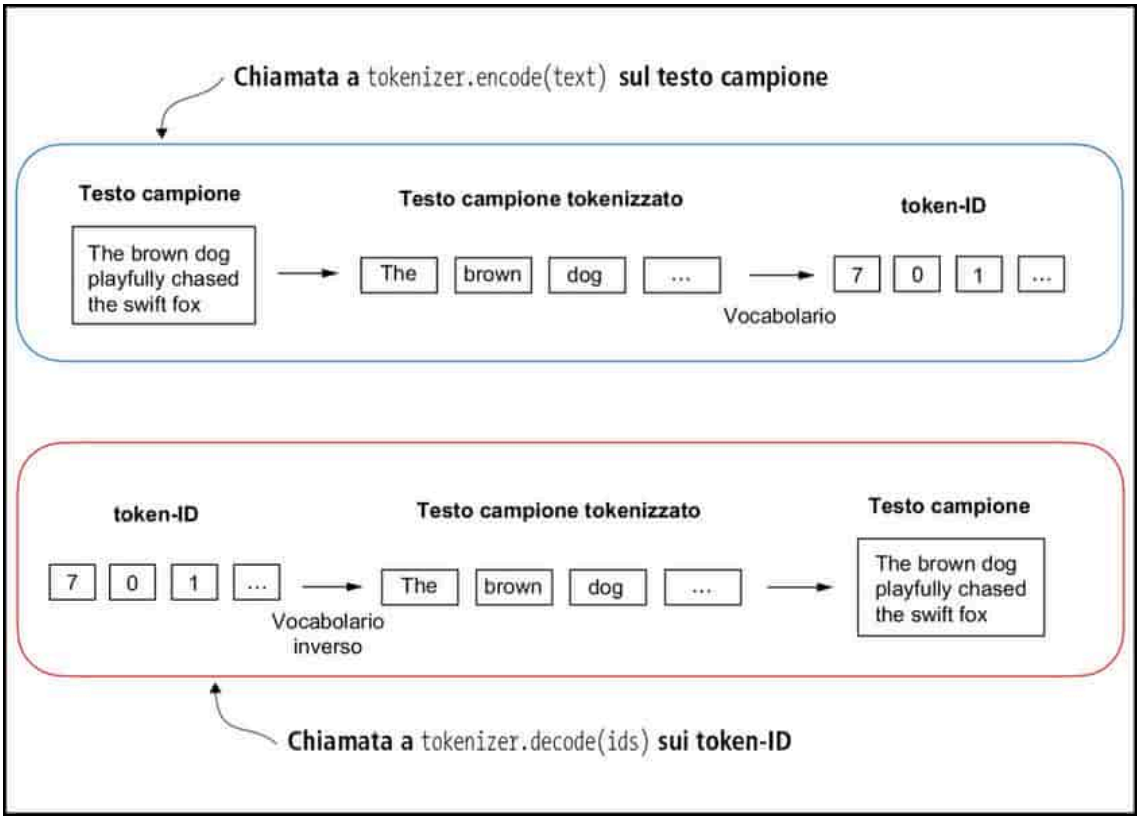


Figura 2.8 Le implementazioni del tokenizer condividono due metodi comuni: un metodo encoder e un metodo decoder. Il primo prende il testo campione, lo divide in singoli token e converte i token in token-ID tramite il vocabolario. Il metodo decoder prende i token-ID, li converte di nuovo in token di testo e infine concatena i token di testo in testo naturale.

Sulla base di questo output, possiamo vedere che il metodo `decode` ha riconvertito correttamente i token-ID nel testo originale.

Fin qui, tutto bene. Abbiamo implementato un tokenizer in grado di tokenizzare e de-tokenizzare il testo in base a un frammento del dataset di addestramento. Applichiamolo ora a un nuovo campione di testo non contenuto nel dataset di addestramento:

```
text = "Hello, do you like tea?"
print(tokenizer.encode(text))
```

L'esecuzione di questo codice provocherà il seguente errore:

```
KeyError: 'Hello'
```

Il problema è che la parola `"Hello"` non è stata usata nel racconto *The Verdict*, quindi, non è contenuta nel vocabolario. Ciò evidenzia la necessità di impiegare, per i modelli LLM, dataset di addestramento ampi e diversificati, per estendere il vocabolario.

Ora, proveremo ulteriormente il tokenizer su testi contenenti parole sconosciute, e discuteremo di ulteriori token speciali che possono essere utilizzati per fornire ulteriore contesto a un modello LLM in addestramento.

Aggiunta di token contestuali speciali

Dobbiamo modificare il tokenizer per gestire anche le parole sconosciute. Dobbiamo anche occuparci dell'uso e dell'aggiunta di token contestuali speciali, che possono migliorare la capacità di un modello di comprendere il contesto e altre informazioni rilevanti presenti nel testo. Questi token speciali possono includere marcatori per parole sconosciute e i delimitatori del documento, per esempio. In particolare, modificheremo il vocabolario e il tokenizer, `SimpleTokenizerV2`, per supportare due nuovi token, `<|unk|>` ed `<|endoftext|>`, come mostra la Figura 2.9.

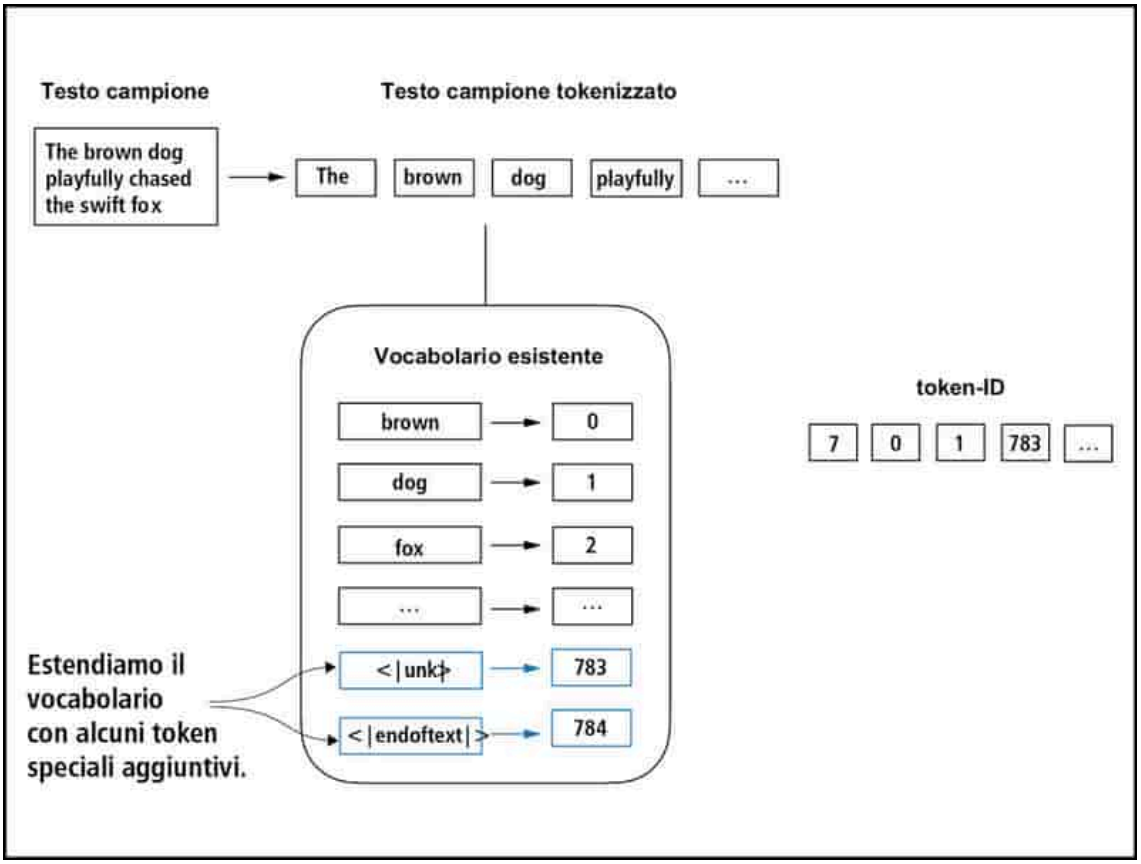


Figura 2.9 Aggiungiamo al vocabolario alcuni token speciali per gestire determinati contesti. Per esempio, aggiungiamo un token <|unk|> per rappresentare parole nuove e sconosciute, che non facevano parte dei dati di addestramento e quindi non fanno parte del vocabolario esistente. Inoltre, aggiungiamo un token <|endoftext|>, che possiamo usare per separare due fonti testuali non correlate.

Possiamo modificare il tokenizer per usare un token `<|unk|>` se incontra una parola che non fa parte del vocabolario. Inoltre, aggiungiamo un token separatore, tra testi non correlati. Per esempio, quando si addestrano modelli LLM di tipo GPT su più documenti o libri, è pratica comune inserire un token separatore fra ogni documento o libro, come mostra la Figura 2.10. Ciò aiuta il modello LLM a capire che, sebbene queste fonti testuali siano concatenate per l'addestramento, non sono correlate.

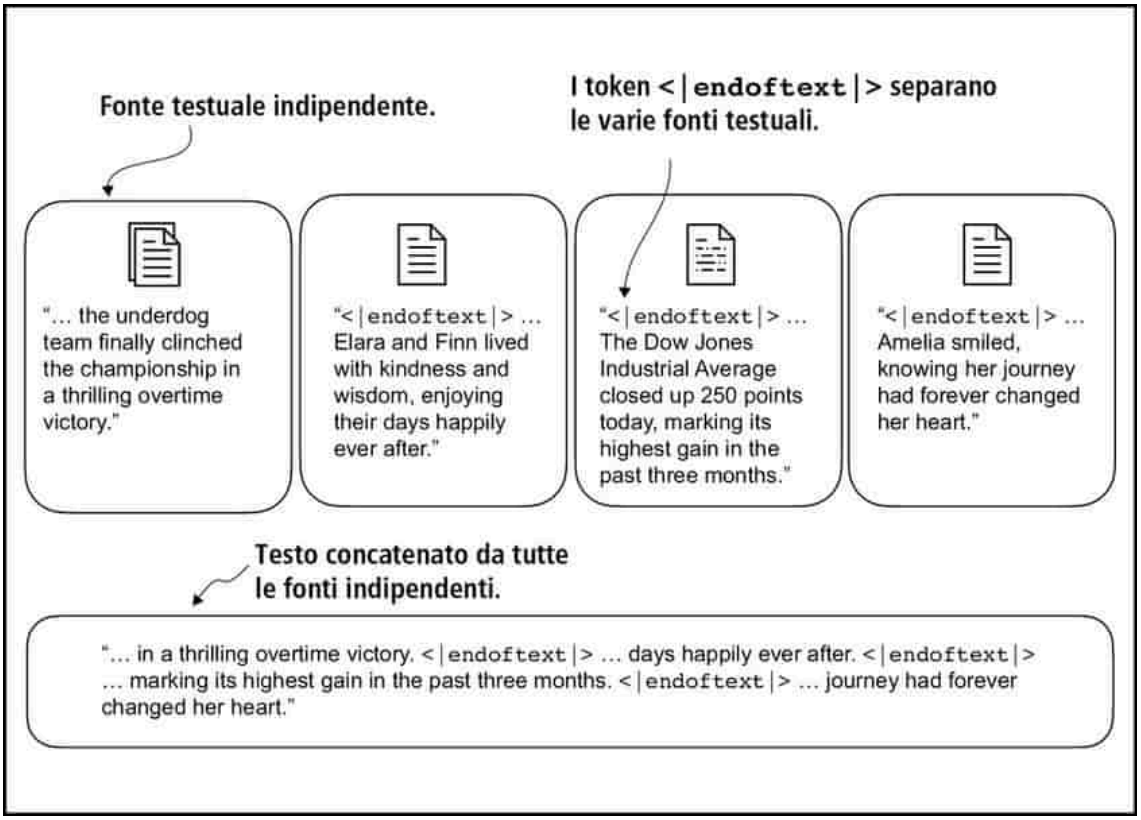


Figura 2.10 Quando lavoriamo con sorgenti di testo indipendenti, aggiungiamo il token <|endoftext|> tra di esse. Questi token agiscono come marcatori, segnalando l'inizio/la fine di un segmento, consentendo un'elaborazione e una comprensione più efficaci da parte del modello LLM.

Modifichiamo ora il vocabolario per includere questi due token speciali, `<|unk|>` ed `<|endoftext|>`, aggiungendoli al nostro elenco di tutte le parole univoche:

```
all_tokens = sorted(list(set(preprocessed)))
all_tokens.extend(["<|endoftext|>", "<|unk|>"])
vocab = {token:integer for integer,token in enumerate(all_tokens)}

print(len(vocab.items()))
```

In base all'output di questa istruzione `print`, ora le dimensioni del vocabolario sono pari a 1.132 (le dimensioni precedenti erano 1.130).

Come ulteriore rapido check, visualizziamo gli ultimi cinque elementi del vocabolario aggiornato:

```
for i, item in enumerate(list(vocab.items())[-5:]):
    print(item)
```

Il codice produce il seguente output:

```
('younger', 1127)
('your', 1128)
('yourself', 1129)
('<|endoftext|>', 1130)
('<|unk|>', 1131)
```

Sulla base di questo output, possiamo confermare che i due nuovi token speciali sono stati effettivamente incorporati con successo nel vocabolario. Ora, adattiamo il tokenizer del Listato 2.3, come mostrato nel Listato 2.4.

Listato 2.4 Un semplice tokenizer di testo che gestisce anche le parole sconosciute.

```
class SimpleTokenizerV2:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = { i:s for s,i in vocab.items()}

    def encode(self, text):
        preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', text)
        preprocessed = [item.strip() for item in preprocessed if item.strip()]
        preprocessed = [item if item in self.str_to_int (1)
                        else "<|unk|>" for item in preprocessed]

        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    def decode(self, ids):
        text = " ".join([self.int_to_str[i] for i in ids])
        text = re.sub(r'\s+([,.:;?!"()\'])', r'\1', text) (2)
        return text
```

(1) Sostituisce le parole sconosciute con token `<|unk|>`.

(2) Sostituisce gli spazi prima dei segni di punteggiatura specificati.

Rispetto al `SimpleTokenizerV1` implementato nel Listato 2.3, il nuovo `SimpleTokenizerV2` sostituisce le parole sconosciute con i token `<|unk|>`.

Ora proviamo questo nuovo tokenizer. Per farlo, useremo un semplice esempio di testo che concateniamo da due frasi indipendenti e non correlate:

```
text1 = "Hello, do you like tea?"
text2 = "In the sunlit terraces of the palace."
text = " <|endoftext|> ".join((text1, text2))
print(text)
```

L’output è il seguente:

```
Hello, do you like tea? <|endoftext|> In the sunlit terraces of the palace.
```

Ora tokenizziamo il testo campione utilizzando `SimpleTokenizerV2` sul vocabolario creato in precedenza nel Listato 2.2:

```
tokenizer = SimpleTokenizerV2(vocab)
print(tokenizer.encode(text))
```

Questo stampa i seguenti token-ID:

```
[1131, 5, 355, 1126, 628, 975, 10, 1130, 55, 988, 956, 984, 722, 988, 1131, 7]
```

Possiamo vedere che l’elenco dei token-ID contiene `1130` per il token separatore `<|endoftext|>` e due token `1131`, utilizzati per le parole sconosciute.

Per un rapido controllo di integrità, de-tokenizziamo il testo:

```
print(tokenizer.decode(tokenizer.encode(text)))
```

L’output è il seguente:

```
<|unk|>, do you like tea? <|endoftext|> In the sunlit terraces of the <|unk|>.
```

Confrontando questo testo de-tokenizzato col testo di input originale, scopriamo che il dataset di addestramento, il racconto di Edith Wharton *The Verdict*, non contiene le parole “Hello” e “palace”.

A seconda del modello LLM, alcuni ricercatori prendono in considerazione anche altri token speciali, come i seguenti.

- `[BOS]` (*inizio sequenza*): questo token segna l’inizio di un testo. Indica al modello LLM dove inizia un frammento dei contenuti.
- `[EOS]` (*fine sequenza*): questo token è posizionato alla fine di un testo ed è particolarmente utile quando si concatenano più testi non correlati, in modo simile a `<|endoftext|>`. Per esempio, quando si combinano due diversi articoli di Wikipedia o libri, il token `[EOS]` indica dove finisce uno e inizia il successivo.
- `[PAD]` (*padding*): quando si addestrano modelli LLM con dimensioni dei “lotti”, *batch*, maggiori di 1, il batch potrebbe contenere testi di lunghezze variabili. Per garantire che tutti i

testi abbiano la stessa lunghezza, i testi più corti vengono "allungati" utilizzando il token `[PAD]`, fino alla lunghezza del testo più lungo nel batch.

Il tokenizer utilizzato per i modelli GPT non necessita di nessuno di questi token; per semplicità utilizza solo un token `<|endoftext|>`, analogo al token `[EOS]`. `<|endoftext|>` è utilizzato anche per il padding. Tuttavia, come esploreremo nei prossimi capitoli, quando si effettua l'addestramento su input in batch, in genere utilizziamo una maschera, il che significa che non ci occupiamo del padding dei token. Pertanto, il token specifico scelto per il padding diventa irrilevante.

Inoltre, il tokenizer utilizzato per i modelli GPT non utilizza un token `<|unk|>` per le parole fuori dal vocabolario. Al suo posto, i modelli GPT utilizzano un tokenizer di *codifica a coppie di byte*, che suddivide le parole in unità sub-parola, di cui parleremo in seguito.

BPE: codifica a coppie di byte

Esaminiamo ora uno schema di tokenizzazione più sofisticato, basato su un concetto chiamato codifica a coppie di byte (BPE - *Byte Pair Encoding*). Il tokenizer BPE è stato utilizzato per addestrare modelli LLM come GPT-2, GPT-3 e il modello originale utilizzato in ChatGPT.

Poiché implementare la codifica BPE può essere relativamente complicato, useremo una libreria open source Python chiamata *tiktoken* (`https://github.com/openai/tiktoken`), che implementa l'algoritmo BPE in modo molto efficiente, basandosi su codice sorgente Rust. In modo analogo a quanto avviene per altre librerie Python, possiamo installare la libreria tiktoken tramite l'installer `pip` dal terminale:

```
pip install tiktoken
```

Il codice che utilizzeremo è basato su tiktoken 0.7.0. Potete usare il seguente codice per controllare la versione che avete attualmente installato:

```
from importlib.metadata import version
import tiktoken
print("tiktoken version:", version("tiktoken"))
```

Una volta installato, possiamo creare un'istanza del tokenizer BPE da tiktoken come segue:

```
tokenizer = tiktoken.get_encoding("gpt2")
```

L'utilizzo di questo tokenizer è simile al `SimpleTokenizerV2` che abbiamo implementato in precedenza tramite un metodo `encode`:

```
text = (
    "Hello, do you like tea? <|endoftext|> In the sunlit terraces"
    "of someunknownPlace."
)
integers = tokenizer.encode(text, allowed_special={"<|endoftext|>"})
print(integers)
```

Il codice visualizza i seguenti token-ID:

```
[15496, 11, 466, 345, 588, 8887, 30, 220, 50256, 554, 262, 4252, 18250, 8812,
2114, 286, 617, 34680, 27271, 13]
```

Possiamo convertire nuovamente i token-ID in testo utilizzando il metodo `decode`, simile a quello del nostro `SimpleTokenizerV2`:

```
strings = tokenizer.decode(integers)
print(strings)
```

Il codice produce il seguente output:

```
Hello, do you like tea? <|endoftext|> In the sunlit terraces of someunknownPlace.
```

Possiamo fare due osservazioni degne di nota in base ai token-ID e al testo decodificato. Innanzitutto, al token `<|endoftext|>` viene assegnato un token-ID relativamente grande, vale a dire, `50256`. Infatti, il tokenizer BPE, che era utilizzato per addestrare modelli come GPT-2, GPT-3 e il modello originale utilizzato in ChatGPT, ha un vocabolario di 50.257 token totali, e a `<|endoftext|>` è assegnato il token-ID più grande.

In secondo luogo, il tokenizer BPE codifica e decodifica correttamente le parole sconosciute, come `someunknownPlace`. Il tokenizer BPE può gestire qualsiasi parola sconosciuta. Come riesce a farlo senza usare token `<|unk|>`?

L'algoritmo che è alla base dell'encoder BPE scompone le parole che non sono nel suo vocabolario in unità di sub-parole più piccole o perfino in singoli caratteri, consentendogli di gestire parole non presenti nel vocabolario. Quindi, grazie all'algoritmo BPE, se durante la tokenizzazione il tokenizer incontra una parola non familiare, può rappresentarla come una sequenza di token di sub-parole o caratteri, come mostra la Figura 2.11.

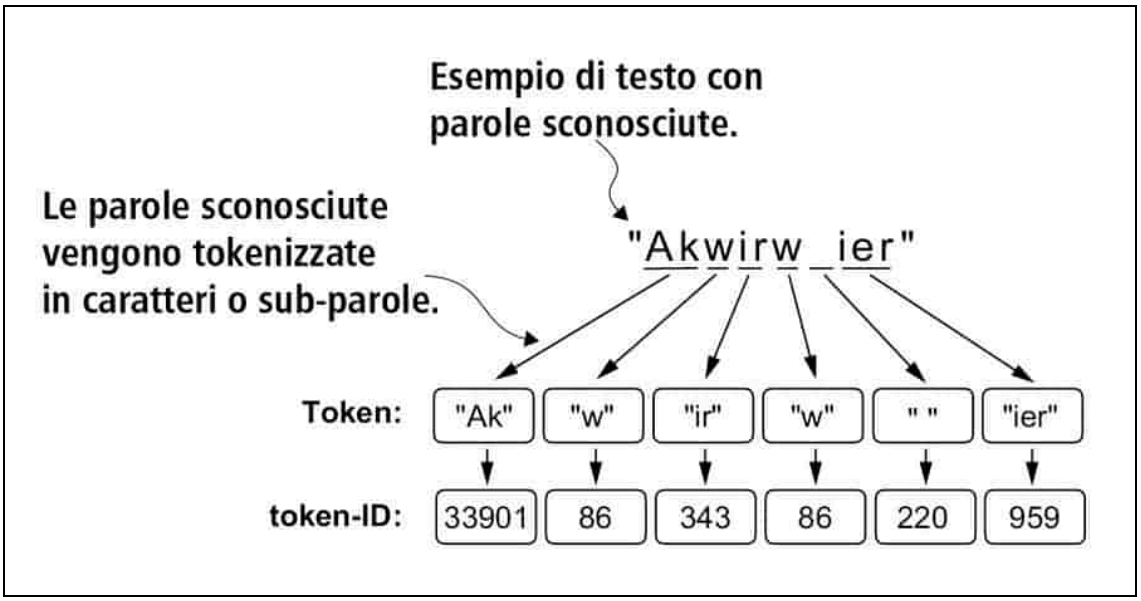


Figura 2.11 I tokenizer BPE scompongono le parole sconosciute in sub-parole e singoli caratteri. In questo modo, un tokenizer BPE può analizzare qualsiasi parola e non ha bisogno di sostituire le parole sconosciute con token speciali, come <|unk|>.

La capacità di scomporre le parole sconosciute in singoli caratteri garantisce che il tokenizer e, di conseguenza, il modello LLM con esso addestrato possano elaborare qualsiasi testo, anche se contiene parole che non erano presenti nei suoi dati di addestramento.

Esercizio 2.1 Codifica a coppie di byte di parole sconosciute
Provate il tokenizer BPE della libreria *tiktoken* sulle parole sconosciute “Akwirw ier” e mostrate i singoli token-ID. Quindi, richiamate la funzione `decode` su ciascuno degli interi risultanti in questo elenco, per riprodurre la mappatura rappresentata nella Figura 2.11. Infine, richiamate il metodo `decode` sui token-ID, per verificare che riesca a ricostruire l’input originale, “Akwirw ier”.

Una discussione e un’implementazione dettagliate dell’encoder BPE esulano dagli scopi di questo libro, ma in breve, costruisce il proprio vocabolario unendo iterativamente i caratteri frequenti in sub-parole più frequenti, e poi le sub-parole più frequenti in parole. Per esempio, l’encoder BPE inizia aggiungendo al suo vocabolario tutti i singoli caratteri (“a”, “b” e così via). Nella fase successiva, unisce le combinazioni di caratteri che si verificano più frequentemente insieme in sub-parole. Per esempio, “d” ed “e” possono essere unite nella sub-parola “de”, che è comune in molte parole. Le unioni sono determinate da un limite legato alla frequenza minima.

Campionamento dei dati a finestra scorrevole

Il passo successivo nella creazione degli embedding per il modello LLM è quello di generare le coppie input-target necessarie per addestrare un modello LLM. Come si formano queste coppie input-target? Come già sappiamo, i modelli LLM vengono pre-addestrati prevedendo la parola successiva in un testo, come mostra la Figura 2.12.

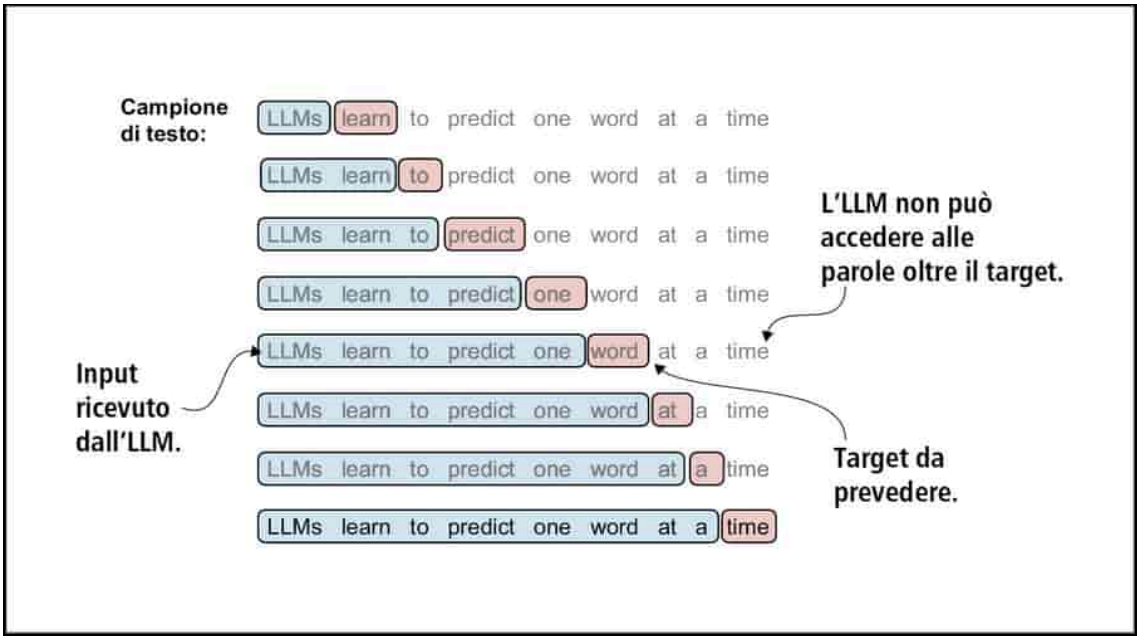


Figura 2.12 Dato un campione di testo, estraiamo i blocchi di input come sub-campioni che servono come input per il modello LLM, e il compito di previsione del modello LLM durante l'addestramento è quello di prevedere la parola successiva che segue il blocco di input. Durante l'addestramento, mascheriamo tutte le parole che vanno oltre il target. Notate che il testo mostrato in questa figura deve essere sottoposto a tokenizzazione prima che il modello LLM possa elaborarlo; tuttavia, per semplicità, questa figura omette il passo di tokenizzazione.

Implementiamo un *data loader* (un "caricatore di dati") che recupera le coppie input-target (Figura 2.12) dal dataset di addestramento usando un approccio a finestra scorrevole. Per iniziare, tokenizzeremo l'intero racconto *The Verdict* usando il tokenizer BPE:

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

enc_text = tokenizer.encode(raw_text)
print(len(enc_text))
```

L'esecuzione di questo codice produrrà in output il valore `5145`, il numero totale di token nel dataset di addestramento, dopo

l’applicazione del tokenizer BPE.

Ora, a scopo dimostrativo rimuoviamo dal dataset i primi 50 token, poiché nei passi successivi si ha un passo leggermente più interessante:

```
enc_sample = enc_text[50:]
```

Uno dei modi più semplici e intuitivi per creare le coppie input-target per l’attività di previsione della parola successiva consiste nel creare due variabili, `x` e `y`, dove `x` contiene i token di input e `y` contiene i target, che sono gli input spostati di `1`:

```
context_size = 4 (1)
x = enc_sample[:context_size]
y = enc_sample[1:context_size+1]
print(f"x: {x}")
print(f"y:      {y}")
```

(1) Le dimensioni del contesto determinano quanti token sono inclusi nell’input.

L’esecuzione del codice precedente produce il seguente output:

```
x: [290, 4920, 2241, 287]
y:      [4920, 2241, 287, 257]
```

Elaborando gli input insieme ai target, che sono gli input spostati di una posizione, possiamo creare i compiti di previsione della parola successiva (vedi Figura 2.12), come segue:

```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(context, "---->", desired)
```

Il codice produce il seguente output:

```
[290] ----> 4920
[290, 4920] ----> 2241
[290, 4920, 2241] ----> 287
[290, 4920, 2241, 287] ----> 257
```

Tutto ciò che rimane a sinistra della freccia (`--->`) si riferisce all’input che riceverebbe il modello LLM, e il token-ID sul lato destro della freccia rappresenta il token-ID target che il modello LLM dovrebbe prevedere. Ripetiamo il codice precedente, ma convertiamo i token-ID in testo:

```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(tokenizer.decode(context), "---->", tokenizer.decode([desired]))
```

Gli output seguenti mostrano l’aspetto degli input e degli output in formato testuale:

```
and ----> established
and established ----> himself
and established himself ----> in
and established himself in ----> a
```

Abbiamo ora creato le coppie input-target che possiamo utilizzare per l’addestramento del modello LLM.

C’è solo un altro compito prima di poter trasformare i token in embedding: implementare un data loader efficiente che iteri sul dataset di input e restituisca gli input e i target come tensori di PyTorch, che possono essere considerati come array multidimensionali. In particolare, siamo interessati a restituire due tensori: un tensore di input contenente il testo che il modello LLM vede; un tensore target che include i target che il modello LLM deve prevedere, come mostra la Figura 2.13. La figura mostra i token sotto forma di stringhe a scopo puramente illustrativo; l’implementazione del codice opererà direttamente sui token-ID, poiché il metodo `encode` del tokenizer BPE esegue in un unico passo sia la tokenizzazione sia la conversione in token-ID.

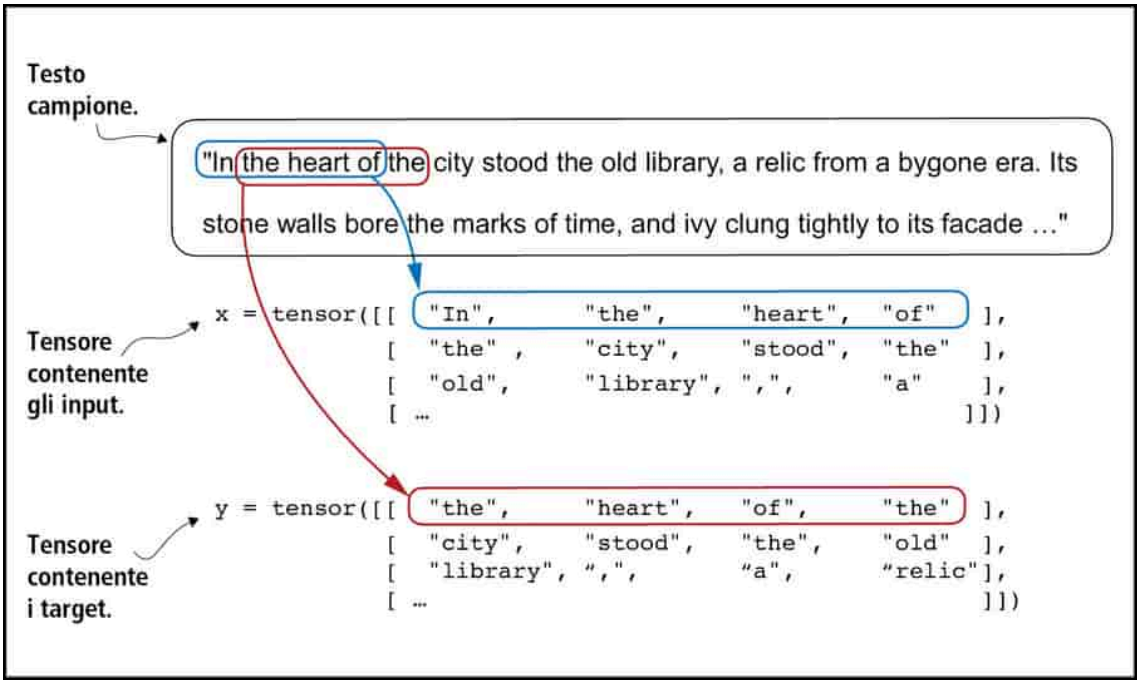


Figura 2.13 Per implementare data loader efficienti, raccogliamo gli input in un tensore, x, in cui ogni riga rappresenta un contesto di input. Un secondo tensore, y, contiene i corrispondenti target da prevedere (le parole successive), che vengono creati spostando l'input di una posizione.

NOTA

Per un'implementazione efficiente del data loader, utilizzeremo le classi `Dataset` e `DataLoader` di PyTorch. Per ulteriori informazioni e indicazioni sull'installazione di PyTorch, consultate l'Appendice A.

Il codice per la classe del dataset è rappresentato nel Listato 2.5.

Listato 2.5 Un dataset per input e target in batch.

```
import torch
from torch.utils.data import Dataset, DataLoader

class GPTDatasetV1(Dataset):
    def __init__(self, txt, tokenizer, max_length, stride):
        self.input_ids = []
        self.target_ids = []

        token_ids = tokenizer.encode(txt) (1)

        for i in range(0, len(token_ids) - max_length, stride): (2)
            input_chunk = token_ids[i:i + max_length]
```

```
            target_chunk = token_ids[i + 1: i + max_length + 1]
            self.input_ids.append(torch.tensor(input_chunk))
            self.target_ids.append(torch.tensor(target_chunk))

    def __len__(self): (3)
        return len(self.input_ids)

    def __getitem__(self, idx): (4)
        return self.input_ids[idx], self.target_ids[idx]
```

(1) Tokenizza l'intero testo.

(2) Utilizza una finestra scorrevole per suddividere il libro in sequenze sovrapposte di lunghezza massima `max_lenght`.

(3) Restituisce il numero totale di righe nel dataset.

(4) Restituisce una singola riga dal dataset.

La classe `GPTDatasetV1` si basa sulla classe `Dataset` di PyTorch e definisce come le singole righe vengono recuperate dal dataset, dove ogni riga è costituita da un certo numero di token-ID (basati su `max_length`) assegnati a un tensore `input_chunk`. Il tensore `target_chunk` contiene i target corrispondenti. Vi consiglio di continuare la lettura per vedere l'aspetto dei dati restituiti da questo dataset quando lo combiniamo con un `DataLoader` PyTorch: tutto sarà molto più chiaro.

> **NOTA**
>
> Se non conoscete la struttura delle classi `Dataset` di PyTorch, usate nel Listato 2.5, consultate l'Appendice A, che spiega la struttura generale e l'utilizzo delle classi `Dataset` e `DataLoader` di PyTorch.

Il codice del Listato 2.6 utilizza `GPTDatasetV1` per caricare gli input in batch tramite il `DataLoader` di PyTorch.

Listato 2.6 Un data loader per generare i batch.

```
def create_dataloader_v1(txt, batch_size=4, max_length=256,
                         stride=128, shuffle=True, drop_last=True,
                         num_workers=0):
    tokenizer = tiktoken.get_encoding("gpt2") (1)
    dataset = GPTDatasetV1(txt, tokenizer, max_length, stride) (2)
    dataloader = DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=shuffle,
        drop_last=drop_last, (3)
```

```
        num_workers=num_workers (4)
    )

    return dataloader
```

(1) Inizializza il tokenizer.

(2) Crea il dataset.

(3) `drop_last=True` elimina l'ultimo batch se è più breve del `batch_size` specificato, per evitare problemi durante l'addestramento.

(4) Il numero di processi CPU da utilizzare per la pre-elaborazione.

Proviamo a sottoporre a test il `dataloader` con dimensioni del batch pari a `1` per un modello LLM con una dimensione di contesto pari a `4` per cercare di capire come funzionano, insieme, la classe `GPTDatasetV1` del Listato 2.5 e la funzione `create_dataloader_v1` del Listato 2.6:

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

dataloader = create_dataloader_v1(raw_text, batch_size=1, max_length=4, stride=1,
shuffle=False)
data_iter = iter(dataloader) (1)
first_batch = next(data_iter)
print(first_batch)
```

(1) Converte il data loader in un iteratore Python per recuperare l'elemento successivo tramite la funzione `next()` di Python.

L'esecuzione del codice precedente produce il seguente output:

```
[tensor([[40, 367, 2885, 1464]]), tensor([[367, 2885, 1464, 1807]])]
```

La variabile `first_batch` contiene due tensori: il primo memorizza i token-ID di input, e il secondo memorizza i token-ID target. Poiché `max_length` è impostato su `4`, i due tensori contengono entrambi quattro ID dei token. Notate che le dimensioni di input pari a `4` sono piuttosto piccole, e sono state scelte solo per semplicità. È pratica comune addestrare i modelli LLM con dimensioni di input pari ad almeno `256`.

Per comprendere il significato di `stride=1`, recuperiamo un altro batch da questo dataset:

```
second_batch = next(data_iter)
print(second_batch)
```

Il secondo batch ha il seguente contenuto:

```
[tensor([[367, 2885, 1464, 1807]]), tensor([[2885, 1464, 1807, 3619]])]
```

Se confrontiamo il primo e il secondo batch, possiamo vedere che i token-ID del secondo batch sono spostati di una posizione (per esempio, il secondo ID del primo batch è `367`, che è il primo ID del secondo batch). L'impostazione `stride` determina il numero di posizioni di cui gli input si spostano tra i batch, emulando un approccio a finestra scorrevole, come mostra la Figura 2.14.

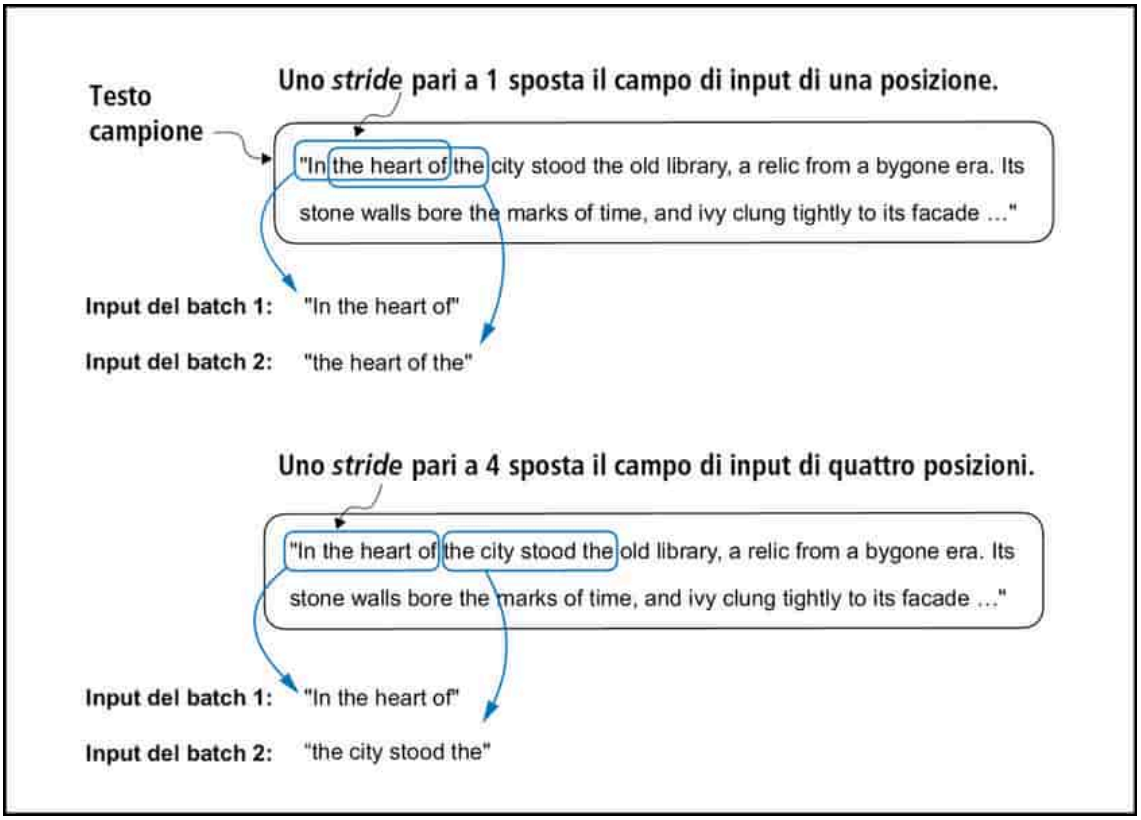


Figura 2.14 Quando creiamo più batch dal dataset di input, facciamo scorrere una finestra di input sul testo. Se il passo (stride) è impostato a 1, spostiamo la finestra di input di una posizione quando creiamo il batch successivo. Se impostiamo stride uguale alle dimensioni della finestra di input, possiamo impedire la sovrapposizioni tra i batch.

Esercizio 2.2 Data loader con diverse dimensioni di passo (stride) e contesto
Per comprendere meglio il funzionamento del data loader, provate a eseguirlo con impostazioni differenti, per esempio `max_length=2` e `stride=2`, oppure `max_length=8` e `stride=2`.

Le dimensioni del batch pari a 1, come quelle che abbiamo campionato finora dal data loader, sono utili solo a scopo illustrativo. Se avete esperienza col deep learning, potreste sapere che batch di piccole dimensioni richiedono meno memoria durante l'addestramento, ma portano ad aggiornamenti del modello più "rumorosi". Proprio come nel deep learning, le dimensioni del batch sono un compromesso e un iperparametro con cui sperimentare durante l'addestramento dei modelli LLM.

Vediamo ora brevemente come possiamo utilizzare il data loader per effettuare il campionamento con dimensioni del batch maggiori di `1`:

```
dataloader = create_dataloader_v1(
    raw_text, batch_size=8, max_length=4, stride=4,
    shuffle=False
)

data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Inputs:\n", inputs)
print("\nTargets:\n", targets)
```

Questo codice produce il seguente output:

```
Inputs:
 tensor([[ 40,   367,  2885,  1464],
        [ 1807,  3619,   402,   271],
        [10899,  2138,   257,  7026],
        [15632,   438,  2016,   257],
        [  922,  5891,  1576,   438],
        [  568,   340,   373,   645],
        [ 1049,  5975,   284,   502],
        [  284,  3285,   326,    11]]
 )

Targets:
 tensor([[ 367, 2885, 1464, 1807],
         [ 3619, 402, 271, 10899],
         [ 2138, 257, 7026, 15632],
         [ 438, 2016, 257, 922],
         [ 5891, 1576, 438, 568],
         [ 340, 373, 645, 1049],
```

```
        [ 5975, 284, 502, 284],
        [ 3285, 326, 11, 287]]
)
```

Notate che aumentiamo lo `stride` a `4` per utilizzare completamente il dataset (non saltiamo una sola parola). Ciò evita qualsiasi sovrapposizione tra i batch, poiché una sovrapposizione maggiore potrebbe portare a un aumento dell'overfitting.

Creazione degli embedding dei token

L'ultimo passo nella preparazione del testo di input per l'addestramento del modello LLM consiste nel convertire i token-ID in vettori di embedding, come mostra la Figura 2.15. Come passo preliminare, dobbiamo inizializzare i pesi di embedding con valori casuali. Questa inizializzazione serve come punto di partenza per il processo di apprendimento del modello LLM. Nel Capitolo 5, ottimizzeremo i pesi di embedding nell'ambito dell'addestramento del modello LLM.

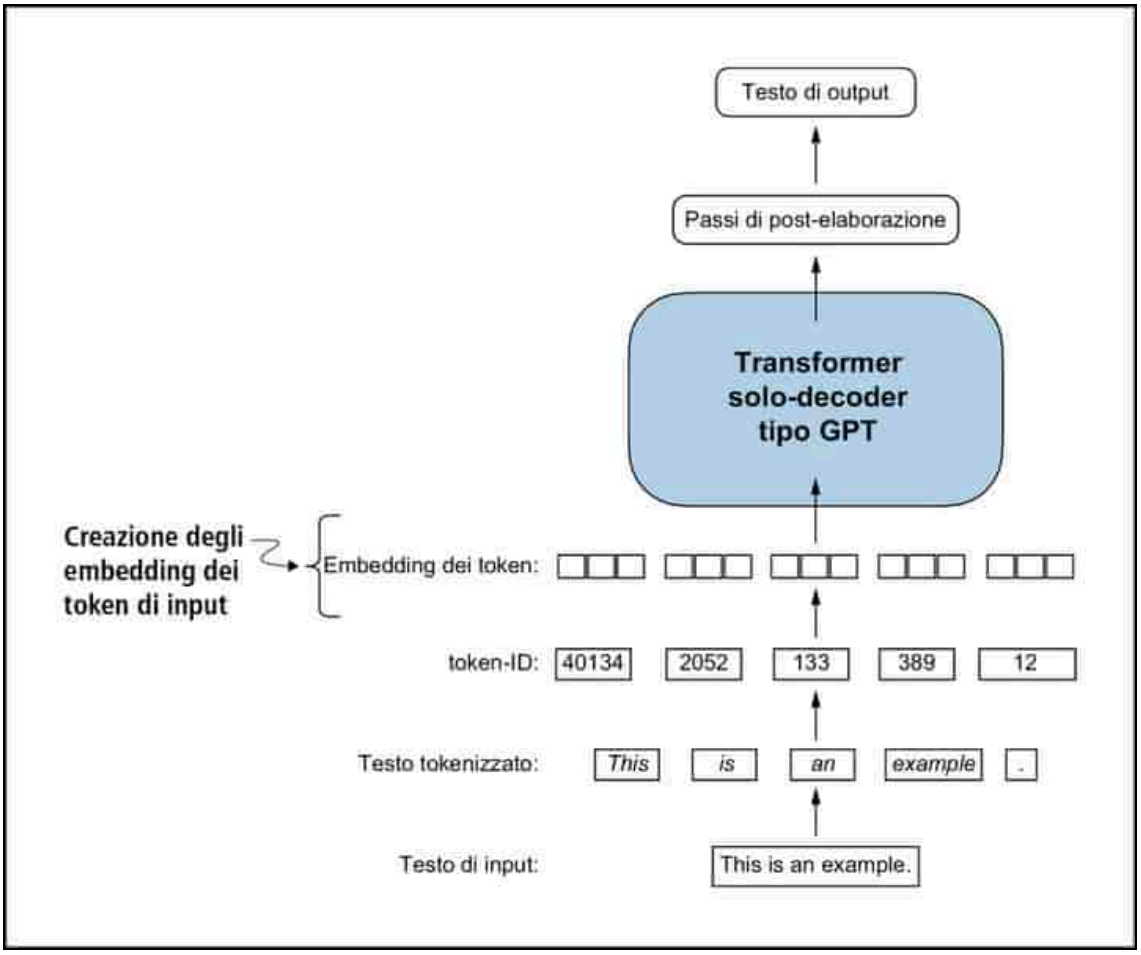


Figura 2.15 La preparazione comporta la tokenizzazione del testo, la conversione dei token di testo in token-ID e la conversione dei token-ID in vettori di embedding. Qui, consideriamo i token-ID creati in precedenza per creare i vettori di embedding dei token.

È necessaria una rappresentazione vettoriale continua, o *embedding*, poiché i modelli LLM di tipo GPT sono reti neurali profonde addestrate con l'algoritmo di retropropagazione (*backpropagation*).

NOTA

Se non avete familiarità col modo in cui le reti neurali vengono addestrate con la retropropagazione, consultate l'Appendice A.

Vediamo come funziona la conversione da token-ID a vettori di embedding con un esempio pratico. Supponiamo di avere i seguenti

quattro token di input, con ID `2`, `3`, `5` e `1`:

```
input_ids = torch.tensor([2, 3, 5, 1])
```

Per semplicità, supponiamo di avere un piccolo vocabolario di sole sei parole (invece delle 50.257 parole del vocabolario del tokenizer BPE) e di voler creare embedding di dimensioni pari a `3` (in GPT-3, gli embedding hanno dimensioni pari a 12.288):

```
vocab_size = 6
output_dim = 3
```

Utilizzando `vocab_size` e `output_dim`, possiamo creare un layer di embedding in PyTorch, impostando il seme casuale su `123` per motivi di riproducibilità:

```
torch.manual_seed(123)
embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
print(embedding_layer.weight)
```

L’istruzione `print` mostra la matrice dei pesi del layer di embedding:

```
Parameter containing:
tensor([[ 0.3374, -0.1778, -0.1690],
        [ 0.9178, 1.5810, 1.3010],
        [ 1.2753, -0.2010, -0.1606],
        [-0.4015, 0.9666, -1.1481],
        [-1.1589, 0.3255, -0.6315],
        [-2.8400, -0.7849, -1.4096]],
        requires_grad=True
)
```

La matrice dei pesi del layer di embedding contiene piccoli valori casuali. Questi valori vengono ottimizzati durante l’addestramento del modello LLM nell’ambito del fine-tuning del modello stesso. Inoltre, possiamo vedere che la matrice dei pesi è di sei righe per tre colonne. C’è una riga per ciascuno dei sei possibili token presenti nel vocabolario e c’è una colonna per ciascuna delle tre dimensioni dell’embedding.

Ora applichiamolo a un token-ID per ottenere il vettore di embedding:

```
print(embedding_layer(torch.tensor([3])))
```

Il vettore di embedding restituito è il seguente:

```
tensor([[-0.4015, 0.9666, -1.1481]], grad_fn=<EmbeddingBackward0>)
```

Se confrontiamo il vettore di embedding per il token-ID `3` con la matrice di embedding precedente, vediamo che è identico alla quarta riga (Python inizia dall'indice zero, quindi è la riga corrispondente all'indice `3`). In altre parole, il layer di embedding è essenzialmente un'operazione di ricerca che recupera le righe dalla matrice dei pesi del layer di embedding tramite un token-ID.

NOTA

Per coloro che hanno familiarità con la codifica one-hot, l'approccio del layer di embedding qui descritto è essenzialmente un modo più efficiente per implementare la codifica one-hot seguita dalla moltiplicazione di matrici in un layer completamente connesso, che è illustrato nel codice disponibile su GitHub all'indirizzo https://mng.bz/ZEB5. Poiché il layer di embedding è solo un'implementazione più efficiente equivalente all'approccio codifica one-hot più moltiplicazione di matrici, può essere visto come un layer della rete neurale che può essere ottimizzato tramite retropropagazione.

Abbiamo visto come convertire un singolo token-ID in un vettore di embedding tridimensionale. Applichiamolo ora a tutti e quattro gli ID di input (`torch.tensor ([2, 3, 5, 1])`):

```
print(embedding_layer(input_ids))
```

L'output rivela che il risultato è una matrice 4 × 3:

```
tensor([[ 1.2753, -0.2010, -0.1606],
        [-0.4015, 0.9666, -1.1481],
        [-2.8400, -0.7849, -1.4096],
        [ 0.9178, 1.5810, 1.3010]],
        grad_fn=<EmbeddingBackward0>
)
```

Ogni riga di questa matrice di output viene ottenuta tramite un'operazione di ricerca dalla matrice dei pesi di embedding, come mostra la Figura 2.16.

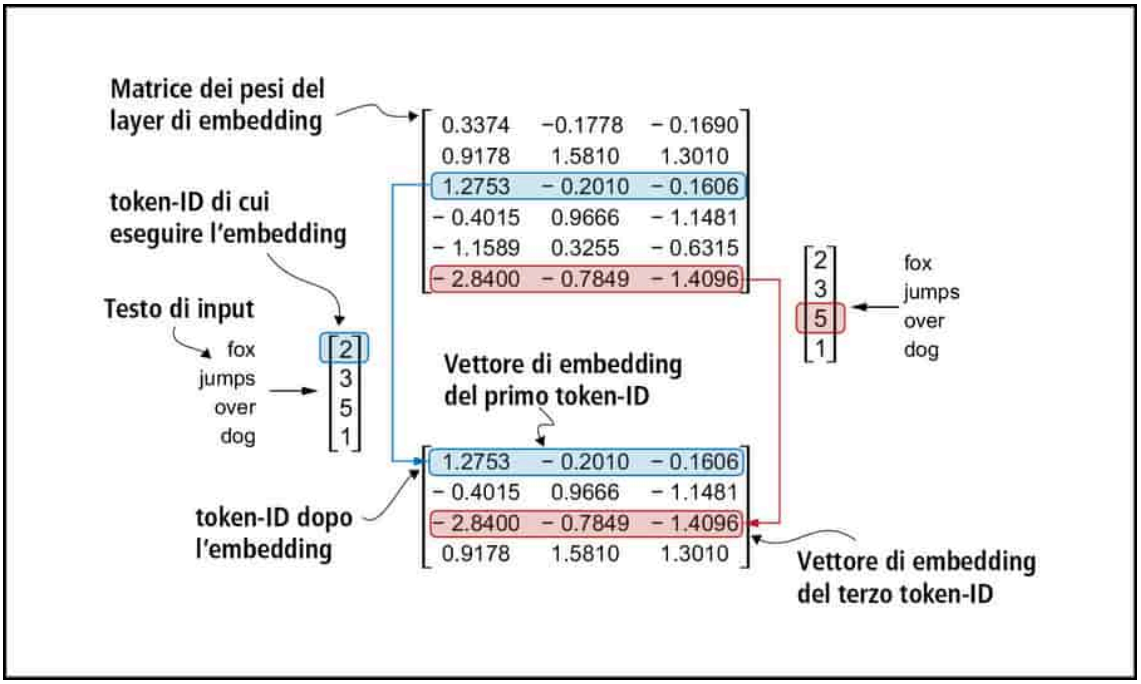


Figura 2.16 I layer di embedding eseguono un'operazione di ricerca, recuperando il vettore di embedding corrispondente al token-ID dalla matrice dei pesi del layer di embedding. Per esempio, il vettore di embedding del token-ID 5 è la sesta riga della matrice dei pesi del layer di embedding (è la sesta anziché la quinta riga perché Python inizia a contare da 0). Supponiamo che i token-ID siano stati prodotti dal piccolo vocabolario creato in precedenza.

Dopo aver creato i vettori di embedding dai token-ID, aggiungeremo una piccola modifica a questi vettori per codificare le informazioni posizionali di un token all'interno di un testo.

Codifica della posizione delle parole

In linea di principio, gli embedding dei token sono un input adatto per un modello LLM. Tuttavia, una piccola lacuna dei modelli LLM è che il loro meccanismo di auto-attenzione (vedi il Capitolo 3) non ha

il concetto di posizione o di ordine dei token all'interno di una sequenza. Il modo in cui funziona il layer di embedding introdotto in precedenza è che lo stesso token-ID viene sempre mappato sulla stessa rappresentazione vettoriale, indipendentemente da dove si trova posizionato il token-ID nella sequenza di input, come mostra la Figura 2.17.

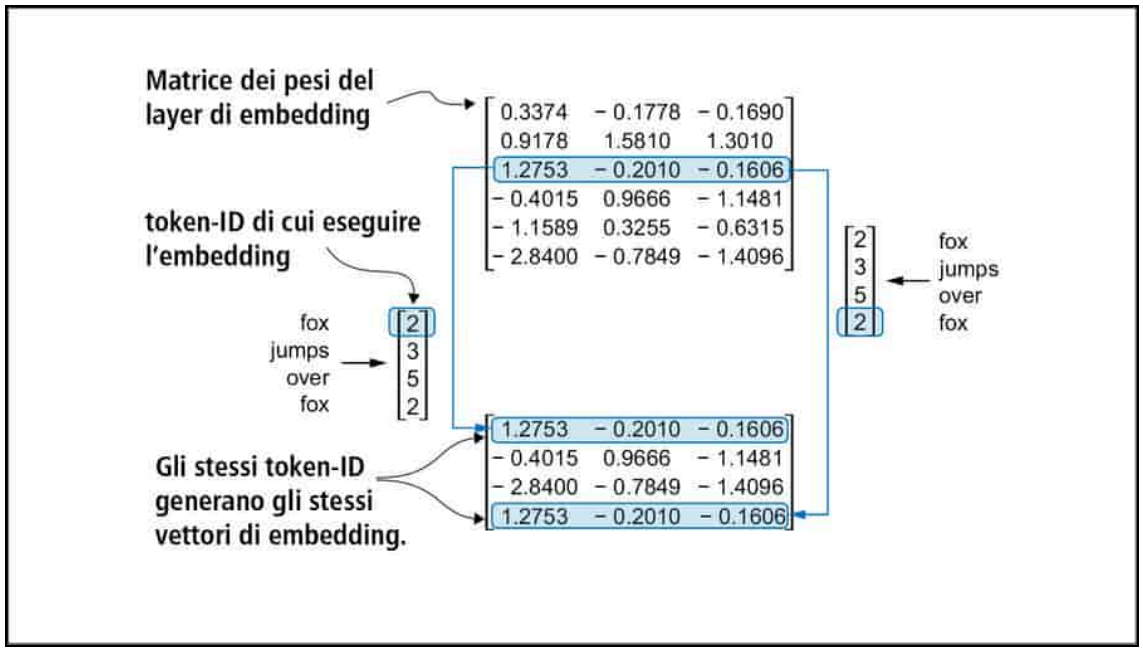


Figura 2.17 Il layer di embedding converte un token-ID nella stessa rappresentazione vettoriale, indipendentemente da dove si trova nella sequenza di input. Per esempio, il token-ID 5, nella prima o quarta posizione nel vettore di input, produrrà sempre lo stesso vettore di embedding.

In linea di principio, l'embedding deterministico e indipendente dalla posizione del token-ID è valido per scopi di riproducibilità. Tuttavia, poiché il meccanismo di auto-attenzione dei modelli LLM è anch'esso indipendente dalla posizione, è utile iniettare anche delle informazioni di posizione nel modello LLM.

Per ottenere questo risultato, possiamo usare due ampie categorie di embedding sensibili alla posizione: embedding posizionali relativi

e assoluti. Gli embedding posizionali assoluti sono direttamente associati a specifiche posizioni in una sequenza. Per ogni posizione nella sequenza di input, all'embedding dei token viene aggiunto un embedding univoco, per comunicare la sua esatta posizione. Per esempio, il primo token avrà un suo embedding posizionale, il secondo token un altro embedding, differente, e così via, come mostra la Figura 2.18.

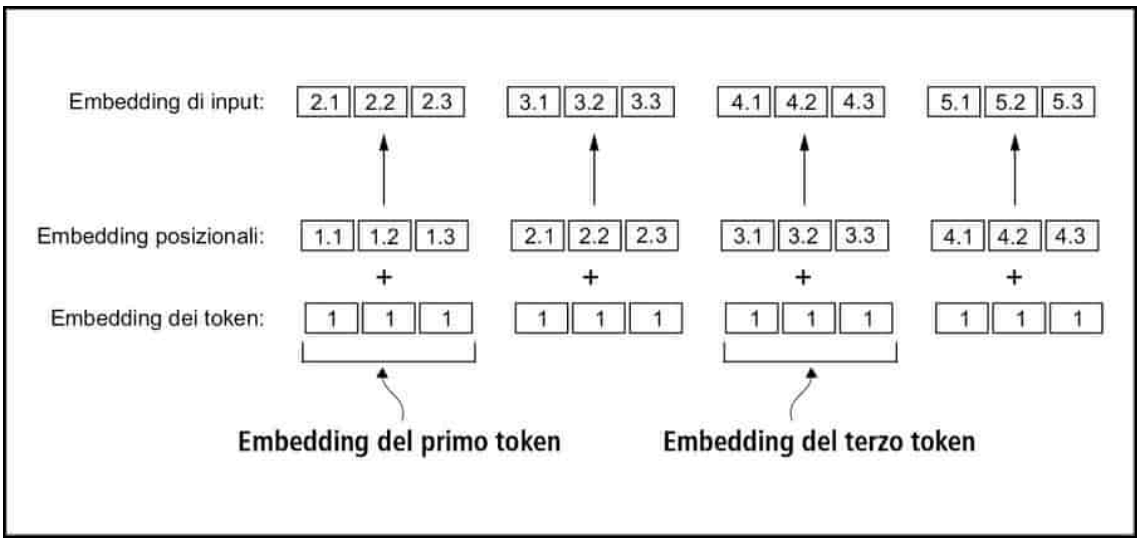


Figura 2.18 Gli embedding posizionali vengono aggiunti al vettore di embedding dei token per creare gli embedding di input per un modello LLM. I vettori posizionali hanno le stesse dimensioni degli embedding dei token originali. Gli embedding dei token sono mostrati con valore 1 per semplicità.

Invece di concentrarsi sulla posizione assoluta di un token, l'enfasi degli embedding posizionali relativi è sulla posizione relativa o sulla distanza tra token. Ciò significa che il modello apprende le relazioni in termini di "quanto distanti" anziché "in quale posizione esatta". Il vantaggio è che il modello può generalizzare meglio sequenze di lunghezza variabile, anche se non ha visto tale lunghezza durante l'addestramento.

Entrambi i tipi di embedding posizionali mirano ad aumentare la capacità dei modelli LLM di comprendere l'ordine e le relazioni tra i token, garantendo previsioni più accurate e consapevoli del contesto. La scelta spesso dipende dall'applicazione specifica e dalla natura dei dati elaborati.

I modelli GPT di OpenAI utilizzano embedding posizionali assoluti, che vengono ottimizzati durante il processo di addestramento anziché essere fissi o predefiniti come le codifiche posizionali del modello a transformer originale. Questo processo di ottimizzazione fa parte dell'addestramento del modello stesso. Per ora, creiamo gli embedding posizionali iniziali per creare gli input per il modello LLM.

In precedenza, per semplicità ci siamo concentrati su dimensioni dell'embedding molto piccole. Ora, prendiamo in considerazione dimensioni dell'embedding più realistiche e utili, e codifichiamo i token di input in una rappresentazione vettoriale a 256 dimensioni, che è più piccola di quella utilizzata dal modello GPT-3 originale (in GPT-3, le dimensioni dell'embedding sono pari a 12.288) ma comunque ragionevole per la sperimentazione. Inoltre, supponiamo che i token-ID siano stati creati dal tokenizer BPE che abbiamo implementato in precedenza, che ha una dimensioni del vocabolario pari a 50.257:

```
vocab_size = 50257
output_dim = 256
token_embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
```

Utilizzando il precedente `token_embedding_layer`, se eseguiamo il campionamento dei dati dal data loader, incorporiamo ogni token di ogni batch in un vettore a 256 dimensioni. Se le dimensioni del batch sono pari a 8 con quattro token ciascuno, il risultato sarà un tensore 8 × 4 × 256.

Per prima cosa creiamo un'istanza del data loader (Listato 2.6):

```
max_length = 4
dataloader = create_dataloader_v1(raw_text, batch_size=8, max_length=max_length,
                                  stride=max_length, shuffle=False
)
data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Token IDs:\n", inputs)
print("\nInputs shape:\n", inputs.shape)
```

Il codice precedente produce il seguente output:

```
Token IDs:
 tensor([[   40,   367,  2885,  1464],
         [ 1807,  3619,   402,   271],
         [10899,  2138,   257,  7026],
         [15632,   438,  2016,   257],
         [  922,  5891,  1576,   438],
         [  568,   340,   373,   645],
         [ 1049,  5975,   284,   502],
         [  284,  3285,   326,    11]]
 )

Inputs shape:
 torch.Size([8, 4])
```

Come potete vedere, il tensore dei token-ID ha dimensioni pari a 8 × 4, il che significa che il batch di dati è composto da otto campioni di testo con quattro token ciascuno.

Ora utilizziamo il layer di embedding per incorporare questi token-ID in vettori a 256 dimensioni:

```
token_embeddings = token_embedding_layer(inputs)
print(token_embeddings.shape)
```

La chiamata alla funzione `print` produce il seguente output:

```
torch.Size([8, 4, 256])
```

L’output del tensore a 8 × 4 × 256 dimensioni mostra che ogni token-ID è ora sottoposto a embedding come un vettore a 256 dimensioni.

Per un approccio a embedding assoluto (proprio del modello GPT), dobbiamo solo creare un altro layer di embedding che abbia le stesse dimensioni dell’embedding del layer `token_embedding_layer`:

```
context_length = max_length
pos_embedding_layer = torch.nn.Embedding(context_length, output_dim)
pos_embeddings = pos_embedding_layer(torch.arange(context_length))
print(pos_embeddings.shape)
```

L’input per `pos_embeddings` è solitamente un vettore segnaposto `torch.arange(context_length)`, che contiene una sequenza di numeri 0, 1, ..., fino alla lunghezza massima dell’input meno 1. `context_length` è una variabile che rappresenta le dimensioni di input supportate dal modello LLM. Qui, la scegliamo in modo simile alla lunghezza massima del testo di input. Nella realtà, il testo di input può essere più lungo della lunghezza del contesto, nel qual caso dobbiamo troncare il testo.

L’output dell’istruzione `print` è il seguente:

```
torch.Size([4, 256])
```

Come potete vedere, il tensore di embedding posizionale è costituito da quattro vettori a 256 dimensioni. Ora possiamo aggiungerli direttamente agli embedding dei token, dove PyTorch aggiungerà il tensore 4 × 256 dimensionale `pos_embeddings` a ciascun tensore di embedding dei token 4 × 256 dimensionale in ciascuno degli otto batch:

```
input_embeddings = token_embeddings + pos_embeddings
print(input_embeddings.shape)
```

L’output è il seguente:

```
torch.Size([8, 4, 256])
```

Gli `input_embeddings` che abbiamo creato, come riassunto nella Figura 2.19, sono gli esempi di input dopo l’embedding, che ora possono essere elaborati dai moduli LLM, che inizieremo a implementare nel prossimo capitolo.

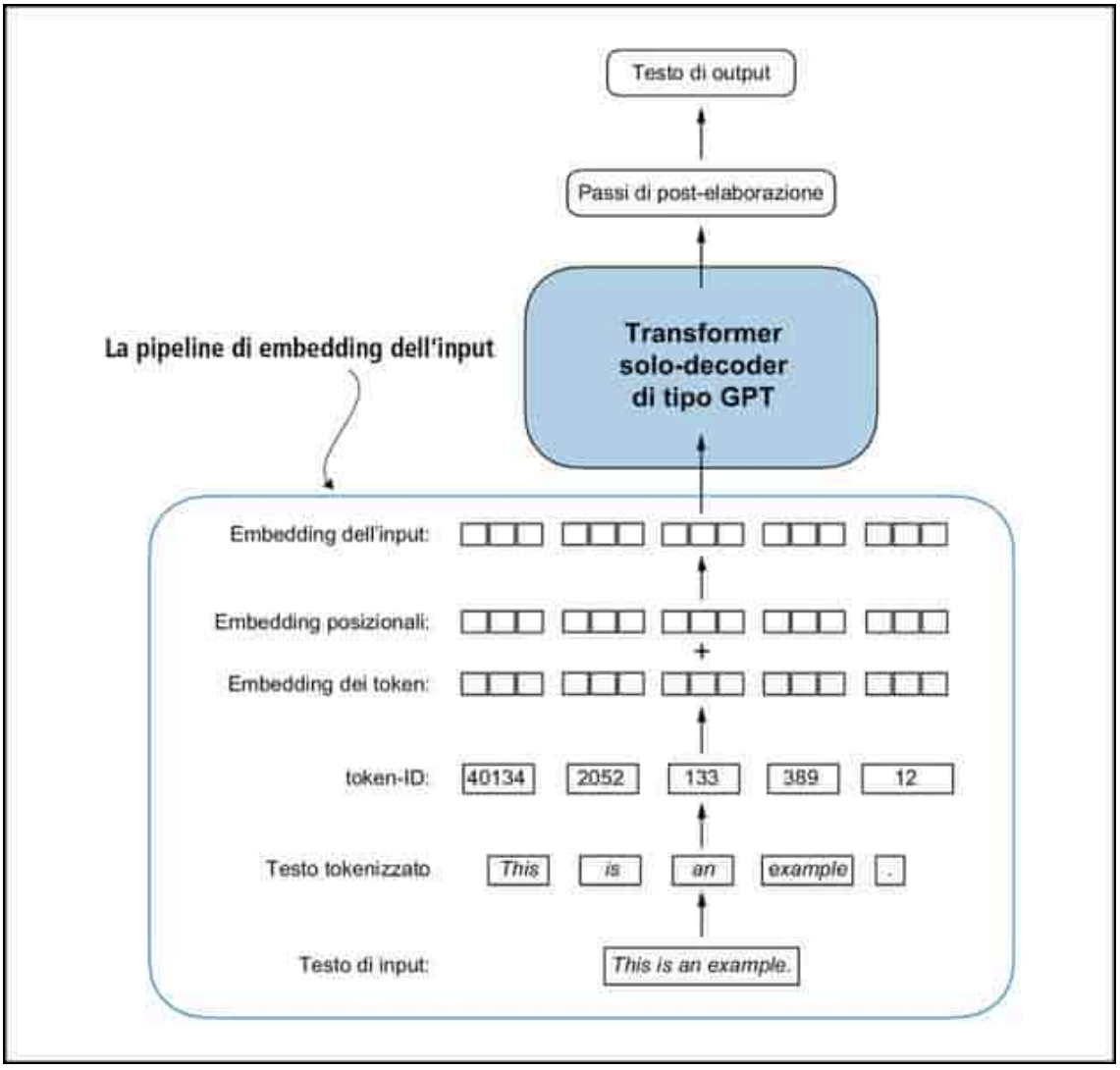


Figura 2.19 Nella pipeline di elaborazione dell'input, il testo di input viene prima suddiviso in singoli token. Questi token vengono poi convertiti in token-ID utilizzando un vocabolario. I token-ID vengono convertiti in vettori di embedding cui vengono aggiunti embedding posizionali di dimensioni simili, con conseguenti embedding di input che vengono utilizzati come input per i layer principali del modello LLM.

Riepilogo

- I modelli LLM richiedono che i dati testuali siano convertiti in vettori numerici, noti come embedding, poiché non possono elaborare il testo grezzo. Gli embedding trasformano i dati discreti (come le parole o le immagini) in spazi vettoriali continui, rendendoli compatibili con le attività delle reti neurali.
- Come primo passo, il testo grezzo viene suddiviso in token, che possono essere parole o caratteri. Quindi, i token vengono convertiti in rappresentazioni intere, denominate token-ID.
- Possono essere aggiunti dei token speciali, come `<|unk|>` e `<|endoftext|>`, per migliorare la comprensione del modello e gestire vari contesti, come parole sconosciute, o per contrassegnare il confine tra più testi.
- Il tokenizer BPE (*Byte Pair Encoding*) utilizzato da modelli LLM come GPT-2 e GPT-3 può gestire in modo efficiente le parole sconosciute, suddividendole in unità sub-parola o singoli caratteri.
- Utilizziamo un approccio a finestra scorrevole sui dati tokenizzati per generare coppie input-target per l'addestramento del modello LLM.
- I layer di embedding di PyTorch funzionano come un'operazione di ricerca, recuperando i vettori corrispondenti ai token-ID. I vettori di embedding risultanti forniscono rappresentazioni continue di token, il che è fondamentale per l'addestramento dei modelli di deep learning, come i modelli LLM.
- Mentre gli embedding dei token forniscono rappresentazioni vettoriali coerenti per ogni token, non hanno il senso della posizione del token in una sequenza. Per rimediare, esistono due tipi principali di embedding posizionali: assoluti e relativi. I

modelli GPT di OpenAI utilizzano embedding posizionali assoluti, che vengono aggiunti ai vettori di embedding dei token e poi ottimizzati durante l’addestramento del modello.

Capitolo 3

Programmazione dei meccanismi di attenzione

A questo punto, sapete come preparare il testo di input per l'addestramento dei modelli LLM suddividendo il testo in singoli token di parole e sub-parole, che possono essere codificati in rappresentazioni vettoriali, ovvero "embedding", per il modello LLM.

Ora, esamineremo una parte integrante dell'architettura LLM, i *meccanismi di attenzione*, illustrati nella Figura 3.1. Li esamineremo ampiamente in modo isolato e ci concentreremo su di essi a livello "meccanico". Quindi programmeremo le parti rimanenti del modello LLM che attuano il meccanismo di auto-attenzione, per vederlo in azione e per creare un modello in grado di generare testo.

In questo capitolo implementiamo il meccanismo di attenzione, un importante elemento costitutivo dei modelli LLM di tipo GPT.

Implementeremo quattro diverse varianti di meccanismi di attenzione, come mostra la Figura 3.2. Queste varianti si basano l'una sull'altra, e l'obiettivo è quello di arrivare a un'implementazione compatta ed efficiente dell'attenzione multi-head che potremo poi collegare all'architettura LLM che programmeremo nel prossimo capitolo.

Il problema della modellazione di sequenze lunghe

Prima di immergerci nel meccanismo di *auto-attenzione* su cui si basano i modelli LLM, consideriamo il problema con architetture pre-LLM che non includono meccanismi di attenzione. Supponiamo di voler sviluppare un modello di traduzione linguistica da una lingua a un'altra. Come mostra la Figura 3.3, non possiamo semplicemente tradurre un testo parola per parola, a causa delle diverse strutture grammaticali della lingua di origine e di destinazione.

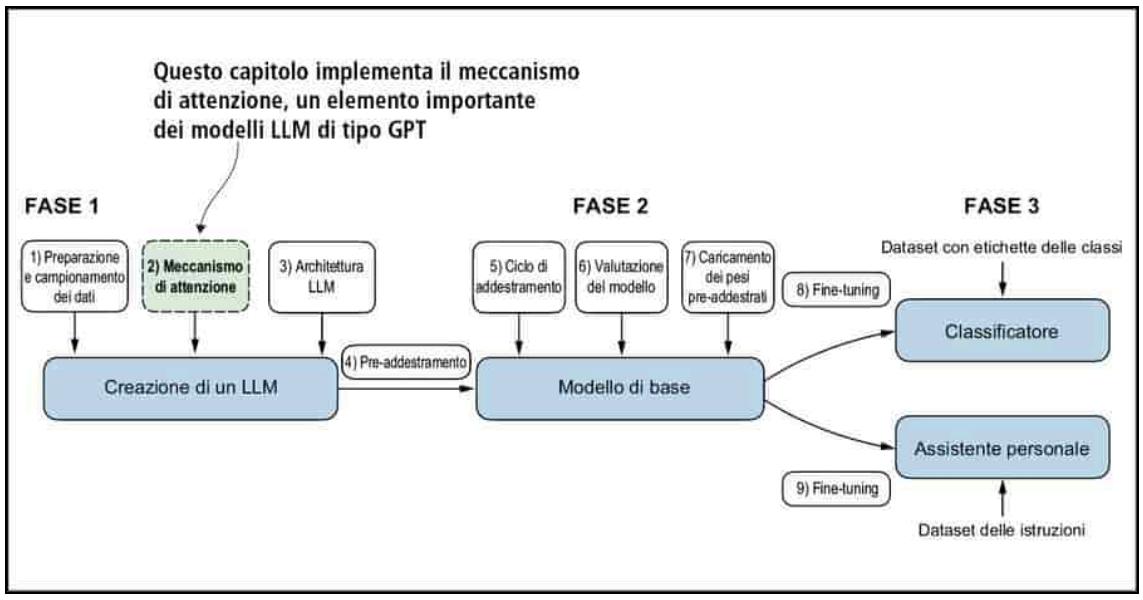


Figura 3.1 Le tre fasi principali della programmazione di un modello LLM. Questo capitolo si concentra sul Passo 2 della Fase 1: implementazione dei meccanismi di attenzione, che sono parte integrante dell'architettura LLM.

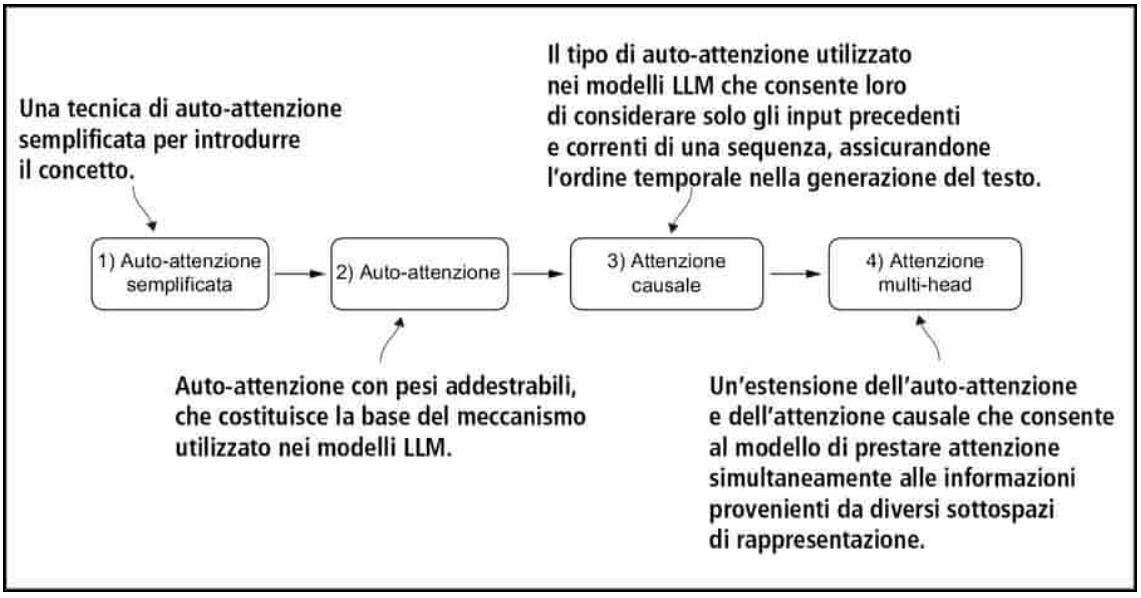


Figura 3.2 I diversi meccanismi di attenzione che programmeremo in questo capitolo, iniziando con una versione semplificata dell'auto-attenzione, prima di aggiungere i pesi addestrabili. Il meccanismo di attenzione causale aggiunge una maschera all'auto-attenzione, la quale consente al modello LLM di generare una parola alla volta. Infine, l'attenzione multi-head organizza il meccanismo di attenzione in più "teste", consentendo al modello di catturare in parallelo vari aspetti dei dati di input.

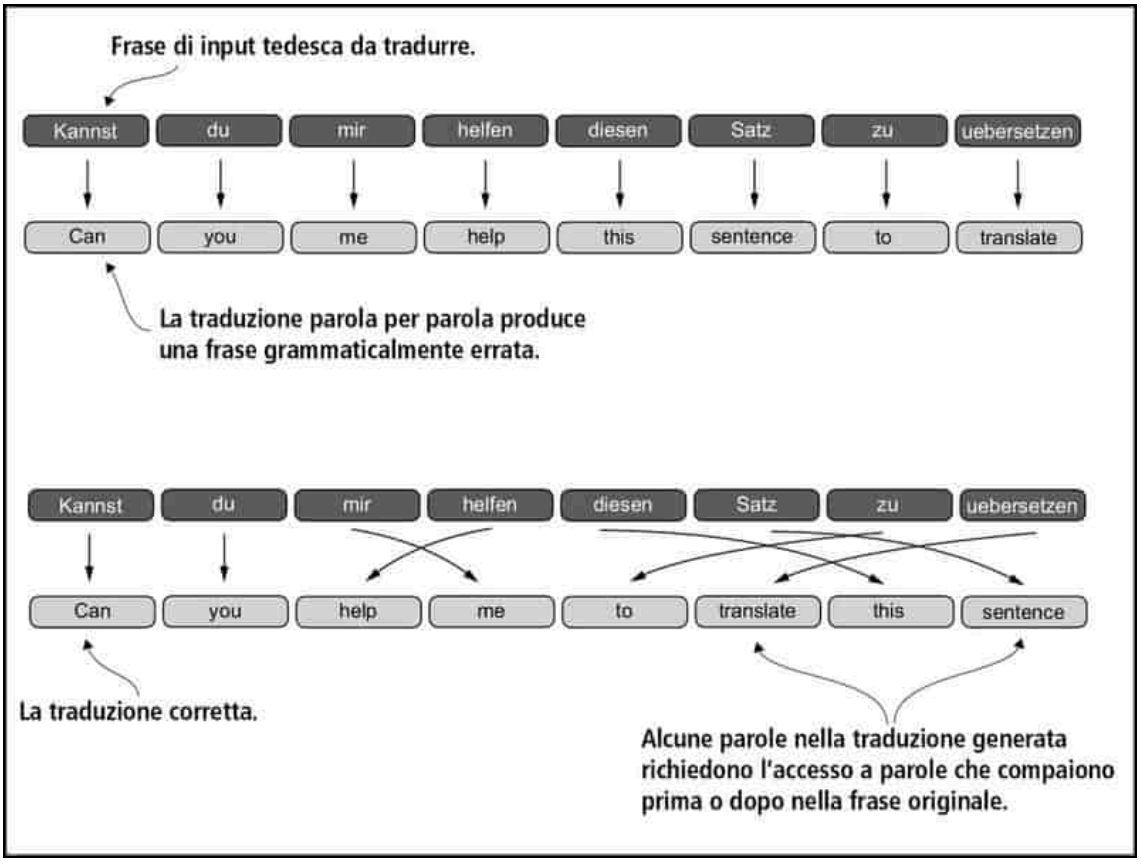


Figura 3.3 Nel tradurre un testo da una lingua a un'altra, come dal tedesco all'inglese, non è possibile tradurre semplicemente parola per parola. Al contrario, il processo di traduzione richiede una comprensione contestuale e un allineamento grammaticale.

Per risolvere questo problema, di solito si usa una rete neurale profonda con due sub-moduli, un *encoder* e un *decoder.* Il compito dell'encoder è quello di leggere ed elaborare l'intero testo; il decoder, poi, produce il testo tradotto.

Prima dell'avvento dei transformer, l'architettura encoder-decoder più utilizzata per la traduzione linguistica era costituita dalle *reti neurali ricorrenti* (RNN). In una rete neurale ricorrente, gli output dei passi precedenti vengono immessi come input del passo corrente, cosa che le rende adatte per dati sequenziali come il testo. Se non avete familiarità con le reti neurali ricorrenti, non preoccupatevi: non

avete bisogno di conoscere dettagliatamente il loro funzionamento per seguire questa discussione; la nostra attenzione, qui, è più sul concetto generale della configurazione encoder-decoder.

In una rete neurale ricorrente a encoder-decoder, il testo di input viene immesso nell'encoder, che lo elabora in sequenza. A ogni passo, l'encoder aggiorna il suo stato nascosto (i valori interni nei layer nascosti), cercando di catturare l'intero significato della frase in input nello stato nascosto finale, come mostra la Figura 3.4. Il decoder, poi, prende questo stato nascosto finale per iniziare a generare la frase tradotta, una parola alla volta. Inoltre, a ogni passo aggiorna il suo stato nascosto, che dovrebbe contenere il contesto necessario per la previsione della parola successiva.

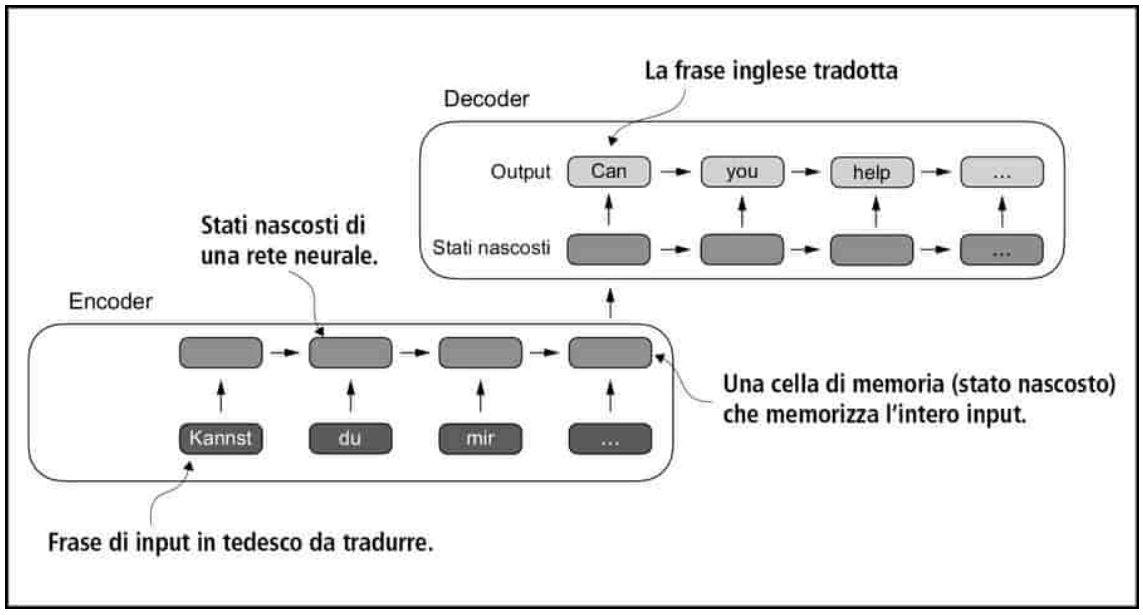


Figura 3.4 Prima dell'avvento dei modelli a transformer, le reti neurali ricorrenti a encoder-decoder erano una scelta molto utilizzata per la traduzione automatica. L'encoder prende come input una sequenza di token dalla lingua di origine, dove uno stato nascosto (un layer intermedio della rete neurale) dell'encoder codifica una rappresentazione compressa dell'intera sequenza di input. Quindi, il decoder usa il suo stato nascosto corrente per iniziare la traduzione, token per token.

Sebbene non abbiamo bisogno di conoscere il funzionamento interno di queste reti neurali ricorrenti a encoder-decoder, l'idea chiave è che la parte encoder elabora l'intero testo di input in uno stato nascosto (una cella di memoria). Il decoder, poi, impiega questo stato nascosto per produrre l'output. Potete paragonare questo stato nascosto a un vettore di embedding, un concetto che abbiamo discusso nel Capitolo 2.

Il grande limite delle reti neurali ricorrenti a encoder-decoder è che la rete non può accedere direttamente agli stati nascosti precedenti dell'encoder durante la fase decoder. Di conseguenza, si basa esclusivamente sullo stato nascosto corrente, che incapsula tutte le informazioni rilevanti. Ciò può portare a una perdita di contesto, specialmente in frasi complesse in cui le dipendenze potrebbero estendersi su lunghe distanze.

Fortunatamente, non è essenziale comprendere il funzionamento delle reti neurali ricorrenti per costruire un modello LLM. Basta ricordare che è proprio questo difetto delle reti neurali ricorrenti a encoder-decoder ad aver motivato la progettazione dei meccanismi di attenzione.

Cattura delle dipendenze nei dati con i meccanismi di attenzione

Sebbene le reti neurali ricorrenti siano in grado di tradurre brevi frasi, non funzionano bene per testi più lunghi, in quanto non hanno accesso alle parole precedenti nell'input. Una delle principali carenze di questo approccio è che la rete neurale ricorrente deve ricordare l'intero input, codificato in un singolo stato nascosto prima di passarlo al decoder (Figura 3.4).

Pertanto, nel 2014 i ricercatori hanno sviluppato il *meccanismo di attenzione Bahdanau* per le reti neurali ricorrenti (dal nome del primo autore dell'articolo in cui è trattata, per maggiori informazioni, vedi l'Appendice B), che modifica la rete neurale ricorrente a encoder-decoder in modo che in ogni passo il decoder possa accedere selettivamente a diverse parti della sequenza di input, come mostra la Figura 3.5.

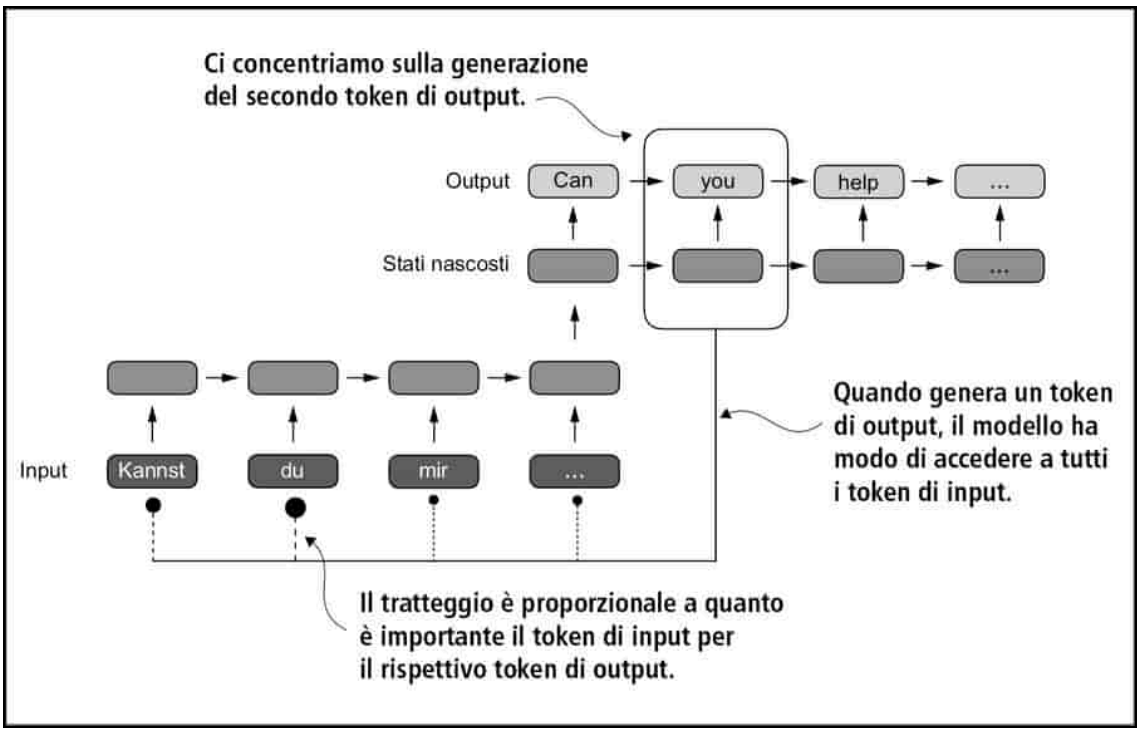


Figura 3.5 Utilizzando un meccanismo di attenzione, la parte decoder di generazione del testo della rete può accedere in modo selettivo a tutti i token di input. Ciò significa che alcuni token di input divengono più importanti di altri per generare un dato token di output. L'importanza è determinata dai pesi di attenzione, che calcoleremo in seguito. Notate che questa figura mostra l'idea generale alla base del meccanismo di attenzione e non raffigura l'implementazione esatta del meccanismo Bahdanau, che è un metodo dedicato alle reti neurali ricorrenti, le quali non rientrano negli ambiti di questo libro.

È interessante notare che solo tre anni dopo, i ricercatori hanno scoperto che le architetture a rete neurale ricorrente non sono

necessarie per costruire reti neurali profonde per l'elaborazione del linguaggio naturale, e hanno proposto l'architettura a *transformer* (discussa nel Capitolo 1) che include un meccanismo di auto-attenzione ispirato al meccanismo di attenzione di Bahdanau.

L'auto-attenzione è un meccanismo che consente a ogni posizione nella sequenza di input di considerare la rilevanza di (o di "fare attenzione a") tutte le altre posizioni della stessa sequenza nel calcolare la rappresentazione di una sequenza. L'auto-attenzione è un componente chiave dei modelli LLM di oggi, basati sull'architettura a transformer, come la serie GPT.

Questo capitolo si concentra sulla codifica e sulla comprensione di questo meccanismo di auto-attenzione utilizzato nei modelli di tipo GPT, come mostra la Figura 3.6. Nel prossimo capitolo, programmeremo le parti rimanenti del modello LLM.

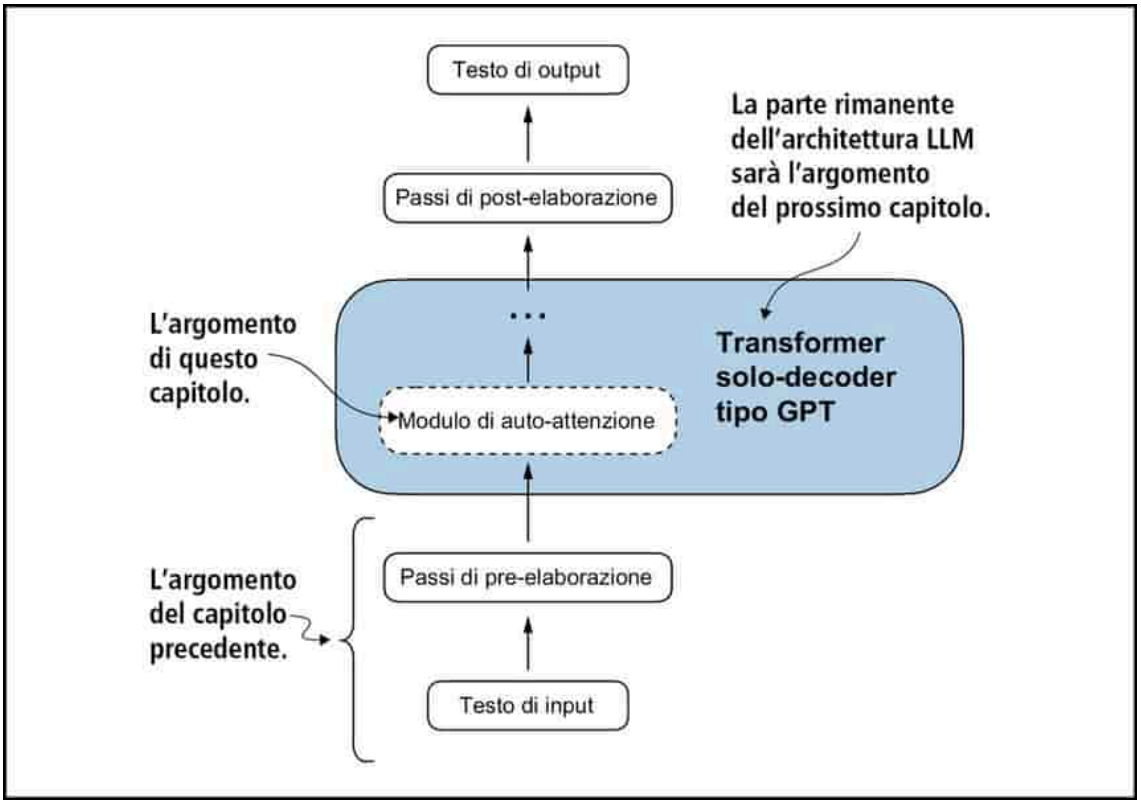


Figura 3.6 L'auto-attenzione è un meccanismo dei transformer utilizzato per generare rappresentazioni di input più efficienti, consentendo a ogni posizione di una sequenza di "interagire con" e di "pesare l'importanza di" tutte le altre posizioni all'interno della stessa sequenza. In questo capitolo, programmeremo da zero questo meccanismo di auto-attenzione, prima di programmare le parti rimanenti del modello LLM di tipo GPT, nel prossimo capitolo.

Attenzione alle diverse parti dell'input con l'auto-attenzione

Ora parleremo del funzionamento interno del meccanismo di auto-attenzione e impareremo a programmarlo da zero. L'auto-attenzione è la pietra angolare di ogni modello LLM basato sull'architettura a transformer. Questo argomento potrebbe richiedere molta

concentrazione e attenzione (il gioco di parole è voluto), ma una volta che ne avrete afferrato i fondamenti, avrete acquisito uno degli aspetti più difficili di questo libro e in generale dell'implementazione di modelli LLM.

Che cosa significa "auto", in "auto-attenzione"

Nell'auto-attenzione, quell'"auto" si riferisce alla capacità del meccanismo di calcolare i pesi di attenzione correlando diverse posizioni all'interno di una singola sequenza di input. Esso valuta e apprende le relazioni e le dipendenze tra le varie parti dell'input, come le parole in una frase o i pixel in un'immagine.

Al contrario, nei meccanismi di attenzione tradizionali, l'attenzione è rivolta alle relazioni tra elementi di due sequenze diverse, come nei modelli sequenza-sequenza in cui l'attenzione può essere tra una sequenza di input e una sequenza di output, come nell'esempio rappresentato nella Figura 3.5.

Poiché l'auto-attenzione può sembrare un argomento complesso, soprattutto la prima volta che se ne parla, inizieremo esaminandone una versione semplificata. Poi, implementeremo il meccanismo di auto-attenzione con pesi addestrabili utilizzato nei modelli LLM.

Un meccanismo di auto-attenzione semplificato, senza pesi addestrabili

Cominciamo implementando una variante semplificata dell'auto-attenzione, senza pesi addestrabili, come vediamo nella Figura 3.7. Qui l'obiettivo è solo quello di illustrare alcuni concetti chiave dell'auto-attenzione, prima di aggiungere i pesi addestrabili.

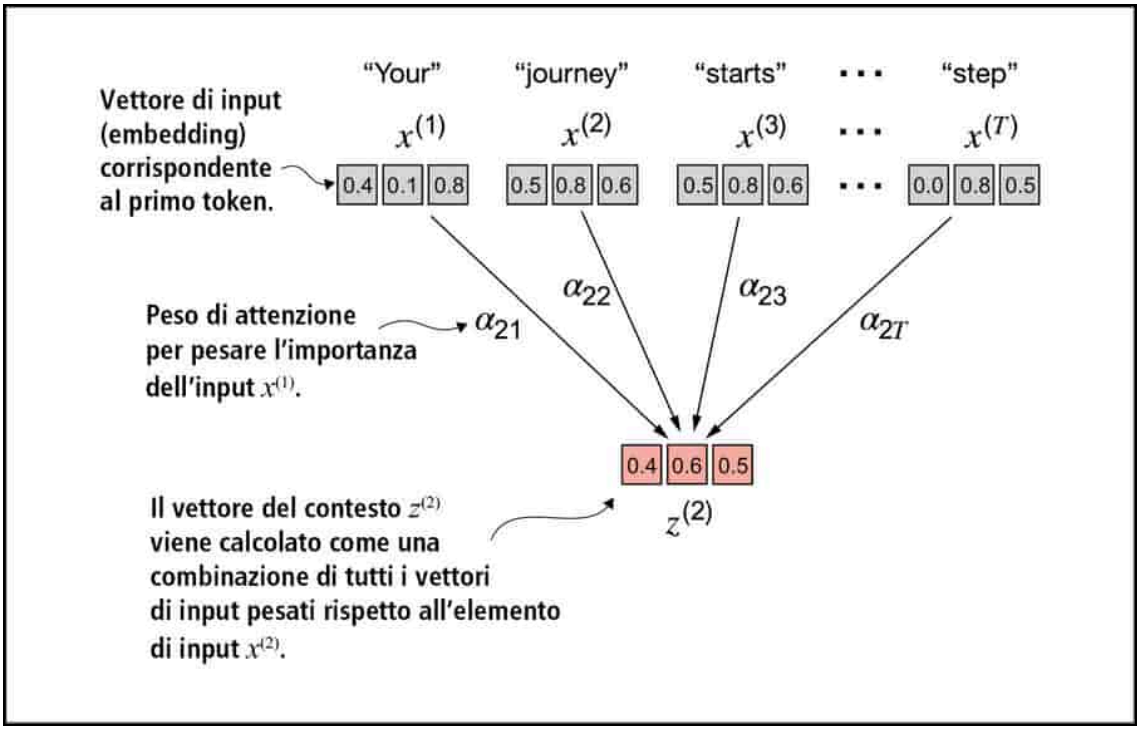


Figura 3.7 L'obiettivo dell'auto-attenzione è quello di calcolare per ogni elemento di input un vettore del contesto che combina le informazioni da tutti gli altri elementi di input. In questo esempio, calcoliamo il vettore del contesto $z^{(2)}$. L'importanza o il contributo di ogni elemento di input per il calcolo di $z^{(2)}$ è determinato dai pesi di attenzione da α_{21} ad α_{2T}. Nel calcolare $z^{(2)}$, i pesi di attenzione vengono calcolati rispetto all'elemento di input $x^{(2)}$ e a tutti gli altri input.

La Figura 3.7 mostra una sequenza di input, indicata come *x*, composta da *T* elementi da $x^{(1)}$ a $x^{(T)}$. In genere questa sequenza rappresenta un testo, per esempio una frase, che è già stata trasformata in embedding dei token.

Per esempio, considerate il testo di input (inglese): "Your journey starts with one step". In questo caso, ogni elemento della sequenza, come $x^{(1)}$, corrisponde a un vettore di embedding *d*-dimensionale che rappresenta un determinato token, come "Your". La Figura 3.7 mostra questi vettori di input come embedding tridimensionali.

Nell’auto-attenzione, il nostro obiettivo è quello di calcolare i vettori di contesto $z^{(i)}$ per ogni elemento $x^{(i)}$ nella sequenza di input. Un *vettore del contesto* può essere interpretato come un vettore di embedding arricchito.

Per illustrare questo concetto, concentriamoci sul vettore di embedding del secondo elemento di input, $x^{(2)}$ (che corrisponde al token “journey”), e sul suo vettore del contesto, $z^{(2)}$, mostrato nella parte inferiore della Figura 3.7. Questo vettore del contesto migliorato, $z^{(2)}$, è un embedding che contiene informazioni su $x^{(2)}$ e su tutti gli altri elementi di input, da $x^{(1)}$ a $x^{(T)}$.

I vettori di contesto svolgono un ruolo cruciale nell’auto-attenzione. Il loro scopo è quello di creare rappresentazioni arricchite di ogni elemento di una sequenza di input (come una frase) incorporando informazioni da tutti gli altri elementi presenti nella sequenza (Figura 3.7). Ciò è essenziale nei modelli LLM, che devono comprendere tutte le relazioni e la pertinenza di tutte le parole di una frase. In seguito, aggiungeremo i pesi addestrabili, che aiutano un modello LLM a imparare a costruire questi vettori di contesto in modo che siano rilevanti per il modello LLM, così da consentirgli di generare il token successivo. Ma prima, implementiamo un meccanismo di auto-attenzione semplificato per poi calcolare questi pesi e il vettore del contesto risultante, un passo alla volta.

Considerate la seguente frase di input, che è già stata incorporata in vettori tridimensionali (vedi il Capitolo 2). Ho scelto un embedding di piccole dimensioni per motivi di spazio nella pagina:

```
import torch
inputs = torch.tensor(
  [[0.43, 0.15, 0.89], # Your     (x^1)
   [0.55, 0.87, 0.66], # journey  (x^2)
   [0.57, 0.85, 0.64], # starts   (x^3)
   [0.22, 0.58, 0.33], # with     (x^4)
   [0.77, 0.25, 0.10], # one      (x^5)
   [0.05, 0.80, 0.55]] # step     (x^6)
)
```

Il primo passo per implementare l'auto-attenzione è quello di calcolare i valori intermedi **ω**, i *punteggi di attenzione*, come mostra la Figura 3.8. A causa dei limiti di spazio, la figura mostra i valori del tensore `inputs` precedente in una versione troncata; per esempio, `0.87` è troncato in `0.8`. In questa versione troncata, gli embedding delle parole "journey" e "starts" possono apparire simili, ma solo per caso.

La Figura 3.8 mostra come calcoliamo i punteggi di attenzione intermedi tra il token della query e ogni token di input. Determiniamo questi punteggi calcolando il prodotto scalare del token della query, $x^{(2)}$, con ogni altro token di input:

```
query = inputs[1] (1)
attn_scores_2 = torch.empty(inputs.shape[0])
for i, x_i in enumerate(inputs):
    attn_scores_2[i] = torch.dot(x_i, query)
print(attn_scores_2)
```

(1) La query riguarda il secondo token di input.

I punteggi di attenzione calcolati sono i seguenti:

```
tensor([0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865])
```

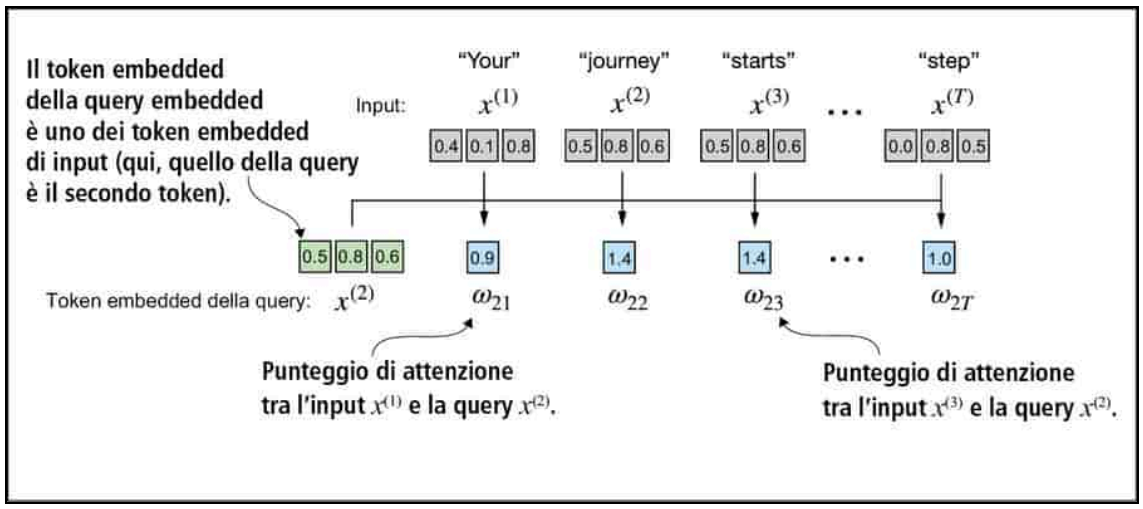


Figura 3.8 L'obiettivo generale è quello di illustrare il calcolo del vettore del contesto $z^{(2)}$ utilizzando per la query il secondo elemento di input, $x^{(2)}$. Questa figura mostra il primo passo intermedio, calcolando i punteggi di attenzione ω tra il token della query, $x^{(2)}$, e tutti

gli altri elementi di input come un prodotto scalare. Notate che i numeri sono troncati a una cifra dopo il punto decimale, ma solo per ridurne l'ingombro.

Il prodotto scalare

Il prodotto scalare è essenzialmente un modo conciso per moltiplicare due vettori, elemento per elemento, e poi sommare i prodotti, come segue:

```
res = 0.
for idx, element in enumerate(inputs[0]):
    res += inputs[0][idx] * query[idx]
print(res)
print(torch.dot(inputs[0], query))
```

L'output conferma che la somma della moltiplicazione elemento per elemento fornisce gli stessi risultati del prodotto scalare:

```
tensor(0.9544)
tensor(0.9544)
```

Oltre a essere uno strumento matematico che combina due vettori per produrre un valore scalare, il prodotto scalare è una misura di similitudine, perché quantifica quanto siano allineati due vettori: un prodotto scalare più elevato indica un grado maggiore di allineamento o di similitudine tra i vettori. Nel contesto dei meccanismi di auto-attenzione, il prodotto scalare determina la misura in cui ogni elemento di una sequenza "presta attenzione a" qualsiasi altro elemento: più elevato è il prodotto scalare, più elevato è il punteggio di similitudine e anche l'attenzione tra due elementi.

Nel passo successivo, come mostra la Figura 3.9, normalizziamo ciascuno dei punteggi di attenzione calcolati in precedenza. L'obiettivo principale della normalizzazione è quello di ottenere pesi di attenzione a somma 1. Questa normalizzazione è una convenzione utile per l'interpretazione e il mantenimento della stabilità dell'addestramento in un modello LLM. Ecco un metodo semplice per ottenere questo passo di normalizzazione:

```
attn_weights_2_tmp = attn_scores_2 / attn_scores_2.sum()
print("Attention weights:", attn_weights_2_tmp)
print("Sum:", attn_weights_2_tmp.sum())
```

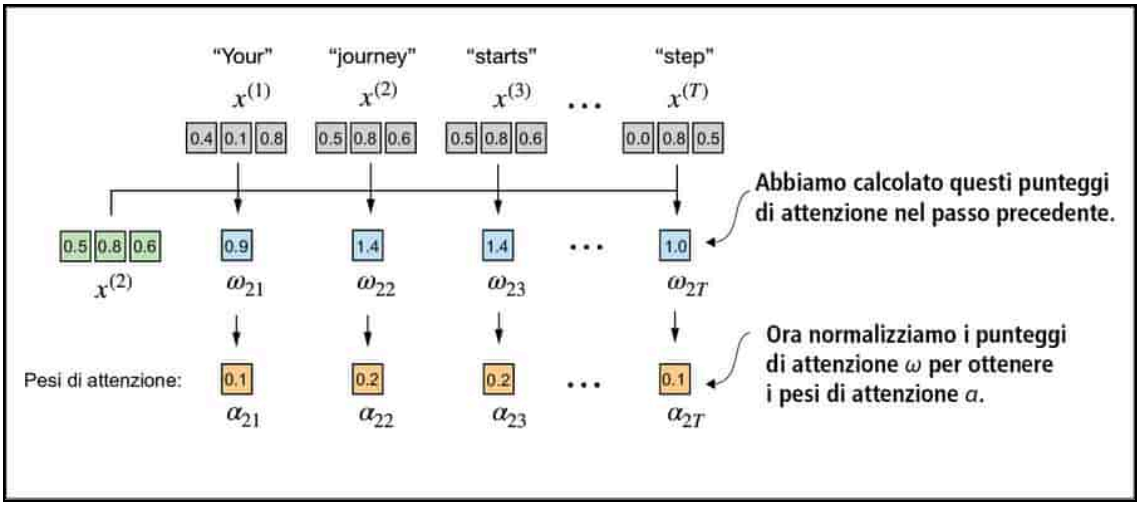


Figura 3.9 Dopo aver calcolato i punteggi di attenzione da ω_{21} a ω_{2T} rispetto alla query di input $x^{(2)}$, il passo successivo consiste nell'ottenere i pesi di attenzione da α_{21} ad α_{2T} normalizzando i punteggi di attenzione.

Come mostra l'output, ora la somma dei pesi di attenzione è `1`:

```
Attention weights: tensor([0.1455, 0.2278, 0.2249, 0.1285, 0.1077, 0.1656])
Sum: tensor(1.0000)
```

All'atto pratico, è pratica comune e consigliabile usare la funzione `softmax()` per la normalizzazione. Questo approccio è migliore nella gestione dei valori estremi e offre proprietà dei gradienti più favorevoli durante l'addestramento. Quella che segue è un'implementazione semplice della funzione `softmax()` per la normalizzazione dei punteggi di attenzione:

```
def softmax_naive(x):
    return torch.exp(x) / torch.exp(x).sum(dim=0)

attn_weights_2_naive = softmax_naive(attn_scores_2)
print("Attention weights:", attn_weights_2_naive)
print("Sum:", attn_weights_2_naive.sum())
```

Come mostra l'output, anche la funzione `softmax()` soddisfa l'obiettivo e normalizza i pesi di attenzione in modo che la loro somma sia `1`:

```
Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
```

```
Sum: tensor(1.)
```

Inoltre, la funzione `softmax()` assicura che i pesi di attenzione siano sempre positivi. Ciò rende l'output interpretabile come una probabilità o un'importanza relativa, dove pesi più elevati indicano una maggiore importanza.

Notate che questa implementazione semplificata (`softmax_naive()`) può incontrare problemi di instabilità numerica, come overflow e underflow, quando si ha a che fare con valori di input molto grandi o piccoli. Pertanto, nella pratica è consigliabile utilizzare l'implementazione PyTorch di `softmax()`, che è stata ampiamente ottimizzata per le prestazioni:

```
attn_weights_2 = torch.softmax(attn_scores_2, dim=0)
print("Attention weights:", attn_weights_2)
print("Sum:", attn_weights_2.sum())
```

In questo caso, la funzione produce gli stessi risultati della nostra `softmax_naive()`:

```
Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
Sum: tensor(1.)
```

Ora che abbiamo calcolato i pesi di attenzione normalizzati, siamo pronti per il passo finale, rappresentato nella Figura 3.10: calcolare il vettore del contesto $z^{(2)}$ moltiplicando i token embedded di input, $x^{(i)}$, per i pesi di attenzione corrispondenti, e poi sommando i vettori risultanti. Quindi, il vettore del contesto $z^{(2)}$ è la somma ponderata di tutti i vettori di input, ottenuta moltiplicando ogni vettore di input per il corrispondente peso di attenzione:

```
query = inputs[1] (1)
context_vec_2 = torch.zeros(query.shape)
for i,x_i in enumerate(inputs):
context_vec_2 += attn_weights_2[i]*x_i
print(context_vec_2)
```

(1) Il secondo token di input è quello della query.

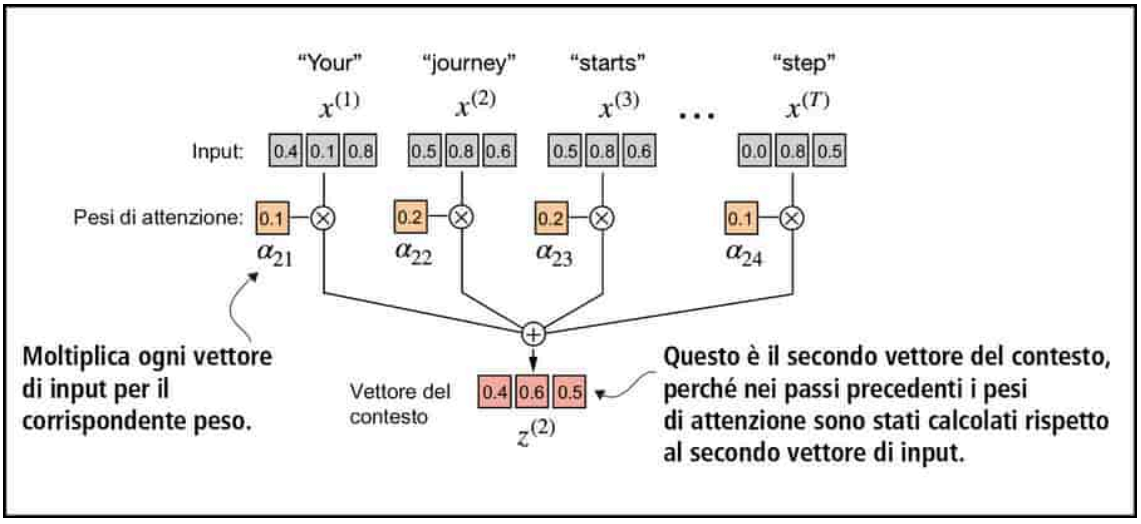


Figura 3.10 Il passo finale, dopo aver calcolato e normalizzato i punteggi di attenzione per ottenere i pesi di attenzione per la query $x^{(2)}$, è quello di calcolare il vettore del contesto $z^{(2)}$. Questo vettore è una combinazione di tutti i vettori di input da $x^{(1)}$ a $x^{(T)}$ pesati in base ai pesi di attenzione.

I risultati di questo calcolo sono i seguenti:

```
tensor([0.4419, 0.6515, 0.5683])
```

Ora, generalizzeremo questa procedura per il calcolo dei vettori di contesto in modo da calcolare tutti i vettori di contesto, simultaneamente.

Calcolo dei pesi di attenzione per tutti i token di input

Finora abbiamo calcolato i pesi di attenzione e il vettore del contesto per l'input 2, come mostra la riga evidenziata nella Figura 3.11. Ora estendiamo questo calcolo per calcolare i pesi di attenzione e i vettori di contesto per tutti gli input.

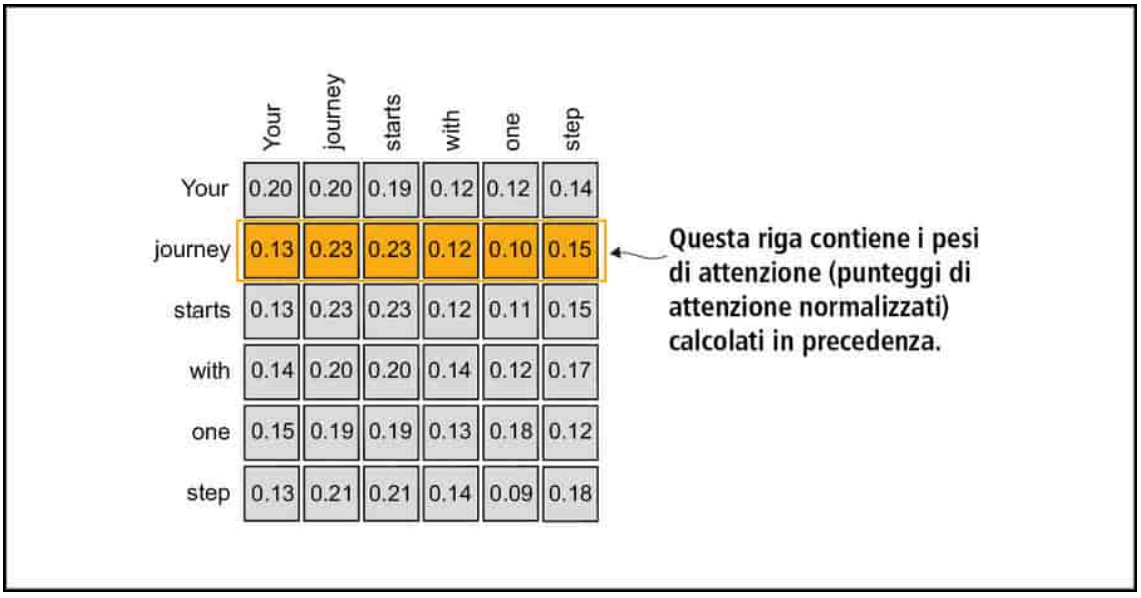


Figura 3.11 La riga evidenziata mostra i pesi di attenzione avendo come query il secondo elemento di input. Ora generalizziamo il calcolo per ottenere tutti gli altri pesi di attenzione. Notate che i numeri in questa figura sono troncati a due cifre dopo il punto decimale solo per ridurre l'ingombro. I valori in ogni riga devono avere somma 1.0, il 100 percento.

Seguiamo gli stessi tre passi di prima (vedi Figura 3.12), tranne per il fatto che apportiamo alcune modifiche al codice per calcolare tutti i vettori di contesto, non solo il secondo, $z^{(2)}$:

```
attn_scores = torch.empty(6, 6)
for i, x_i in enumerate(inputs):
    for j, x_j in enumerate(inputs):
        attn_scores[i, j] = torch.dot(x_i, x_j)
print(attn_scores)
```

I punteggi di attenzione risultanti sono i seguenti:

```
tensor([[0.9995, 0.9544, 0.9422, 0.4753, 0.4576, 0.6310],
        [0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865],
        [0.9422, 1.4754, 1.4570, 0.8296, 0.7154, 1.0605],
        [0.4753, 0.8434, 0.8296, 0.4937, 0.3474, 0.6565],
        [0.4576, 0.7070, 0.7154, 0.3474, 0.6654, 0.2935],
        [0.6310, 1.0865, 1.0605, 0.6565, 0.2935, 0.9450]]
)
```

Figura 3.12 Nel Passo 1, aggiungiamo un ulteriore ciclo for per calcolare i prodotti scalari per tutte le coppie di input.

Ogni elemento del tensore rappresenta un punteggio di attenzione tra ogni coppia di input, come abbiamo visto nella Figura 3.11. Notate che i valori nella figura sono normalizzati, motivo per cui differiscono dai punteggi di attenzione non normalizzati del tensore precedente. Ci occuperemo della normalizzazione in un secondo momento.

Quando abbiamo calcolato il precedente tensore dei punteggi di attenzione, abbiamo utilizzato i cicli `for` di Python. Tuttavia, i cicli `for` sono lenti e possiamo ottenere gli stessi risultati utilizzando la moltiplicazione di matrici:

```
attn_scores = inputs @ inputs.T
print(attn_scores)
```

Possiamo confermare anche visivamente che i risultati sono gli stessi di prima:

```
tensor([[0.9995, 0.9544, 0.9422, 0.4753, 0.4576, 0.6310],
        [0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865],
        [0.9422, 1.4754, 1.4570, 0.8296, 0.7154, 1.0605],
        [0.4753, 0.8434, 0.8296, 0.4937, 0.3474, 0.6565],
        [0.4576, 0.7070, 0.7154, 0.3474, 0.6654, 0.2935],
        [0.6310, 1.0865, 1.0605, 0.6565, 0.2935, 0.9450]]
)
```

Nel Passo 2 della Figura 3.12, normalizziamo ogni riga in modo che la somma dei valori in ogni riga sia `1`:

```
attn_weights = torch.softmax(attn_scores, dim=-1)
print(attn_weights)
```

Ciò restituisce il seguente tensore dei pesi di attenzione, che corrisponde ai valori mostrati nella Figura 3.10:

```
tensor([[0.2098, 0.2006, 0.1981, 0.1242, 0.1220, 0.1452],
        [0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581],
        [0.1390, 0.2369, 0.2326, 0.1242, 0.1108, 0.1565],
        [0.1435, 0.2074, 0.2046, 0.1462, 0.1263, 0.1720],
        [0.1526, 0.1958, 0.1975, 0.1367, 0.1879, 0.1295],
        [0.1385, 0.2184, 0.2128, 0.1420, 0.0988, 0.1896]]
)
```

Nel contesto dell'utilizzo di PyTorch, il parametro `dim` in funzioni come `torch.softmax()` specifica la dimensione del tensore di input in base alla quale verrà calcolata la funzione. Impostando `dim=-1`, stiamo chiedendo alla funzione `softmax()` di applicare la normalizzazione in base all'ultima dimensione del tensore `attn_scores`. Se `attn_scores` è un tensore bidimensionale (per esempio, con una forma a [righe, colonne]), verrà normalizzato per colonne, in modo che i valori in ogni riga (colonna per colonna) abbiano somma `1`.

Possiamo verificare che la somma delle righe sia effettivamente 1:

```
row_2_sum = sum([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
print("Row 2 sum:", row_2_sum)
print("All row sums:", attn_weights.sum(dim=-1))
```

Il risultato è il seguente:

```
Row 2 sum: 1.0
All row sums: tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000])
```

Nel terzo e ultimo passo della Figura 3.12, utilizziamo questi pesi di attenzione per calcolare tutti i vettori di contesto tramite la moltiplicazione di matrici:

```
all_context_vecs = attn_weights @ inputs
print(all_context_vecs)
```

Nel tensore di output risultante, ogni riga contiene un vettore del contesto tridimensionale:

```
tensor([[0.4421, 0.5931, 0.5790],
        [0.4419, 0.6515, 0.5683],
        [0.4431, 0.6496, 0.5671],
        [0.4304, 0.6298, 0.5510],
        [0.4671, 0.5910, 0.5266],
        [0.4177, 0.6503, 0.5645]]
)
```

Possiamo verificare che il codice sia corretto confrontando la seconda riga col vettore del contesto $z^{(2)}$ che abbiamo calcolato in precedenza:

```
print("Previous 2nd context vector:", context_vec_2)
```

Sulla base del risultato, possiamo vedere che `context_vec_2`, calcolato in precedenza, corrisponde esattamente alla seconda riga di questo tensore:

```
Previous 2nd context vector: tensor([0.4419, 0.6515, 0.5683])
```

Questo conclude la guida alla programmazione di un semplice meccanismo di auto-attenzione. Ora, aggiungeremo i pesi addestrabili, per consentire al modello LLM di apprendere dai dati e di migliorare le prestazioni su attività specifiche.

Implementazione dell'auto-attenzione con pesi addestrabili

Il nostro prossimo passo consisterà nell'implementazione del meccanismo di auto-attenzione utilizzato nell'architettura a transformer originale, nei modelli GPT e nella maggior parte degli altri modelli LLM più diffusi. Questo meccanismo di auto-attenzione è chiamato *attenzione a prodotto scalare*. La Figura 3.13 mostra come questo meccanismo di auto-attenzione si colloca nel contesto più ampio dell'implementazione di un modello LLM.

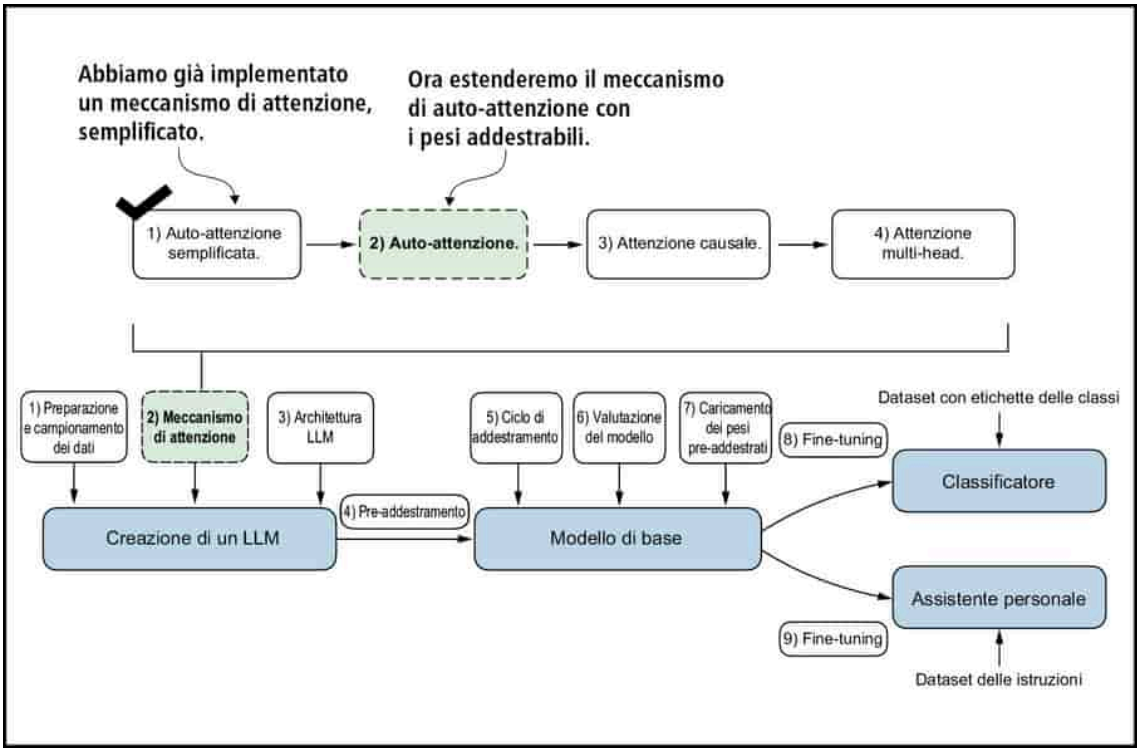


Figura 3.13 In precedenza, abbiamo programmato un meccanismo di attenzione semplificato per comprendere il funzionamento di base dei meccanismi di attenzione. Ora, aggiungiamo a questo meccanismo di attenzione i pesi addestrabili. In seguito, estenderemo questo meccanismo di auto-attenzione aggiungendo una maschera causale e più “teste”.

Come illustra la Figura 3.13, il meccanismo di auto-attenzione con pesi addestrabili si basa sui concetti precedenti: vogliamo calcolare i vettori di contesto come somme pesate dei vettori di input di un determinato elemento di input. Come vedrete, ci sono solo lievi differenze rispetto al meccanismo di auto-attenzione di base che abbiamo programmato in precedenza.

La differenza più rilevante è l’introduzione delle matrici dei pesi, che vengono aggiornate durante l’addestramento del modello. Queste matrici dei pesi addestrabili sono fondamentali affinché il modello (in particolare, il modulo di attenzione all’interno del modello) possa

imparare a produrre vettori di contesto “buoni”. Addestreremo poi il modello LLM nel Capitolo 5.

Affronteremo questo meccanismo di auto-attenzione in due paragrafi. Innanzitutto, lo programmeremo passo dopo passo, come prima. In secondo luogo, organizzeremo il codice in una classe Python compatta, che può essere importata nell’architettura LLM.

Calcolo dei pesi di attenzione passo dopo passo

Implementeremo il meccanismo di auto-attenzione passo dopo passo introducendo le tre matrici dei pesi addestrabili W_q, W_k e W_v. Queste tre matrici vengono utilizzate per proiettare i token embedded di input, $x^{(i)}$, rispettivamente nei vettori delle query, delle chiavi e dei valori, come mostra la Figura 3.14.

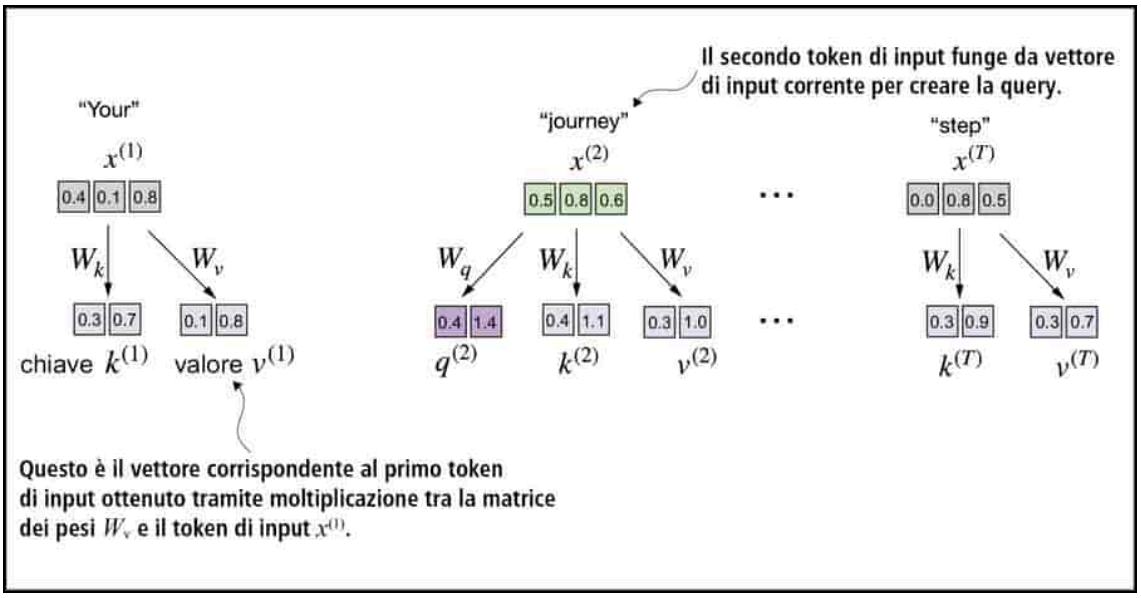


Figura 3.14 Nel primo passo del meccanismo di auto-attenzione con matrici dei pesi addestrabili, calcoliamo i vettori delle query (q), delle chiavi (k) e dei valori (v) per gli elementi di input *x*. In modo analogo a quanto detto nei paragrafi precedenti, designiamo il secondo input, $x^{(2)}$, come input della query. Il vettore delle query $q^{(2)}$ è ottenuto tramite

una moltiplicazione di matrici tra l'input $x^{(2)}$ e la matrice dei pesi W_q. Analogamente, otteniamo i vettori delle chiavi e dei valori tramite una moltiplicazione di matrici che coinvolge le matrici dei pesi W_k e W_v.

In precedenza, abbiamo definito come query il secondo elemento di input $x^{(2)}$ quando abbiamo calcolato i pesi di attenzione semplificati per calcolare il vettore del contesto $z^{(2)}$. Ora lo generalizziamo per calcolare tutti i vettori di contesto $z^{(1)}$ … $z^{(T)}$ per la frase di input di sei parole "Your journey starts with one step".

Allo stesso modo, iniziamo calcolando, a scopo illustrativo, un solo vettore del contesto, $z^{(2)}$. Quindi modificheremo il codice per calcolare tutti i vettori di contesto.

Cominciamo definendo alcune variabili:

```
x_2 = inputs[1] (1)
d_in = inputs.shape[1] (2)
d_out = 2 (3)
```

(1) Il secondo elemento di input.

(2) La dimensione dell'embedding di input, `d=3`.

(3) La dimensione dell'embedding di output, `d_out=2`.

Notate che nei modelli di tipo GPT, le dimensioni di input e output sono solitamente le stesse, ma per seguire meglio il calcolo, qui utilizzeremo dimensioni di input (`d_in=3`) e di output (`d_out=2`) differenti.

Ora, inizializziamo le tre matrici dei pesi W_q, W_k e W_v mostrate nella Figura 3.14:

```
torch.manual_seed(123)
W_query = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
W_key = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
W_value = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
```

Impostiamo `requires_grad=False` solo per ridurre l'ingombro negli output, ma se dovessimo usare le matrici dei pesi per l'addestramento del modello, imposteremmo `requires_grad=True` per aggiornare queste matrici durante l'addestramento del modello.

Ora, calcoliamo i vettori delle query, delle chiavi e dei valori:

```
query_2 = x_2 @ W_query
key_2 = x_2 @ W_key
value_2 = x_2 @ W_value
print(query_2)
```

L'output per la query genera un vettore bidimensionale, poiché abbiamo impostato a 2 il numero di colonne della matrice dei pesi corrispondente, tramite `d_out`:

```
tensor([0.4306, 1.4551])
```

Parametri dei pesi vs pesi di attenzione

Nelle matrici dei pesi *W*, il termine "pesi" è un'abbreviazione di "parametri dei pesi", i valori di una rete neurale che vengono ottimizzati durante l'addestramento. È importante non fare confusione con i pesi di attenzione. Come abbiamo già visto, i pesi di attenzione determinano la misura in cui un vettore del contesto dipende dalle diverse parti dell'input (ovvero, in che misura la rete si concentra sulle diverse parti dell'input).

In sintesi, i parametri dei pesi sono i coefficienti fondamentali, appresi, che definiscono le connessioni della rete, mentre i pesi di attenzione sono valori dinamici e specifici del contesto.

Anche se il nostro obiettivo temporaneo è solo quello di calcolare un vettore del contesto, $z^{(2)}$, abbiamo ancora bisogno dei vettori delle chiavi e dei valori per tutti gli elementi di input, poiché sono coinvolti nel calcolo dei pesi di attenzione rispetto alla query $q^{(2)}$ (vedi Figura 3.14).

Possiamo ottenere tutte le chiavi e i valori tramite la moltiplicazione di matrici:

```
keys = inputs @ W_key
values = inputs @ W_value
print("keys.shape:", keys.shape)
print("values.shape:", values.shape)
```

Come potete vedere dagli output, abbiamo proiettato con successo i sei token di input da uno spazio di embedding tridimensionale a uno bidimensionale:

```
keys.shape: torch.Size([6, 2])
```

```
values.shape: torch.Size([6, 2])
```

Il secondo passo consiste nel calcolare i punteggi di attenzione, come mostra la Figura 3.15.

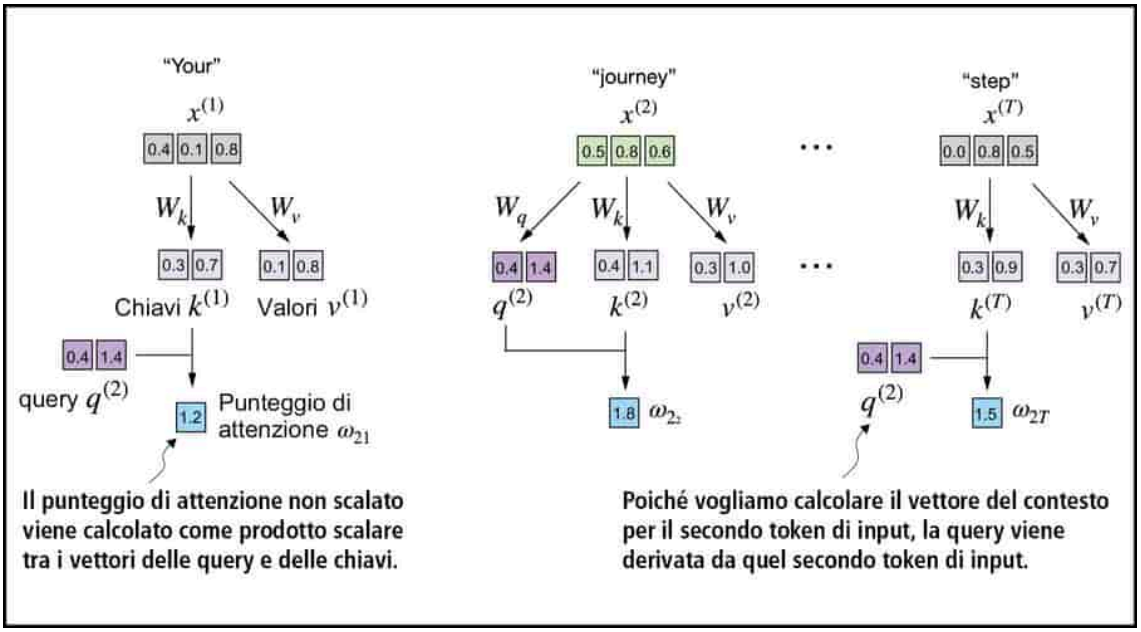


Figura 3.15 Il calcolo dei punteggi di attenzione prevede un prodotto scalare simile a quello che abbiamo utilizzato nel meccanismo di auto-attenzione semplificato. L'aspetto nuovo, qui, è che non stiamo calcolando il prodotto scalare direttamente tra gli elementi di input, ma utilizzando la query e la chiave ottenute trasformando gli input tramite le rispettive matrici dei pesi.

Per prima cosa, calcoliamo il punteggio di attenzione ω_{22}:

```
keys_2 = keys[1] (1)
attn_score_22 = query_2.dot(keys_2)
print(attn_score_22)
```

(1) Ricordatevi che in Python gli indici partono da 0.

Il risultato per il punteggio di attenzione non normalizzato è il seguente:

```
tensor(1.8524)
```

Ancora una volta, possiamo generalizzare questo calcolo a tutti i punteggi di attenzione tramite la moltiplicazione di matrici:

```
attn_scores_2 = query_2 @ keys.T (1)
print(attn_scores_2)
```

(1) Tutti i punteggi di attenzione per la query specificata.

Come potete vedere, per un rapido check, il secondo elemento dell'output corrisponde all'`attn_score_22` che abbiamo calcolato in precedenza:

```
tensor([1.2705, 1.8524, 1.8111, 1.0795, 0.5577, 1.5440])
```

Ora, vogliamo passare dai punteggi di attenzione ai pesi di attenzione, come mostra la Figura 3.16. Calcoliamo i pesi di attenzione riducendo in scala i punteggi di attenzione e utilizzando la funzione `softmax()`. Tuttavia, ora per scalarli li dividiamo per la radice quadrata delle dimensioni dell'embedding delle chiavi (calcolare la radice quadrata equivale matematicamente a elevare alla potenza `0.5`):

```
d_k = keys.shape[-1]
attn_weights_2 = torch.softmax(attn_scores_2 / d_k**0.5, dim=-1)
print(attn_weights_2)
```

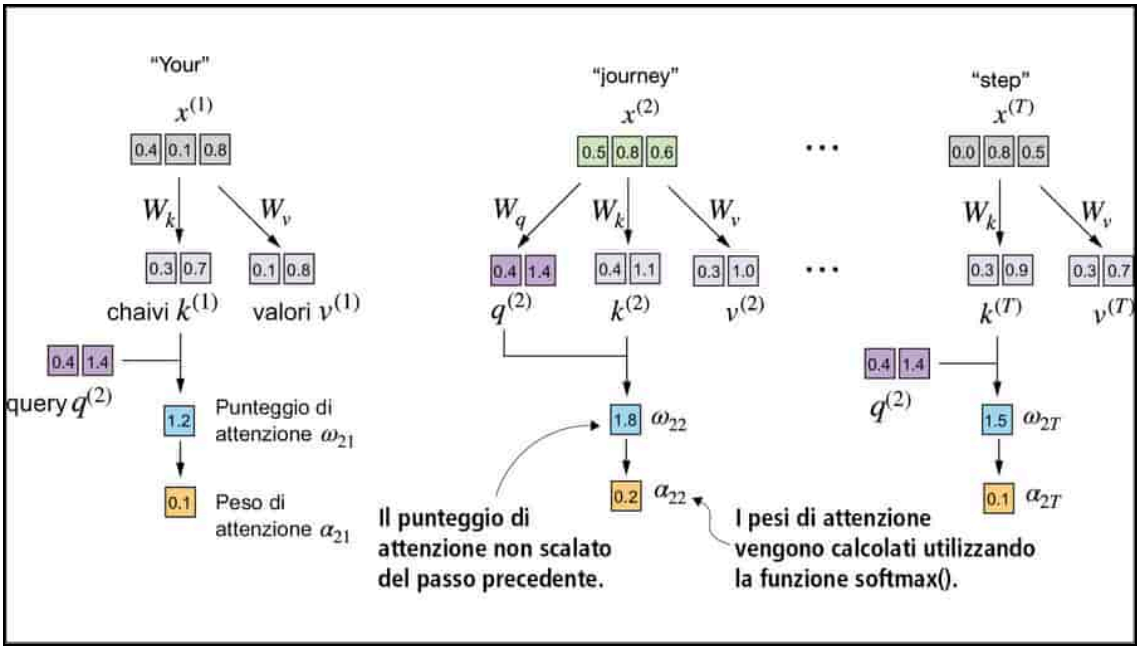


Figura 3.16 Dopo aver calcolato i punteggi di attenzione ω, il passo successivo consiste nel normalizzare tali punteggi utilizzando la funzione softmax() per ottenere i pesi di

attenzione α.

I pesi di attenzione risultanti sono i seguenti:

```
tensor([0.1500, 0.2264, 0.2199, 0.1311, 0.0906, 0.1820])
```

La logica alla base dell'uso dell'attenzione a prodotto scalato

Lo scopo della normalizzazione tramite le dimensioni dell'embedding è quello di migliorare le prestazioni dell'addestramento evitando gradienti troppo piccoli. Per esempio, aumentando le dimensioni dell'embedding, in genere maggiori di 1.000 per modelli LLM di tipo GPT, grandi prodotti scalari possono dare luogo a gradienti molto piccoli durante la retropropagazione, a causa della funzione `softmax()` che viene applicata. Man mano che i prodotti scalari aumentano, la funzione `softmax()` si comporta più come una funzione a gradini, con conseguenti gradienti prossimi allo zero. Questi piccoli gradienti possono rallentare drasticamente l'apprendimento o causarne la stagnazione.

La scalatura tramite radice quadrata delle dimensioni dell'embedding è il motivo per cui questo meccanismo di auto-attenzione è anche chiamato attenzione a prodotto scalato (*scaled-dot product attention*).

Ora, il passo finale consiste nel calcolare i vettori di contesto, come mostra la Figura 3.17.

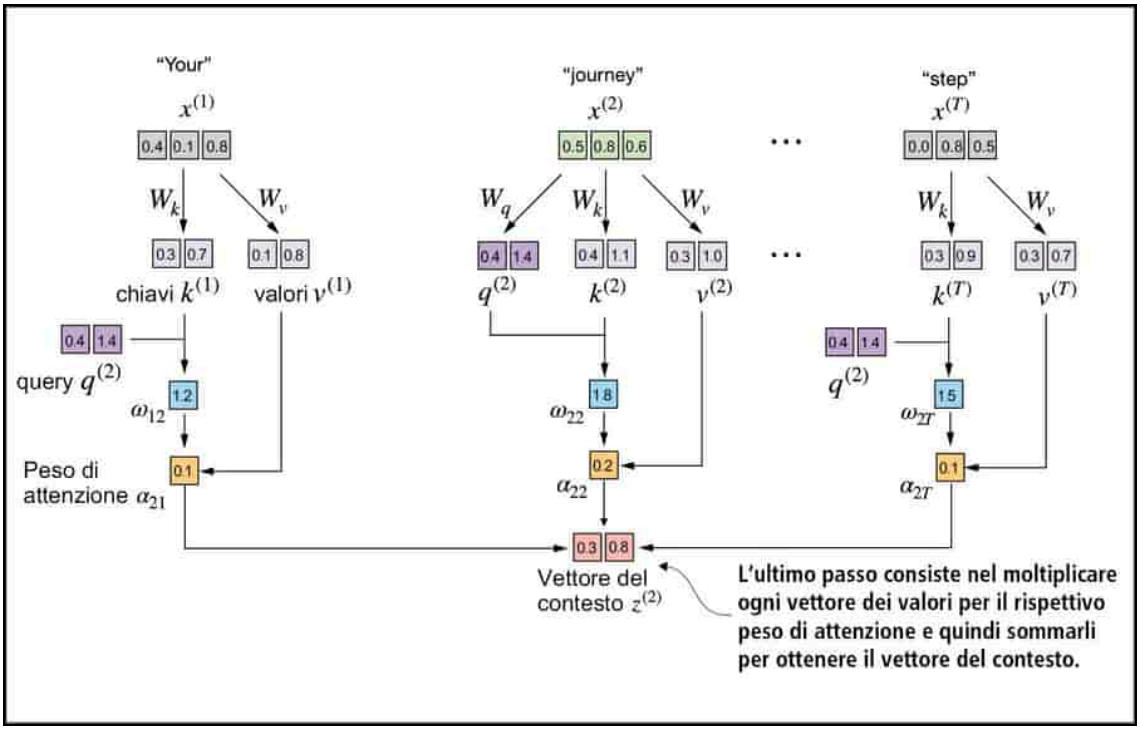


Figura 3.17 Nella fase finale del calcolo dell'auto-attenzione, calcoliamo il vettore del contesto combinando tutti i vettori dei valori tramite i pesi di attenzione.

In modo analogo a quando abbiamo calcolato il vettore del contesto come una somma ponderata sui vettori di input, ora calcoliamo il vettore del contesto come una somma ponderata sui vettori dei valori. Qui, i pesi di attenzione servono come fattore di ponderazione che valuta la rispettiva importanza di ciascun vettore dei valori. Inoltre, come prima, possiamo usare la moltiplicazione di matrici per ottenere l'output in un unico passo:

```
context_vec_2 = attn_weights_2 @ values
print(context_vec_2)
```

Il contenuto del vettore risultante è il seguente:

```
tensor([0.3061, 0.8210])
```

Finora abbiamo calcolato un singolo vettore del contesto, $z^{(2)}$. Ora, generalizzeremo il codice per calcolare tutti i vettori di contesto della sequenza di input, da $z^{(1)}$ a $z^{(T)}$.

Perché query, chiavi e valori?

I termini “chiave”, “query” e “valore” nel contesto dei meccanismi di attenzione sono tratti dal dominio dei database, dove concetti simili vengono utilizzati per archiviare, cercare e recuperare informazioni.

Una *query* è analoga a una query di ricerca in un database. Rappresenta l’elemento corrente (per esempio, una parola o un token in una frase) sul quale il modello si concentra o che cerca di comprendere. La query viene utilizzata per sondare le altre parti della sequenza di input per determinare quanta attenzione dedicare loro.

La *chiave* è come una chiave di un database, usata per l’indicizzazione e la ricerca. Nel meccanismo di attenzione, ogni elemento della sequenza di input (per esempio, ogni parola di una frase) ha una sua chiave. Queste chiavi sono usate per trovare corrispondenza con le query.

Il *valore*, in questo contesto, è simile al valore in una coppia chiave-valore di un database. Rappresenta il contenuto effettivo o la rappresentazione degli elementi di input. Una volta che il modello determina quali chiavi (e quindi quali parti dell’input) sono più rilevanti per la query (l’elemento su cui ci concentriamo), recupera i valori corrispondenti.

Implementazione di una classe Python compatta di auto-attenzione

A questo punto, abbiamo eseguito molti passi per calcolare gli output di auto-attenzione. Lo scopo era principalmente illustrativo, in modo da poter procedere un passo alla volta. All’atto pratico, nell’ottica dell’implementazione di un modello LLM che svolgeremo nel prossimo capitolo, è utile organizzare questo codice in una classe Python, come mostrato nel Listato 3.1.

Listato 3.1 Una classe compatta di auto-attenzione.

```
import torch.nn as nn
class SelfAttention_v1(nn.Module):
    def __init__(self, d_in, d_out):
        super().__init__()
        self.W_query = nn.Parameter(torch.rand(d_in, d_out))
        self.W_key = nn.Parameter(torch.rand(d_in, d_out))
        self.W_value = nn.Parameter(torch.rand(d_in, d_out))

    def forward(self, x):
        keys = x @ self.W_key
        queries = x @ self.W_query
        values = x @ self.W_value
```

```
        attn_scores = queries @ keys.T # omega
        attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
        context_vec = attn_weights @ values
        return context_vec
```

In questo codice PyTorch, `SelfAttention_v1` è una classe derivata da `nn.Module`, un elemento fondamentale dei modelli PyTorch che fornisce le funzionalità necessarie per la creazione e la gestione dei layer del modello.

Il metodo `__init__` inizializza le matrici dei pesi addestrabili (`W_query`, `W_key` e `W_value`) con le query, le chiavi e i valori, per trasformare le dimensioni di input `d_in` nelle dimensioni dell'output `d_out`.

Durante il passo *forward*, che utilizza il metodo `forward`, calcoliamo i punteggi di attenzione (`attn_scores`) moltiplicando le query e le chiavi, e normalizzando questi punteggi con `softmax()`. Infine, creiamo un vettore del contesto "pesando" i valori con questi punteggi di attenzione normalizzati.

Possiamo utilizzare questa classe come segue:

```
torch.manual_seed(123)
sa_v1 = SelfAttention_v1(d_in, d_out)
print(sa_v1(inputs))
```

Poiché `inputs` contiene sei vettori di embedding, otterremo una matrice che memorizza i sei vettori di contesto:

```
tensor([[0.2996, 0.8053],
        [0.3061, 0.8210],
        [0.3058, 0.8203],
        [0.2948, 0.7939],
        [0.2927, 0.7891],
        [0.2990, 0.8040]],
        grad_fn=<MmBackward0>
)
```

Per un rapido check, notate che la seconda riga (`[0.3061, 0.8210]`) corrisponde al contenuto di `context_vec_2` nel paragrafo precedente. La Figura 3.18 riassume il meccanismo di auto-attenzione che abbiamo appena implementato.

L’auto-attenzione coinvolge le matrici dei pesi addestrabili W_q, W_k e W_v. Queste matrici trasformano i dati di input in, rispettivamente, query, chiavi e valori, che sono componenti cruciali del meccanismo di attenzione. Man mano che il modello viene esposto a più dati durante l’addestramento, regola questi pesi addestrabili, come vedremo nei prossimi capitoli.

Possiamo migliorare ulteriormente l’implementazione di `SelfAttention_v1` utilizzando layer `nn.Linear` di PyTorch, che eseguono efficacemente la moltiplicazione di matrici quando le unità di bias sono disabilitate.

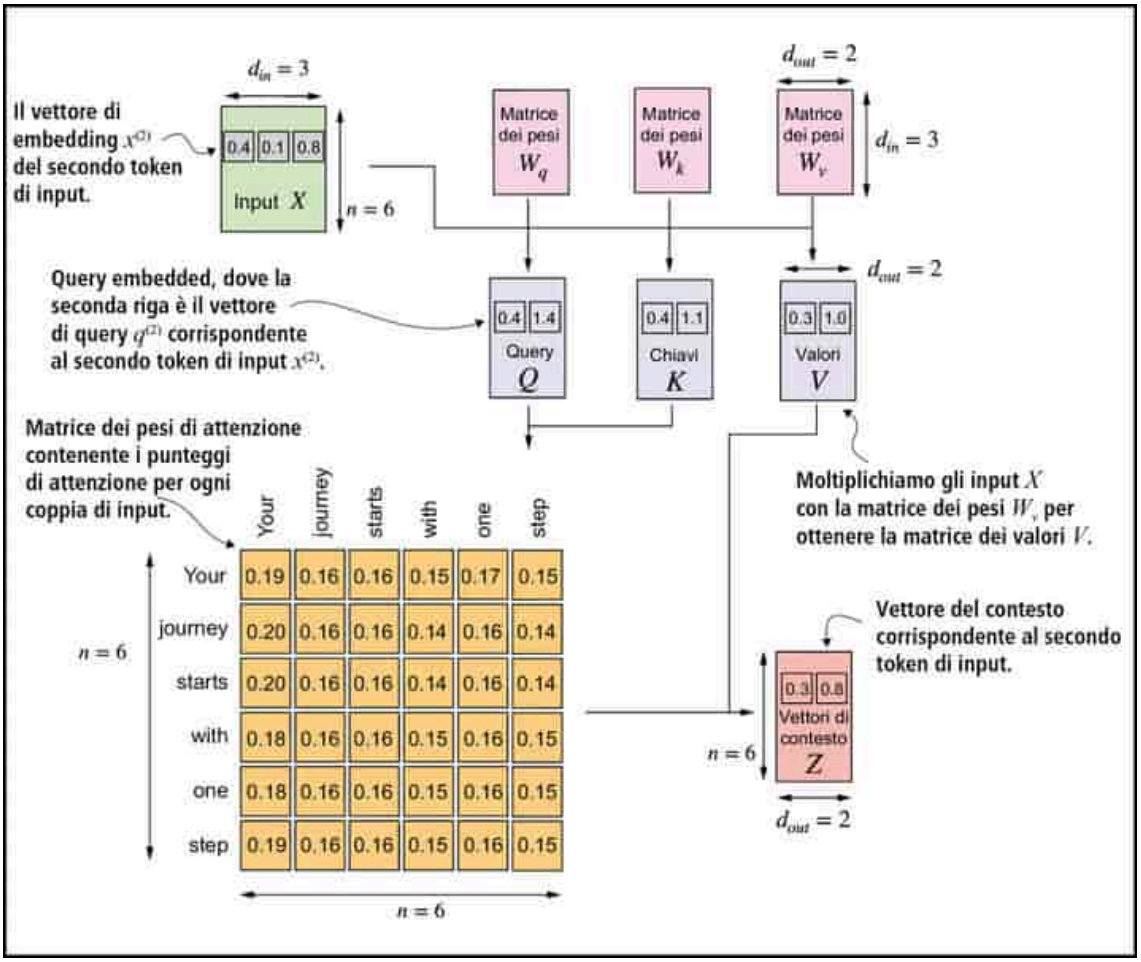


Figura 3.18 Nell’auto-attenzione, trasformiamo i vettori di input nella matrice di input X con le tre matrici dei pesi, W_q, W_k e W_v. Ora calcoliamo la matrice dei pesi di attenzione

in base alle query (Q) e alle chiavi (K) risultanti. Utilizzando i pesi e i valori di attenzione (V), calcoliamo poi i vettori di contesto (Z). Per chiarezza visuale, ci concentriamo su un singolo testo di input con *n* token, non su un batch di più input. Di conseguenza, in questo contesto il tensore di input tridimensionale viene semplificato in una matrice bidimensionale. Questo approccio consente una visualizzazione e una comprensione più semplici dei processi coinvolti. Per coerenza con le figure successive, i valori nella matrice di attenzione non rappresentano i pesi di attenzione reali. I numeri in questa figura sono troncati a due cifre dopo la virgola decimale per ridurne l'ingombro. I valori in ogni riga dovrebbero avere somma 1, il 100 percento.

Inoltre, un vantaggio significativo dell'utilizzo di `nn.Linear` anziché implementare manualmente `nn.Parameter(torch.rand(...))` è che `nn.Linear` ha uno schema ottimizzato di inizializzazione dei pesi, il che contribuisce a un addestramento del modello più stabile ed efficace.

Listato 3.2 Una classe di auto-attenzione che utilizza layer nn.Linear di PyTorch.

```
class SelfAttention_v2(nn.Module):
    def __init__(self, d_in, d_out, qkv_bias=False):
        super().__init__()
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)

    def forward(self, x):
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)
        attn_scores = queries @ keys.T
        attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
        context_vec = attn_weights @ values
        return context_vec
```

Potete utilizzare `SelfAttention_v2` in modo simile a `SelfAttention_v1`:

```
torch.manual_seed(789)
sa_v2 = SelfAttention_v2(d_in, d_out)
print(sa_v2(inputs))
```

L'output è il seguente:

```
tensor([[-0.0739, 0.0713],
        [-0.0748, 0.0703],
        [-0.0749, 0.0702],
        [-0.0760, 0.0685],
        [-0.0763, 0.0679],
```

```
        [-0.0754, 0.0693]],
        grad_fn=<MmBackward0>
)
```

Notate che `SelfAttention_v1` e `SelfAttention_v2` forniscono output differenti perché utilizzano pesi iniziali differenti per le matrici dei pesi, poiché `nn.Linear` utilizza uno schema più sofisticato di inizializzazione dei pesi.

Esercizio 3.1 Confronto tra SelfAttention_v1 e SelfAttention_v2

Notate che `nn.Linear` in `SelfAttention_v2` utilizza uno schema di inizializzazione dei pesi differente da `nn.Parameter(torch.rand(d_in, d_out))` utilizzato in `SelfAttention_v1`, il che fa in modo che i due meccanismi producano risultati differenti. Per verificare che entrambe le implementazioni, `SelfAttention_v1` e `SelfAttention_v2`, operano in modo simile, possiamo trasferire le matrici dei pesi da un oggetto `SelfAttention_v2` a un oggetto `SelfAttention_v1`, in modo che entrambi gli oggetti producano gli stessi risultati.

Il vostro compito è quello di assegnare correttamente i pesi da un'istanza di `SelfAttention_v2` a un'istanza di `SelfAttention_v1`. Per fare ciò, dovete comprendere la relazione che sussiste tra i pesi in entrambe le versioni (suggerimento: `nn.Linear` memorizza la matrice dei pesi in una forma trasposta). Dopo l'esercizio, dovreste osservare che entrambe le istanze producono gli stessi output.

Ora, apporteremo dei miglioramenti al meccanismo di auto-attenzione, concentrandoci in modo specifico sull'incorporazione di elementi causali e multi-head. L'aspetto causale implica la modifica del meccanismo di attenzione, per impedire al modello di accedere a informazioni future nella sequenza, il che è fondamentale per attività come la modellazione linguistica, in cui ogni previsione di parola dovrebbe dipendere solo dalle parole precedenti.

Il componente multi-head comporta la suddivisione del meccanismo di attenzione in più "teste". Ogni testa apprende diversi aspetti dei dati, consentendo al modello di occuparsi simultaneamente di informazioni provenienti da più sottospazi di

rappresentazione, in più posizioni. Ciò migliora le prestazioni del modello in attività complesse.

Nascondere le parole future con l’attenzione causale

Per molti compiti tipici di un modello LLM, vorremmo che il meccanismo di auto-attenzione prendesse in considerazione solo i token che compaiono prima della posizione attuale nel prevedere il token successivo in una sequenza. L’attenzione causale, anche nota come *attenzione con maschera*, è una forma specializzata di auto-attenzione. Limita un modello a considerare solo gli input precedenti e attuali in una sequenza nell’elaborare un qualsiasi token durante il calcolo dei punteggi di attenzione. Questo è in contrasto col meccanismo standard di auto-attenzione, che consente l’accesso all’intera sequenza di input.

Ora, quindi, modificheremo il meccanismo standard di auto-attenzione per creare un meccanismo di *attenzione causale*, che è essenziale per sviluppare il modello LLM dei prossimi capitoli. Per ottenere questo risultato nei modelli LLM di tipo GPT, per ogni token elaborato mascheriamo i token futuri, successivi rispetto al token corrente nel testo di input, come mostra la Figura 3.19. Mascheriamo i pesi di attenzione sopra la diagonale, e normalizziamo i pesi di attenzione non mascherati in modo che siano a somma 1 in ogni riga. In seguito, implementeremo questa procedura di mascheramento e normalizzazione nel codice.

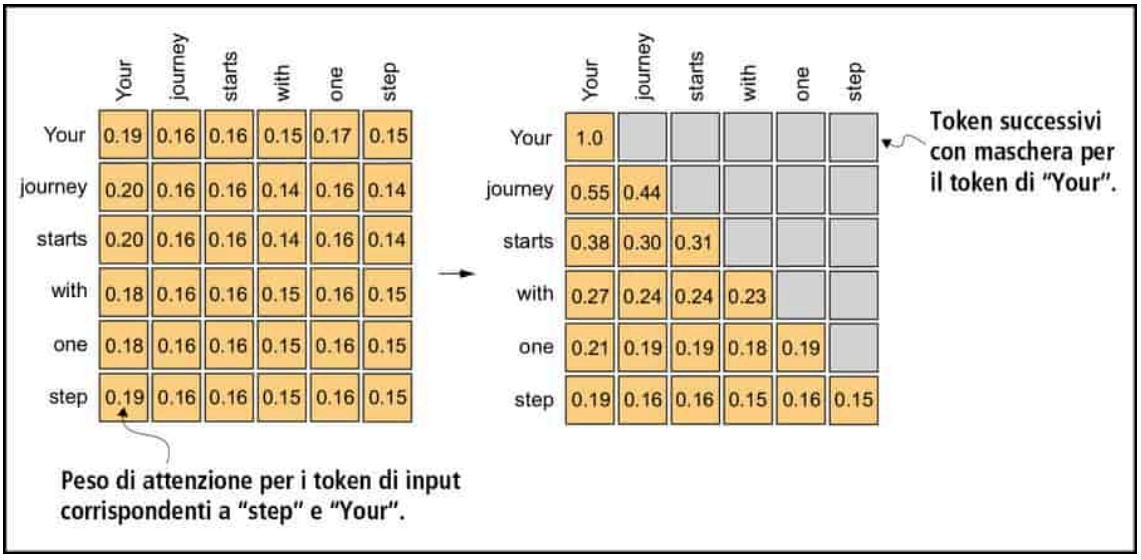


Figura 3.19 Nell'attenzione causale, mascheriamo i pesi di attenzione sopra la diagonale, in modo tale che dato un input, il modello LLM non possa accedere ai token successivi nel calcolare i vettori di contesto utilizzando i pesi di attenzione. Per esempio, per la parola "journey" nella seconda riga, manteniamo solo i pesi di attenzione per le parole precedenti, solo "Your" in questo caso, per la posizione corrente ("journey").

Applicazione di una maschera di attenzione causale

Il nostro prossimo passo consiste nell'implementare la maschera di attenzione causale. Per implementare i passi per applicare una maschera di attenzione causale così da ottenere i pesi di attenzione con maschera, come riassunto nella Figura 3.20, impieghiamo i punteggi e i pesi di attenzione del paragrafo precedente per programmare il meccanismo di attenzione causale.

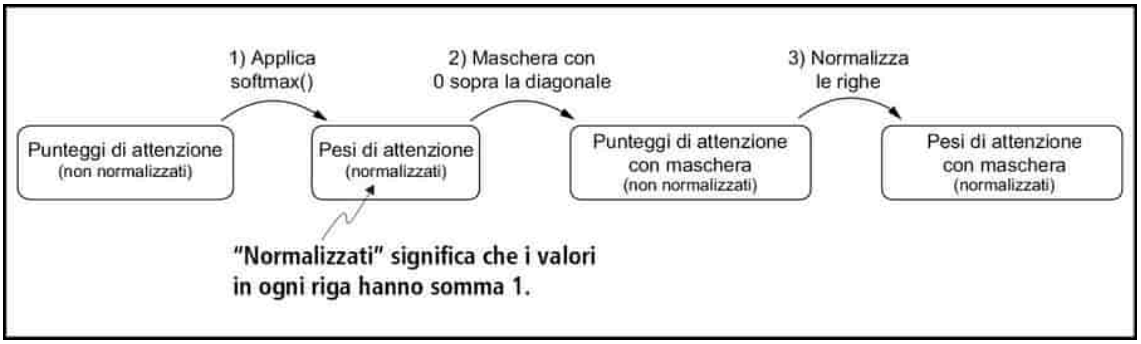


Figura 3.20 Un modo per ottenere la matrice dei pesi di attenzione con maschera dell'attenzione causale consiste nell'applicare la funzione softmax() ai punteggi di attenzione, azzerando gli elementi sopra la diagonale e normalizzando la matrice risultante.

Nel primo passo, calcoliamo i pesi di attenzione utilizzando la funzione `softmax()`, come abbiamo fatto in precedenza:

```
queries = sa_v2.W_query(inputs) (1)
keys = sa_v2.W_key(inputs)
attn_scores = queries @ keys.T
attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
print(attn_weights)
```

(1) Per praticità, riutilizza le matrici delle query e delle chiavi dell'oggetto `SelfAttention_v2` del paragrafo precedente.

Ciò si traduce nei seguenti pesi di attenzione:

```
tensor([[0.1921, 0.1646, 0.1652, 0.1550, 0.1721, 0.1510],
        [0,2041, 0.1659, 0.1662, 0.1496, 0.1665, 0.1477],
        [0.2036, 0.1659, 0.1662, 0.1498, 0.1664, 0.1480],
        [0.1869, 0.1667, 0.1668, 0.1571, 0.1661, 0.1564],
        [0.1830. 0.1669, 0.1670. 0.1588, 0.1658, 0.1585],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
        grad_fn=<SoftmaxBackward0>
)
```

Possiamo implementare il secondo passo utilizzando la funzione PyTorch `tril`, per creare una maschera in cui i valori sopra la diagonale sono pari a zero:

```
context_length = attn_scores.shape[0]
mask_simple = torch.tril(torch.ones(context_length, context_length))
print(mask_simple)
```

La maschera risultante è la seguente:

```
tensor([[1., 0., 0., 0., 0., 0.],
        [1., 1., 0., 0., 0., 0.],
        [1., 1., 1., 0., 0., 0.],
        [1., 1., 1., 1., 0., 0.],
        [1., 1., 1., 1., 1., 0.],
        [1., 1., 1., 1., 1., 1.]]
)
```

Ora possiamo moltiplicare questa maschera per i pesi di attenzione, per azzerare i valori sopra la diagonale:

```
masked_simple = attn_weights*mask_simple
print(masked_simple)
```

Come potete vedere, gli elementi sopra la diagonale sono stati azzerati correttamente:

```
tensor([[0.1921, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2041, 0.1659, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2036, 0.1659, 0.1662, 0.0000, 0.0000, 0.0000],
        [0.1869, 0.1667, 0.1668, 0.1571, 0.0000, 0.0000],
        [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
        grad_fn=<MulBackward0>
)
```

Il terzo passo consiste nel rinormalizzare i pesi di attenzione per riportarli a somma 1 in ogni riga. Possiamo ottenere questo risultato dividendo ogni elemento di ogni riga per la somma in ogni riga:

```
row_sums = masked_simple.sum(dim=-1, keepdim=True)
masked_simple_norm = masked_simple / row_sums
print(masked_simple_norm)
```

Il risultato è una matrice dei pesi di attenzione in cui i pesi di attenzione sopra la diagonale vengono azzerati e la somma delle righe è pari a 1:

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
        [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
        [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
        grad_fn=<DivBackward0>
)
```

Fughe di informazioni

Quando applichiamo una maschera e poi rinormalizziamo i pesi di attenzione, inizialmente potrebbe sembrare che le informazioni provenienti da token successivi (che intendiamo mascherare) possano ancora influenzare il token corrente, perché

i loro valori fanno parte del calcolo di `softmax()`. Tuttavia, l'idea è che, quando rinormalizziamo i pesi di attenzione dopo il mascheramento, in sostanza ricalcoliamo `softmax()` su un sottoinsieme più piccolo (poiché le posizioni mascherate non contribuiscono più al valore prodotto da `softmax()`).

L'eleganza matematica di `softmax()` sta nel fatto che, nonostante inizialmente nel denominatore siano incluse tutte le posizioni, dopo il mascheramento e la rinormalizzazione l'effetto delle posizioni mascherate viene annullato: non contribuiscono più in alcun modo significativo al punteggio prodotto da `softmax()`.

In termini più semplici, dopo il mascheramento e la rinormalizzazione, la distribuzione dei pesi di attenzione è come se fosse stata calcolata solo tra le posizioni non mascherate. Ciò garantisce che non ci siano trasferimenti di informazioni da token successivi (o comunque con maschera).

A questo punto potremmo anche concludere la nostra implementazione dell'attenzione causale, ma possiamo ancora migliorarla. Prendiamo una proprietà matematica della funzione `softmax()` e implementiamo il calcolo dei pesi di attenzione con maschera in modo più efficiente in meno passi, come mostra la Figura 3.21.

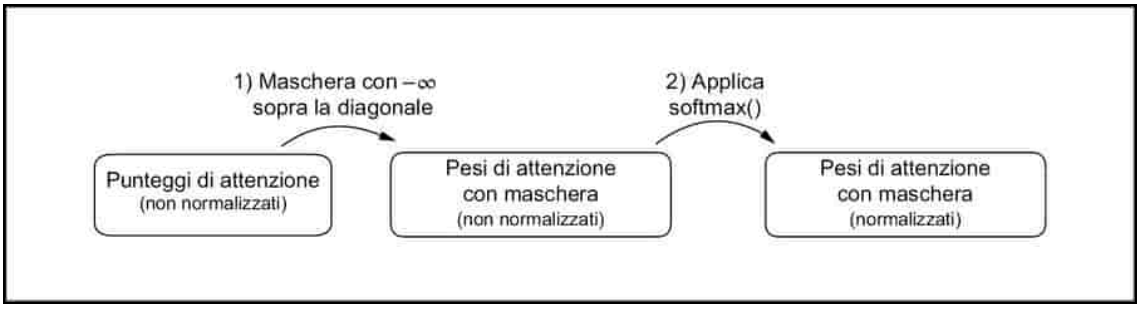


Figura 3.21 Un modo più efficiente per ottenere la matrice dei pesi di attenzione con maschera nell'attenzione causale consiste nel mascherare i punteggi di attenzione con valori infiniti negativi prima di applicare la funzione softmax().

La funzione `softmax()` converte i suoi input in una distribuzione di probabilità. Quando in un riga sono presenti valori a infinito negativo (-∞), la funzione `softmax()` la tratta come una probabilità zero (matematicamente, questo è perché $e^{-\infty}$ è prossima a 0).

Possiamo implementare questo “trucco” di mascheramento, più efficiente, creando una maschera con tanti 1 sopra la diagonale, e poi sostituendo questi `1` con valori di infinito negativo (`-inf`):

```
mask = torch.triu(torch.ones(context_length, context_length), diagonal=1)
masked = attn_scores.masked_fill(mask.bool(), -torch.inf)
print(masked)
```

Il risultato è la seguente maschera:

```
tensor([[0.2899, -inf, -inf, -inf, -inf, -inf],
        [0.4656, 0.1723, -inf, -inf, -inf, -inf],
        [0.4594, 0.1703, 0.1731, -inf, -inf, -inf],
        [0.2642, 0.1024, 0.1036, 0.0186, -inf, -inf],
        [0.2183, 0.0874, 0.0882, 0.0177, 0.0786, -inf],
        [0.3408, 0.1270, 0.1290, 0.0198, 0.1290, 0.0078]],
        grad_fn=<MaskedFillBackward0>
)
```

Ora, tutto ciò che dobbiamo fare è applicare la funzione `softmax()` a questi risultati mascherati, e il gioco è fatto:

```
attn_weights = torch.softmax(masked / keys.shape[-1]**0.5, dim=1)
print(attn_weights)
```

Come potete vedere dall’output, la somma dei valori in ogni riga è 1, e non è necessaria alcuna ulteriore normalizzazione:

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
        [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
        [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
        grad_fn=<SoftmaxBackward0>
)
```

Ora potremmo usare i pesi di attenzione modificati per calcolare i vettori di contesto tramite `context_vec = attn_weights @ values`, come abbiamo fatto in precedenza. Tuttavia, prima eseguiremo un altro piccolo ritocco al meccanismo di attenzione causale, utile per ridurre l’overfitting durante l’addestramento dei modelli LLM.

Ulteriore mascheramento dei pesi di attenzione con il dropout

Il *dropout*, nel deep learning, è una tecnica in cui determinate unità a layer nascosto, selezionate casualmente, vengono ignorate durante l'addestramento, di fatto "eliminandole". Questo metodo aiuta a prevenire l'overfitting, assicurando che un modello non diventi eccessivamente dipendente da un set specifico di unità a layer nascosto. È importante sottolineare che il dropout viene utilizzato solo durante l'addestramento, e poi viene disabilitato.

Nell'architettura a transformer, inclusi modelli come GPT, in genere il dropout nel meccanismo di attenzione viene applicato in due momenti specifici: dopo aver calcolato i pesi di attenzione o dopo aver applicato i pesi di attenzione ai vettori dei valori. Qui applicheremo la maschera di dropout dopo aver calcolato i pesi di attenzione, come mostra la Figura 3.22, perché questa è la variante più comune, nella pratica.

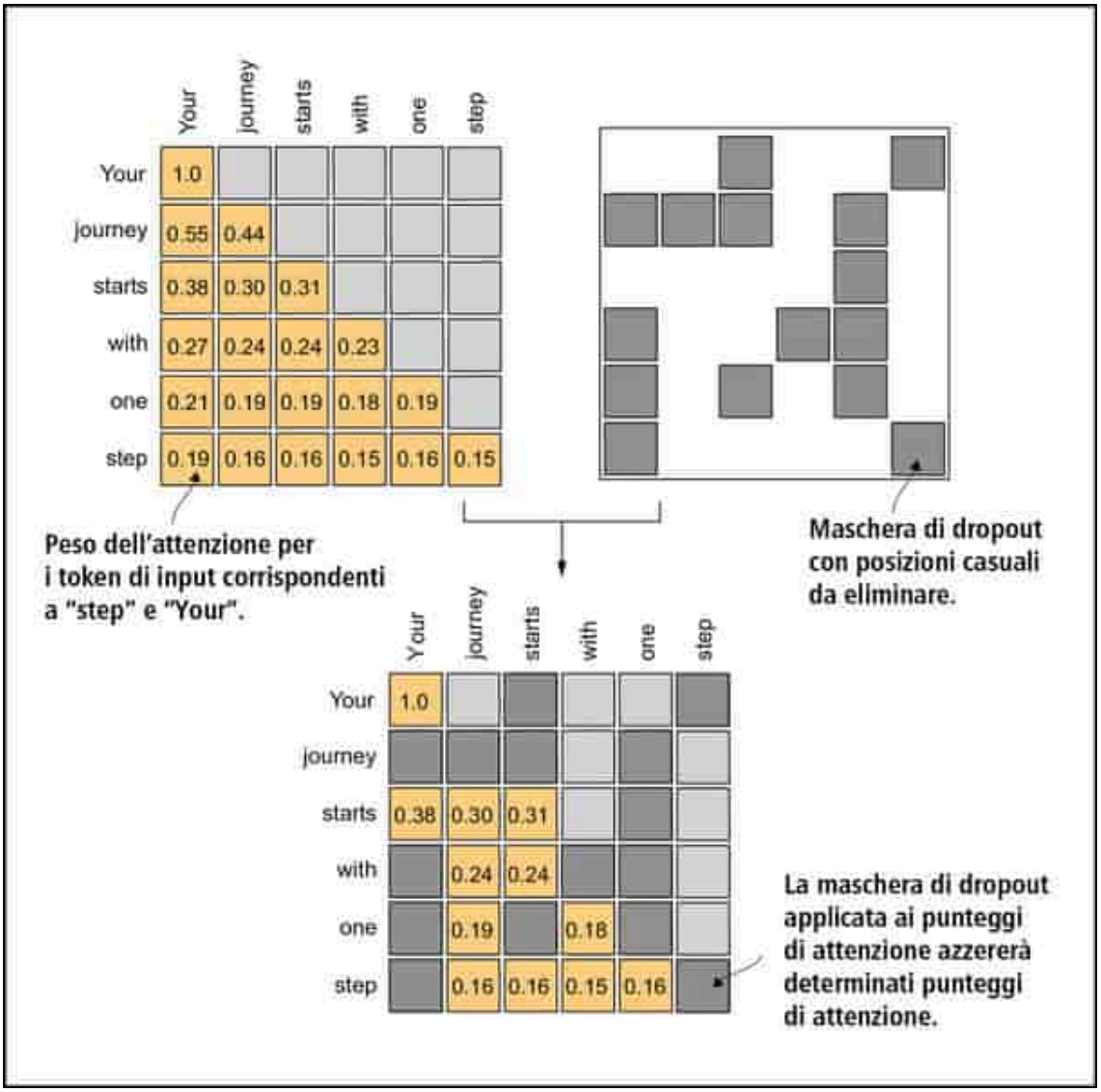


Figura 3.22 Utilizzando la maschera di attenzione causale (in alto a sinistra), applichiamo una maschera di dropout aggiuntiva (in alto a destra) per azzerare ulteriori pesi di attenzione e ridurre l'overfitting durante l'addestramento.

Nel seguente esempio di codice, utilizziamo un tasso di dropout del 50 percento, il che equivale a mascherare metà dei pesi di attenzione. Quando addestreremo il modello GPT nei prossimi capitoli, utilizzeremo un tasso di dropout inferiore, per esempio del 10 o del

20 percento). Per semplicità, applichiamo prima l'implementazione del dropout di PyTorch a un tensore 6 × 6 costituito da tutti 1:

```
torch.manual_seed(123)
dropout = torch.nn.Dropout(0.5) (1)
example = torch.ones(6, 6) (2)
print(dropout(example))
```

(1) Scegliamo un tasso di dropout del 50%.

(2) Qui, creiamo una matrice di tutti 1.

Come potete vedere, circa la metà dei valori è stata azzerata:

```
tensor([[2., 2., 0., 2., 2., 0.],
        [0., 0., 0., 2., 0., 2.],
        [2., 2., 2., 2., 0., 2.],
        [0., 2., 2., 0., 0., 2.],
        [0., 2., 0., 2., 0., 2.],
        [0., 2., 2., 2., 2., 0.]]
)
```

Quando si applica il dropout a una matrice di pesi di attenzione con un tasso del 50 percento, una metà (casuale) degli elementi presenti nella matrice verrà impostata a zero. Per compensare la riduzione degli elementi attivi, i valori degli elementi rimanenti nella matrice vengono aumentati di un fattore di 1 / 0.5 = 2. Questa estensione di scala è fondamentale per mantenere l'equilibrio complessivo dei pesi di attenzione, assicurando che l'influenza media del meccanismo di attenzione rimanga coerente sia durante la fase di addestramento sia durante l'inferenza.

Ora applichiamo il dropout alla matrice dei pesi di attenzione:

```
torch.manual_seed(123)
print(dropout(attn_weights))
```

La matrice dei pesi di attenzione risultante ora ha altri elementi azzerati e i rimanenti ridimensionati a 1:

```
tensor([[2.0000, 0.0000, 0 .0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.7599, 0.6194, 0.6206, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.4921, 0.4925, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.3966, 0.0000, 0.3775, 0.0000, 0.0000],
        [0.0000, 0.3327, 0.3331, 0.3084, 0.3331, 0.0000]],
       grad_fn=<MulBackward0>
)
```

Tenete presente che i dropout potrebbero apparire diversi a seconda del sistema operativo; per ulteriori informazioni su questa incoerenza, leggete il tracker dei problemi di PyTorch all'indirizzo https://github.com/pytorch/pytorch/issues/121595.

Ora che avete acquisito una comprensione dell'attenzione causale e del mascheramento a dropout, possiamo sviluppare una classe Python. Questa classe è progettata per facilitare l'applicazione efficiente di queste due tecniche.

Implementazione di una classe di attenzione causale compatta

Ora incorporeremo l'attenzione causale e le modifiche relative al dropout nella classe Python `SelfAttention` che abbiamo sviluppato in precedenza. Questa classe servirà da modello per lo sviluppo dell'*attenzione multi-head*, l'ultima classe di attenzione che implementeremo.

Ma prima di iniziare, assicuriamoci che il codice possa gestire lotti (*batch*) costituiti da più di un input, in modo che la classe `CausalAttention` supporti i batch prodotti dal data loader implementato nel Capitolo 2.

Per semplicità, per simulare tali batch, duplichiamo l'esempio del testo di input:

```
batch = torch.stack((inputs, inputs), dim=0)
print(batch.shape) (1)
```

(1) Due input di sei token ciascuno; ogni token ha dimensioni dell'embedding pari a 3.

Ciò si traduce in un tensore tridimensionale costituito da due testi di input con sei token ciascuno, dove ogni token è un vettore di embedding tridimensionale:

```
torch.Size([2, 6, 3])
```

La seguente classe `CausalAttention` è simile alla classe `SelfAttention` implementata in precedenza, fatta eccezione per il fatto che abbiamo aggiunto i componenti di dropout e di maschera causale.

Listato 3.3 Una classe compatta di attenzione causale.

```
class CausalAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length, dropout, qkv_bias=False):
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.dropout = nn.Dropout(dropout) (1)
        self.register_buffer( (2)
            'mask',
            torch.triu(torch.ones(context_length, context_length),
            diagonal=1)
        )

    def forward(self, x):
        b, num_tokens, d_in = x.shape (3)
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)

        attn_scores = queries @ keys.transpose(1, 2) (3)
        attn_scores.masked_fill_(self.mask.bool()[:num_tokens, :num_tokens], -
torch.inf) (4)
        attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
        attn_weights = self.dropout(attn_weights)

        context_vec = attn_weights @ values
        return context_vec
```

(1) Rispetto alla precedente classe `SelfAttention_v1`, abbiamo aggiunto un layer di dropout.

(2) Anche la chiamata `register_buffer` è una nuova aggiunta (ulteriori informazioni sono fornite nel testo).

(3) Trasponiamo le dimensioni 1 e 2, mantenendo le dimensioni del batch nella prima posizione `(0)`.

(4) In PyTorch, le operazioni con underscore finale vengono eseguite in loco, evitando duplicazioni non necessarie in memoria.

Tutte le righe di codice aggiunte dovrebbero ormai essere familiari, ma abbiamo aggiunto una chiamata `self.register_buffer()` nel metodo

`__init__`. L’uso di `register_buffer` in PyTorch non è strettamente necessario per tutti i casi d’uso, ma offre parecchi vantaggi, qui. Per esempio, quando usiamo la classe `CausalAttention` nel modello LLM, i buffer vengono automaticamente spostati sul dispositivo appropriato (CPU o GPU) insieme al modello, un aspetto che sarà rilevante durante il suo addestramento. Ciò significa che non dobbiamo assicurarci manualmente che questi tensori siano sullo stesso dispositivo dei parametri del modello, evitando i relativi errori.

Possiamo usare la classe `CausalAttention` come segue, come abbiamo fatto in precedenza con `SelfAttention`:

```
torch.manual_seed(123)
context_length = batch.shape[1]
ca = CausalAttention(d_in, d_out, context_length, 0.0)
context_vecs = ca(batch)
print("context_vecs.shape:", context_vecs.shape)
```

Il vettore del contesto risultante è un tensore tridimensionale in cui ogni token è rappresentato da un embedding bidimensionale:

```
context_vecs.shape: torch.Size([2, 6, 2])
```

La Figura 3.23 riassume quanto abbiamo realizzato finora. Ci siamo concentrati sul concetto e sull’implementazione dell’attenzione causale nelle reti neurali. Ora, amplieremo questo concetto e implementeremo un modulo di attenzione multi-head che implementa in parallelo più meccanismi di attenzione causale.

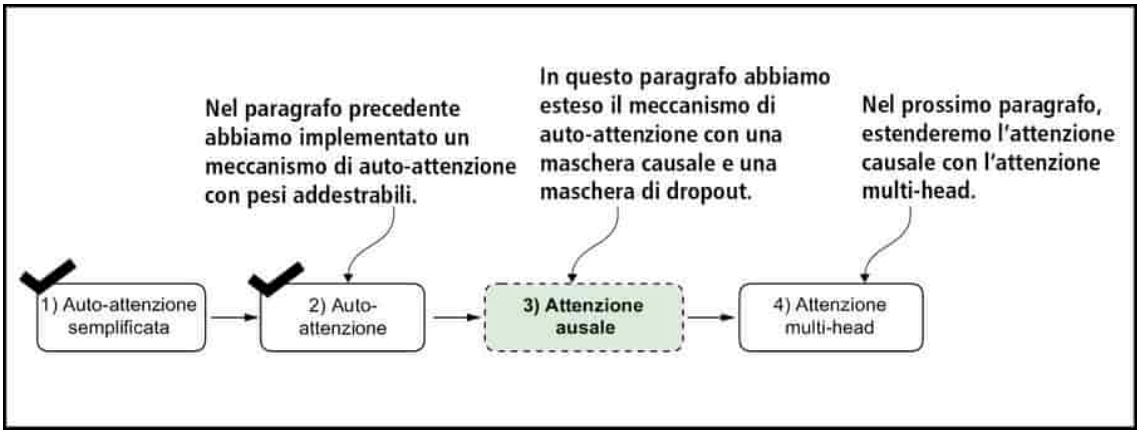


Figura 3.23 Ecco quello che abbiamo fatto finora. Abbiamo iniziato con un meccanismo di attenzione semplificato, abbiamo aggiunto pesi addestrabili e poi abbiamo aggiunto una maschera di attenzione causale. Ora, estenderemo il meccanismo di attenzione causale e programmeremo l'attenzione multi-head, che useremo nel modello LLM.

Estensione dell'attenzione single-head all'attenzione multi-head

Il nostro ultimo passo sarà estendere la classe di attenzione causale precedentemente implementata su più "teste", con l'*attenzione multi-head*.

Il termine "multi-head" si riferisce alla divisione del meccanismo di attenzione su più "teste", ciascuna operante in modo indipendente. In questo contesto, un singolo modulo di attenzione causale può essere considerato single-head: un solo set di pesi di attenzione che elaborano l'input in sequenza.

Affronteremo quindi l'espansione dall'attenzione causale all'attenzione multi-head. Innanzitutto, costruiremo intuitivamente un modulo di attenzione multi-head impilando più moduli `CausalAttention`. Quindi implementeremo lo stesso modulo di attenzione

multi-head in un modo più complicato ma più efficiente dal punto di vista computazionale.

Uno stack di più layer di attenzione single-head

In termini pratici, implementare l'attenzione multi-head implica la creazione di più istanze del meccanismo di auto-attenzione (vedi Figura 3.18), ciascuna con i propri pesi, per poi combinare i loro output. L'utilizzo di più istanze del meccanismo di auto-attenzione può essere computazionalmente pesante, ma è fondamentale per il riconoscimento di pattern complessi per cui sono noti i modelli LLM basati su transformer.

La Figura 3.24 illustra la struttura di un modulo di attenzione multi-head, costituito da più moduli di attenzione single-head (vedi la Figura 3.18 precedente), impilati uno sopra l'altro.

Come ho accennato in precedenza, l'idea su cui si basa l'attenzione multi-head è quella di eseguire il meccanismo di attenzione più volte (in parallelo) con diverse proiezioni lineari apprese, ovvero i risultati della moltiplicazione dei dati di input (come i vettori delle query, delle chiavi e dei valori nei meccanismi di attenzione) per una matrice dei pesi. Nel codice, possiamo ottenere questo risultato implementando una semplice classe `MultiHeadAttentionWrapper` che impila più istanze del modulo `CausalAttention` che abbiamo implementato precedentemente.

Listato 3.4 Una classe wrapper per implementare l'attenzione multi-head.

```
class MultiHeadAttentionWrapper(nn.Module):
    def __init__(self, d_in, d_out, context_length, dropout, num_heads,
qkv_bias=False):
        super().__init__()
        self.heads = nn.ModuleList(
            [CausalAttention(d_in, d_out, context_length, dropout, qkv_bias)
                for _ in range(num_heads)
            ]
```

```
        )

    def forward(self, x):
        return torch.cat([head(x) for head in self.heads], dim=-1)
```

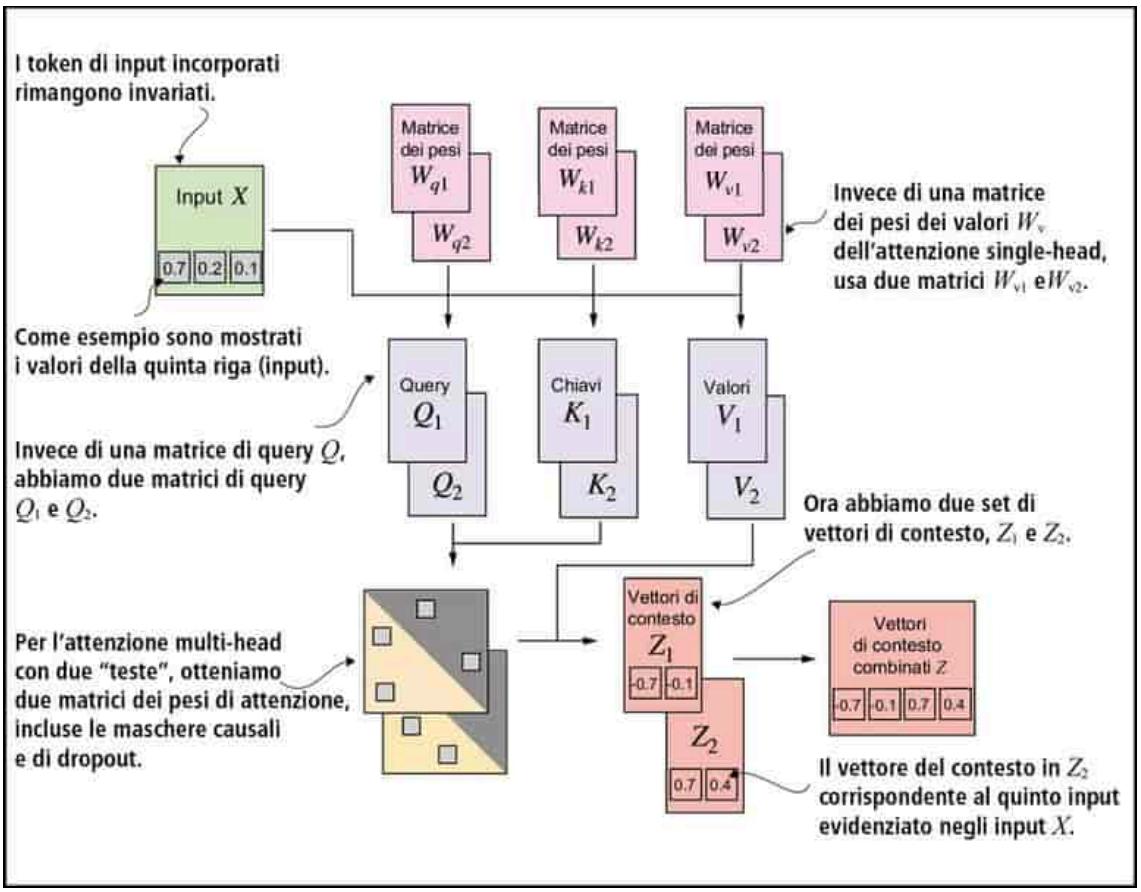


Figura 3.24 Il modulo di attenzione multi-head include due moduli di attenzione single-head impilati uno sull'altro. Quindi, invece di usare una singola matrice W_v per calcolare le matrici dei valori, in un modulo di attenzione multi-head con due "teste", abbiamo due matrici dei pesi dei valori: W_{v1} e W_{v2}. Lo stesso vale per le altre matrici dei pesi, W_Q e W_k. Otteniamo due set di vettori di contesto Z_1 e Z_2 che possiamo combinare in una singola matrice vettore del contesto Z.

Per esempio, se utilizziamo questa classe `MultiHeadAttentionWrapper` con due "teste" (con `num_heads=2`) e dimensioni dell'output di `CausalAttention` pari a 2 (`d_out=2`), otteniamo un vettore del contesto quadridimensionale (`d_out*num_heads=4`), come mostra la Figura 3.25.

Per illustrare ulteriormente questo concetto con un esempio concreto, possiamo utilizzare la classe `MultiHeadAttentionWrapper` in modo simile alla classe `CausalAttention` precedente:

```
torch.manual_seed(123)
context_length = batch.shape[1] # This is the number of tokens
d_in, d_out = 3, 2
mha = MultiHeadAttentionWrapper(d_in, d_out, context_length, 0.0, num_heads=2)
context_vecs = mha(batch)

print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)
```

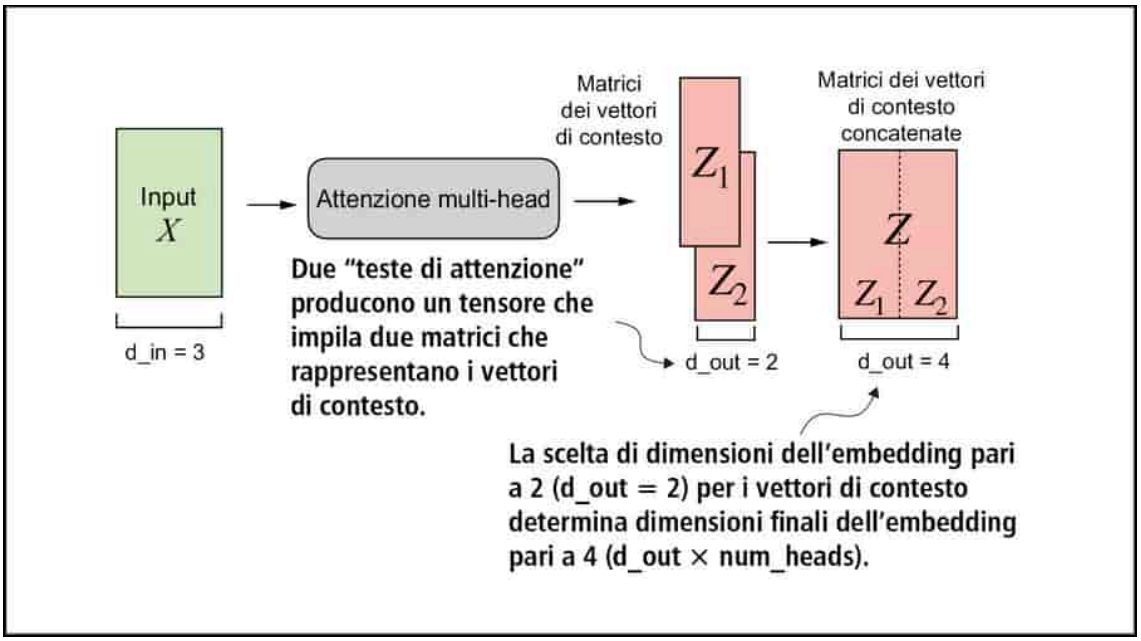


Figura 3.25 Utilizzando MultiHeadAttentionWrapper, abbiamo specificato il numero di "teste di attenzione" (con num_heads). Se impostiamo num_heads=2, come in questo esempio, otteniamo un tensore con due set di matrici dei vettori di contesto. In ogni matrice dei vettori di contesto, le righe rappresentano i vettori di contesto corrispondenti ai token e le colonne corrispondono alle dimensioni dell'embedding, specificate tramite d_out=4. Concateniamo queste matrici dei vettori di contesto lungo le colonne. Poiché abbiamo due teste e dimensioni dell'embedding pari a 2, le dimensioni finali dell'embedding saranno pari a 2 × 2 = 4.

Ciò si traduce nel seguente tensore, che rappresenta i vettori di contesto:

```
tensor([[[-0.4519,  0.2216,  0.4772,  0.1063],
         [-0.5874,  0.0058,  0.5891,  0.3257],
         [-0.6300, -0.0632,  0.6202,  0.3860],
         [-0.5675, -0.0843,  0.5478,  0.3589],
         [-0.5526, -0.0981,  0.5321,  0.3428],
         [-0.5299, -0.1081,  0.5077,  0.3493]],

        [[-0.4519,  0.2216,  0.4772,  0.1063],
         [-0.5874,  0.0058,  0.5891,  0.3257],
         [-0.6300, -0.0632,  0.6202,  0.3860],
         [-0.5675, -0.0843,  0.5478,  0.3589],
         [-0.5526, -0.0981,  0.5321,  0.3428],
         [-0.5299, -0.1081,  0.5077,  0.3493]]],
        grad_fn=<CatBackward0>
)
context_vecs.shape: torch.Size([2, 6, 4])
```

La prima dimensione del tensore `context_vecs` risultante è 2, poiché abbiamo due testi di input (i testi di input sono duplicati, quello è il motivo per cui i vettori di contesto sono esattamente identici). La seconda dimensione si riferisce ai 6 token di ogni input. La terza dimensione si riferisce all'embedding quadridimensionale di ogni token.

Esercizio 3.2 Restituzione di vettori di embedding bidimensionali

Modificate gli argomenti di input della chiamata a `MultiHeadAttentionWrapper(..., num_heads=2)` in modo tale che i vettori di contesto di output siano bidimensionali invece che quadridimensionali, mantenendo l'impostazione `num_heads=2`. Suggerimento: non è necessario modificare l'implementazione della classe; basta modificare uno degli altri argomenti di input.

A questo punto, abbiamo implementato un `MultiHeadAttentionWrapper` che combina più moduli di attenzione single-head. Tuttavia, questi vengono elaborati in sequenza tramite `[head(x) for head in self.heads]` nel metodo `forward`. Possiamo migliorare questa implementazione elaborando le "teste" in parallelo. Un modo per ottenere questo risultato consiste nel calcolare simultaneamente gli output per tutte le "teste di attenzione" tramite una moltiplicazione di matrici.

Implementazione dell’attenzione multi-head con suddivisione del peso

Finora, abbiamo creato un `MultiHeadAttentionWrapper` per implementare l’attenzione multi-head semplicemente impilando più moduli di attenzione single-head. L’abbiamo fatto mediante l’istanziazione e la combinazione di più oggetti `CausalAttention`.

Invece di mantenere due classi distinte, `MultiHeadAttentionWrapper` e `CausalAttention`, possiamo combinare questi concetti in una singola classe `MultiHeadAttention`. Inoltre, oltre a unire il codice di `MultiHeadAttentionWrapper` col codice di `CausalAttention`, apporteremo altre modifiche per implementare l’attenzione multi-head in modo più efficiente.

In `MultiHeadAttentionWrapper`, le “teste” vengono implementate creando una serie di oggetti `CausalAttention` (`self.heads`), ciascuno dei quali rappresenta una “testa di attenzione”, distinta. La classe `CausalAttention` esegue il meccanismo di attenzione indipendentemente e poi i risultati di ogni “testa” vengono concatenati. Al contrario, la seguente classe `MultiHeadAttention` integra la funzionalità multi-head in una singola classe. Divide l’input in più “teste” rimodellando i tensori di query, chiavi e valori e poi combina i risultati di queste “teste” dopo aver calcolato l’attenzione.

Ma, prima di approfondire l’argomento, esaminiamo la classe `MultiHeadAttention`.

Listato 3.5 Una classe di attenzione multi-head efficiente.

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length, dropout, num_heads,
qkv_bias=False):
        super().__init__()
        assert (d_out % num_heads == 0), "d_out must be divisible by num_heads"

        self.d_out = d_out
```

```
        self.num_heads = num_heads
        self.head_dim = d_out // num_heads (1)
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.out_proj = nn.Linear(d_out, d_out) (2)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer("mask",
            torch.triu(torch.ones(context_length, context_length),
            diagonal=1)
        )

    def forward(self, x):
        b, num_tokens, d_in = x.shape
        keys = self.W_key(x) (3)
        queries = self.W_query(x) (3)
        values = self.W_value(x) (3)

        keys = keys.view(b, num_tokens, self.num_heads, self.head_dim) (4)
        values = values.view(b, num_tokens, self.num_heads, self.head_dim) (4)
        queries = queries.view(b, num_tokens, self.num_heads, self.head_dim) (4)

        keys = keys.transpose(1, 2) (5)
        queries = queries.transpose(1, 2) (5)
        values = values.transpose(1, 2) (5)

        attn_scores = queries @ keys.transpose(2, 3) (6)
        mask_bool = self.mask.bool()[:num_tokens, :num_tokens] (7)

        attn_scores.masked_fill_(mask_bool, -torch.inf) (8)

        attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
        attn_weights = self.dropout(attn_weights)

        context_vec = (attn_weights @ values).transpose(1, 2) (9)

        context_vec = context_vec.contiguous().view(b, num_tokens, self.d_out)
(10)
        context_vec = self.out_proj(context_vec) (11)
        return context_vec
```

(1) Riduce la dimensione della proiezione per adattarla alla dimensione di output desiderata.

(2) Utilizza un layer `Linear` per combinare gli output delle “teste”.

(3) Forma del tensore: `(b, num_tokens, d_out)`.

(4) Suddividiamo implicitamente la matrice aggiungendo una dimensione `num_heads`. Quindi consideriamo l’ultima dim: `(b, num_tokens, d_out) -> (b, num_tokens, num_heads, head_dim)`.

(5) Traspone dalla forma `(b, num_tokens, num_heads, head_dim)` alla forma `(b, num_heads, num_tokens, head_dim)`.

(6) Calcola il prodotto scalare per ogni “testa”.
(7) Maschere troncate al numero di token.
(8) Utilizza la maschera per compilare i punteggi di attenzione.
(9) Forma del tensore: `(b, num_tokens, n_heads, head_dim)`.
(10) Combina le “teste”, dove `self.d_out = self.num_heads * self.head_dim`.
(11) Aggiunge una proiezione lineare facoltativa.

Anche se la rimodellazione (`.view`) e la trasposizione (`.transpose`) dei tensori all’interno della classe `MultiHeadAttention` sembra molto complicata, la classe `MultiHeadAttention` implementa lo stesso concetto del precedente `MultiHeadAttentionWrapper`.

A livello generale, nella precedente `MultiHeadAttentionWrapper`, abbiamo impilato più layer di attenzione single-head, che abbiamo combinato in un layer di attenzione multi-head. La classe `MultiHeadAttention` adotta un approccio integrato: inizia con un livello multi-head e poi divide internamente questo livello nelle singole “teste di attenzione”, come mostra la Figura 3.26.

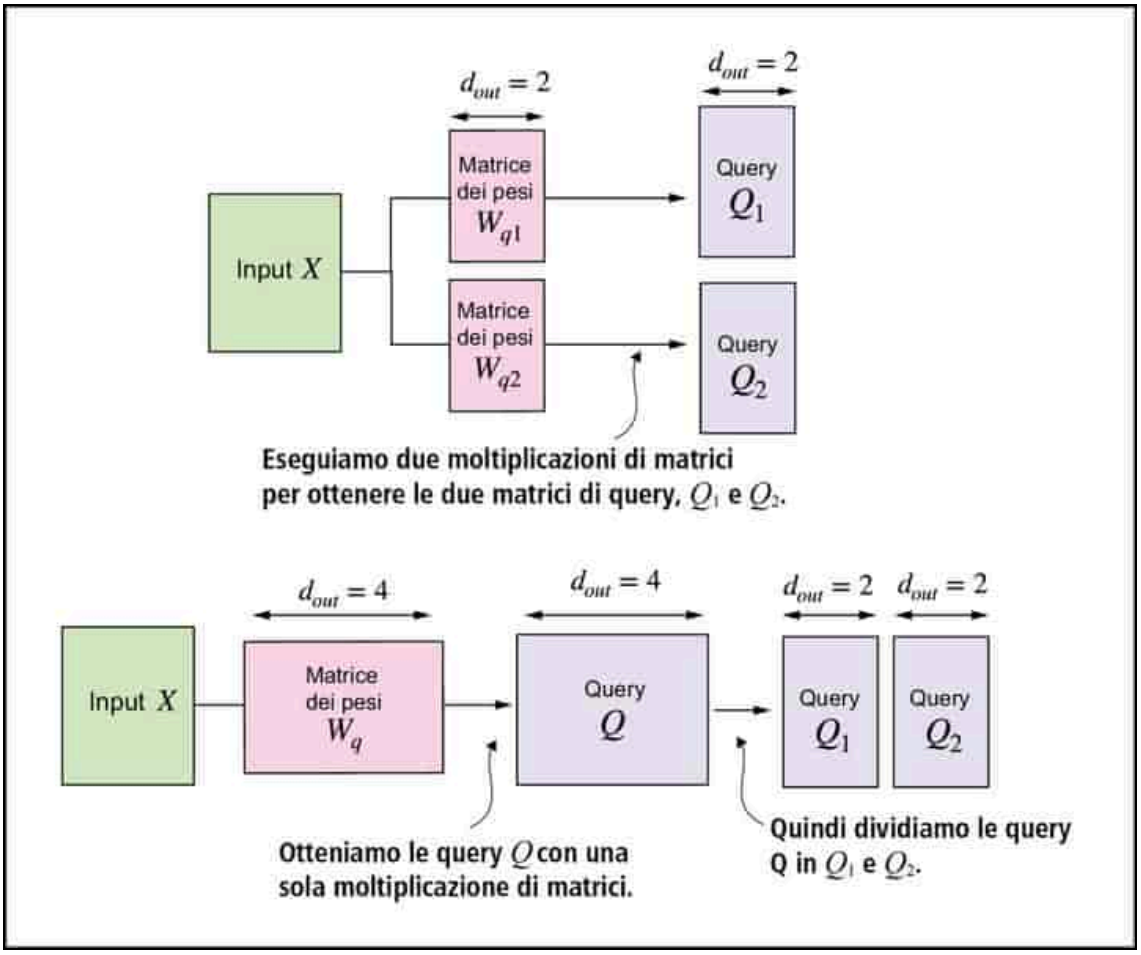


Figura 3.26 Nella classe MultiHeadAttentionWrapper con due “teste di attenzione”, abbiamo inizializzato due matrici dei pesi, W_{q1} e $W_{q2,}$ e calcolato due matrici di query, Q_1 e Q_2 (in alto). Nella classe MultiheadAttention, inizializziamo una matrice dei pesi più grande W_q, eseguiamo una moltiplicazione fra matrici con gli input per ottenere una matrice di query Q, quindi dividiamo la matrice di query in Q_1 e Q_2 (in basso). Facciamo lo stesso per le chiavi e i valori, che non sono mostrati solo per brevità.

La suddivisione dei tensori di query, chiavi e valori viene ottenuta tramite operazioni di rimodellamento e trasposizione dei tensori utilizzando i metodi `.view` e `.transpose` di PyTorch. L’input viene prima trasformato (tramite layer `Linear` per query, chiavi e valori) e poi rimodellato per rappresentare più “teste”.

L’operazione chiave consiste nel dividere la dimensione `d_out` in `num_heads` e `head_dim`, dove `head_dim = d_out / num_heads`. Questa suddivisione è ottenuta utilizzando il metodo `.view`: un tensore di dimensioni `(b, num_tokens, d_out)` viene rimodellato in base alle dimensioni `(b, num_tokens, num_heads, head_dim)`.

I tensori vengono quindi trasposti per portare le dimensioni `num_heads` alle dimensioni `num_tokens`, con conseguente forma `(b, num_heads, num_tokens, head_dim)`. Questa trasposizione è fondamentale per allineare correttamente le query, le chiavi e i valori tra le diverse “teste” ed eseguire in modo efficiente le moltiplicazioni di matrici in batch.

Per illustrare questa moltiplicazione di matrici in batch, supponiamo di avere il seguente tensore:

```
a = torch.tensor([[[[0.2745, 0.6584, 0.2775, 0.8573], (1)
                    [0.8993, 0.0390, 0.9268, 0.7388],
                    [0.7179, 0.7058, 0.9156, 0.4340]],

                   [[0.0772, 0.3565, 0.1479, 0.5331],
                    [0.4066, 0.2318, 0.4545, 0.9737],
                    [0.4606, 0.5159, 0.4220, 0.5786]]]]
)
```

(1) La forma di questo tensore è `(b, num_heads, num_tokens, head_dim) = (1, 2, 3, 4)`.

Ora eseguiamo una moltiplicazione di matrici tra il tensore e una vista del tensore in cui abbiamo trasposto le ultime due dimensioni, `num_tokens` e `head_dim`:

```
print(a @ a.transpose(2, 3))
```

Il risultato è il seguente:

```
tensor([[[[1.3208, 1.1631, 1.2879],
          [1.1631, 2.2150, 1.8424],
          [1.2879, 1.8424, 2.0402]],

         [[0.4391, 0.7003, 0.5903],
          [0.7003, 1.3737, 1.0620],
          [0.5903, 1.0620, 0.9912]]]]
)
```

In questo caso, l’implementazione della moltiplicazione di matrici di PyTorch gestisce il tensore di input quadridimensionale in modo che la moltiplicazione venga eseguita tra le ultime due dimensioni `(num_tokens, head_dim)` e poi ripetuta per le singole “teste”.

Per esempio, quella precedente diventa un modo più compatto per calcolare la moltiplicazione della matrice per ogni “testa”, separatamente:

```
first_head = a[0, 0, :, :]
first_res = first_head @ first_head.T
print("First head:\n", first_res)

second_head = a[0, 1, :, :]
second_res = second_head @ second_head.T
print("\nSecond head:\n", second_res)
```

I risultati sono esattamente gli stessi che abbiamo ottenuto utilizzando la moltiplicazione di matrici in batch `print(a @ a.transpose(2, 3))`:

```
First head:
 tensor([[1.3208, 1.1631, 1.2879],
         [1.1631, 2.2150, 1.8424],
         [1.2879, 1.8424, 2.0402]]
 )
Second head:
 tensor([[0.4391, 0.7003, 0.5903],
         [0.7003, 1.3737, 1.0620],
         [0.5903, 1.0620, 0.9912]]
 )
```

Continuando con `MultiHeadAttention`, dopo aver calcolato i pesi di attenzione e i vettori di contesto, i vettori di contesto di tutte le “teste” vengono trasposti nuovamente nella forma `(b, num_tokens, num_heads, head_dim)`. Questi vettori vengono quindi rimodellati (appiattiti) nella forma `(b, num_tokens, d_out)`, combinando efficacemente gli output di tutte le “teste”.

Inoltre, dopo aver combinato le “teste” abbiamo aggiunto a `MultiHeadAttention` un layer di proiezione di output (`self.out_proj`), che non è presente nella classe `CausalAttention`. Questo layer di proiezione di

output non è strettamente necessario (vedi l'Appendice B per maggiori dettagli), ma è comunemente utilizzato in molte architetture LLM, motivo per cui l'ho aggiunto qui per completezza.

Anche se la classe `MultiHeadAttention` sembra più complicata rispetto a `MultiHeadAttentionWrapper` a causa dell'ulteriore rimodellamento e trasposizione dei tensori, è più efficiente. Il motivo è che ci serve soltanto una moltiplicazione di matrici per calcolare le chiavi, per esempio, `keys = self.W_key(x)` (lo stesso vale per le query e i valori). In `MultiHeadAttentionWrapper`, abbiamo dovuto ripetere questa moltiplicazione di matrici, che è uno dei passi più costosi computazionalmente, per ogni "testa di attenzione".

La classe `MultiHeadAttention` può essere utilizzata in modo simile alle classi `SelfAttention` e `CausalAttention` implementate in precedenza:

```
torch.manual_seed(123)
batch_size, context_length, d_in = batch.shape
d_out = 2
mha = MultiHeadAttention(d_in, d_out, context_length, 0.0, num_heads=2)
context_vecs = mha(batch)
print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)
```

I risultati mostrano che le dimensioni dell'output sono controllate direttamente dall'argomento `d_out`:

```
tensor([[[0.3190, 0.4858],
         [0.2943, 0.3897],
         [0.2856, 0.3593],
         [0.2693, 0.3873],
         [0.2639, 0.3928],
         [0.2575, 0.4028]],

        [[0.3190, 0.4858],
         [0.2943, 0.3897],
         [0.2856, 0.3593],
         [0.2693, 0.3873],
         [0.2639, 0.3928],
         [0.2575, 0.4028]]],
      grad_fn=<ViewBackward0>
)
context_vecs.shape: torch.Size([2, 6, 2])
```

Abbiamo implementato la classe `MultiHeadAttention` che useremo quando implementeremo e addestreremo il modello LLM. Notate che,

anche se il codice è pienamente funzionale, ho usato dimensioni dell'embedding e numeri di "teste di attenzione" relativamente piccoli, con lo scopo di mantenere leggibili gli output.

Per fare un confronto, il modello GPT-2 più piccolo (117 milioni di parametri) impiega 12 teste di attenzione e le dimensioni dell'embedding del vettore del contesto sono pari a 768. Il modello GPT-2 più grande (1,5 miliardi di parametri) impiega 25 teste di attenzione e le dimensioni dell'embedding del vettore del contesto sono pari a 1600. Le dimensioni dell'embedding degli input (token) e degli embedding del contesto sono le stesse (`d_in = d_out`).

Esercizio 3.3 Inizializzazione dei moduli di attenzione di dimensioni pari a quelle di GPT-2

Utilizzando la classe `MultiHeadAttention`, inizializzate un modulo di attenzione multi-head che abbia lo stesso numero di teste di attenzione del modello GPT-2 più piccolo (ovvero 12). Assicuratevi inoltre di utilizzare le dimensioni degli embedding di input e output di GPT-2 (768 dimensioni). Notate che il modello GPT-2 più piccolo supporta una lunghezza di contesto pari a 1024 token.

Riepilogo

- I meccanismi di attenzione trasformano gli elementi di input in rappresentazioni vettoriali avanzate del contesto che incorporano informazioni su tutti gli input.
- Un meccanismo di auto-attenzione calcola la rappresentazione del vettore del contesto come una somma ponderata degli input.
- In un meccanismo di attenzione semplificato, i pesi di attenzione vengono calcolati tramite prodotti scalari.
- Il prodotto scalare è un metodo conciso per moltiplicare due vettori elemento per elemento e poi sommare i prodotti.
- Le moltiplicazioni di matrici, sebbene non siano strettamente necessarie, ci aiutano a implementare i calcoli in modo più

efficiente e compatto, sostituendo i cicli `for` nidificati.

- Nei meccanismi di auto-attenzione utilizzati nei modelli LLM (attenzione basata sul prodotto scalare), includiamo delle matrici di pesi addestrabili per calcolare le trasformazioni intermedie degli input: query, valori e chiavi.
- Quando lavoriamo con modelli LLM che leggono e generano testo da sinistra a destra, aggiungiamo una maschera di attenzione causale per impedire al modello LLM di accedere ai token successivi.
- Oltre alle maschere di attenzione causale per azzerare i pesi di attenzione, possiamo aggiungere una maschera di dropout per ridurre l'overfitting nei modelli LLM.
- I moduli di attenzione nei modelli LLM basati su transformer coinvolgono molteplici istanze di attenzione causale: un'attenzione multi-head.
- Possiamo creare un modulo di attenzione multi-head impilando più istanze di moduli di attenzione causale.
- Un metodo più efficiente per creare moduli di attenzione multi-head usa le moltiplicazioni di matrici in batch.

Capitolo 4

Implementazione di un modello GPT da zero per generare testo

Abbiamo parlato del meccanismo di *attenzione multi-head*, uno dei componenti principali dei modelli LLM e lo abbiamo programmato. Ora, programmeremo gli altri elementi costitutivi di un modello LLM e li assembleremo a formare un modello di tipo GPT che nel prossimo capitolo addestreremo per generare testo di tipo umano.

L'architettura LLM cui fa riferimento la Figura 4.1, è composta da diversi blocchi costitutivi. Inizieremo con una panoramica dell'architettura del modello, prima di esaminare in modo più dettagliato i singoli componenti.

Programmazione di un'architettura LLM

I modelli LLM, come GPT (che sta per *Generative Pretrained Transformer*), sono grandi architetture a reti neurali profonde progettate per generare nuovo testo, una parola (o token) alla volta. Tuttavia, nonostante le loro dimensioni, l'architettura del modello è meno complicata di quanto si possa pensare, poiché molti dei suoi componenti sono semplicemente ripetuti, come vedremo più avanti. La Figura 4.2 fornisce una panoramica di un modello LLM di tipo GPT, con i suoi componenti principali evidenziati.

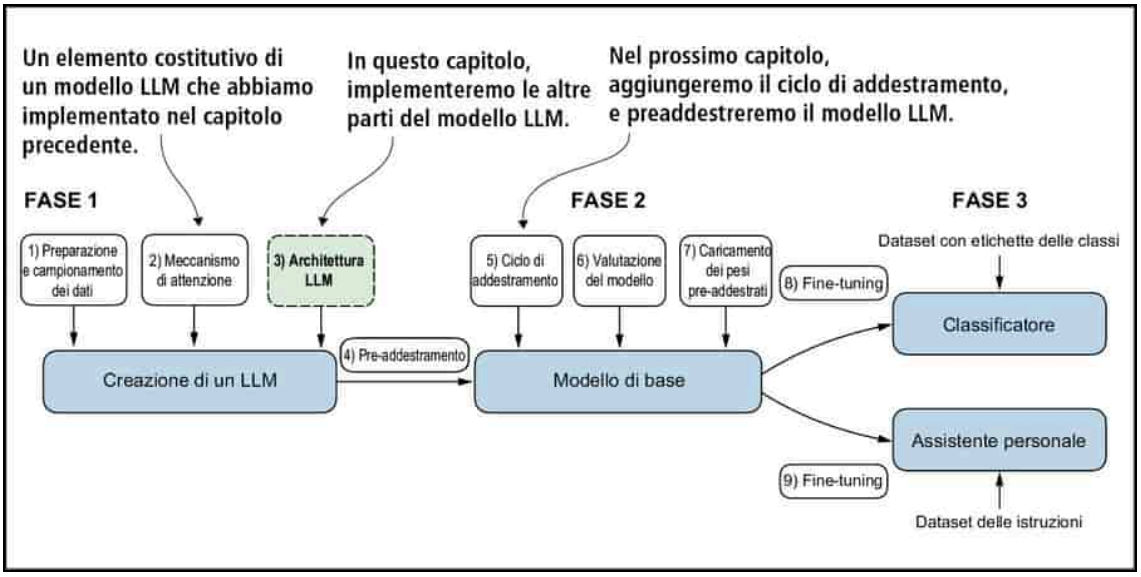


Figura 4.1 Le tre fasi principali della programmazione di un modello LLM. Questo capitolo si concentra sul Passo 3 della Fase 1: implementazione dell'architettura LLM.

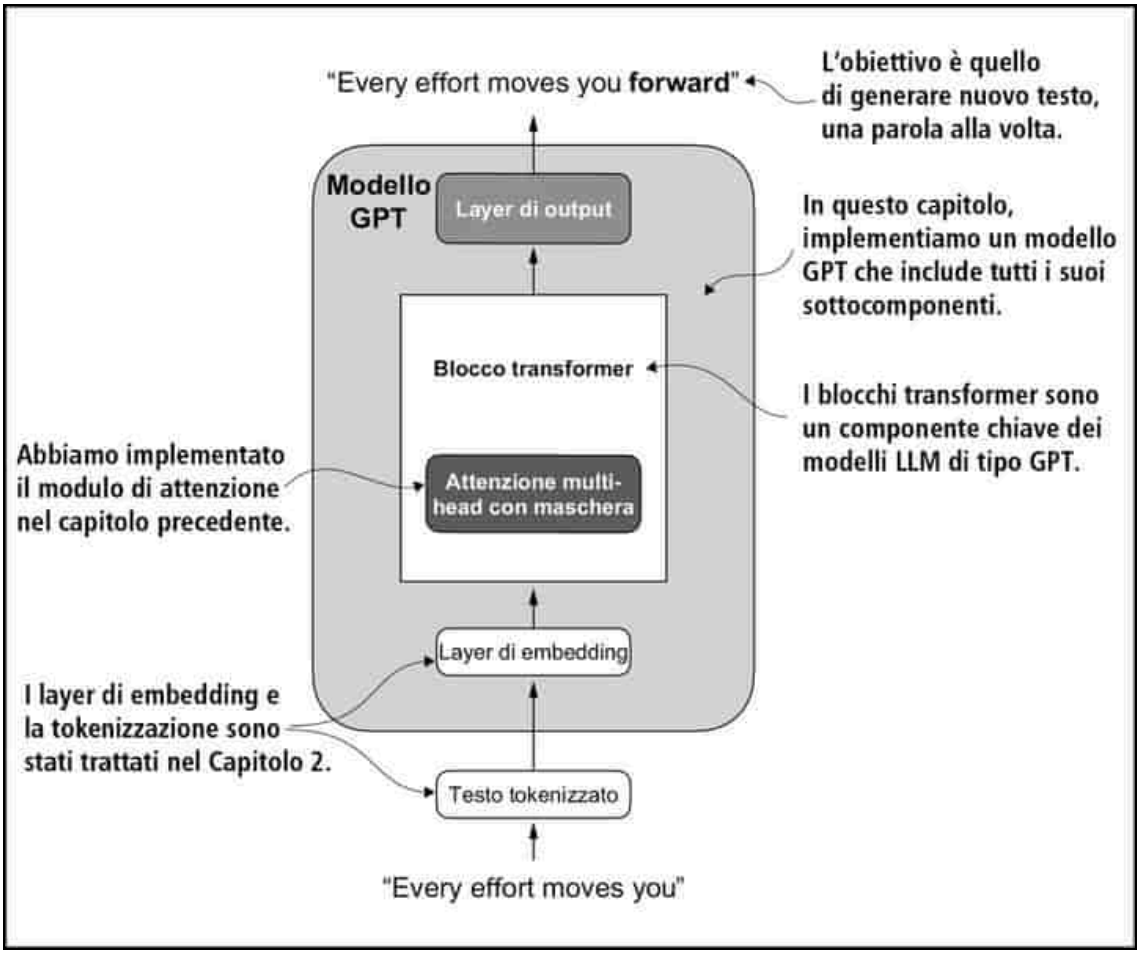


Figura 4.2 Un modello GPT. Oltre ai layer di embedding, è costituito da uno o più blocchi transformer contenenti il modulo di attenzione multi-head con maschera che abbiamo implementato in precedenza.

Abbiamo già trattato diversi aspetti dell'architettura LLM, come la tokenizzazione e l'embedding dell'input, e il modulo di attenzione multi-head con maschera. Ora, vogliamo implementare la struttura fondamentale del modello GPT, inclusi i suoi *blocchi transformer*, che in seguito addestreremo per generare testo di tipo umano.

In precedenza, per semplicità abbiamo utilizzato dimensioni dell'embedding più piccole, per assicurare che i concetti e gli esempi potessero rientrare in una pagina. Ora, stiamo passando alle dimensioni di un piccolo modello GPT-2, in particolare alla versione

più piccola, con 124 milioni di parametri, come descritto in *Language Models Are Unsupervised Multitask Learners*, di Radford et al. (https://mng.bz/yoBq). Notate che, mentre il report originale menziona 117 milioni di parametri, questa cifra è stata corretta, successivamente. Nel Capitolo 6, ci concentreremo sul caricamento di pesi pre-addestrati nella nostra implementazione e sul suo adattamento per modelli GPT-2 più grandi, con 345, 762 e 1.542 milioni di parametri.

Nel contesto del deep learning e dei modelli LLM come GPT, il termine "parametri" si riferisce ai pesi addestrabili del modello. Questi pesi sono essenzialmente le variabili interne del modello, che vengono regolate e ottimizzate durante il processo di addestramento per minimizzare una determinata funzione loss. Questa ottimizzazione consente al modello di apprendere dai dati di addestramento.

Per esempio, in un layer della rete neurale rappresentato da una matrice (o tensore) di pesi di 2.048 × 2.048 dimensioni, ogni elemento di questa matrice è un parametro. Poiché ci sono 2.048 righe e 2.048 colonne, il numero totale di parametri in questo layer è 2.048 moltiplicato per 2.048, che equivale a 4.194.304 parametri.

GPT-2 vs GPT-3

Ci stiamo concentrando su GPT-2 perché OpenAI ha reso pubblici i pesi del modello pre-addestrato, che caricheremo nella nostra implementazione nel Capitolo 6. GPT-3 è fondamentalmente lo stesso in termini di architettura del modello, tranne per il fatto che è scalato dagli 1,5 miliardi di parametri di GPT-2 ai 175 miliardi di parametri di GPT-3, ed è addestrato su più dati. Al momento della stesura, i pesi per GPT-3 non sono di pubblico dominio. GPT-2, inoltre, è una scelta migliore per imparare a implementare modelli LLM, poiché può essere eseguito su un singolo computer portatile, mentre GPT-3 richiede un cluster di GPU per l'addestramento e l'inferenza. Secondo Lambda Labs (https://lambdalabs.com), ci vorrebbero 355 anni per addestrare GPT-3 su una singola GPU di datacenter V100 e 665 anni su una singola GPU RTX 8000 di livello consumer.

Specifichiamo la configurazione del piccolo modello GPT-2 tramite il seguente dizionario Python, che utilizzeremo nei prossimi esempi di codice:

```
GPT_CONFIG_124M = {
    "vocab_size": 50257,     # Dimensioni vocabolario
    "context_length": 1024,  # Lunghezza contesto
    "emb_dim": 768,          # Dimensioni embedding
    "n_heads": 12,           # Numero di "teste di attenzione"
    "n_layers": 12,          # Numero di layer
    "drop_rate": 0.1,        # Tasso di dropout
    "qkv_bias": False        # Bias Query-Chiave-Valore
}
```

Nel dizionario `GPT_CONFIG_124M` utilizziamo variabili dal nome sintetico per chiarezza e per evitare che le righe di codice divengano troppo lunghe.

- `vocab_size` si riferisce a un vocabolario di 50.257 parole, come quello utilizzato dal tokenizer BPE (vedi il Capitolo 2).
- `context_length` indica il numero massimo di token di input che il modello può gestire tramite gli embedding posizionali (vedi il Capitolo 2).
- `emb_dim` rappresenta le dimensioni dell'embedding, trasformando ogni token in un vettore 768-dimensionale.
- `n_heads` indica il numero di "teste di attenzione" nel meccanismo multi-head (vedi il Capitolo 3).
- `n_layers` specifica il numero di blocchi transformer del modello, di cui parleremo fra poco.
- `drop_rate` indica l'intensità del meccanismo di dropout (`0.1` implica un dropout casuale del 10 percento delle unità nascoste) per evitare l'overfitting (vedi il Capitolo 3).
- `qkv_bias` determina se includere un vettore di bias nei layer `Linear` multi-head per i calcoli di query, chiavi e valori. Inizialmente lo disabiliteremo, seguendo le norme dei modelli LLM odierni, ma

lo rivisiteremo più avanti quando caricheremo nel modello i pesi di GPT-2 pre-addestrati di OpenAI (vedi il Capitolo 6).

Utilizzando questa configurazione, implementeremo un'architettura GPT fittizia (`DummyGPTModel`), come mostra la Figura 4.3. Ciò ci fornirà una visione d'insieme di come tutti i pezzi si incastrano e su quali altri componenti dobbiamo programmare per assemblare l'architettura completa del modello GPT.

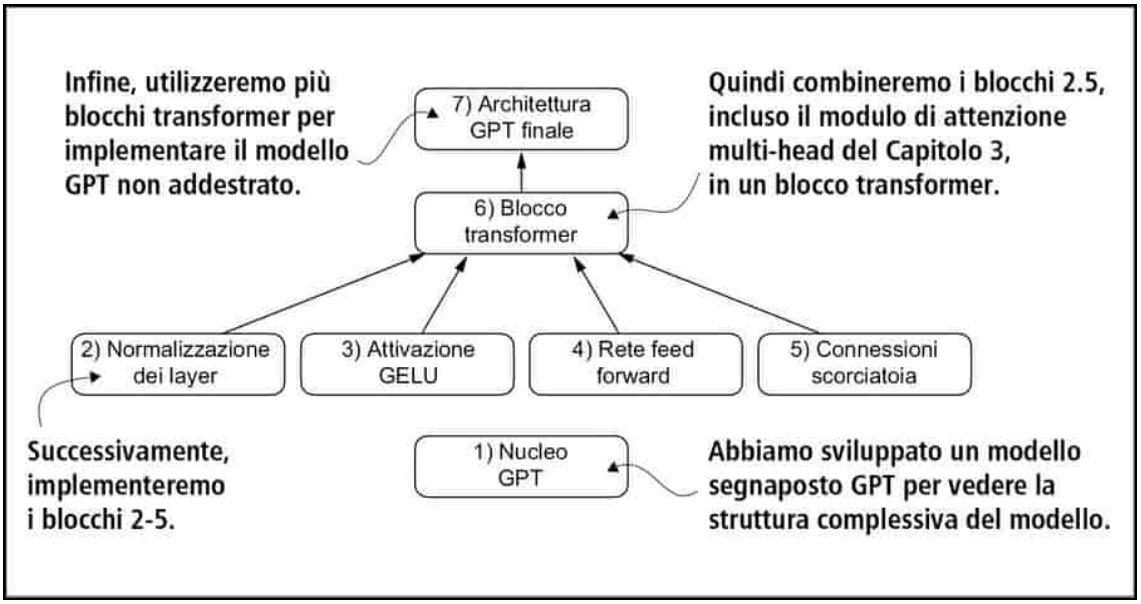


Figura 4.3 L'ordine in cui programmiamo l'architettura GPT. Iniziamo col nucleo di GPT, un'architettura fittizia, prima di arrivare ai singoli pezzi e assemblarli in un blocco transformer per l'architettura GPT finale.

I blocchi numerati nella Figura 4.3 illustrano l'ordine in cui affrontiamo i singoli concetti richiesti per programmare l'architettura GPT finale. Inizieremo col Passo 1, un nucleo GPT fittizio che chiameremo `DummyGPTModel`.

Listato 4.1 Una classe fittizia per l'architettura del modello GPT.

```
import torch
import torch.nn as nn
```

```
class DummyGPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])
        self.trf_blocks = nn.Sequential(
            *[DummyTransformerBlock(cfg)
                for _ in range(cfg["n_layers"])
            ]
        )
        self.final_norm = DummyLayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(cfg["emb_dim"], cfg["vocab_size"], bias=False)

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)
        pos_embeds = self.pos_emb(torch.arange(seq_len, device=in_idx.device))
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits

class DummyTransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()

    def forward(self, x):
        return x

class DummyLayerNorm(nn.Module):
    def __init__(self, normalized_shape, eps=1e-5):
        super().__init__()

    def forward(self, x):
        return x
```

La classe `DummyGPTModel` di questo codice definisce una versione semplificata di un modello di tipo GPT che utilizza il modulo PyTorch per reti neurali (`nn.Module`). L'architettura del modello nella classe `DummyGPTModel` è costituita da token, embedding posizionali, dropout, una serie di blocchi transformer (`DummyTransformerBlock`), un layer di normalizzazione finale (`DummyLayerNorm`) e un layer di output lineare (`out_head`). La configurazione viene passata tramite un dizionario Python, per esempio, il dizionario `GPT_CONFIG_124M` che abbiamo creato in precedenza.

Il metodo `forward` descrive il flusso dei dati attraverso il modello: calcola gli embedding dei token e li posiziona per gli indici di input, applica il dropout, elabora i dati attraverso i blocchi transformer, applica la normalizzazione e infine produce i risultati col layer di output lineare.

Il codice riportato nel Listato 4.1 è già funzionante. Tuttavia, per ora, notate che utilizziamo dei “segnaposto” (`DummyLayerNorm` e `DummyTransformerBlock`) per il blocco transformer e il layer di normalizzazione, che svilupperemo in seguito.

Ora, prepareremo i dati di input e inizializzeremo un nuovo modello GPT per illustrarne l’utilizzo. Basandoci sulla nostra programmazione del tokenizer (vedi il Capitolo 2), consideriamo ora una panoramica di alto livello di come i dati fluiscono in un modello GPT, come mostra la Figura 4.4.

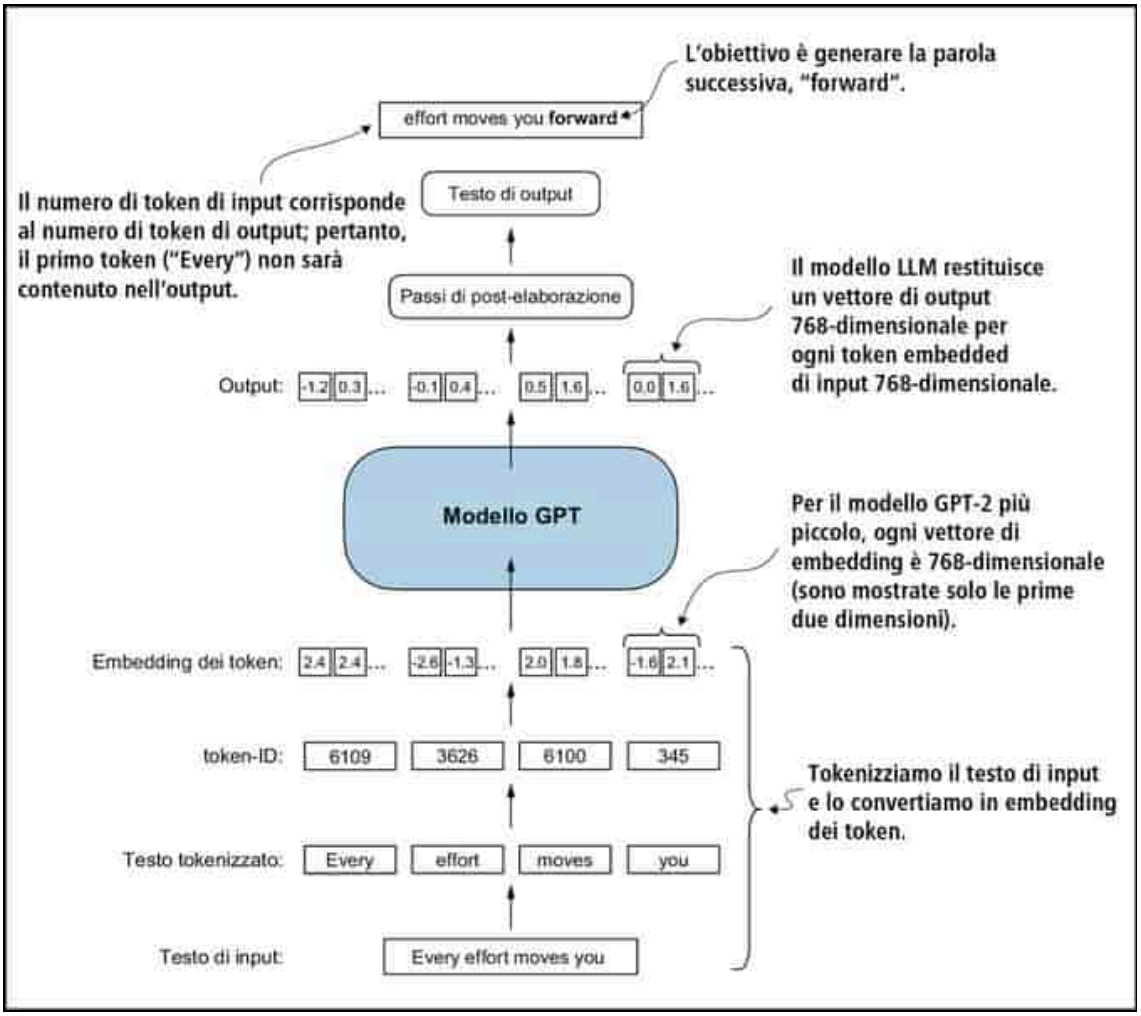


Figura 4.4 Una panoramica generale che mostra come i dati di input vengono tokenizzati, sottoposti a embedding e immessi nel modello GPT. Notate che nella nostra classe DummyGPTClass programmata in precedenza, l'embedding dei token viene gestito all'interno del modello GPT. Nei modelli LLM, le dimensioni del token di input sottoposto a embedding corrispondono in genere alle dimensioni dell'output. Gli embedding di output, qui, rappresentano i vettori di contesto (vedi il Capitolo 3).

Per implementare questi passi, tokenizziamo un batch composto da due input di testo utilizzando il tokenizer tiktoken del Capitolo 2:

```
import tiktoken

tokenizer = tiktoken.get_encoding("gpt2")
batch = []
txt1 = "Every effort moves you"
```

```
txt2 = "Every day holds a"

batch.append(torch.tensor(tokenizer.encode(txt1)))
batch.append(torch.tensor(tokenizer.encode(txt2)))
batch = torch.stack(batch, dim=0)
print(batch)
```

I token-ID risultanti per i due testi sono i seguenti:

```
tensor([[6109, 3626, 6100,  345], (1)
        [6109, 1110, 6622,  257]] (1)
)
```

(1) La prima riga corrisponde al primo testo, la seconda riga al secondo testo.

Ora, inizializziamo una nuova istanza di `DummyGPTModel` da 124 milioni di parametri e le forniamo il `batch` tokenizzato:

```
torch.manual_seed(123)
model = DummyGPTModel(GPT_CONFIG_124M)
logits = model(batch)
print("Output shape:", logits.shape)
print(logits)
```

Gli output del modello, comunemente denominati *logit*, sono i seguenti:

```
Output shape: torch.Size([2, 4, 50257])
tensor([[[-1.2034,  0.3201, -0.7130, ..., -1.5548, -0.2390, -0.4667],
         [-0.1192,  0.4539, -0.4432, ...,  0.2392,  1.3469,  1.2430],
         [ 0.5307,  1.6720, -0.4695, ...,  1.1966,  0.0111,  0.5835],
         [ 0.0139,  1.6755, -0.3388, ...,  1.1586, -0.0435, -1.0400]],

        [[-1.0908,  0.1798, -0.9484, ..., -1.6047,  0.2439, -0.4530],
         [-0.7860,  0.5581, -0.0610, ...,  0.4835, -0.0077,  1.6621],
         [ 0.3567,  1.2698, -0.6398, ..., -0.0162, -0.1296,  0.3717],
         [-0.2407, -0.7349, -0.5102, ...,  2.0057, -0.3694,  0.1814]]],
        grad_fn=<UnsafeViewBackward0>
)
```

Il tensore di output ha due righe, corrispondenti ai due campioni di testo. Ogni campione di testo è costituito da quattro token; ogni token è un vettore 50.257-dimensionale, che corrispondono alle dimensioni del vocabolario del tokenizer.

L’embedding è 50.257-dimensionale, perché ciascuna di queste dimensioni fa riferimento a un token univoco nel vocabolario. Quando implementeremo il codice di post-elaborazione,

convertiremo questi vettori 50.257-dimensionali in token-ID, che possiamo poi decodificare in parole.

Ora che abbiamo esaminato l'intera architettura GPT e i suoi input e output, programmiamo i singoli elementi, iniziando con la classe del layer di normalizzazione che sostituirà il `DummyLayerNorm` del codice precedente.

Normalizzazione delle attivazioni con la normalizzazione dei layer

L'addestramento delle reti neurali profonde con molti layer può rivelarsi impegnativo a causa di problemi come la sparizione o l'esplosione dei gradienti. Questi problemi portano a dinamiche di addestramento instabili, e non consentono alla rete di regolare efficacemente i propri pesi, il che significa che il processo di apprendimento fatica a trovare un set di parametri (pesi) per la rete neurale che minimizzi la funzione loss. In altre parole, la rete ha difficoltà ad apprendere i pattern esistenti nei dati a un livello che le consentirebbe di fare previsioni o decisioni accurate.

> **NOTA**
> Se non avete familiarità con l'addestramento delle reti neurali e con i concetti dei gradienti, potete trovare una breve introduzione a questi concetti nell'Appendice A. Tuttavia, non è richiesta una profonda comprensione matematica dei gradienti per seguire i contenuti di questo libro.

Implementiamo ora la *normalizzazione dei layer* per migliorare la stabilità e l'efficienza dell'addestramento della rete neurale. L'idea alla base della normalizzazione dei layer consiste nel regolare le attivazioni (output) di un layer della rete neurale per avere media 0 e varianza 1, o varianza unitaria. Questa regolazione accelera la convergenza dei pesi e garantisce un addestramento coerente e affidabile. Nelle architetture GPT-2 e dei transformer odierni, in

genere la normalizzazione dei layer viene applicata prima e dopo il modulo di attenzione multi-head e, come abbiamo visto col layer fittizio `DummyLayerNorm`, prima del layer di output finale. La Figura 4.5 fornisce una panoramica del funzionamento della normalizzazione dei layer.

Possiamo ricreare l'esempio rappresentato nella Figura 4.5 tramite il seguente codice, in cui implementiamo un layer della rete neurale con cinque input e sei output che applichiamo a due esempi di input:

```
torch.manual_seed(123)
batch_example = torch.randn(2, 5) (1)
layer = nn.Sequential(nn.Linear(5, 6), nn.ReLU())
out = layer(batch_example)
print(out)
```

(1) Crea due esempi di addestramento con cinque dimensioni (caratteristiche) ciascuno.

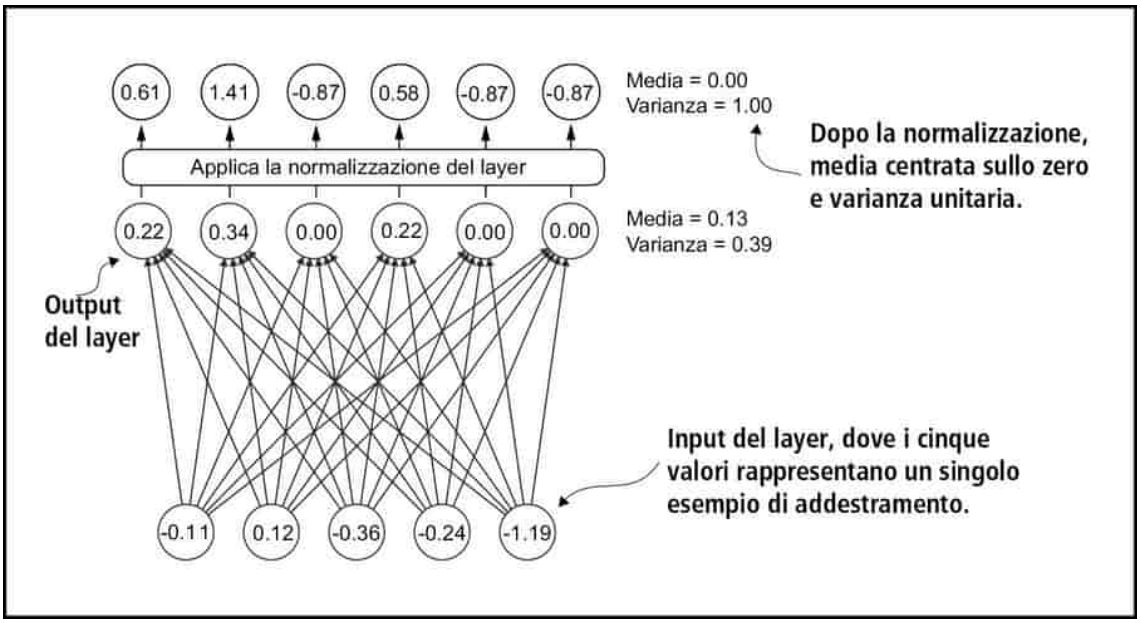


Figura 4.5 Un'illustrazione della normalizzazione di un layer in cui i sei output del layer, chiamati anche attivazioni, vengono normalizzati in modo tale da avere media 0 e varianza 1.

Questo codice produce il seguente tensore, dove la prima riga elenca gli output del layer per il primo input e la seconda riga elenca gli output del layer per la seconda riga:

```
tensor([[0.2260, 0.3470, 0.0000, 0.2216, 0.0000, 0.0000],
        [0.2133, 0.2394, 0.0000, 0.5198, 0.3297, 0.0000]],
       grad_fn=<ReluBackward0>
)
```

Il layer della rete neurale che abbiamo programmato è costituito da un livello `Linear` seguito da una funzione di attivazione non lineare, `ReLU` (abbreviazione di *Rectified Linear Unit*), che è una funzione di attivazione standard nelle reti neurali. Se non avete familiarità con `ReLU`, sappiate che limita gli input negativi a 0, assicurando che un layer produca solo valori positivi, il che spiega perché l'output del layer risultante non contiene alcun valore negativo. Più avanti, useremo un'altra funzione di attivazione più sofisticata.

Prima di applicare a questi output la normalizzazione del layer, esaminiamo la media e la varianza:

```
mean = out.mean(dim=-1, keepdim=True)
var = out.var(dim=-1, keepdim=True)
print("Mean:\n", mean)
print("Variance:\n", var)
```

L'output è il seguente:

```
Mean:
tensor([[0.1324],
        [0.2170]],
       grad_fn=<MeanBackward1>
)
Variance:
tensor([[0.0231],
        [0.0398]],
       grad_fn=<VarBackward0>
)
```

In questo caso, la prima riga del tensore `Mean` contiene la media della prima riga di input, e la seconda riga contiene la media della seconda riga di input.

L'uso di `keepdim=True` in operazioni come il calcolo della media o della varianza assicura che il tensore di output mantenga lo stesso

numero di dimensioni del tensore di input, nonostante il fatto che l'operazione riduce il tensore lungo la dimensione specificata tramite `dim`. Per esempio, senza `keepdim=True`, il valore restituito per la media del tensore sarebbe un vettore bidimensionale `[0,1324, 0,2170]` invece di una matrice di dimensioni 2 × 1, `[[0.1324], [0,2170]]`.

Il parametro `dim` specifica la dimensione lungo la quale deve essere eseguito il calcolo statistico (in questo caso, media o varianza) in un tensore. Come spiega la Figura 4.6, per un tensore bidimensionale (come una matrice), usare `dim=-1` per operazioni come il calcolo della media o della varianza equivale a usare `dim=1`. Questo perché quel `-1` si riferisce all'ultima dimensione del tensore, che in un tensore bidimensionale corrisponde alle colonne. In seguito, quando aggiungeremo al modello GPT la normalizzazione dei layer, che produce tensori tridimensionali con la forma `[batch_size, num_tokens, embedding_size]`, potremo usare `dim=-1` per la normalizzazione nell'ultima dimensione, evitando un passaggio da `dim=1` a `dim=2`.

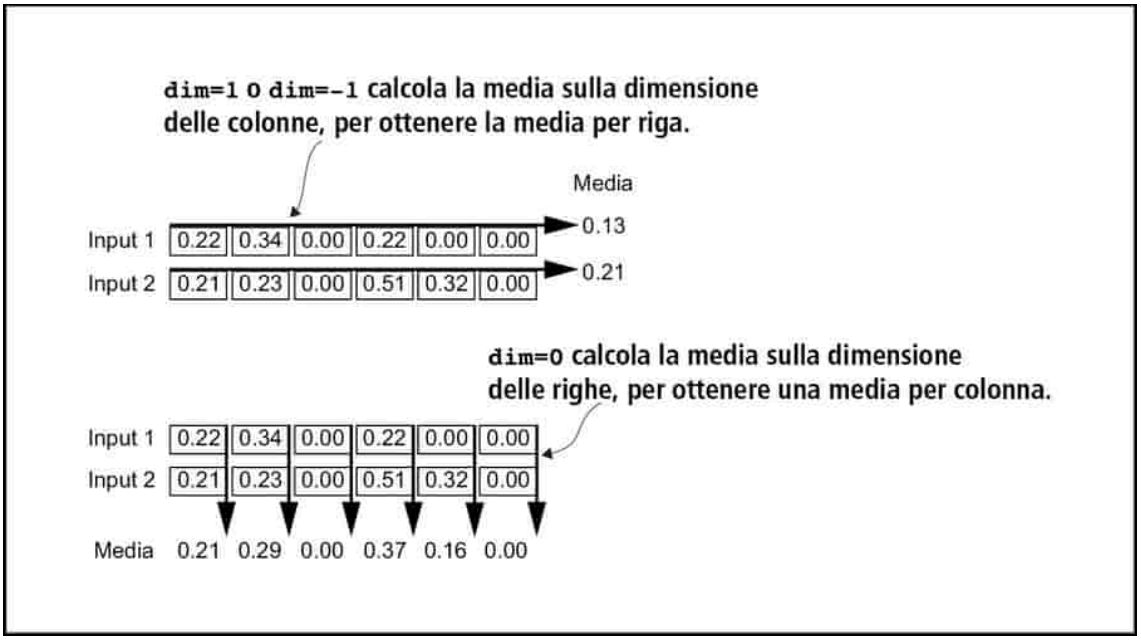


Figura 4.6 Un'illustrazione del parametro dim quando si calcola la media di un tensore. Per esempio, se abbiamo un tensore bidimensionale (una matrice) con dimensioni [rows, columns], utilizzando dim=0 verrà eseguita l'operazione su più righe (verticalmente, come mostrato in basso), con un output che aggrega i dati per colonna. Utilizzando dim=1 o dim=-1 verrà eseguita l'operazione su più colonne (orizzontalmente, come mostrato in alto), con un output che aggrega i dati per riga.

Ora, applichiamo la normalizzazione del layer agli output dei layer ottenuti in precedenza. L'operazione consiste nel sottrarre la media e dividerla per la radice quadrata della varianza (calcolando la deviazione standard):

```
out_norm = (out - mean) / torch.sqrt(var)
mean = out_norm.mean(dim=-1, keepdim=True)
var = out_norm.var(dim=-1, keepdim=True)
print("Normalized layer outputs:\n", out_norm)
print("Mean:\n", mean)
print("Variance:\n", var)
```

Come potete vedere dai risultati, gli output del layer normalizzato, che ora contengono anche valori negativi, hanno media 0 e varianza 1:

```
Normalized layer outputs:
tensor([[ 0.6159, 1.4126, -0.8719, 0.5872, -0.8719, -0.8719],
        [-0.0189, 0.1121, -1.0876, 1.5173, 0.5647, -1.0876]],
       grad_fn=<DivBackward0>
)
Mean:
tensor([[-5.9605e-08],
        [1.9868e-08]],
       grad_fn=<MeanBackward1>
)
Variance:
tensor([[1.],
        [1.]],
       grad_fn=<VarBackward0>
)
```

Notate che il valore `-5.9605e-08` nel tensore di output è il valore in notazione scientifica di $-5{,}9605 \times 10^{-8}$, ovvero 0.000000059605 in forma decimale. Questo valore è molto vicino a 0, ma non è esattamente 0 a causa di piccoli errori numerici che possono accumularsi a causa della limitata precisione con cui i computer rappresentano i numeri.

Per migliorare la leggibilità, possiamo anche disattivare la notazione scientifica nel mostrare i valori dei tensori, impostando `sci_mode` su `False`:

```
torch.set_printoptions(sci_mode=False)
print("Mean:\n", mean)
print("Variance:\n", var)
```

L’output è il seguente:

```
Mean:
tensor([[ 0.0000],
        [ 0.0000]],
       grad_fn=<MeanBackward1>
)
Variance:
tensor([[1.],
        [1.]],
       grad_fn=<VarBackward0>
)
```

Finora abbiamo programmato e applicato la normalizzazione dei layer seguendo un processo progressivo. Ora incapsuliamo questo processo in un modulo PyTorch che in seguito potremo usare nel modello GPT.

Listato 4.2 Una classe di normalizzazione dei layer.

```
class LayerNorm(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.eps = 1e-5
        self.scale = nn.Parameter(torch.ones(emb_dim))
        self.shift = nn.Parameter(torch.zeros(emb_dim))

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)
        norm_x = (x - mean) / torch.sqrt(var + self.eps)
        return self.scale * norm_x + self.shift
```

Questa specifica implementazione della normalizzazione dei layer opera sull'ultima dimensione del tensore di input `x`, che rappresenta la dimensione dell'embedding (`emb_dim`). La variabile `eps` è una piccola costante (*epsilon*) aggiunta alla varianza per impedire la divisione per zero durante la normalizzazione. `scale` (scala) e `shift` (spostamento) sono due parametri addestrabili (delle stesse dimensioni dell'input) che il modello LLM regola automaticamente durante l'addestramento, se si determina che ciò migliorerebbe le prestazioni del modello nel suo compito di addestramento. Ciò consente al modello di apprendere la scala e lo spostamento che meglio si adattano ai dati che sta elaborando.

Varianza col bias

Nel metodo di calcolo della varianza, utilizziamo un dettaglio implementativo impostando `unbiased=False`. Per coloro che volessero comprendere il suo significato, nel calcolo della varianza, dividiamo per il numero di input *n*. Questo approccio non applica la correzione di Bessel, che tipicamente usa *n*-1 invece di *n* al denominatore, per correggere il bias nella stima della varianza del campione. Questa decisione produce una stima distorta (a causa del bias) della varianza. Per i modelli LLM, dove le dimensioni dell'embedding *n* sono significativamente grandi, la differenza tra l'utilizzo di *n* e *n – 1* è praticamente trascurabile. Ho scelto questo approccio per garantire la compatibilità con i layer di normalizzazione del modello GPT-2 e perché riflette il comportamento standard di TensorFlow, utilizzato per implementare il modello GPT-2 originale. L'utilizzo di tale impostazione garantisce

che il metodo sia compatibile con i pesi pre-addestrati che caricheremo nel Capitolo 6.

Proviamo ora il modulo `LayerNorm` e applichiamolo all'input batch:

```
ln = LayerNorm(emb_dim=5)
out_ln = ln(batch_example)
mean = out_ln.mean(dim=-1, keepdim=True)
var = out_ln.var(dim=-1, unbiased=False, keepdim=True)
print("Mean:\n", mean)
print("Variance:\n", var)
```

I risultati mostrano che il codice di normalizzazione dei layer si comporta come previsto, e normalizza i valori di ciascuno dei due input in modo che abbiano media 0 e varianza 1:

```
Mean:
tensor([[ -0.0000],
        [ 0.0000]],
       grad_fn=<MeanBackward1>
)
Variance:
tensor([[1.0000],
        [1.0000]],
       grad_fn=<VarBackward0>
)
```

Abbiamo appena trattato due dei blocchi costitutivi di cui avremo bisogno per implementare l'architettura GPT, come mostra la Figura 4.7. Ora, esamineremo GELU, una delle funzioni di attivazione utilizzate nei modelli LLM, al posto della tradizionale funzione ReLU utilizzata in precedenza.

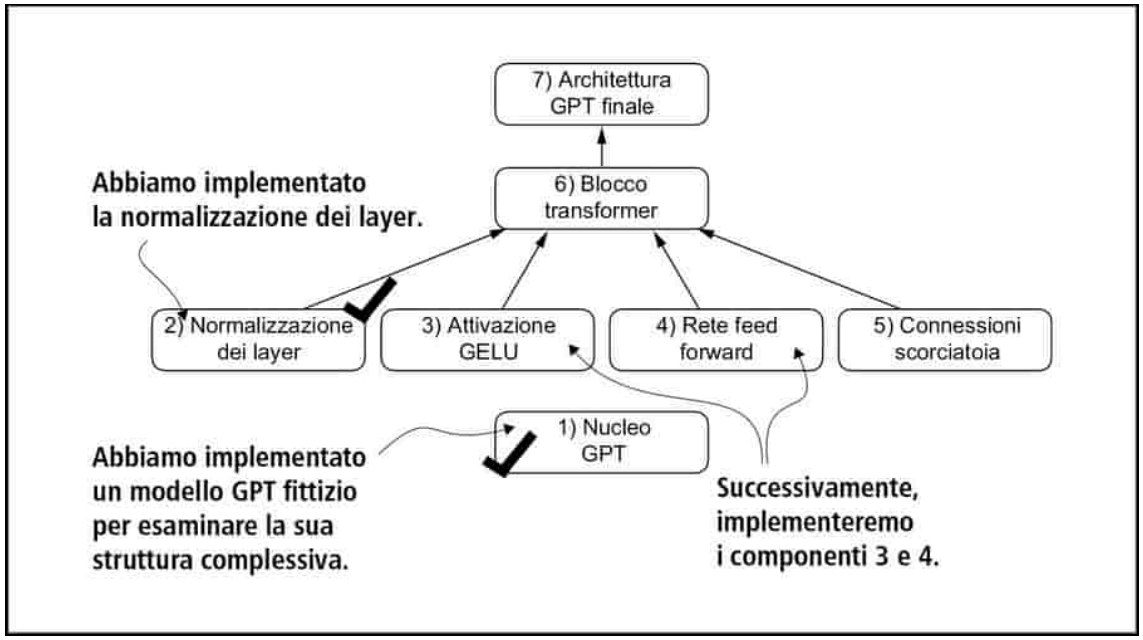


Figura 4.7 I blocchi costitutivi necessari per l'architettura GPT. Finora abbiamo completato il nucleo di GPT e la normalizzazione dei layer. Ora, ci concentreremo sull'attivazione GELU e sulla rete feed forward.

Normalizzazione dei layer vs normalizzazione del batch

Se avete familiarità con la normalizzazione del batch, un metodo di normalizzazione comune e tradizionale per le reti neurali, potreste chiedervi che legame abbia con la normalizzazione dei layer. A differenza della normalizzazione del batch, che normalizza in base alle dimensioni del batch, la normalizzazione dei layer normalizza in base alle dimensioni delle caratteristiche. Spesso i modelli LLM richiedono molte risorse di calcolo e l'hardware disponibile o lo specifico caso d'uso potrebbero determinare le dimensioni del batch durante l'addestramento o l'inferenza. Poiché la normalizzazione dei layer normalizza ogni input indipendentemente dalle dimensioni del batch, offre maggiore flessibilità e stabilità in questi scenari. Ciò è particolarmente utile per l'addestramento distribuito o quando si distribuiscono i modelli in ambienti dotati di risorse limitate.

Implementazione di una rete feed forward con attivazioni GELU

Ora, implementeremo un piccolo sub-modulo di rete neurale utilizzato nell'ambito del blocco transformer nei modelli LLM. Iniziamo implementando la funzione di attivazione *GELU*, che svolge un ruolo cruciale in questo sub-modulo di rete neurale.

NOTA

Per ulteriori informazioni sull'implementazione delle reti neurali in PyTorch, vedi l'Appendice A.

Storicamente, nel deep learning è stata comunemente utilizzata la funzione di attivazione ReLU, per la sua semplicità ed efficacia in varie architetture a reti neurali. Tuttavia, nei modelli LLM, vengono impiegate diverse altre funzioni di attivazione, oltre alla tradizionale ReLU. Due esempi degni di nota sono GELU (*Gaussian Error Linear Unit*) e SwiGLU (*Swish-Gated Linear Unit*).

GELU e SwiGLU sono funzioni di attivazione più complesse e fluide, che incorporano rispettivamente unità lineari gaussiane e a sigmoide e offrono prestazioni migliori per i modelli di deep learning, a differenza del più semplice ReLU.

La funzione di attivazione GELU può essere implementata in diversi modi; la versione esatta è definita come GELU(x) = x×**φ**(x), dove **φ**(x) è la funzione di distribuzione cumulativa della distribuzione gaussiana standard. All'atto pratico, tuttavia, è pratica comune implementare un'approssimazione computazionalmente più economica (anche il modello GPT-2 originale è stato addestrato con questa approssimazione, che è stata trovata tramite *curve fitting*):

$$GELU(x) \approx 0.5 \cdot x \cdot \left(1 + tanh\left[\sqrt{\frac{2}{\pi}} \cdot \left(x + 0.044715 \cdot x^3\right)\right]\right)$$

Nel codice possiamo implementare questa funzione come un modulo PyTorch.

Listato 4.3 Un'implementazione della funzione di attivazione GELU.

```
class GELU(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return 0.5 * x * (1 + torch.tanh(torch.sqrt(torch.tensor(2.0 / torch.pi))
*
            (x + 0.044715 * torch.pow(x, 3))
        ))
```

Ora, per farci un'idea dell'aspetto di questa funzione GELU e di come essa si confronta con la funzione ReLU, rappresentiamo graficamente queste funzioni una accanto all'altra:

```
import matplotlib.pyplot as plt
gelu, relu = GELU(), nn.ReLU()

x = torch.linspace(-3, 3, 100) (1)
y_gelu, y_relu = gelu(x), relu(x)
plt.figure(figsize=(8, 3))
for i, (y, label) in enumerate(zip([y_gelu, y_relu], ["GELU", "ReLU"]), 1):
    plt.subplot(1, 2, i)
    plt.plot(x, y)
    plt.title(f"{label} activation function")
    plt.xlabel("x")
    plt.ylabel(f"{label}(x)")
    plt.grid(True)
plt.tight_layout()
plt.show()
```

(1) Crea 100 punti campione nell'intervallo da -3 a 3

Come potete vedere nel grafico risultante (Figura 4.8), ReLU (a destra) è una funzione lineare a tratti che restituisce l'input solo se è positivo, altrimenti, restituisce zero. GELU (a sinistra) è una funzione regolare e non lineare che approssima ReLU, ma con un gradiente diverso da zero per quasi tutti i valori negativi (tranne per circa $x = -0,75$).

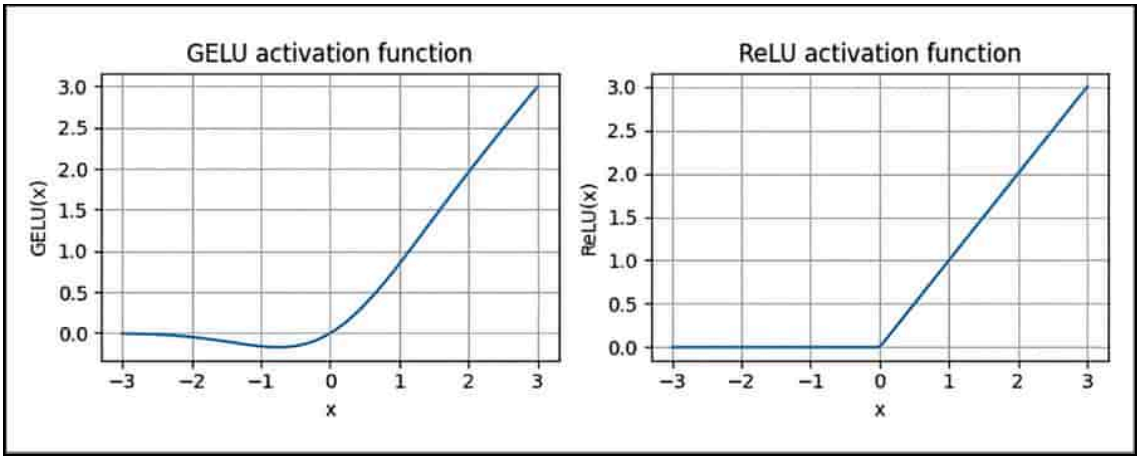


Figura 4.8 I grafici delle funzioni GELU e ReLU prodotti con matplotlib. L'asse x mostra gli input della funzione e l'asse y mostra gli output della funzione.

La fluidità di GELU può offrire migliori proprietà di ottimizzazione durante l'addestramento, poiché consente di apportare aggiustamenti più sfumati ai parametri del modello. Al contrario, ReLU ha un angolo netto a zero (Figura 4.18, a destra), che a volte può renderne più difficile l'ottimizzazione, specialmente in reti molto profonde o con architetture complesse. Inoltre, a differenza di ReLU, che restituisce zero per qualsiasi input negativo, GELU produce un output piccolo ma diverso da zero. Questa caratteristica fa sì che, durante il processo di addestramento, i neuroni che ricevono input negativi possano comunque contribuire al processo di apprendimento, anche se in misura minore rispetto agli input positivi.

Ora, utilizziamo la funzione GELU per implementare il piccolo modulo di rete neurale, `FeedForward`, che utilizzeremo più avanti nel blocco transformer del modello LLM.

Listato 4.4 Un modulo di rete neurale feed forward.

```
class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(cfg["emb_dim"], 4 * cfg["emb_dim"]),
```

```
            GELU(),
            nn.Linear(4 * cfg["emb_dim"], cfg["emb_dim"]),
        )
    def forward(self, x):
        return self.layers(x)
```

Come potete vedere, il modulo `FeedForward` è una piccola rete neurale composta da due layer `Linear` e una funzione di attivazione `GELU`. Nel modello GPT da 124 milioni di parametri, riceve i batch di input con token che hanno dimensioni dell'embedding pari a 768 ciascuno tramite il dizionario `GPT_CONFIG_124M` dove `GPT_CONFIG_124M["emb_dim"] = 768`. La Figura 4.9 mostra come le dimensioni dell'embedding vengono manipolate all'interno di questa piccola rete neurale feed forward quando le passiamo alcuni input.

I tre valori rappresentano le dimensioni del batch (2), il numero di token (3) e le dimensioni dell'embedding (768).

Seguendo l'esempio della Figura 4.9, inizializziamo un nuovo modulo `FeedForward` con dimensioni dell'embedding dei token pari a 768 e gli forniamo un input batch con due campioni e tre token ciascuno:

```
ffn = FeedForward(GPT_CONFIG_124M)
x = torch.rand(2, 3, 768) (1)
out = ffn(x)
print(out.shape)
```

(1) Crea un input di esempio con dimensioni del batch pari a 2.

Come potete vedere, la forma del tensore di output è la stessa del tensore di input:

```
torch.Size([2, 3, 768])
```

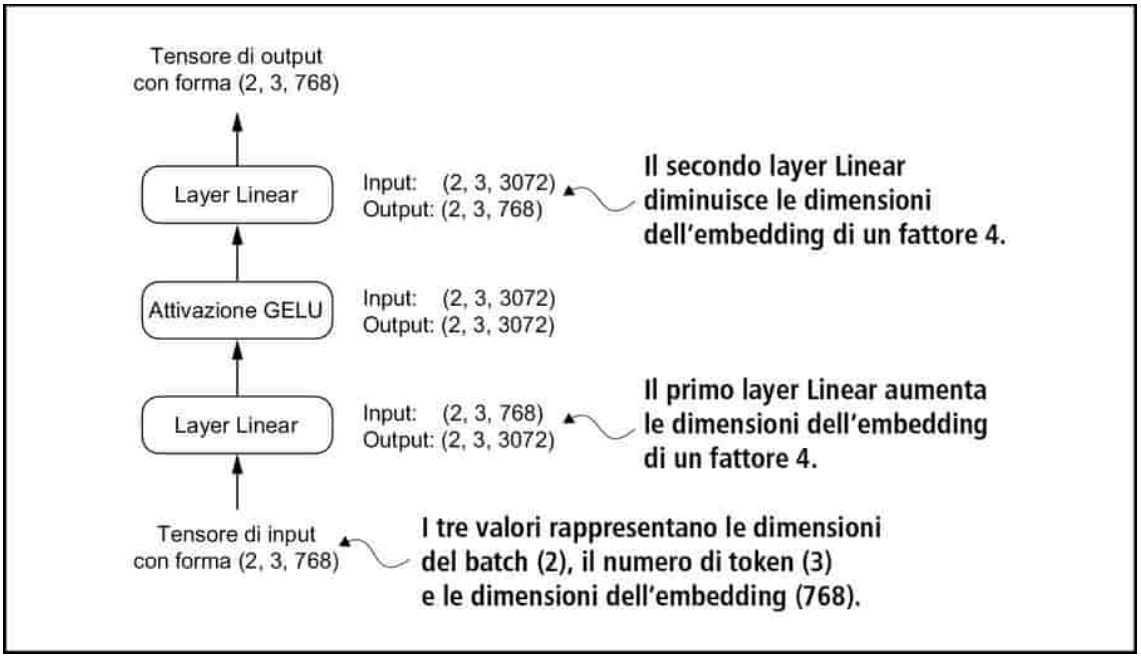


Figura 4.9 Una panoramica delle connessioni tra i layer della rete neurale feed forward. Questa rete neurale può accettare dimensioni del batch e numeri di token variabili nell'input. Tuttavia, le dimensioni dell'embedding per ogni token vengono determinate e fissate quando si inizializzano i pesi.

Il modulo `FeedForward` svolge un ruolo cruciale nel potenziare la capacità del modello di apprendere dai dati e di generalizzare. Sebbene le dimensioni dell'input e dell'output di questo modulo siano le stesse, espande internamente le dimensioni dell'embedding in uno spazio di dimensione superiore attraverso il primo layer lineare, come mostra la Figura 4.10. Questa espansione è seguita da un'attivazione GELU non lineare e quindi da una contrazione alle dimensioni originali con la seconda trasformazione lineare. Tale struttura consente l'esplorazione di uno spazio di rappresentazione più ricco.

Inoltre, l'uniformità nelle dimensioni dell'input e dell'output semplifica l'architettura, consentendo l'impilamento di più layer,

come faremo in seguito, senza la necessità di regolare le dimensioni tra di essi, rendendo così il modello più scalabile.

Come mostra la Figura 4.11, abbiamo implementato la maggior parte dei blocchi costitutivi del modello LLM. Ora, esamineremo il concetto di connessioni scorciatoia che inseriamo tra i diversi layer di una rete neurale, le quali sono importanti per migliorare le prestazioni dell'addestramento nelle architetture a reti neurali profonde.

Aggiunta delle connessioni scorciatoia

Discutiamo il concetto alla base delle *connessioni scorciatoia* (*shortcut connection*), chiamate anche connessioni di salto o residue. In origine, le connessioni scorciatoia erano state proposte per le reti profonde dedicate alla visione artificiale (in particolare, nelle reti residue, *ResNet*) per mitigare il problema dei gradienti evanescenti, che si riferisce al problema per cui i gradienti (che determinano gli aggiornamenti dei pesi durante l'addestramento) diventano progressivamente sempre più piccoli mentre si propagano a ritroso attraverso i layer, rendendo difficile addestrare efficacemente i layer precedenti.

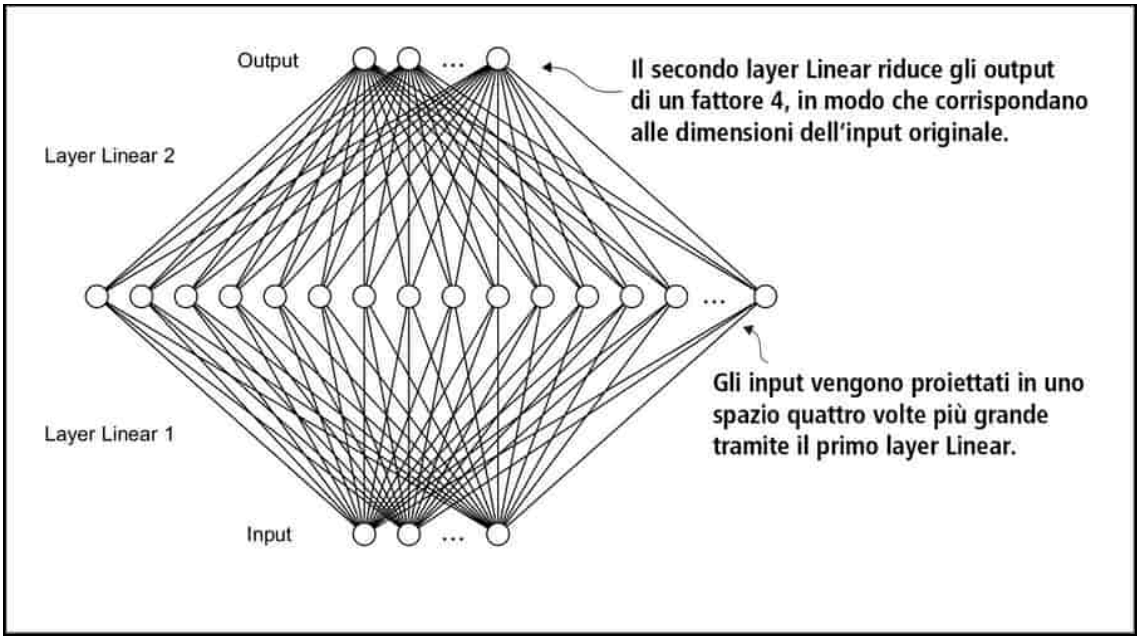


Figura 4.10 Un'illustrazione dell'espansione e contrazione degli output dei layer della rete neurale feed forward. Innanzitutto, gli input si espandono di un fattore 4 da 768 a 3.072 valori. Quindi, il secondo layer comprime i 3.072 valori di nuovo in una rappresentazione 768-dimensionale.

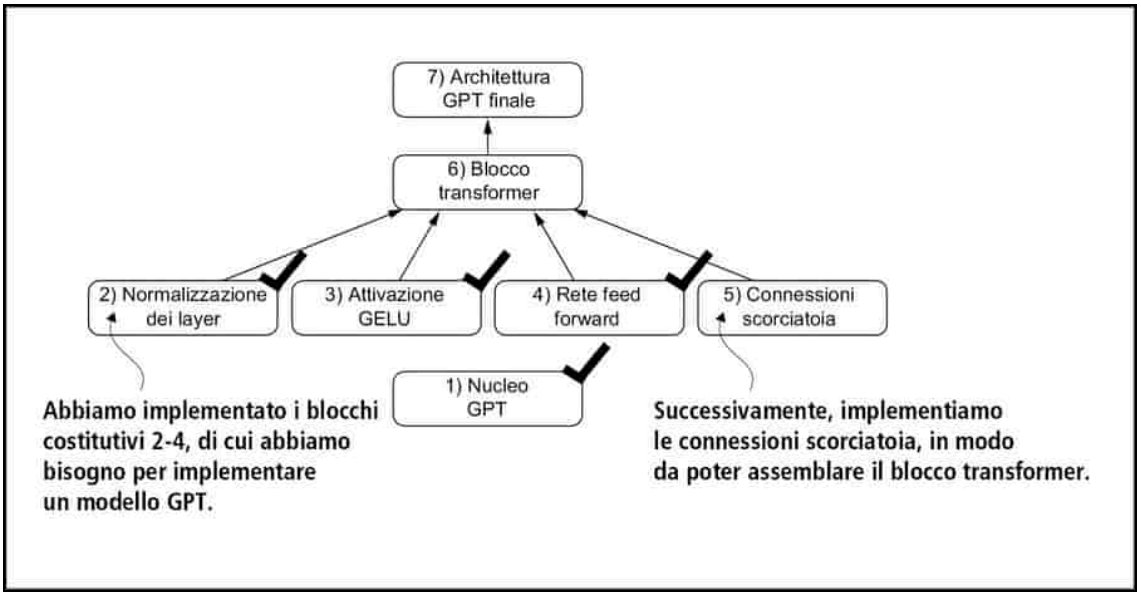


Figura 4.11 I blocchi costitutivi necessari per costruire l'architettura GPT. I segni di spunta neri indicando quelli che abbiamo già trattato.

La Figura 4.12 mostra che una connessione scorciatoia crea un percorso alternativo, più breve, per il gradiente che scorre attraverso la rete saltando uno o più layer, il che si ottiene aggiungendo l'output di un layer all'output di un layer successivo. Ecco perché queste connessioni sono chiamate anche connessioni di salto. Esse svolgono un ruolo cruciale nel preservare il flusso dei gradienti durante il passo all'indietro (backward) dell'addestramento.

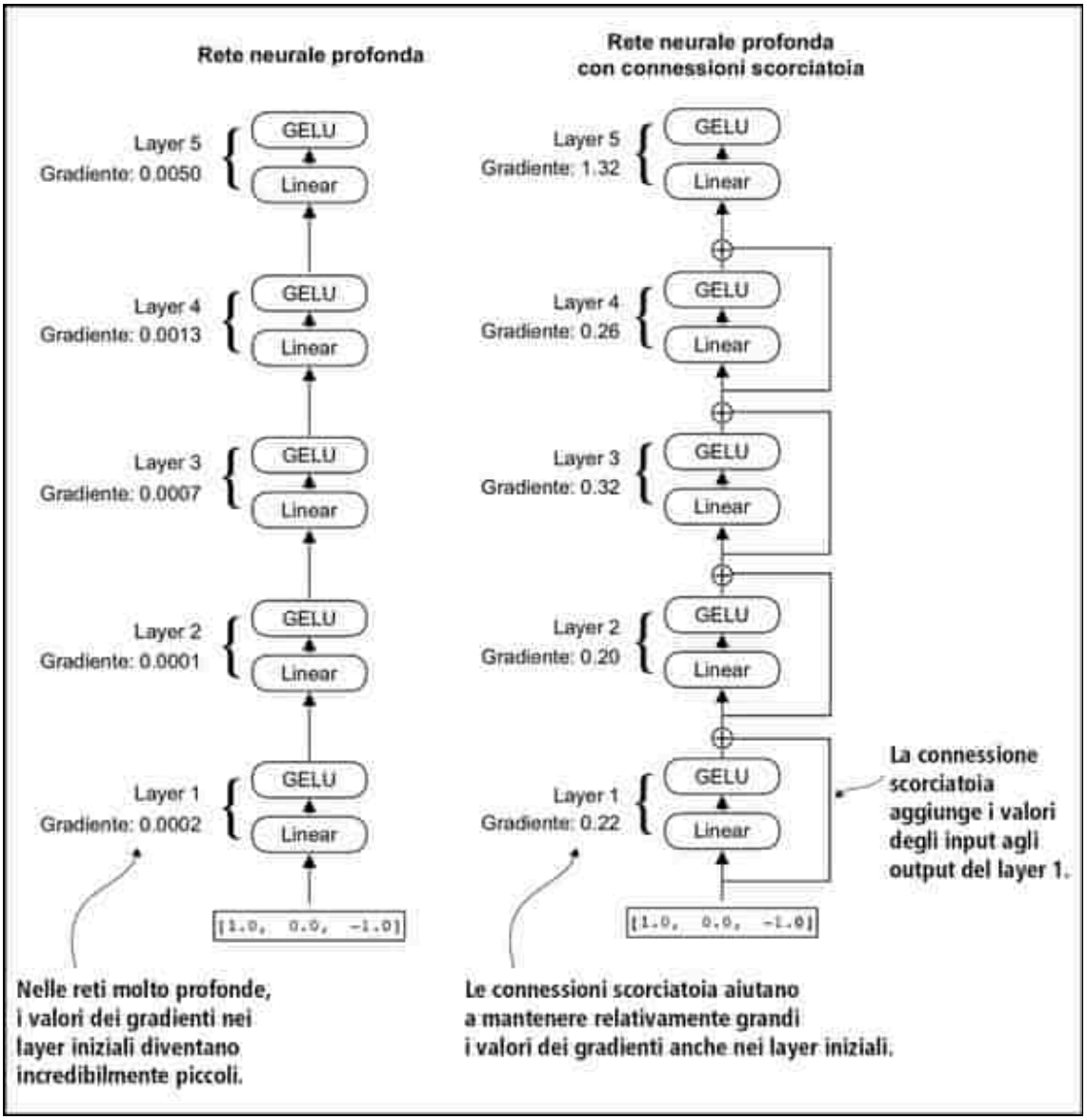


Figura 4.12 Un confronto tra una rete neurale profonda composta da cinque layer senza (a sinistra) e con (a destra) connessioni scorciatoia. Le connessioni scorciatoia implicano l'aggiunta degli input di un layer ai suoi output, creando di fatto un percorso alternativo che bypassa determinati layer. I gradienti denotano il gradiente assoluto medio a ogni layer, che calcoliamo nel Listato 4.5.

Nel Listato 4.5 implementiamo la rete neurale della Figura 4.12 per vedere come possiamo aggiungere le connessioni scorciatoia nel

metodo `forward`.

Listato 4.5 Una rete neurale che illustra le connessioni scorciatoia.

```
class ExampleDeepNeuralNetwork(nn.Module):
    def __init__(self, layer_sizes, use_shortcut):
        super().__init__()
        self.use_shortcut = use_shortcut
        self.layers = nn.ModuleList([ (1)
            nn.Sequential(nn.Linear(layer_sizes[0], layer_sizes[1]), GELU()),
            nn.Sequential(nn.Linear(layer_sizes[1], layer_sizes[2]), GELU()),
            nn.Sequential(nn.Linear(layer_sizes[2], layer_sizes[3]), GELU()),
            nn.Sequential(nn.Linear(layer_sizes[3], layer_sizes[4]), GELU()),
            nn.Sequential(nn.Linear(layer_sizes[4], layer_sizes[5]), GELU())
        ])

    def forward(self, x):
        for layer in self.layers:
            layer_output = layer(x) (2)
            if self.use_shortcut and x.shape == layer_output.shape: (3)
                x = x + layer_output
            else:
                x = layer_output
        return x
```

(1) Implementa cinque livelli.

(2) Calcola l'output del livello corrente.

(3) Controlla se è possibile applicare una connessione scorciatoia.

Il codice implementa una rete neurale profonda con cinque livelli, ciascuno costituito da un layer `Linear` e una funzione di attivazione `GELU`. Nel passo forward, passiamo iterativamente l'input attraverso i layer e, facoltativamente, aggiungiamo le connessioni scorciatoia se l'attributo `self.use_shortcut` è `True`.

Usiamo questo codice per inizializzare una rete neurale senza connessioni scorciatoia. Ogni layer verrà inizializzato in modo tale da accettare un esempio con tre valori di input e restituire tre valori di output. L'ultimo layer restituisce un singolo valore di output:

```
layer_sizes = [3, 3, 3, 3, 3, 1]
sample_input = torch.tensor([[1., 0., -1.]])
torch.manual_seed(123) (1)
model_without_shortcut = ExampleDeepNeuralNetwork(layer_sizes, use_shortcut=False)
```

(1) Specifica un valore casuale per i pesi iniziali ai fini della riproducibilità.

Ora, implementiamo una funzione che calcoli i gradienti nel passo backward del modello:

```
def print_gradients(model, x):
    output = model(x) (1)
    target = torch.tensor([[0.]])

    loss = nn.MSELoss()
    loss = loss(output, target) (2)

    loss.backward() (3)

    for name, param in model.named_parameters():
        if 'weight' in name:
            print(f"{name} has gradient mean of {param.grad.abs().mean().item()}")
```

(1) Passo forward.

(2) Calcola il valore loss in base alla vicinanza tra target e output.

(3) Passo backward per calcolare i gradienti.

Questo codice specifica una funzione `loss` che calcola quanto sono vicini l'output del modello e un target specificato dall'utente (qui, per semplicità, il valore `0`). Quindi, quando richiamiamo `loss.backward()`, PyTorch calcola il gradiente di loss per ogni layer del modello. Possiamo scorrere i parametri dei pesi tramite `model.named_parameters()`. Supponiamo di avere una matrice di parametri dei pesi 3 × 3 per un dato layer. In tal caso, questo layer avrà valori dei gradienti 3 × 3 e visualizziamo il gradiente assoluto medio di questi valori dei gradienti 3 × 3 per ottenere un singolo valore gradiente per il layer, per confrontare più facilmente i gradienti tra i layer.

In breve, `.backward()` è un metodo di PyTorch utile per calcolare i gradienti del valore loss, necessari durante l'addestramento del modello, senza dover implementare noi stessi i calcoli matematici per il calcolo dei gradienti, rendendo così molto più agevole il lavoro.

NOTA

Se non avete familiarità col concetto di gradienti e addestramento delle reti neurali, potete far riferimento all'Appendice A.

Ora usiamo la funzione `print_gradients` e la applichiamo al modello senza le connessioni di salto:

```
print_gradients(model_without_shortcut, sample_input)
```

L'output è il seguente:

```
layers.0.0.weight has gradient mean of 0.00020173587836325169
layers.1.0.weight has gradient mean of 0.0001201116101583466
layers.2.0.weight has gradient mean of 0.0007152041653171182
layers.3.0.weight has gradient mean of 0.001398873864673078
layers.4.0.weight has gradient mean of 0.005049646366387606
```

L'output della funzione `print_gradients` mostra che i gradienti diventano più piccoli man mano che si procede dall'ultimo layer (`layers.4`) al primo (`layers.0`), un fenomeno chiamato *problema di evanescenza dei gradienti*.

Creiamo ora un modello con connessioni di salto e paragoniamoli:

```
torch.manual_seed(123)
model_with_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=True
)
print_gradients(model_with_shortcut, sample_input)
```

L'output è il seguente:

```
layers.0.0.weight has gradient mean of 0.22169792652130127
layers.1.0.weight has gradient mean of 0.20694105327129364
layers.2.0.weight has gradient mean of 0.32896995544433594
layers.3.0.weight has gradient mean of 0.2665732502937317
layers.4.0.weight has gradient mean of 1.3258541822433472
```

L'ultimo layer (`layers.4`) ha ancora un gradiente più grande degli altri layer. Tuttavia, il valore dei gradienti si stabilizza man mano che si procede verso il primo layer (`layers.0`) e non si riduce mai a un valore troppo piccolo.

In conclusione, le connessioni scorciatoia sono importanti per superare le limitazioni poste dal problema di evanescenza dei gradienti nelle reti neurali profonde. Le connessioni scorciatoia sono un elemento fondamentale di modelli molto grandi come quelli LLM e aiuteranno a facilitare l'addestramento, assicurando un flusso dei gradienti coerente tra i livelli quando addestreremo il modello GPT nel prossimo capitolo.

Ora, collegheremo tutti i concetti trattati in precedenza (normalizzazione dei layer, attivazioni GELU, modulo feed forward e connessioni scorciatoia) in un blocco transformer, che è l'ultimo blocco costitutivo di cui abbiamo bisogno per programmare l'architettura GPT.

Collegamento dei layer di attenzione e lineari in un blocco transformer

Ora, implementiamo il *blocco transformer*, un elemento fondamentale di un GPT e di altre architetture LLM. Questo blocco, che viene ripetuto una dozzina di volte nell'architettura GPT-2 da 124 milioni di parametri, combina diversi concetti che abbiamo già trattato in precedenza: attenzione multi-head, normalizzazione dei layer, dropout, layer feed forward e attivazioni GELU. In seguito, collegheremo questo blocco transformer alle parti rimanenti dell'architettura GPT.

La Figura 4.13 mostra un blocco transformer che combina diversi componenti, tra cui il modulo di attenzione multi-head con maschera (vedi il Capitolo 3) e il modulo `FeedForward` che abbiamo implementato in precedenza. Quando un blocco transformer elabora una sequenza di input, ogni elemento della sequenza (per esempio, un token di parola o sub-parola) è rappresentato da un vettore di dimensione fissa (in questo caso, 768-dimensionale). Le operazioni all'interno del blocco transformer, tra cui l'attenzione multi-head e i layer feed forward, sono progettate per trasformare questi vettori in modo da preservarne la dimensionalità.

L'idea è che il meccanismo di auto-attenzione nel blocco di attenzione multi-head identifichi e analizzi le relazioni tra gli

elementi presenti nella sequenza di input. Al contrario, la rete feed forward modifica i dati individualmente in ogni posizione. Questa combinazione non solo consente una comprensione e un'elaborazione più dettagliate dell'input, ma migliora anche la capacità complessiva del modello di gestire pattern complessi nei dati.

Possiamo creare il blocco `TransformerBlock` nel codice.

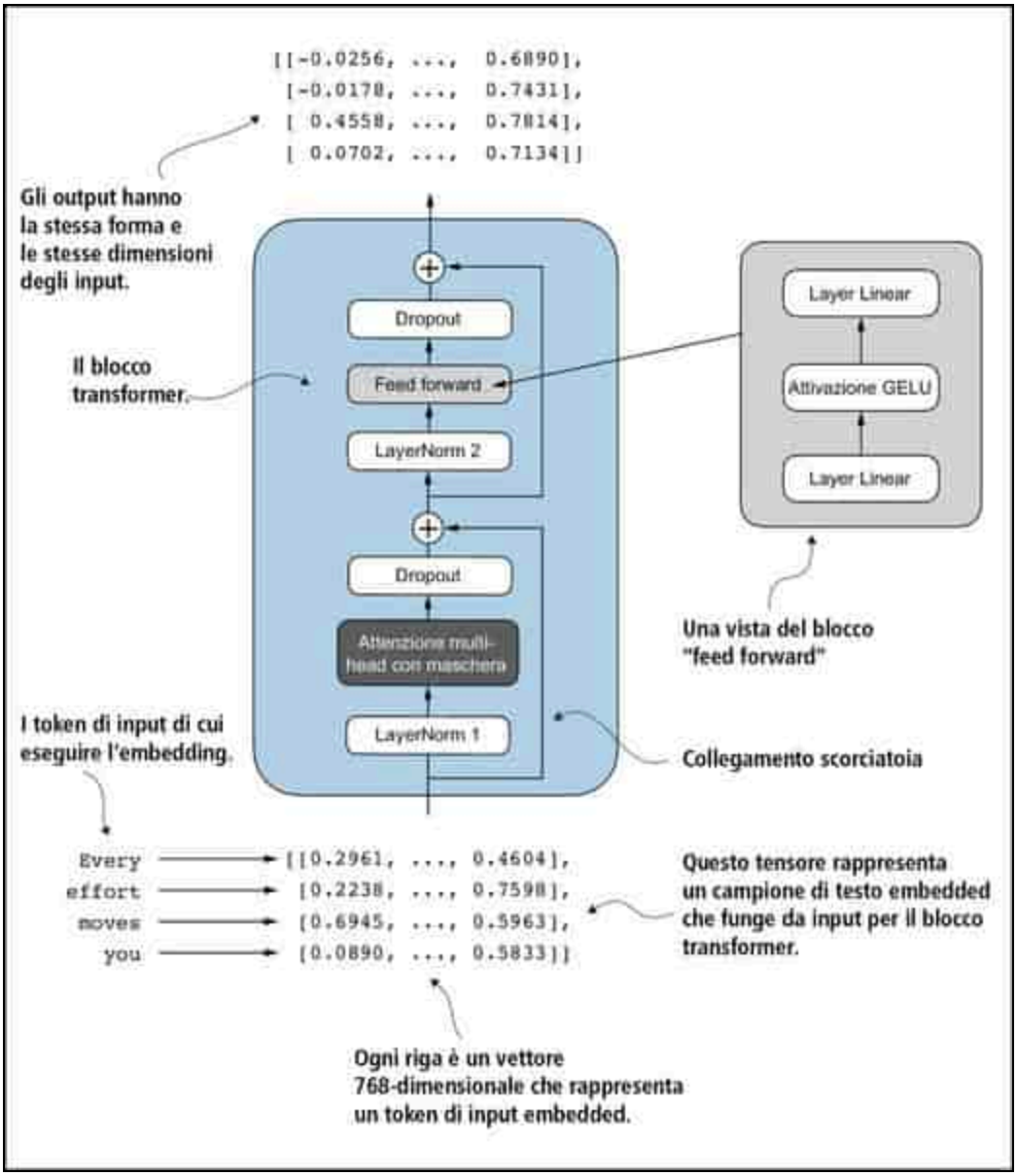


Figura 4.13 La rappresentazione di un blocco transformer. I token di input hanno subìto l'embedding in vettori 768-dimensionali. Ogni riga corrisponde alla rappresentazione vettoriale di un solo token. Gli output del blocco transformer sono vettori delle stesse dimensioni dell'input, e possono quindi essere immessi in layer successivi di un modello LLM.

Listato 4.6 Il componente del blocco transformer di GPT.

```
from chapter03 import MultiHeadAttention

class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate"],
            qkv_bias=cfg["qkv_bias"])
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_shortcut = nn.Dropout(cfg["drop_rate"])

    def forward(self, x):

        shortcut = x (1)
        x = self.norm1(x)
        x = self.att(x)
        x = self.drop_shortcut(x)
        x = x + shortcut (2)

        shortcut = x (3)
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x)
        x = x + shortcut (4)
        return x
```

(1) Collegamento scorciatoia per il blocco di attenzione.
(2) Aggiunge nuovamente l'input originale.
(3) Collegamento scorciatoia per il blocco feed forward.
(4) Aggiunge nuovamente l'input originale.

Il codice definisce una classe `TransformerBlock` in PyTorch che include un meccanismo di attenzione multi-head (`MultiHeadAttention`) e una rete feed forward (`FeedForward`), entrambi configurati in base a un dizionario fornito (`cfg`), come `GPT_CONFIG_124M`.

La normalizzazione dei layer (`LayerNorm`) viene applicata prima di ciascuno di questi due componenti, e il dropout viene applicato dopo di essi per regolarizzare il modello e prevenire l'overfitting. Questo è noto come *Pre-LayerNorm* (normalizzazione pre-layer). Le

architetture più vecchie, come il modello a transformer originale, applicavano la normalizzazione dei layer dopo le reti di auto-attenzione e feed forward, un sistema noto come *Post-LayerNorm* (normalizzazione post-layer), che spesso presenta dinamiche di addestramento peggiori.

La classe implementa anche il passo forward, in cui ogni componente è seguito da una connessione scorciatoia che aggiunge l'input del blocco al suo output. Questa operazione, fondamentale, aiuta i gradienti a fluire attraverso la rete durante l'addestramento e migliora l'apprendimento dei modelli profondi.

Utilizzando il dizionario `GPT_CONFIG_124M` definito in precedenza, creiamo un blocco transformer e gli forniamo alcuni dati di esempio:

```
torch.manual_seed(123)
x = torch.rand(2, 4, 768) (1)
block = TransformerBlock(GPT_CONFIG_124M)
output = block(x)

print("Input shape:", x.shape)
print("Output shape:", output.shape)
```

(1) Crea un campione di input della forma `[batch_size, num_tokens, emb_dim]`.

L'output è il seguente:

```
Input shape: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 768])
```

Come potete vedere, il blocco transformer mantiene le dimensioni di input nel suo output, il che indica che l'architettura a transformer elabora sequenze di dati senza alterarne la forma in tutta la rete.

La conservazione della forma in tutta l'architettura del blocco transformer non è incidentale, ma un aspetto cruciale della sua progettazione. Questa progettazione consente di applicarla efficacemente in un'ampia varietà di attività sequenza-sequenza, in cui ogni vettore di output corrisponde direttamente a un vettore di input, mantenendo una relazione uno-a-uno. Tuttavia, l'output è un

vettore del contesto che incapsula le informazioni dall'intera sequenza di input (vedi il Capitolo 3). Ciò significa che, mentre le dimensioni fisiche della sequenza (lunghezza e dimensioni delle caratteristiche) rimangono invariate mentre attraversa il blocco transformer, il contenuto di ogni vettore di output viene ricodificato per integrare le informazioni contestuali dall'intera sequenza di input.

Ora che il blocco transformer è implementato, abbiamo tutti i blocchi costitutivi necessari per implementare l'architettura GPT. Come mostra la Figura 4.14, il blocco transformer combina la normalizzazione dei layer, la rete feed forward, le attivazioni GELU e le connessioni scorciatoia. Come vedremo alla fine, questo blocco transformer costituirà il componente principale dell'architettura GPT.

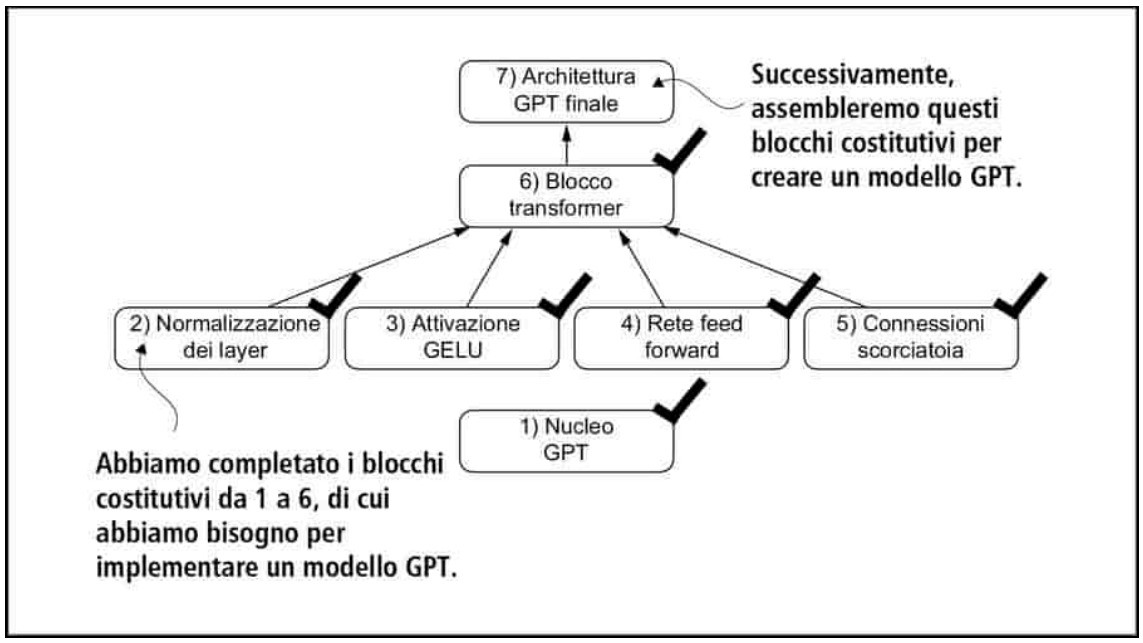


Figura 4.14 I blocchi costitutivi necessari per costruire l'architettura GPT. I segni di spunta neri indicano i blocchi che abbiamo già completato.

Programmazione del modello GPT

Abbiamo iniziato questo capitolo con una panoramica generale di un'architettura GPT che abbiamo chiamato `DummyGPTModel`. In tale implementazione del codice abbiamo mostrato l'input e gli output del modello GPT, ma i suoi elementi costitutivi sono rimasti una "scatola nera", dal momento che utilizzavano le classi fittizie `DummyTransformerBlock` e `DummyLayerNorm`.

Sostituiamo ora le classi fittizie `DummyTransformerBlock` e `DummyLayerNorm` con le vere classi `TransformerBlock` e `LayerNorm` che abbiamo programmato in precedenza, per assemblare una versione completamente funzionante della versione originale da 124 milioni di parametri di GPT-2. Nel Capitolo 5, preaddestreremo un modello GPT-2 e nel Capitolo 6, caricheremo i pesi pre-addestrati da OpenAI.

Prima di assemblare il modello GPT-2 tramite il codice, esaminiamo però la sua struttura complessiva, rappresentata nella Figura 4.15, che include tutti i concetti che abbiamo trattato finora. Come potete vedere, il blocco transformer viene ripetuto più volte in tutta l'architettura di un modello GPT. Nel caso del modello GPT-2 da 124 milioni di parametri, viene ripetuto 12 volte, che specifichiamo tramite l'elemento `n_layers` del dizionario `GPT_CONFIG_124M`. Nel modello GPT-2 più grande, con 1.542 milioni di parametri, questo blocco transformer viene ripetuto 48 volte.

L'output fornito dal blocco transformer finale passa quindi attraverso un ultimo passo di normalizzazione del layer prima di raggiungere il layer di output `Linear`. Questo layer mappa l'output del transformer in uno spazio a elevata dimensionalità (in questo caso, 50.257 dimensioni, corrispondenti alle dimensioni del vocabolario del modello) per prevedere il token successivo nella sequenza.

Passiamo ora a programmare l'architettura rappresentata nella Figura 4.15.

Listato 4.7 Implementazione dell'architettura del modello GPT.

```
class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])

        self.trf_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])]
        )

        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)

        pos_embeds = self.pos_emb(torch.arange(seq_len, device=in_idx.device)) (1)
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits
```

(1) L'impostazione del dispositivo ci consentirà di addestrare il modello su una CPU o una GPU, a seconda del dispositivo su cui risiedono i dati di input.

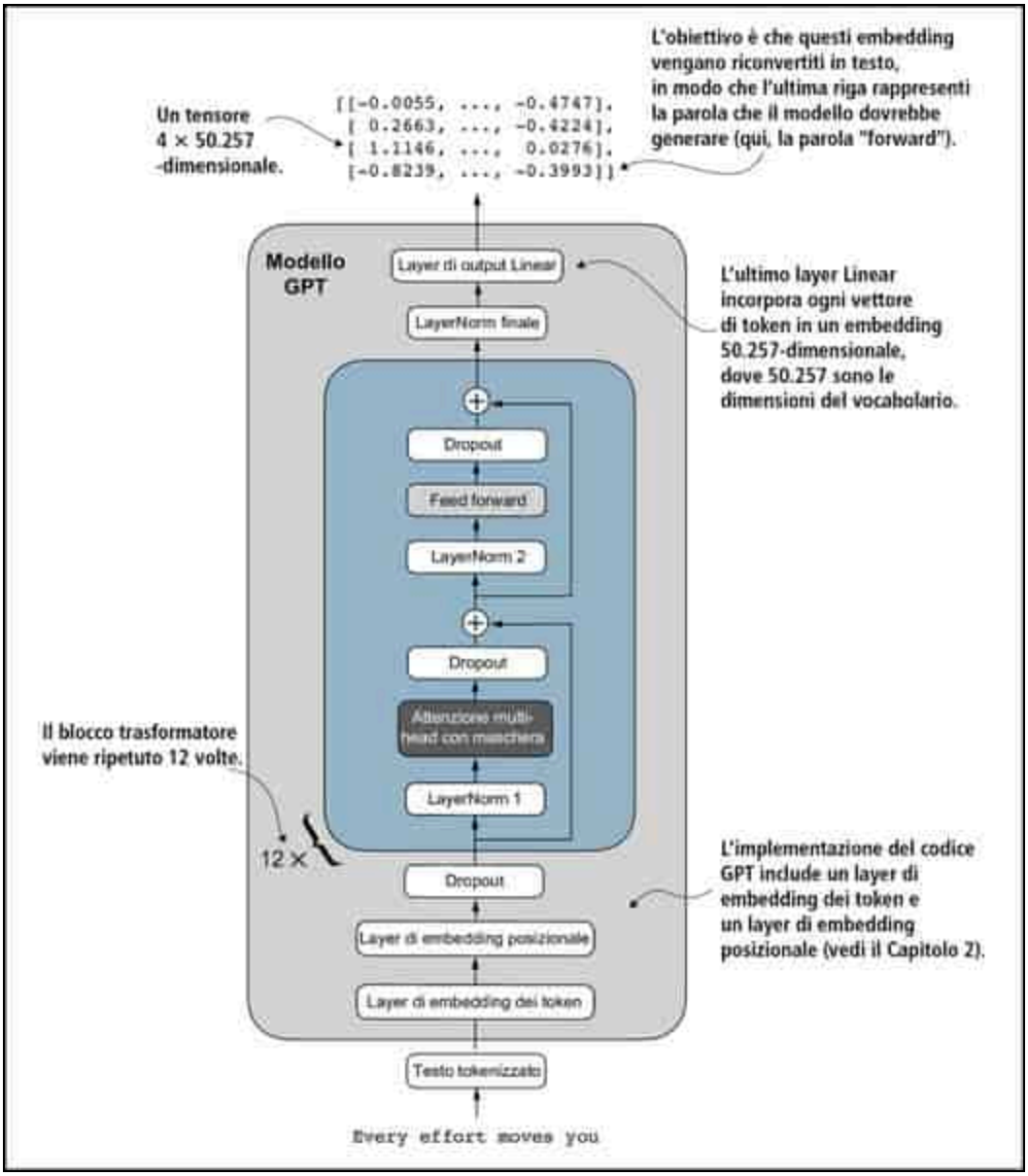


Figura 4.15 Una panoramica dell'architettura del modello GPT che mostra il flusso dei dati. Partendo dal basso, il testo tokenizzato viene prima convertito in embedding dei token, che vengono poi aumentati con l'embedding posizionale. Queste informazioni combinate formano un tensore che viene fatto passare attraverso una serie di blocchi transformer mostrati al centro (ciascuno contenente i layer di attenzione multi-head e feed forward con dropout e normalizzazione), che vengono impilati uno sopra l'altro e ripetuti 12 volte.

Grazie alla classe `TransformerBlock`, la classe `GPTModel` è relativamente compatta.

Il costruttore `__init__` di questa classe `GPTModel` inizializza i layer di tokenizzazione e di embedding posizionale utilizzando le configurazioni passate tramite un dizionario Python, `cfg`. Questi layer di embedding sono responsabili della conversione degli indici dei token di input in vettori densi e dell'aggiunta di informazioni posizionali (vedi il Capitolo 2).

Ora, il metodo `__init__`crea uno stack sequenziale di moduli `TransformerBlock` pari al numero di layer specificato in `cfg`. Dopo i blocchi transformer, viene applicato un layer `LayerNorm`, che standardizza gli output forniti dai blocchi transformer per stabilizzare il processo di apprendimento. Infine, viene definita una "testa" di output lineare senza bias, che proietta l'output del transformer nello spazio del vocabolario del tokenizer, per generare i risultati (`logits`) per ogni token del vocabolario.

Il metodo `forward` prende in input un batch di indici di token, calcola i loro embedding, applica gli embedding posizionali, passa la sequenza attraverso i blocchi transformer, normalizza l'output finale e quindi calcola i `logits`, che rappresentano le probabilità non normalizzate del token successivo. Convertiremo questi `logits` in token e output testuali nel prossimo paragrafo.

Inizializziamo ora il modello GPT da 124 milioni di parametri utilizzando il dizionario `GPT_CONFIG_124M` che passiamo nel parametro `cfg` e forniamogli l'input di testo creato in precedenza:

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)

out = model(batch)
print("Input batch:\n", batch)
print("\nOutput shape:", out.shape)
print(out)
```

Questo codice visualizza il contenuto del batch di input seguito dal tensore di output:

```
Input batch:
 tensor([[6109, 3626, 6100, 345], (1)
         [6109, 1110, 6622, 257]] (2)
 )

Output shape: torch.Size([2, 4, 50257])
 tensor([[[ 0.3613,  0.4222, -0.0711, ...,  0.3483,  0.4661, -0.2838],
          [-0.1792, -0.5660, -0.9485, ...,  0.0477,  0.5181, -0.3168],
          [ 0.7120,  0.0332,  0.1085, ...,  0.1018, -0.4327, -0.2553],
          [-1.0076,  0.3418, -0.1190, ...,  0.7195,  0.4023,  0.0532]],
         [[-0.2564,  0.0900,  0.0335, ...,  0.2659,  0.4454, -0.6806],
          [ 0.1230,  0.3653, -0.2074, ...,  0.7705,  0.2710,  0.2246],
          [ 1.0558,  1.0318, -0.2800, ...,  0.6936,  0.3205, -0.3178],
          [-0.1565,  0.3926,  0.3288, ...,  1.2630, -0.1858,  0.0388]]],
        grad_fn=<UnsafeViewBackward0>
 )
```

(1) ID del token del testo 1.

(2) ID del token del testo 2.

Come potete vedere, il tensore di output ha la forma `[2, 4, 50257]`, poiché abbiamo passato i due testi di input con quattro token ciascuno. L'ultimo valore, `50257`, corrisponde alle dimensioni del vocabolario del tokenizer. Più avanti, vedremo come convertire ciascuno di questi vettori di output 50.257-dimensionali di nuovo in token.

Prima di passare alla programmazione della funzione che converte gli output del modello in testo, diamo un'occhiata all'architettura del modello e analizziamone le dimensioni. Utilizzando il metodo `numel()` ("numero di elementi"), possiamo raccogliere il numero totale di parametri nei tensori dei parametri del modello:

```
total_params = sum(p.numel() for p in model.parameters())
print(f"Total number of parameters: {total_params:,}")
```

Il risultato è il seguente:

```
Total number of parameters: 163,009,536
```

Ora, un lettore curioso potrebbe notare una discrepanza. In precedenza, abbiamo parlato di inizializzazione di un modello GPT

da 124 milioni di parametri, quindi perché il numero effettivo di parametri è 163 milioni?

Il motivo è un concetto chiamato *weight tying*, che è stato utilizzato nell'architettura GPT-2 originale. L'architettura GPT-2 originale riutilizza nel suo layer di output i pesi dal layer di embedding dei token. Per chiarire, esaminiamo le forme del layer di embedding dei token e del layer di output `Linear` che abbiamo inizializzato su `model` tramite `GPTModel`, in precedenza:

```
print("Token embedding layer shape:", model.tok_emb.weight.shape)
print("Output layer shape:", model.out_head.weight.shape)
```

Come potete vedere dagli output, i tensori dei pesi per entrambi questi layer hanno la stessa forma:

```
Token embedding layer shape: torch.Size([50257, 768])
Output layer shape: torch.Size([50257, 768])
```

I layer di embedding dei token e di output sono molto grandi a causa del numero di righe (50.257) nel vocabolario del tokenizer. Rimuoviamo il conteggio dei parametri del layer di output dal conteggio totale del modello GPT-2 in base al vincolo di peso:

```
total_params_gpt2 = (
    total_params - sum(p.numel()
    for p in model.out_head.parameters())
)

print(f"Number of trainable parameters "
      f"considering weight tying: {total_params_gpt2:,}"
)
```

L'output è il seguente:

```
Number of trainable parameters considering weight tying: 124,412,160
```

Come potete vedere, il modello ora è grande solo 124 milioni di parametri, il che corrisponde alle dimensioni originali del modello GPT-2.

Il vincolo di peso riduce l'ingombro complessivo in termini di memoria e la complessità computazionale del modello. Tuttavia, per esperienza, l'utilizzo di layer separati di embedding dei token e di output si traduce in un miglioramento dell'addestramento e delle

prestazioni del modello; pertanto, nella nostra implementazione `GPTModel` utilizziamo layer separati. Lo stesso vale per i modelli LLM attuali. Tuttavia, rivisiteremo e implementeremo il concetto di vincolo di peso più avanti nel Capitolo 6, quando caricheremo i pesi pre-addestrati da OpenAI.

Esercizio 4.1 Numero di parametri nei moduli feed forward e di attenzione
Calcolate e confrontate il numero di parametri contenuti nel modulo feed forward e quelli contenuti nel modulo di attenzione multi-head.

Infine, calcoliamo i requisiti di memoria dei 163 milioni di parametri nel nostro oggetto `GPTModel`:

```
total_size_bytes = total_params * 4 (1)
total_size_mb = total_size_bytes / (1024 * 1024) (2)
print(f"Total size of the model: {total_size_mb:.2f} MB")
```

(1) Calcola le dimensioni totali in byte (supponendo `float32`, 4 byte per parametro).

(2) Converte in megabyte.

Il risultato è il seguente:

```
Total size of the model: 621.83 MB
```

In conclusione, calcolando i requisiti di memoria per i 163 milioni parametri nel nostro oggetto `GPTModel` e supponendo che ogni parametro sia un float da 32 bit che occupa 4 byte, scopriamo che le dimensioni totali del modello sono pari a 621.83 MB, il che dimostra la capacità di archiviazione relativamente ampia richiesta per ospitare anche LLM relativamente piccoli.

Ora che abbiamo implementato l'architettura di `GPTModel` e abbiamo visto che produce tensori numerici di forma `[batch_size, num_tokens, vocab_size]`, scriviamo il codice per convertire questi tensori di output in testo.

Esercizio 4.2 Inizializzazione di modelli GPT più grandi
Abbiamo inizializzato un modello GPT da 124 milioni di parametri, noto come *GPT-2 small*. Senza apportare modifiche al codice oltre all'aggiornamento del file di configurazione, impiegate la classe `GPTModel` per implementare *GPT-2 medium* (usando embedding 1024-dimensionali, 24 blocchi transformer, 16 "teste di attenzione" multi-head), *GPT-2 large* (embedding 1280-dimensionali, 36 blocchi transformer, 20 "teste di attenzione" multi-head) e *GPT-2 XL* (embedding 1600-dimensionali, 48 blocchi transformer, 25 "teste di attenzione" multi-head). Inoltre, calcolate il numero totale di parametri in ogni modello GPT.

Generazione del testo

Ora implementeremo il codice che converte gli output tensoriali del modello GPT di nuovo in testo. Prima di iniziare, rivediamo brevemente come un modello generativo come un LLM genera il testo una parola (o token) alla volta.

La Figura 4.16 illustra il processo passo-passo con cui un modello GPT genera il testo dato un contesto di input, come "Hello, I am". A ogni iterazione, il contesto di input cresce, consentendo al modello di generare testo coerente e appropriato al contesto.

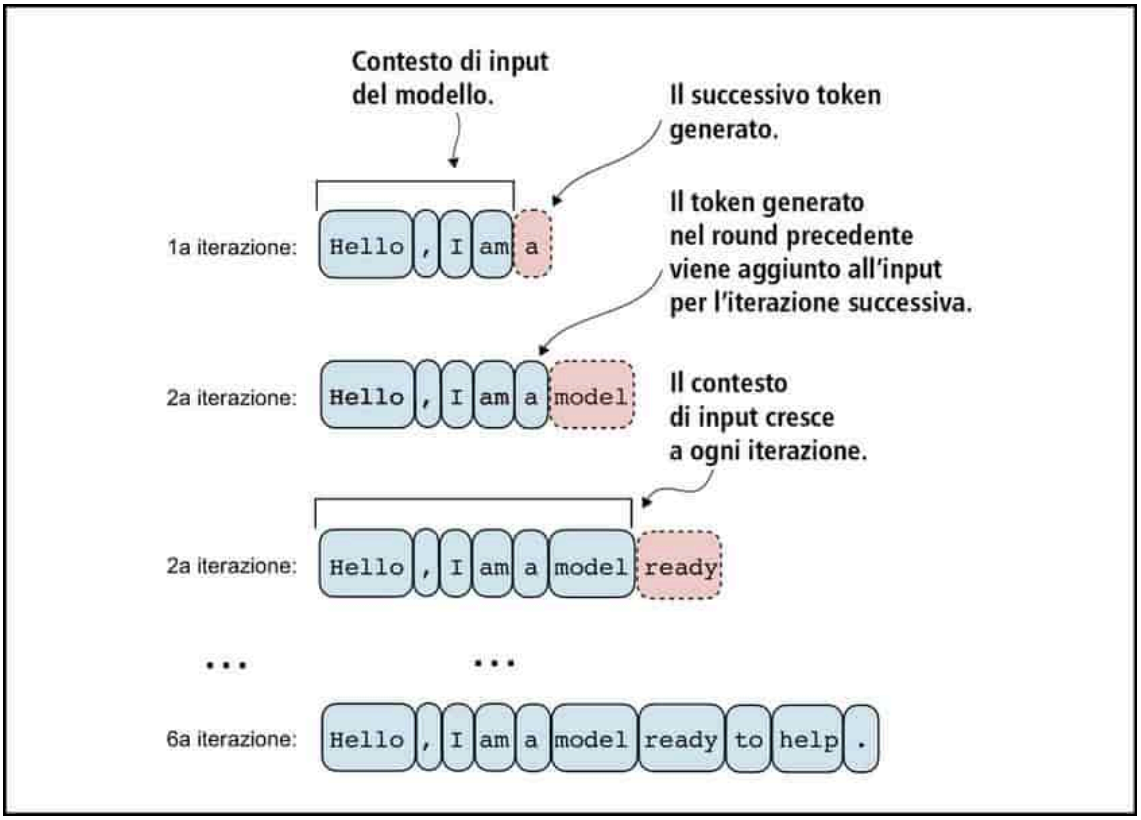


Figura 4.16 Il processo passo-passo con cui un modello LLM genera il testo, un token alla volta. Partendo da un contesto di input iniziale ("Hello, I am"), il modello prevede a ogni iterazione un token successivo, aggiungendolo al contesto di input per il round successivo. Come vediamo, la prima iterazione aggiunge "a", la seconda "model" e la terza "ready", costruendo progressivamente la frase.

Alla sesta iterazione, il modello ha costruito una frase completa: "Hello, I am a model ready to help". Abbiamo visto che la nostra attuale implementazione `GPTModel` produce tensori con forma `[batch_size, num_tokens, vocab_size]`. Ora la domanda è: in che modo un modello GPT passa da questi tensori di output al testo generato?

Il processo mediante il quale un modello GPT passa dai tensori di output al testo generato comporta diversi passi, come mostra la Figura 4.17. Questi passi includono la decodifica dei tensori di

output, la selezione dei token in base a una distribuzione di probabilità e la conversione di questi token in testo.

Il processo di generazione del token successivo dettagliato nella Figura 4.17 illustra un singolo passo in cui il modello GPT genera il token successivo dato il suo input. A ogni passo, il modello produce una matrice con vettori che rappresentano i potenziali token successivi. Il vettore corrispondente al token successivo viene estratto e convertito in una distribuzione di probabilità tramite la funzione `softmax()`. All'interno del vettore contenente i punteggi di probabilità risultanti, si trova l'indice con il valore più elevato, che si traduce nel token-ID corrispondente. Questo token-ID viene poi decodificato nuovamente in testo, producendo il token successivo nella sequenza. Infine, questo token viene aggiunto agli input precedenti, formando una nuova sequenza di input per l'iterazione successiva. Questo processo passo-passo consente al modello di generare testo in sequenza, creando frasi e periodi coerenti a partire dal contesto di input iniziale.

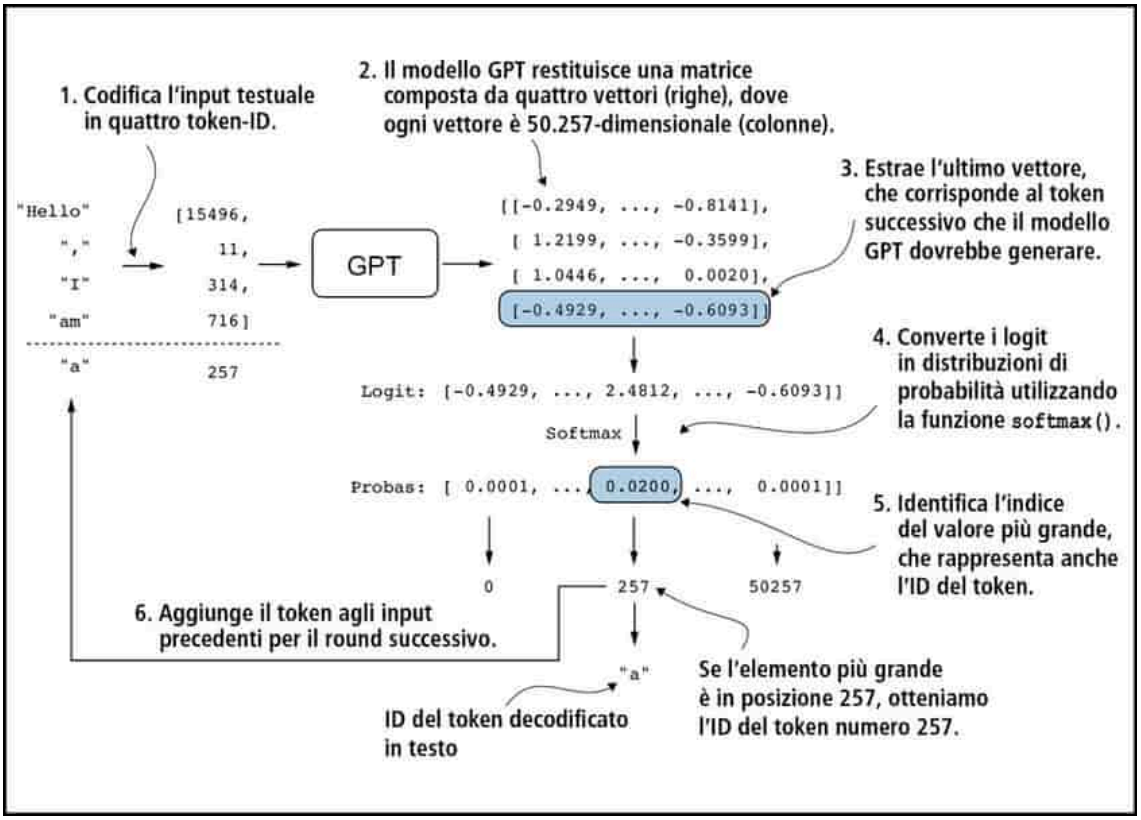


Figura 4.17 La meccanica della generazione del testo in un modello GPT, mostrando una singola iterazione nel processo di generazione dei token. Il processo inizia codificando il testo di input in token-ID, che vengono poi immessi nel modello GPT. Gli output del modello vengono poi riconvertiti in testo e aggiunti al testo di input originale.

All'atto pratico, ripetiamo questo processo per molte iterazioni, come mostra la Figura 4.16, finché non raggiungiamo un numero di token generati specificato dall'utente. Nel codice, possiamo implementare il processo di generazione dei token come mostrato nel Listato 4.8.

Listato 4.8 Una funzione di generazione del testo per il modello GPT.

```
def generate_text_simple(model, idx, max_new_tokens, context_size): (1)
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:] (2)
        with torch.no_grad():
            logits = model(idx_cond)

        logits = logits[:, -1, :] (3)
```

```
        probas = torch.softmax(logits, dim=-1) (4)
        idx_next = torch.argmax(probas, dim=-1, keepdim=True) (5)
        idx = torch.cat((idx, idx_next), dim=1) (6)

    return idx
```

(1) idx è un array (`batch, n_token`) di indici nel contesto corrente.

(2) Taglia il contesto corrente se supera le dimensioni supportate per il contesto, per esempio, se il modello LLM supporta solo cinque token e il contesto è 10-dimensionale, solo gli ultimi cinque token vengono utilizzati come contesto.

(3) Si concentra solo sull'ultimo passo, in modo che (`batch, n_token, vocab_size`) diventi (`batch, vocab_size`).

(4) `probas` ha forma (`batch, vocab_size`).

(5) idx_next ha forma (`batch, 1`).

(6) Aggiunge l'indice campionato alla sequenza in esecuzione, dove `idx` ha forma (`batch, n_token+1`).

Questo codice mostra una semplice implementazione di un ciclo generativo per un modello di linguaggio usando PyTorch. Itera per un determinato numero (specificato) di nuovi token da generare, taglia il contesto corrente per adattarlo alle dimensioni massime del contesto del modello, calcola le previsioni e quindi seleziona il token successivo in base alla previsione di probabilità più elevata.

Per programmare la funzione `generate_text_simple()`, utilizziamo una funzione `softmax()` per convertire i logit in una distribuzione di probabilità, da cui identifichiamo la posizione col valore più elevato tramite `torch.argmax()`. La funzione `softmax()` è "monotona", ovvero conserva l'ordine dei suoi input quando li trasforma in output. Quindi, all'atto pratico, il passaggio a `softmax()` è ridondante, poiché la posizione col punteggio più elevato nel tensore di output della `softmax()` è la stessa che occupa nel tensore logit. In altre parole, potremmo applicare la funzione `torch.argmax()` direttamente al tensore

logit e ottenere risultati identici. Tuttavia, fornisco il codice di conversione per illustrare l'intero processo di trasformazione dei logit in probabilità, che può aggiungere ulteriori conoscenze, in modo che il modello generi il token successivo più probabile (*decodifica greedy*).

Quando implementeremo il codice di addestramento del GPT nel prossimo capitolo, utilizzeremo altre tecniche di campionamento per modificare gli output della softmax() in modo che il modello non selezioni sempre il token più probabile. Ciò introduce maggiore variabilità e creatività al testo generato.

Questo processo di generazione di un token-ID alla volta e di aggiunta al contesto tramite la funzione `generate_text_simple()` è ulteriormente illustrato nella Figura 4.18 (il processo di generazione del token-ID per ogni iterazione è descritto in dettaglio nella Figura 4.17). Generiamo i token-ID in modo iterativo. Per esempio, nell'iterazione 1, il modello riceve i token corrispondenti a "Hello, I am", prevede il token successivo (con ID 257, che è "a") e lo aggiunge all'input. Questo processo viene ripetuto finché il modello non produce la frase completa "Hello, I am a model ready to help" dopo sei iterazioni.

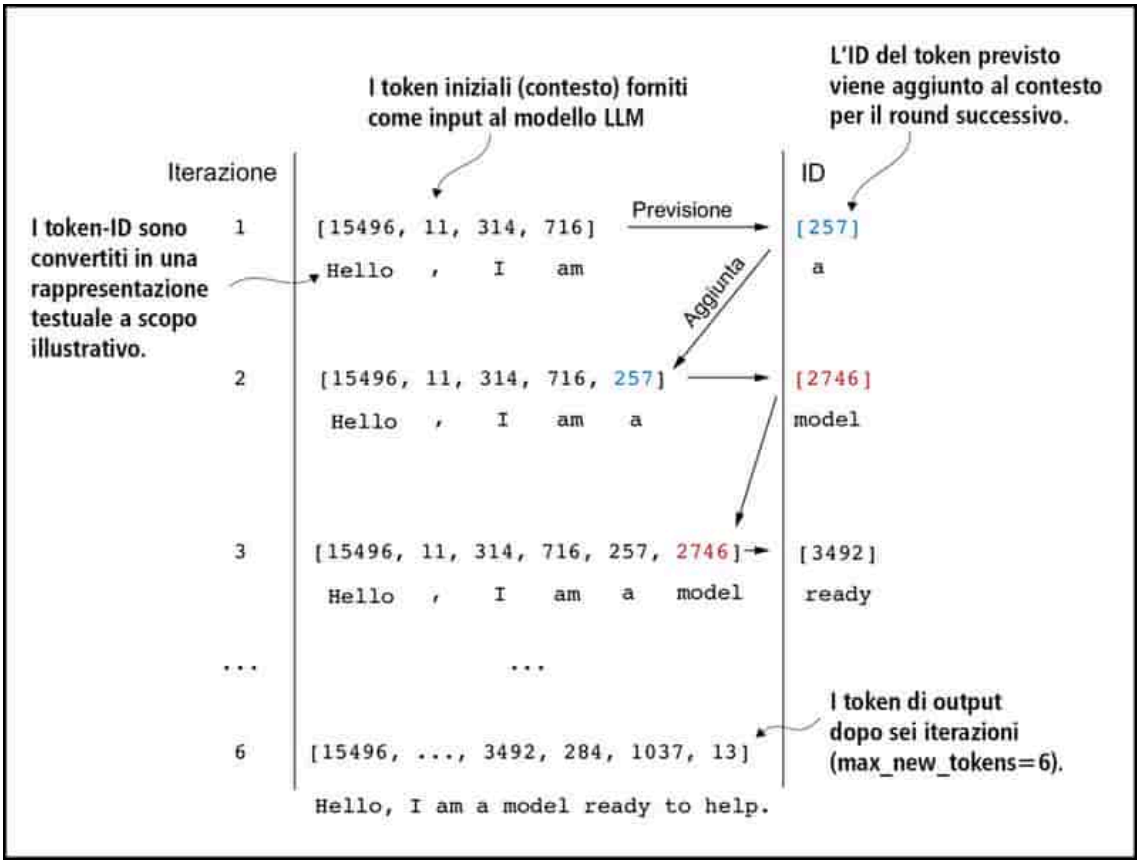


Figura 4.18 Le sei iterazioni di un ciclo di previsione del token, in cui il modello accetta come input una sequenza di token-ID iniziali, prevede il token successivo e aggiunge questo token alla sequenza di input per l'iterazione successiva. I token-ID sono stati tradotti nel testo corrispondente per facilitare la comprensione.

Proviamo ora la funzione `generate_text_simple()` con l'input `"Hello, I am"`. Innanzitutto, codifichiamo il contesto di input in token-ID:

```
start_context = "Hello, I am"
encoded = tokenizer.encode(start_context)
print("encoded:", encoded)
encoded_tensor = torch.tensor(encoded).unsqueeze(0) (1)
print("encoded_tensor.shape:", encoded_tensor.shape)
```

(1) Aggiunge le dimensioni del batch.

Gli ID codificati sono i seguenti:

```
encoded: [15496, 11, 314, 716]
encoded_tensor.shape: torch.Size([1, 4])
```

Ora, mettiamo il modello in modalità `.eval()`. Questo disabilita i componenti casuali come i dropout, che vengono usati solo durante l'addestramento, e impiega la funzione `generate_text_simple()` sul tensore di input codificato:

```
model.eval() (1)
out = generate_text_simple(
    model=model,
    idx=encoded_tensor,
    max_new_tokens=6,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Output:", out)
print("Output length:", len(out[0]))
```

(1) Disabilita il dropout poiché ora non stiamo addestrando il modello.

I token-ID di output risultanti sono i seguenti:

```
Output: tensor([[15496, 11, 314, 716, 27018, 24086, 47843, 30961, 42348, 7267]])
Output length: 10
```

Utilizzando il metodo `.decode` del tokenizer, possiamo riconvertire gli ID in testo:

```
decoded_text = tokenizer.decode(out.squeeze(0).tolist())
print(decoded_text)
```

L'output del modello in formato testuale è il seguente:

```
Hello, I am Featureiman Byeswickattribute argue
```

Come potete vedere, il modello ha generato un testo incomprensibile, che non somiglia affatto al testo coerente `Hello, I am a model ready to help.` Che cosa è successo? Il motivo per cui il modello non è stato in grado di produrre testo coerente è che non lo abbiamo ancora addestrato. Finora, abbiamo solo implementato l'architettura GPT e inizializzato un'istanza del modello GPT con pesi casuali iniziali. L'addestramento del modello è un argomento molto vasto e lo affronteremo nel prossimo capitolo.

Esercizio 4.3 Utilizzo di parametri di dropout distinti

All'inizio di questo capitolo, abbiamo definito un'impostazione globale `drop_rate` nel dizionario `GPT_CONFIG_124M` per impostare il tasso di dropout in vari punti

nell'architettura di `GPTModel`. Modificate il codice per specificare un nuovo valore di dropout per i vari layer di dropout del modello. (Suggerimento: ci sono tre punti distinti in cui abbiamo utilizzato i layer di dropout: il layer di embedding, il layer scorciatoia e il modulo di attenzione multi-head.)

Riepilogo

- La normalizzazione dei layer stabilizza l'addestramento garantendo che gli output di ogni layer abbiano una media e una varianza coerenti.
- Le connessioni scorciatoia saltano uno o più layer, inviando l'output di un layer direttamente a un layer più profondo, il che aiuta ad attenuare il problema dell'evanescenza dei gradienti durante l'addestramento delle reti neurali profonde, come i modelli LLM.
- I blocchi transformer sono un componente strutturale fondamentale dei modelli GPT, e combinano moduli di attenzione multi-head con maschera e reti feedforward completamente connesse che utilizzano la funzione di attivazione GELU.
- I GPT sono modelli LLM con molti blocchi transformer ripetuti con milioni o miliardi di parametri.
- I modelli GPT sono disponibili in varie dimensioni, per esempio con 124, 345, 762 e 1.542 milioni di parametri, che possiamo implementare con la stessa classe Python `GPTModel`.
- La capacità di generazione del testo di un modello LLM di tipo GPT implica la decodifica dei tensori di output in testo, prevedendo sequenzialmente un token alla volta in base a un dato contesto di input.
- Senza addestramento, un modello GPT genera testo incoerente, il che sottolinea l'importanza dell'addestramento del modello

per la generazione di testo coerente.

Capitolo 5

Pre-addestramento su dati non etichettati

Finora abbiamo implementato il campionamento dei dati e il meccanismo di attenzione e programmato l'architettura LLM. Ora è il momento di implementare una funzione di addestramento e pre-addestrare il modello LLM. Impareremo alcune semplici tecniche di valutazione del modello, per misurare la qualità del testo generato, che è un requisito per ottimizzare il modello LLM durante il processo di addestramento. Inoltre, discuteremo come caricare i pesi pre-addestrati, dando al modello LLM un solido punto di partenza per il fine-tuning. La Figura 5.1 mostra il piano generale, evidenziando ciò di cui parleremo in questo capitolo.

Parametri dei pesi

Nel contesto dei modelli LLM e di altri modelli di deep learning, i *pesi* si riferiscono ai parametri regolati dal processo di apprendimento. Questi pesi sono chiamati anche *parametri*. In framework come PyTorch, questi pesi sono memorizzati in layer `Linear`; li abbiamo usati per implementare il modulo di attenzione multi-head nel Capitolo 3 e il `GPTModel` nel Capitolo 4. Dopo aver inizializzato un layer (`new_layer = torch.nn.Linear(...)`), possiamo accedere ai suoi pesi tramite l'attributo `.weight` (`new_layer.weight`). Inoltre, per comodità, PyTorch consente l'accesso diretto a tutti i parametri addestrabili di un modello, inclusi i pesi e i bias, tramite il metodo `model.parameters()`, che utilizzeremo quando implementeremo l'addestramento del modello.

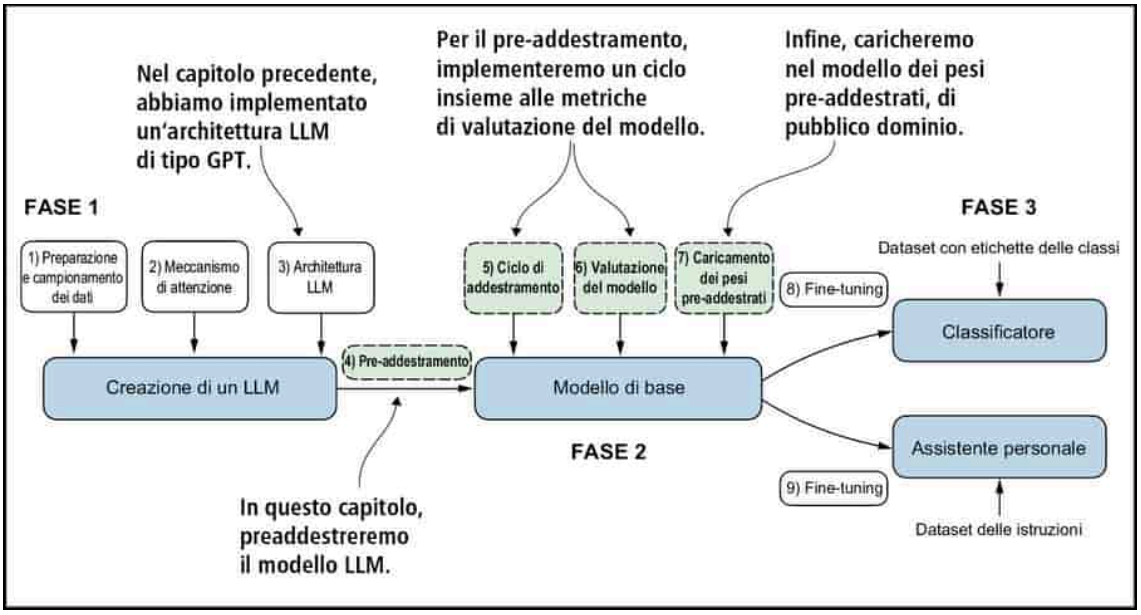


Figura 5.1 Le tre fasi principali della programmazione di un modello LLM. Questo capitolo si concentra sulla Fase 2: pre-addestramento del modello LLM (Fase 4), che include l'implementazione del codice di addestramento (Fase 5), la valutazione delle prestazioni (Fase 6) e il salvataggio e il caricamento dei pesi del modello (Fase 7).

Valutazione dei modelli generativi per il testo

Dopo aver brevemente riepilogato la generazione del testo del Capitolo 4, imposteremo il modello LLM per la generazione del testo e poi discuteremo vari modi per valutare la qualità del testo generato. Poi calcoleremo i valori loss di addestramento e validazione. La Figura 5.2 mostra gli argomenti trattati in questo capitolo, evidenziando questi primi tre passi.

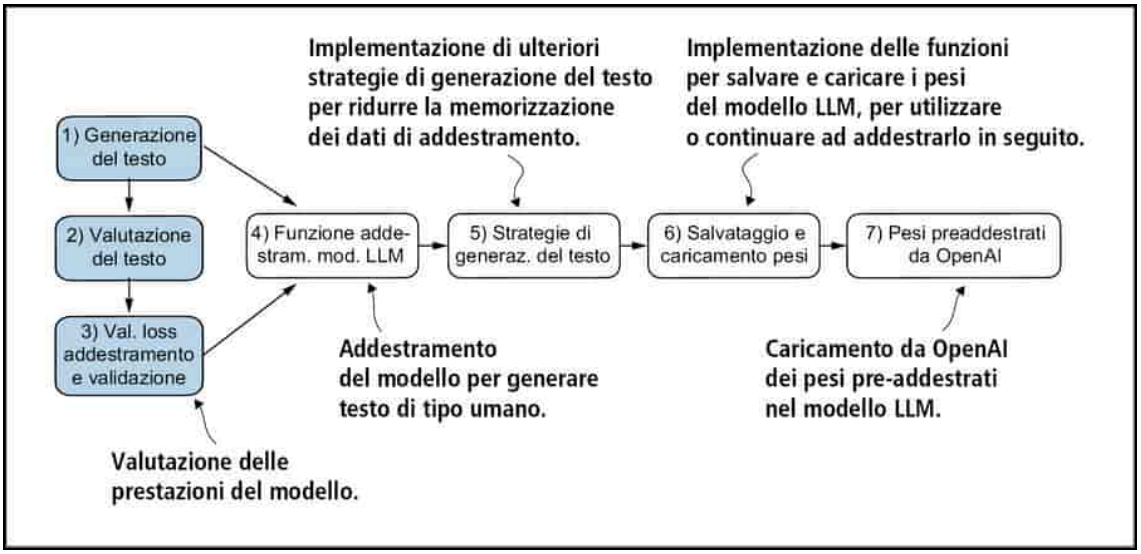


Figura 5.2 Una panoramica degli argomenti trattati in questo capitolo. Iniziamo riepilogando la generazione del testo (Passo 1) prima di passare a discutere le tecniche di base di valutazione del modello (Passo 2) e i valori loss di addestramento e validazione (Passo 3).

Utilizzo di GPT per generare testo

Impostiamo il modello LLM e ricapitoliamo brevemente il processo di generazione del testo implementato nel Capitolo 4. Iniziamo inizializzando il modello GPT che poi valuteremo e addestreremo utilizzando la classe `GPTModel` e il dizionario `GPT_CONFIG_124M` (vedi il Capitolo 4):

```
import torch
from chapter04 import GPTModel

GPT_CONFIG_124M = {
    "vocab_size": 50257,
    "context_length": 256, (1)
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate": 0.1, (2)
    "qkv_bias": False
}
torch.manual_seed(123)
```

```
model = GPTModel(GPT_CONFIG_124M)
model.eval()
```

(1) Riduciamo la lunghezza del contesto da 1024 a `256` token.

(2) È possibile e pratica comune impostare il dropout a `0`.

Considerando il dizionario `GPT_CONFIG_124M`, l'unica modifica che abbiamo apportato rispetto al capitolo precedente è che abbiamo ridotto la lunghezza del contesto (`context_length`) a 256 token. Questa modifica riduce le richieste computazionali di addestramento del modello, rendendo possibile l'addestramento su un normale computer portatile.

Originariamente, il modello GPT-2 con 124 milioni di parametri era configurato per gestire fino a 1024 token. Dopo il processo di addestramento, aggiorneremo l'impostazione delle dimensioni del contesto e caricheremo i pesi pre-addestrati per lavorare con un modello configurato per una lunghezza del contesto di 1024 token.

Utilizzando l'istanza `GPTModel`, impieghiamo la funzione `generate_text_simple()` del Capitolo 4 e introduciamo due funzioni utili: `text_to_token_ids` e `token_ids_to_text`. Queste funzioni facilitano la conversione tra rappresentazioni testuali e token, una tecnica che utilizzeremo in tutto il presente capitolo.

La Figura 5.3 illustra un processo di generazione del testo in tre passi, utilizzando un modello GPT. Innanzitutto, il tokenizer converte il testo di input in una serie di token-ID (vedi il Capitolo 2). In secondo luogo, il modello riceve questi token-ID e genera i logit corrispondenti, che sono vettori che rappresentano la distribuzione di probabilità per ogni token presente nel vocabolario (vedi il Capitolo 4). In terzo luogo, questi logit vengono riconvertiti in token-ID, che il tokenizer decodifica in testo, completando il ciclo dall'input testuale all'output testuale.

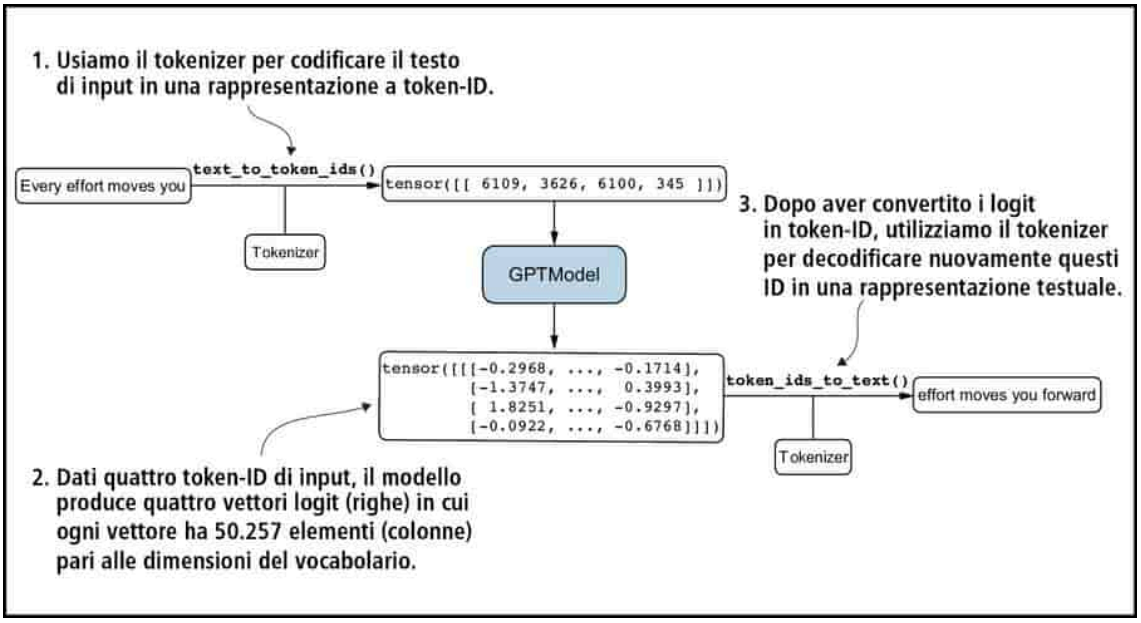


Figura 5.3 La generazione del testo comporta la codifica del testo in token-ID, che il modello LLM elabora in vettori logit. I vettori logit vengono poi riconvertiti in token-ID e infine de-tokenizzati in una rappresentazione testuale.

Possiamo implementare il processo di generazione del testo, come mostrato nel Listato 5.1.

Listato 5.1 Funzioni di servizio per la conversione del testo in token-ID.

```
import tiktoken
from chapter04 import generate_text_simple

def text_to_token_ids(text, tokenizer):
    encoded = tokenizer.encode(text, allowed_special={'<|endoftext|>'})
    encoded_tensor = torch.tensor(encoded).unsqueeze(0) (1)
    return encoded_tensor

def token_ids_to_text(token_ids, tokenizer):
    flat = token_ids.squeeze(0) (2)
    return tokenizer.decode(flat.tolist())

start_context = "Every effort moves you"
tokenizer = tiktoken.get_encoding("gpt2")

token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(start_context, tokenizer),
    max_new_tokens=10,
    context_size=GPT_CONFIG_124M["context_length"]
```

```
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

(1) `.unsqueeze(0)` aggiunge le dimensioni del batch.

(2) Rimuove le dimensioni del batch.

Utilizzando questo codice, il modello genera il seguente testo:

```
Output text:
 Every effort moves you rentingetic wasn? refres RexMeCHicular stren
```

Chiaramente, il modello non sta ancora producendo testo coerente, perché non è ancora stato sottoposto ad alcun addestramento. Per definire che cosa rende il testo “coerente” o “di alta qualità”, dobbiamo però implementare un metodo numerico per valutare il contenuto generato. Questo approccio ci consentirà di monitorare e migliorare le prestazioni del modello durante tutto il suo processo di addestramento.

Ora, calcoleremo una *metrica loss* per gli output generati. Questo valore loss funge da indicatore dei progressi e del successo dell’addestramento. Nei prossimi capitoli, quando perfezioneremo il modello LLM, esamineremo ulteriori metodologie per valutare la qualità del modello.

Calcolo del valore loss nella generazione del testo

Ora, esploriamo le tecniche per la valutazione numerica della qualità del testo generato durante l’addestramento calcolando un *valore loss per la generazione del testo*. Affronteremo questo argomento progressivamente, con un esempio pratico per chiarire e rendere applicabili i concetti, iniziando con un breve riepilogo di come vengono caricati i dati e di come viene generato il testo tramite la funzione `generate_text_simple()`.

La Figura 5.4 illustra il flusso complessivo dal testo di input al testo generato dal modello LLM utilizzando una procedura in cinque

passi. Questo processo di generazione del testo mostra che cosa fa internamente la funzione `generate_text_simple()`. Dobbiamo eseguire questi stessi passi iniziali prima di poter calcolare un valore loss che misuri la qualità del testo generato, cosa che faremo più avanti in questo paragrafo.

La Figura 5.4 delinea il processo di generazione del testo con un piccolo vocabolario di sette token, per poter far rimanere l'immagine su una sola pagina del libro. Tuttavia, il `GPTModel` funziona con un vocabolario molto più ampio, composto da 50.257 parole; pertanto, i token-ID nel codice seguente saranno compresi tra 0 e 50.256 anziché tra 0 e 6.

Inoltre, per semplicità, la Figura 5.4 mostra un singolo esempio di testo (`"every effort moves"`). Nel seguente esempio di codice. che implementa i passi rappresentati nella figura, lavoreremo con due esempi di input per il modello GPT (`"every effort moves"` e `"I really like"`).

Consideriamo questi due esempi di input, che sono già stati mappati sui token-ID (Figura 5.4, Passo 1):

```
inputs = torch.tensor([[16833, 3626, 6100],      # ["every effort moves",
                       [   40, 1107,  588]])     # "I really like"]
```

In base a questi input, i `target` contengono i token-ID che vogliamo che il modello produca:

```
targets = torch.tensor([[3626, 6100,   345],      # [" effort moves you",
                        [1107,  588, 11311]])     # " really like chocolate"]
```

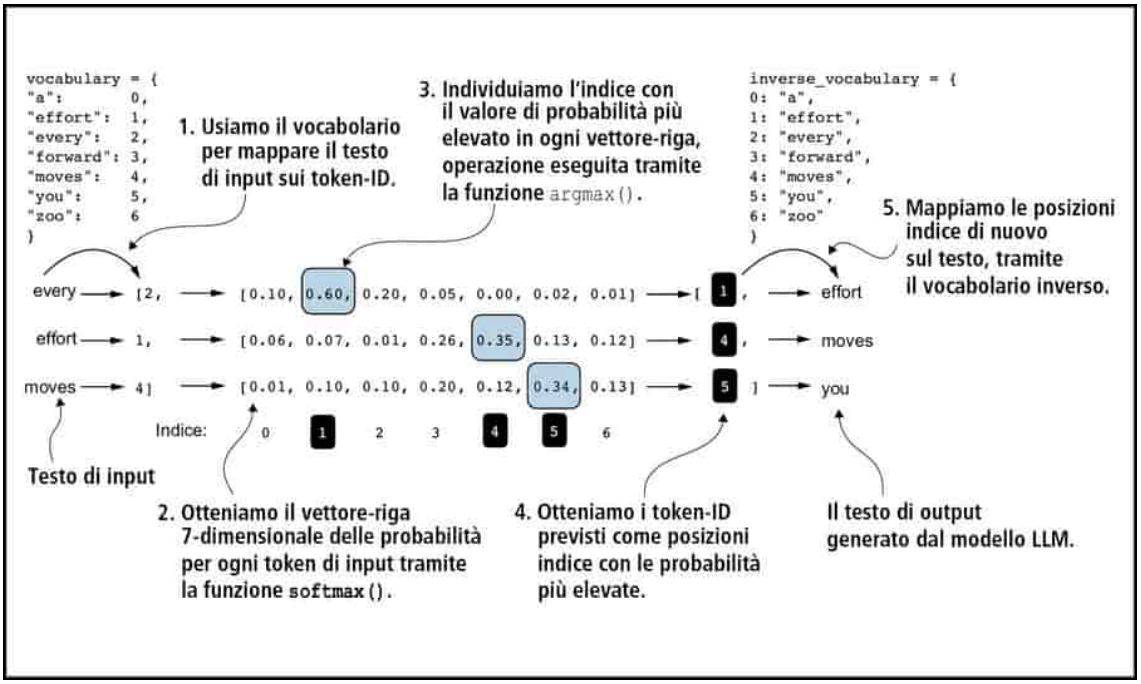


Figura 5.4 Per ciascuno dei tre token di input, mostrati a sinistra, calcoliamo un vettore contenente i punteggi di probabilità corrispondenti a ciascun token presente nel vocabolario. L'indice con punteggio di probabilità più elevato in ciascun vettore rappresenta il token-ID successivo più probabile. Questi token-ID associati ai punteggi di probabilità più elevati vengono selezionati e mappati nuovamente in un testo che rappresenta il testo generato dal modello.

Notate che i `targets` sono gli input, ma spostati di una posizione in avanti, un concetto che abbiamo trattato nel Capitolo 2 durante l'implementazione del data loader. Questa strategia di spostamento è fondamentale per insegnare al modello a predire il token successivo in una sequenza.

Ora inseriamo gli input nel modello per calcolare i vettori logit per i due esempi di input, ciascuno dei quali comprende tre token. Quindi applichiamo la funzione `softmax()` per trasformare questi logit in punteggi di probabilità (`probas`; Figura 5.4, Passo 2):

```
with torch.no_grad(): (1)
    logits = model(inputs)
```

```
probas = torch.softmax(logits, dim=-1) (2)
print(probas.shape)
```

(1) Disabilita il tracciamento dei gradienti, poiché non stiamo ancora eseguendo l'addestramento.

(2) Probabilità di ogni token del vocabolario.

Le dimensioni risultanti del tensore dei punteggi di probabilità (`probas`) sono le seguenti:

```
torch.Size([2, 3, 50257])
```

Il primo numero, `2`, corrisponde ai due esempi di input, ovvero le dimensioni del batch. Il secondo numero, `3`, corrisponde al numero di token in ogni input. Infine, l'ultimo numero corrisponde alla dimensionalità degli embedding, che è determinata dalle dimensioni del vocabolario. In seguito alla conversione da logit a probabilità tramite la funzione `softmax()`, la funzione `generate_text_simple()` poi converte i punteggi di probabilità nuovamente in testi (Figura 5.4, Passi 3-5).

Possiamo completare i Passi 3 e 4 applicando la funzione `argmax()` ai punteggi di probabilità per ottenere i token-ID corrispondenti:

```
token_ids = torch.argmax(probas, dim=-1, keepdim=True)
print("Token IDs:\n", token_ids)
```

Dato che abbiamo due batch di input, ciascuno contenente tre token, applicando la funzione `argmax()` ai punteggi di probabilità (Figura 5.4, Passo 3) si ottengono due set di output, ciascuno con tre token-ID previsti:

```
Token IDs:
 tensor([[[16657], (1)
          [  339],
          [42826]],
         [[49906], (2)
          [29669],
          [41751]]]
 )
```

(1) Primo batch.

(2) Secondo batch.

Infine, il Passo 5 converte nuovamente i token-ID in testo:

```
print(f"Targets batch 1: {token_ids_to_text(targets[0], tokenizer)}")
print(f"Outputs batch 1:"
      f" {token_ids_to_text(token_ids[0].flatten(), tokenizer)}")
```

Quando decodifichiamo questi token, scopriamo che essi sono piuttosto diversi dai token target che vogliamo che il modello generi:

```
Targets batch 1: effort moves you
Outputs batch 1: Armed heNetflix
```

Il modello produce testo casuale che è diverso dal testo target, perché non è stato ancora addestrato. Ora vogliamo valutare numericamente le prestazioni del testo generato dal modello tramite un valore loss (Figura 5.5). Esso non solo sarà utile per misurare la qualità del testo generato, ma sarà anche un elemento fondamentale per implementare la funzione di addestramento, che utilizzeremo per aggiornare i pesi del modello, per migliorare il testo generato.

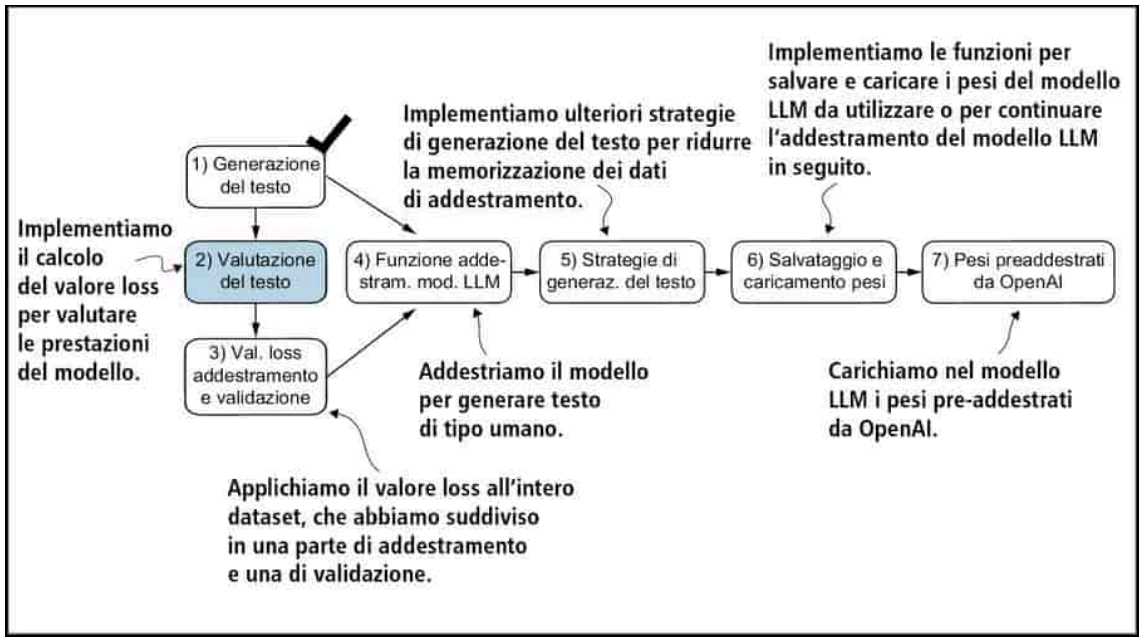


Figura 5.5 Una panoramica degli argomenti trattati in questo capitolo. Abbiamo completato il Passo 1. Ora siamo pronti per implementare la funzione di valutazione del testo (Passo 2).

Parte del processo che implementiamo di valutazione del testo, come mostra la Figura 5.5, consiste nel misurare “quanto” i token generati sono lontani dalle previsioni corrette (i target). La funzione di addestramento che implementeremo in seguito utilizzerà queste informazioni per adattare i pesi del modello per generare un testo più simile (idealmente, corrispondente) al testo target.

L’addestramento del modello mira ad aumentare la probabilità softmax nelle posizioni dell’indice corrispondenti ai token-ID target, come mostra la Figura 5.6. Questa probabilità softmax viene utilizzata anche nella metrica di valutazione che implementeremo in seguito per valutare numericamente gli output generati dal modello: maggiore è la probabilità nelle posizioni corrette, meglio è.

Ricordate che la Figura 5.6 mostra le probabilità softmax per un vocabolario compatto di sette token solo per poter far rientrare tutto in una singola figura. Ciò implica che i valori casuali iniziali si aggireranno intorno a 1/7, che equivale a circa 0,14. Tuttavia, il vocabolario che stiamo utilizzando per il nostro modello GPT-2 ha 50.257 token, quindi la maggior parte delle probabilità iniziali si aggirerà intorno a 0,00002 (1/50.257).

Per ciascuno dei due testi di input, possiamo visualizzare i punteggi delle probabilità softmax iniziali corrispondenti ai token target utilizzando il seguente codice:

```
text_idx = 0
target_probas_1 = probas[text_idx, [0, 1, 2], targets[text_idx]]
print("Text 1:", target_probas_1)

text_idx = 1
target_probas_2 = probas[text_idx, [0, 1, 2], targets[text_idx]]
print("Text 2:", target_probas_2)
```

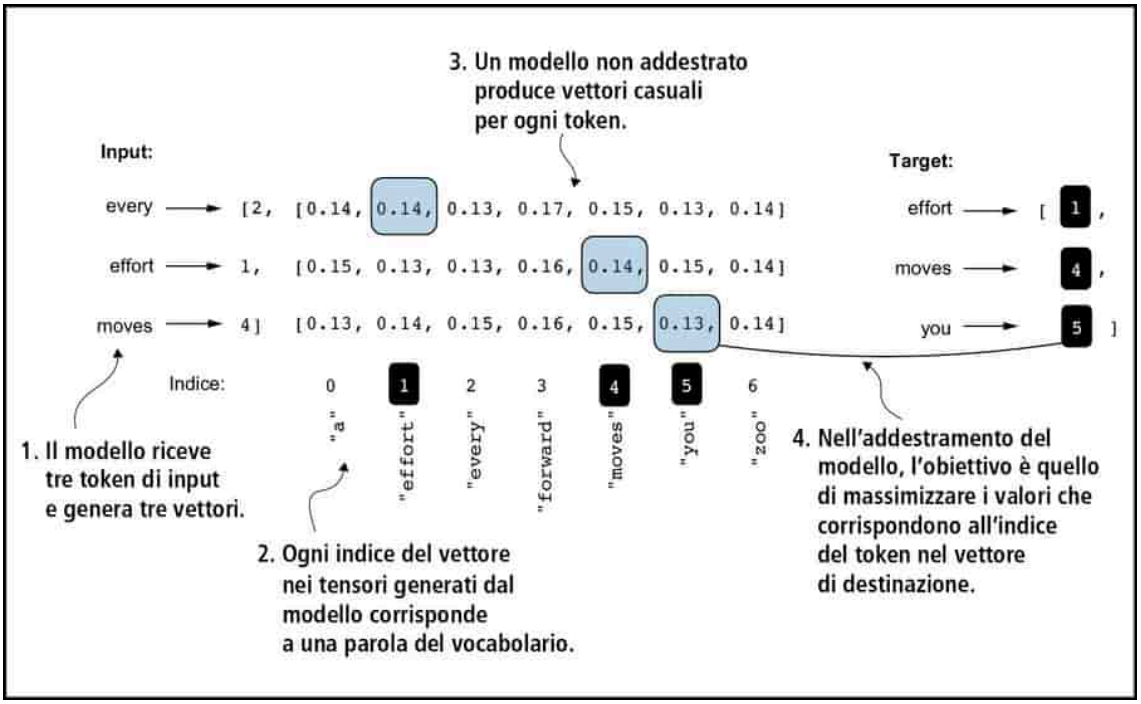


Figura 5.6 Prima dell'addestramento, il modello produce vettori di probabilità casuali del token successivo. L'obiettivo dell'addestramento del modello è garantire che siano massimizzati i valori di probabilità corrispondenti ai token-ID target, evidenziati.

Le tre probabilità dei token-ID target per ogni batch sono i seguenti:

```
Text 1: tensor([7.4541e-05, 3.1061e-05, 1.1563e-05])
Text 2: tensor([1.0337e-05, 5.6776e-05, 4.7559e-06])
```

L'obiettivo dell'addestramento di un modello LLM è quello di massimizzare le probabilità per il token corretto, il che implica l'aumento della sua probabilità rispetto agli altri token. In questo modo, ci assicuriamo che il modello LLM scelga costantemente come token successivo il token target, essenzialmente la parola successiva nella frase.

Retropropagazione

Come massimizziamo i valori della probabilità softmax corrispondenti ai token target? Il quadro generale è che aggiorniamo i pesi del modello in modo che il modello produca valori più elevati per i token-ID che vogliamo generare.

L'aggiornamento dei pesi viene eseguito tramite un processo chiamato *retropropagazione* (*backpropagation*), una tecnica standard per l'addestramento delle reti neurali profonde (vedi l'Appendice A per maggiori dettagli sulla retropropagazione e sull'addestramento dei modelli).
La retropropagazione richiede una funzione `loss`, che calcola la differenza tra l'output previsto dal modello (qui, le probabilità corrispondenti ai token-ID target) e l'output effettivo desiderato. Questa funzione `loss` misura quanto le previsioni del modello siano lontane dai valori target.

Ora, calcoleremo il valore loss per i punteggi di probabilità dei due batch di esempio, `target_probas_1` e `target_probas_2`. I passi principali sono illustrati nella Figura 5.7. Poiché abbiamo già applicato i Passi da 1 a 3 per ottenere `target_probas_1` e `target_probas_2`, procediamo col Passo 4, applicando il *logaritmo* ai punteggi di probabilità:

```
log_probas = torch.log(torch.cat((target_probas_1, target_probas_2)))
print(log_probas)
```

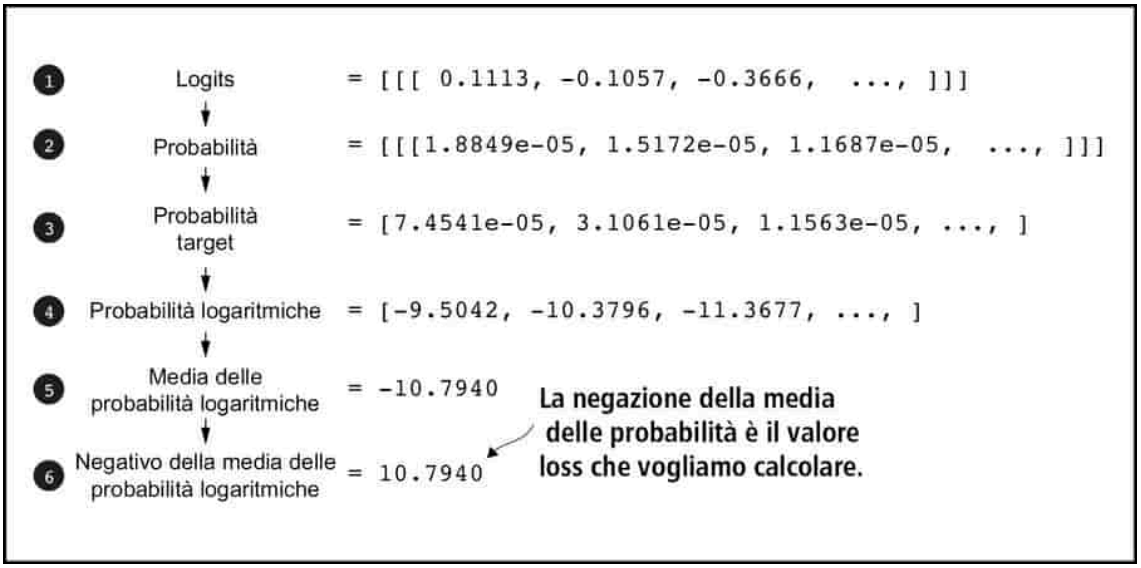


Figura 5.7 Il calcolo del valore loss comporta diversi passi. I Passi da 1 a 3, che abbiamo già completato, calcolano le probabilità del token corrispondenti ai tensori target. Nei Passi da 4 a 6 queste probabilità vengono poi trasformate tramite un logaritmo e poi ne viene calcolata la media.

Ciò si traduce nei seguenti valori:

```
tensor([-9.5042, -10.3796, -11.3677, -11.4798, -9.7764, -12.2561])
```

Lavorare con i logaritmi dei punteggi di probabilità è più gestibile nell'ottimizzazione matematica rispetto alla gestione diretta dei punteggi. Questo argomento esula dall'ambito di questo libro, ma l'ho approfondito nell'Appendice B.

Ora, combiniamo queste probabilità logaritmiche in un unico punteggio calcolando la media (Passo 5 nella Figura 5.7):

```
avg_log_probas = torch.mean(log_probas)
print(avg_log_probas)
```

Il punteggio medio delle probabilità logaritmiche è il seguente:

```
tensor(-10.7940)
```

L'obiettivo è quello di ottenere la media delle probabilità logaritmiche il più vicino possibile a `0` aggiornando i pesi del modello nell'ambito del processo di addestramento. Tuttavia, nel deep learning, è pratica comune non spingere la media delle probabilità logaritmiche proprio fino a 0, ma piuttosto portare a `0` la negazione della media delle probabilità logaritmiche. La negazione della media delle probabilità logaritmiche è semplicemente la media delle probabilità logaritmiche moltiplicata per `-1`, che corrisponde al Passo 6 della Figura 5.7:

```
neg_avg_log_probas = avg_log_probas * -1
print(neg_avg_log_probas)
```

Si ottiene `tensor(10.7940)`. Nel deep learning, il termine per trasformare questo valore negativo, -10.7940, in 10.7940, è noto come *loss a entropia incrociata* (*cross entropy loss*). PyTorch ci torna utile, poiché offre già una funzione `cross_entropy` che si occupa di tutti questi sei passi per noi (Figura 5.7).

Loss a entropia incrociata

In sostanza, il valore loss a entropia incrociata è una metrica diffusa nel machine learning e nel deep learning che misura la differenza tra due distribuzioni di probabilità, in genere la distribuzione reale delle etichette (in questo caso, i token di

un dataset) e la distribuzione prevista da un modello (per esempio, le probabilità dei token generate da un modello LLM).
Nel contesto del machine learning e in particolare in framework come PyTorch, la funzione `cross_entropy` calcola questa metrica per risultati discreti, che è simile alla negazione della media delle probabilità logaritmiche dei token target date le probabilità dei token generate dal modello, rendendo i termini “entropia incrociata” e “negazione della media delle probabilità logaritmiche” correlati e spesso usati come sinonimi nella pratica.

Prima di applicare la funzione `cross_entropy`, ricordiamo brevemente la forma dei logit e dei tensori target:

```
print("Logits shape:", logits.shape)
print("Targets shape:", targets.shape)
```

Abbiamo i seguenti risultati:

```
Logits shape: torch.Size([2, 3, 50257])
Targets shape: torch.Size([2, 3])
```

Come potete vedere, il tensore `logits` ha tre dimensioni: dimensioni del batch, numero di token e dimensioni del vocabolario. Il tensore `targets` ha due dimensioni: dimensioni del batch e numero di token.

Per la funzione `loss cross_entropy` di PyTorch, vogliamo appiattire questi tensori combinandoli sulle dimensioni del batch:

```
logits_flat = logits.flatten(0, 1)
targets_flat = targets.flatten()
print("Flattened logits:", logits_flat.shape)
print("Flattened targets:", targets_flat.shape)
```

Le dimensioni risultanti del tensore sono le seguenti:

```
Flattened logits: torch.Size([6, 50257])
Flattened targets: torch.Size([6])
```

Ricordare che `targets` sono i token-ID che vogliamo siano generati dal modello LLM e `logits` contiene gli output del modello non ridimensionato prima che entrino nella funzione `softmax()` per ottenere i punteggi di probabilità.

In precedenza, abbiamo applicato la funzione `softmax()`, selezionato i punteggi di probabilità corrispondenti agli ID target e calcolato le

negazioni delle medie delle probabilità logaritmiche. La funzione `cross_entropy` di PyTorch si occuperà di tutti questi passi per noi:

```
loss = torch.nn.functional.cross_entropy(logits_flat, targets_flat)
print(loss)
```

Il valore loss risultante è la stesso che abbiamo ottenuto in precedenza applicando manualmente i singoli passi della Figura 5.7:

```
tensor(10.7940)
```

Perplessità

La *perplessità* è una metrica spesso utilizzata insieme al valore loss a entropia incrociata per valutare le prestazioni dei modelli in compiti come la modellazione del linguaggio. Può fornire un modo più interpretabile per comprendere l'incertezza di un modello nel predire il token successivo in una sequenza.

La perplessità misura con quale efficacia la distribuzione di probabilità prevista dal modello corrisponda alla distribuzione effettiva delle parole del dataset. In modo analogo a quanto detto per il valore loss, una perplessità inferiore indica che le previsioni del modello sono più vicine alla distribuzione effettiva.

La perplessità può essere calcolata come `perplexity = torch.exp(loss)`, che restituisce `tensor(48725.8203)` se applicato al valore loss calcolato in precedenza.

La perplessità è spesso considerata più interpretabile del valore loss grezzo, perché indica le dimensioni effettive del vocabolario sul quale il modello è incerto a ogni passo. Nell'esempio dato, ciò si tradurrebbe nel fatto che il modello non è sicuro di quale tra i 48.725 token presenti nel vocabolario generare come token successivo.

Abbiamo calcolato il valore loss per due piccoli input di testo a scopo illustrativo. Ora, applicheremo il calcolo del valore loss agli interi set di addestramento e di validazione.

Calcolo dei valori loss dei set di addestramento e di validazione

Dobbiamo prima preparare i set di addestramento e validazione che utilizzeremo per addestrare il modello LLM. Quindi, come evidenziato nella Figura 5.8, calcoleremo l'entropia incrociata per i

set di addestramento e validazione, che è un componente importante del processo di addestramento del modello.

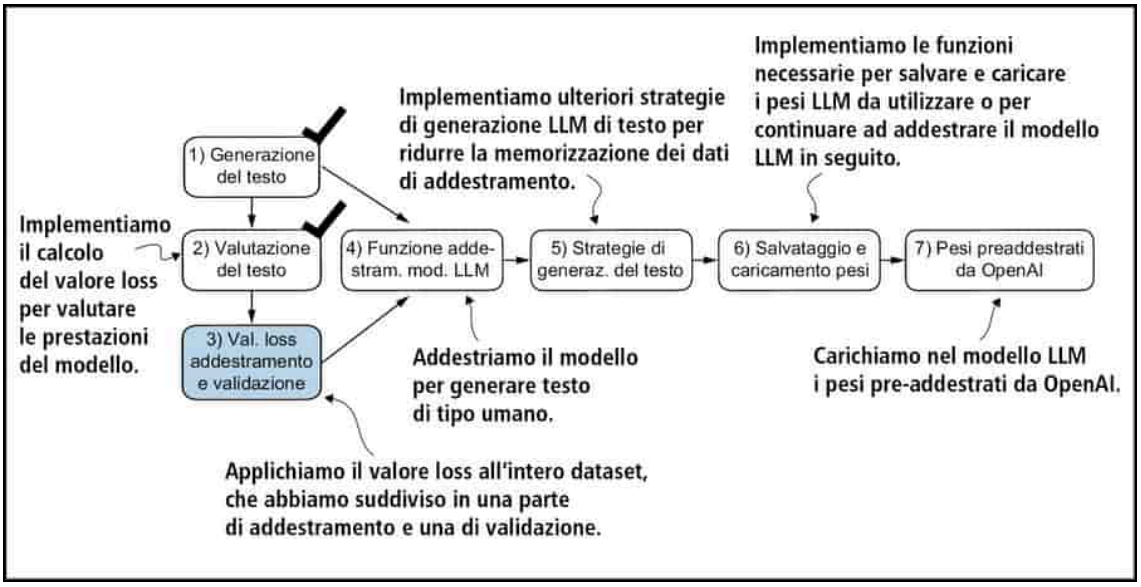


Figura 5.8 Dopo aver completato i Passi 1 e 2, incluso il calcolo del valore loss a entropia incrociata, ora possiamo applicare questo calcolo all'intero dataset di testi che utilizzeremo per l'addestramento del modello.

Per calcolare il valore loss sui set di addestramento e validazione, utilizziamo un dataset di testi molto piccolo, il racconto breve *The Verdict* di Edith Wharton, con cui abbiamo già lavorato nel Capitolo 2. Selezionando un testo di pubblico dominio, fughiamo ogni preoccupazione relativa ai diritti di utilizzo. Inoltre, l'utilizzo di un dataset così piccolo consente l'esecuzione in pochi minuti di esempi di codice su un normale computer portatile, anche senza una GPU di fascia alta, il che è particolarmente utile per scopi didattici.

NOTA

I lettori interessati possono anche utilizzare il codice supplementare di questo libro per preparare un dataset su larga scala composto dagli oltre 60.000 libri di pubblico dominio del Progetto Gutenberg e addestrare un modello LLM su questi (vedi l'Appendice D per i dettagli).

Il costo del pre-addestramento dei modelli LLM

Per mettere in prospettiva la portata del nostro progetto, considerate l'addestramento del modello Llama 2 da 7 miliardi di parametri, un modello LLM relativamente popolare e disponibile a tutti. Questo modello ha richiesto 184.320 ore su costose GPU A100, elaborando 2 trilioni di token.

Al momento della stesura, l'utilizzo di un server cloud 8 × A100 su AWS costa circa 30 dollari all'ora. Una stima approssimativa del costo totale dell'addestramento di un tale LLM raggiunge circa 690.000 dollari (184.320 ore divise per 8 e moltiplicate per 30 dollari).

Il seguente codice carica il racconto breve *The Verdict*:

```
file_path = "the-verdict.txt"
with open(file_path, "r", encoding="utf-8") as file:
    text_data = file.read()
```

Dopo aver caricato il dataset, possiamo controllare il numero di caratteri e token in esso presenti:

```
total_characters = len(text_data)
total_tokens = len(tokenizer.encode(text_data))
print("Characters:", total_characters)
print("Tokens:", total_tokens)
```

L'output è il seguente:

```
Characters: 20479
Tokens: 5145
```

Con solo 5.145 token, il testo potrebbe sembrare troppo piccolo per addestrare un modello LLM, ma come ho già detto qui lo scopo è puramente educativo, in modo che possiamo eseguire il codice in pochi minuti invece che in settimane. Inoltre, più tardi caricheremo nel codice i pesi pre-addestrati da OpenAI.

Ora, dividiamo il dataset in un set di addestramento e un set di validazione e utilizziamo i data loader del Capitolo 2 per preparare i batch per l'addestramento del modello LLM. Questo processo è rappresentato nella Figura 5.9. A causa di vincoli di spazio nella pagina, utilizziamo `max_length=6`. Tuttavia, per i data loader effettivi, occorre impostare `max_length` uguale alla lunghezza del contesto di 256 token supportata dal modello LLM, in modo che esso veda testi più lunghi durante l'addestramento.

NOTA

Per motivi di semplicità ed efficienza, stiamo addestrando il modello con dati di addestramento presentati in blocchi di dimensioni simili. Tuttavia, all'atto pratico, può anche essere utile addestrare un modello LLM con input di lunghezza variabile, per aiutare il modello LLM a generalizzare meglio su diversi tipi di input.

Per implementare la suddivisione e il caricamento dei dati, definiamo innanzitutto un `train_ratio` per utilizzare il 90 percento dei dati per l'addestramento e il restante 10 percento come dati di validazione, per la valutazione del modello durante l'addestramento:

```
train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
train_data = text_data[:split_idx]
val_data = text_data[split_idx:]
```

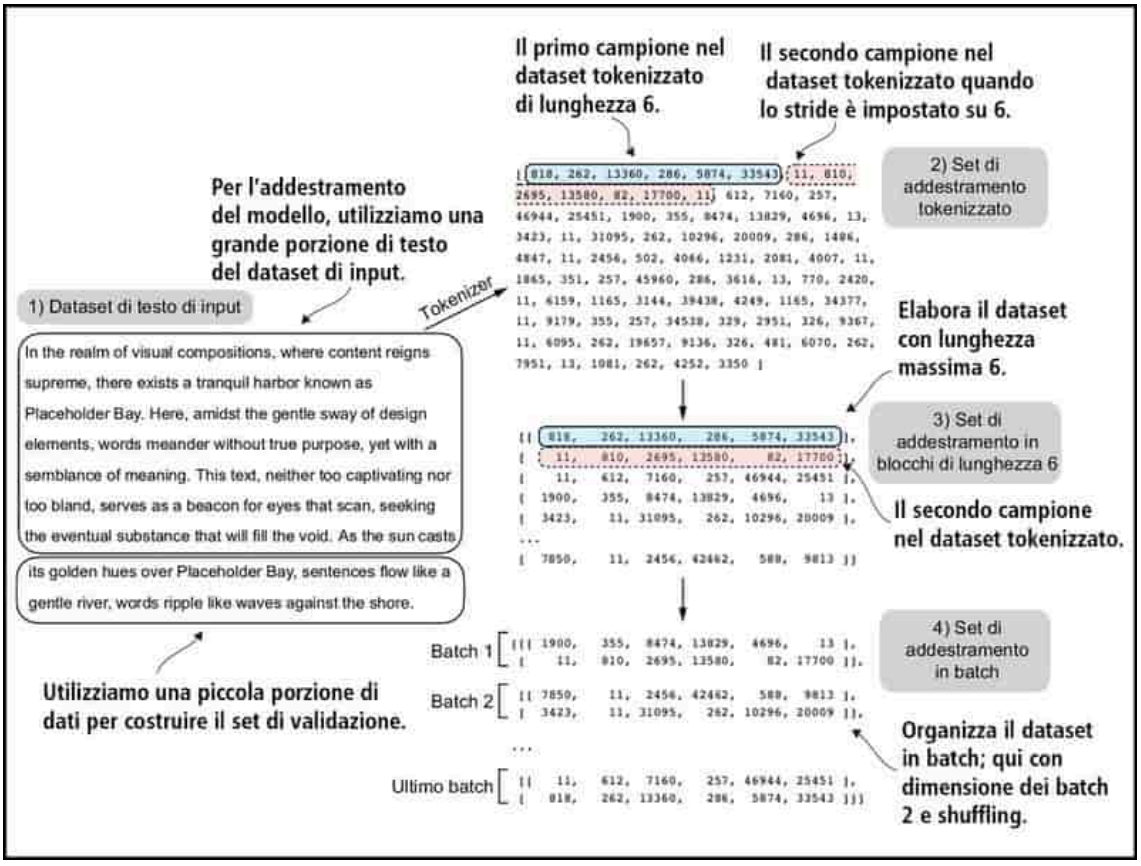


Figura 5.9 Quando prepariamo i data loader, dividiamo il testo di input in set di addestramento e validazione. Quindi tokenizziamo il testo (mostrato solo per la porzione del set di addestramento, per semplicità) e dividiamo il testo tokenizzato in blocchi di

lunghezza specificata dall'utente (qui, 6). Infine, mescoliamo le righe e organizziamo il testo suddiviso in blocchi in batch (qui, dimensioni del batch pari a 2), che possiamo usare per l'addestramento del modello.

Utilizzando i sottoinsiemi `train_data` e `val_data`, ora possiamo creare il rispettivo data loader riutilizzando il codice `create_dataloader_v1` del Capitolo 2:

```
from chapter02 import create_dataloader_v1
torch.manual_seed(123)

train_loader = create_dataloader_v1(
    train_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=True,
    shuffle=True,
    num_workers=0
)
val_loader = create_dataloader_v1(
    val_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=False,
    shuffle=False,
    num_workers=0
)
```

Abbiamo utilizzato una dimensione di batch relativamente piccola per ridurre la richiesta di risorse computazionali, perché stavamo lavorando con un dataset molto piccolo. All'atto pratico, non è insolito addestrare i modelli LLM con dimensioni del batch di 1024 o più.

Come controllo facoltativo, possiamo iterare sui data loader per assicurarci che siano stati creati correttamente:

```
print("Train loader:")
for x, y in train_loader:
    print(x.shape, y.shape)

print("\nValidation loader:")
for x, y in val_loader:
    print(x.shape, y.shape)
```

Dovremmo ottenere i seguenti output:

```
Train loader:
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
Validation loader:
torch.Size([2, 256]) torch.Size([2, 256])
```

Sulla base del codice di output precedente, nel set di addestramento abbiamo nove batch con due campioni e 256 token ciascuno. Poiché abbiamo assegnato solo il 10 percento dei dati alla validazione, c'è un solo batch di validazione composto da due esempi di input. Come previsto, i dati di input (`x`) e i dati target (`y`) hanno la stessa forma (le dimensioni del batch moltiplicate per il numero di token in ogni batch), poiché i target sono gli input spostati di una posizione, come abbiamo già visto nel Capitolo 2.

Ora, implementiamo una funzione di servizio per calcolare il valore loss a entropia incrociata di un dato batch restituito tramite il loader di addestramento e validazione:

```
def calc_loss_batch(input_batch, target_batch, model, device):
    input_batch = input_batch.to(device)   (1)
    target_batch = target_batch.to(device) (1)
    logits = model(input_batch)
    loss = torch.nn.functional.cross_entropy(
        logits.flatten(0, 1), target_batch.flatten()
    )
    return loss
```

(1) Il trasferimento a un dato dispositivo consente di trasferire i dati a una GPU.

Ora possiamo utilizzare questa funzione di servizio `calc_loss_batch()`, che calcola il valore loss per un singolo batch, per implementare la seguente funzione `calc_loss_loader()`, che calcola il valore loss su tutti i batch campionati da un dato loader.

Listato 5.2 Funzione per calcolare il valore loss di addestramento e validazione.

```
def calc_loss_loader(data_loader, model, device, num_batches=None):
    total_loss = 0.
    if len(data_loader) == 0:
        return float("nan")
    elif num_batches is None:
        num_batches = len(data_loader) (1)
    else:
        num_batches = min(num_batches, len(data_loader)) (2)
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            total_loss += loss.item() (3)
        else:
            break
    return total_loss / num_batches (4)
```

(1) Itera su tutti i batch se non è specificato alcun `num_batches` fisso.

(2) Riduce il numero di batch per corrispondere al numero totale di batch nel loader se `num_batches` supera il numero di batch nel loader.

(3) Somma il valore loss per ogni batch.

(4) Calcola la media del loss su tutti i batch.

Per default, la funzione `calc_loss_loader()` itera su tutti i batch di un data loader, accumula il valore loss nella variabile `total_loss`, e poi calcola la media del valore loss sul numero totale dei batch. In alternativa, possiamo specificare un numero più piccolo di batch tramite `num_batches`, per velocizzare la valutazione durante l'addestramento del modello.

Vediamo ora questa funzione `calc_loss_loader()` in azione, applicandola ai loader dei set di addestramento e di validazione:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device) (1)
with torch.no_grad(): (2)
    train_loss = calc_loss_loader(train_loader, model, device) (3)
    val_loss = calc_loss_loader(val_loader, model, device)
print("Training loss:", train_loss)
print("Validation loss:", val_loss)
```

(1) Se la macchina ha una GPU CUDA, il modello LLM si addestrerà sulla GPU senza apportare modifiche al codice.

(2) Disabilita il tracciamento dei gradienti per motivi di efficienza, perché non stiamo ancora eseguendo l'addestramento.
(3) Tramite l'impostazione `device`, ci assicuriamo che i dati vengano caricati sullo stesso dispositivo del modello LLM.

I valori di loss risultanti sono i seguenti:

```
Training loss: 10.98758347829183
Validation loss: 10.98110580444336
```

I valori di loss sono relativamente alti perché il modello non è ancora stato addestrato. Per fare un paragone, il valore loss si avvicina a `0` se il modello impara a generare i token successivi man mano che appaiono nei set di addestramento e validazione.

Ora che abbiamo un modo per misurare la qualità del testo generato, addestreremo il modello LLM a ridurre questo valore loss, in modo che diventi sempre più efficace nel generare testo, come mostra la Figura 5.10.

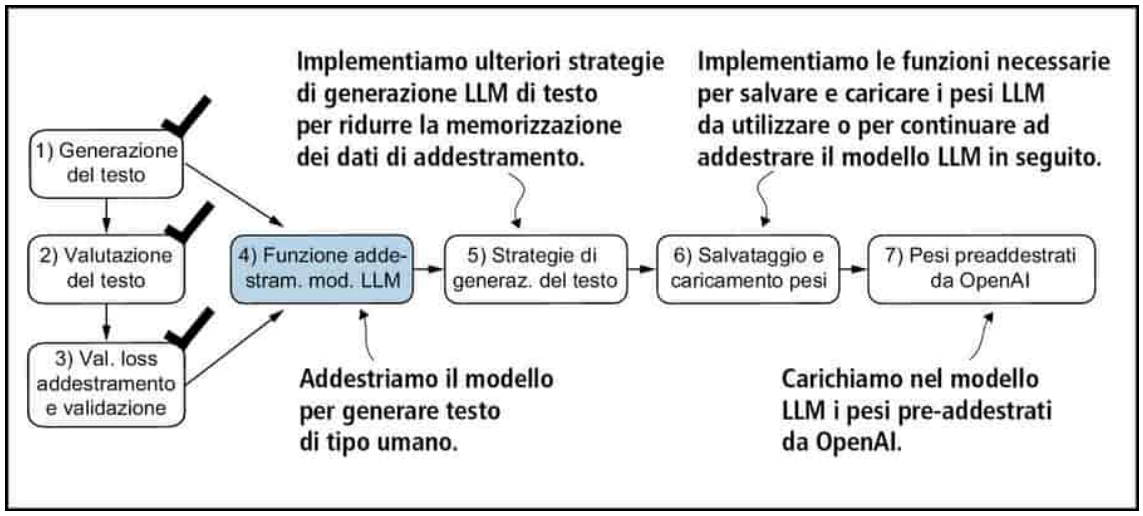


Figura 5.10 Abbiamo riepilogato il processo di generazione del testo (Passo 1) e implementato alcune semplici tecniche di valutazione del modello (Passo 2) per calcolare i valori loss dei set di addestramento e di validazione (Passo 3). Ora, passeremo alle funzioni di addestramento, e preaddestreremo il modello LLM (Passo 4).

Ora, ci concentreremo sul pre-addestramento del modello LLM. Dopo l'addestramento del modello, implementeremo altre strategie di generazione del testo, e proveremo a salvare e caricare i pesi dei modelli pre-addestrati.

Addestramento di un modello LLM

È finalmente giunto il momento di implementare il codice per il pre-addestramento del modello LLM, il `GPTModel`. Per farlo, ci concentriamo su un ciclo di addestramento semplice, con lo scopo di mantenere il codice conciso e leggibile.

NOTA

I lettori interessati possono scoprire tecniche più avanzate, tra cui *Learning Rate Warmup*, *Cosine Annealing* e *Gradient Clipping*, nell'Appendice D.

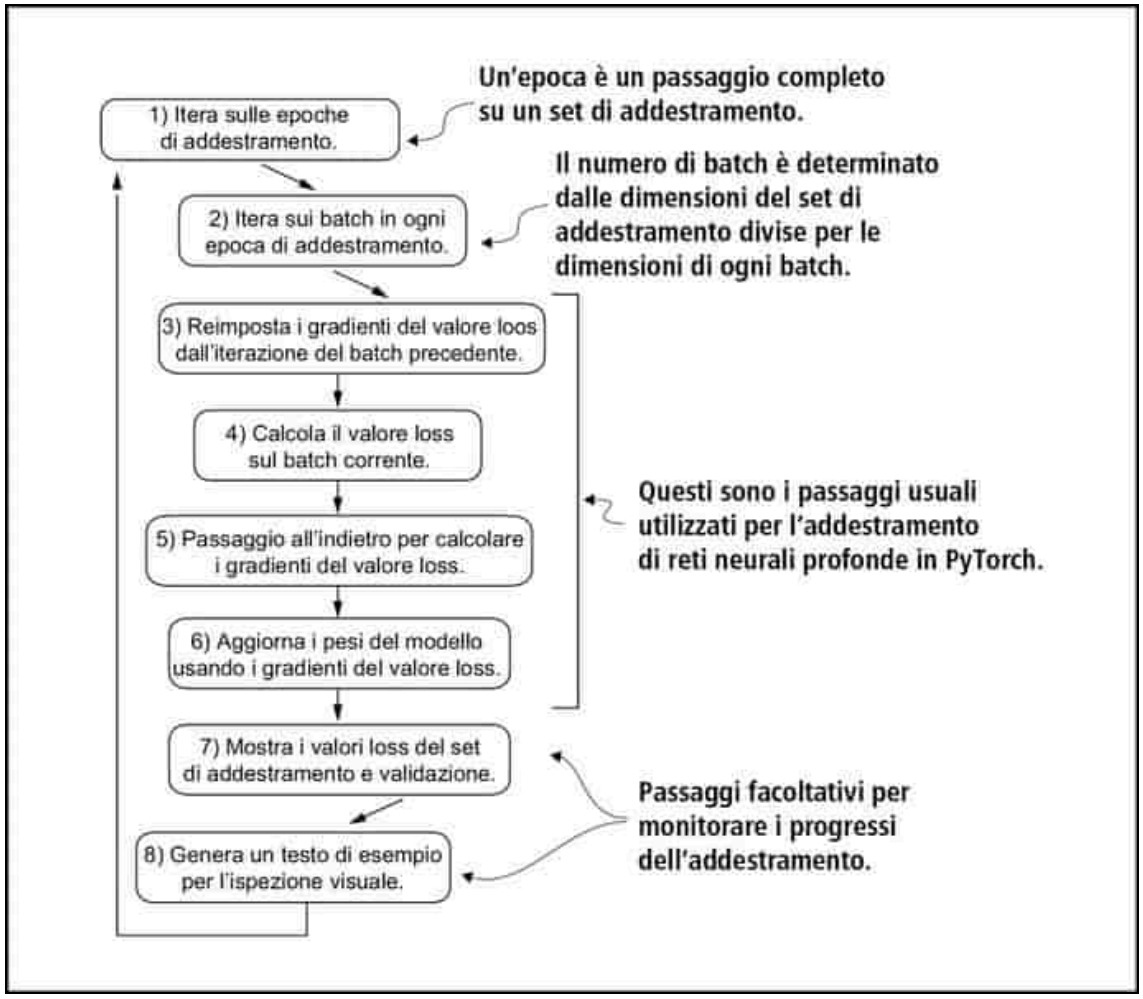


Figura 5.11 Un tipico ciclo per l'addestramento delle reti neurali profonde in PyTorch è costituito da numerosi passi, che iterano sui batch del set di addestramento per diverse epoche. A ogni ciclo, calcoliamo il valore loss per ogni batch del set di addestramento, per determinare i gradienti del valore loss, che utilizziamo per aggiornare i pesi del modello in modo che il valore loss del set di addestramento sia minimizzato.

Il diagramma di flusso nella Figura 5.11 raffigura un tipico flusso di lavoro di addestramento di reti neurali con PyTorch, che utilizziamo anche per questo modello LLM. Delinea otto passi, a partire dall'iterazione su ogni epoca, l'elaborazione dei batch, la reimpostazione dei gradienti, il calcolo del valore loss e dei nuovi gradienti e l'aggiornamento dei pesi, per concludere con i passi di

monitoraggio, come la stampa dei valori loss e la generazione di campioni di testo.

> **NOTA**
> Se non avete familiarità con l'addestramento delle reti neurali profonde con PyTorch e uno qualsiasi di questi passi non vi è familiare, prendete in considerazione la lettura dell'Appendice A.

Possiamo implementare questo flusso di addestramento tramite la funzione `train_model_simple()`.

Listato 5.3 La funzione principale per il pre-addestramento dei modelli LLM.

```
def train_model_simple(model, train_loader, val_loader, optimizer, device,
num_epochs,
                       eval_freq, eval_iter, start_context, tokenizer):
    train_losses, val_losses, track_tokens_seen = [], [], [] (1)
    tokens_seen, global_step = 0, -1

    for epoch in range(num_epochs): (2)
        model.train()
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad() (3)
            loss = calc_loss_batch(input_batch, target_batch, model, device)
            loss.backward() (4)
            optimizer.step() (5)
            tokens_seen += input_batch.numel()
            global_step += 1

            if global_step % eval_freq == 0: (6)
                train_loss, val_loss = evaluate_model(model, train_loader,
val_loader,
                                                      device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                track_tokens_seen.append(tokens_seen)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, "
                      f"Val loss {val_loss:.3f}"
                )

        generate_and_print_sample(model, tokenizer, device, start_context) (7)
    return train_losses, val_losses, track_tokens_seen
```

(1) Inizializza le liste per tracciare i valori loss e i token visti.
(2) Avvia il ciclo di addestramento principale.
(3) Reimposta i gradienti di loss dall'iterazione del batch precedente.
(4) Calcola i gradienti di loss.
(5) Aggiorna i pesi del modello utilizzando i gradienti di loss.

(6) Fase di valutazione, facoltativa.
(7) Mostra un testo di esempio dopo ogni epoca.

Notate che la funzione `train_model_simple()` che abbiamo appena creato usa due funzioni che non abbiamo ancora definito: `evaluate_model()` e `generate_and_print_sample()`.

La funzione `evaluate_model()` corrisponde al Passo 7 della Figura 5.11: mostra i valori loss del set di addestramento e di validazione dopo ogni aggiornamento del modello, per permetterci di valutare se l'addestramento migliora il modello. Più specificamente, la funzione `evaluate_model()` calcola il valore loss sul set di addestramento e di validazione, assicurandosi che il modello sia in modalità di valutazione con il tracciamento dei gradienti e il dropout disabilitati nel calcolo del valore loss sul set di addestramento e di validazione:

```
def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval() (1)
    with torch.no_grad(): (2)
        train_loss = calc_loss_loader(train_loader, model, device,
num_batches=eval_iter)
        val_loss = calc_loss_loader(val_loader, model, device,
num_batches=eval_iter)
    model.train()
    return train_loss, val_loss
```

(1) Il dropout viene disabilitato durante la valutazione per ottenere risultati stabili e riproducibili.
(2) Disabilita il tracciamento dei gradienti, che non è richiesto durante la valutazione, per ridurre il sovraccarico computazionale.

Simile a `evaluate_model()`, `generate_and_print_sample()` è una funzione di servizio che utilizziamo per monitorare se il modello migliora durante l'addestramento. In particolare, la funzione `generate_and_print_sample()` accetta come input un frammento di testo (`start_context`), lo converte in un token-ID e lo invia al modello LLM perché generi un campione di testo utilizzando la funzione `generate_text_simple()` che abbiamo utilizzato in precedenza:

```
def generate_and_print_sample(model, tokenizer, device, start_context):
    model.eval()
    context_size = model.pos_emb.weight.shape[0]
    encoded = text_to_token_ids(start_context, tokenizer).to(device)
    with torch.no_grad():
        token_ids = generate_text_simple(model=model, idx=encoded,
max_new_tokens=50,
                                         context_size=context_size
        )
decoded_text = token_ids_to_text(token_ids, tokenizer)
print(decoded_text.replace("\n", " ")) (1)
model.train()
```

(1) Visualizzazione compatta.

Mentre la funzione `evaluate_model()` fornisce una stima numerica dei progressi nell'addestramento del modello, la funzione `generate_and_print_sample()` fornisce un esempio di testo concreto, generato dal modello, per valutarne l'efficacia durante l'addestramento.

> **AdamW**
> Gli ottimizzatori *Adam* sono una scelta molto utilizzata per l'addestramento delle reti neurali profonde. Tuttavia, nel ciclo di addestramento, optiamo per l'ottimizzatore *AdamW*, una variante di Adam che migliora l'approccio a decadimento dei pesi, volto a minimizzare la complessità del modello e prevenire l'overfitting penalizzando i pesi più grandi. Questa regolazione consente a AdamW di ottenere una regolarizzazione più efficace e una migliore generalizzazione; per questo AdamW viene frequentemente utilizzato nell'addestramento dei modelli LLM.

Vediamo il tutto in azione addestrando un'istanza di `GPTModel` per 10 epoche utilizzando un ottimizzatore `AdamW` e la funzione `train_model_simple()` definita in precedenza:

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
optimizer = torch.optim.AdamW(model.parameters(),
                              lr=0.0004, weight_decay=0.1
)
num_epochs = 10
train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=5,
    start_context="Every effort moves you", tokenizer=tokenizer
)
```

(1) Il metodo `.parameters()` restituisce tutti i parametri dei pesi addestrabili del modello.

L'esecuzione della funzione `train_model_simple()` avvia il processo di addestramento, che impiega circa 5 minuti per essere completato su un MacBook Air o un laptop simile. L'output prodotto da questa esecuzione è il seguente:

```
Ep 1 (Step 000000): Train loss 9.781, Val loss 9.933
Ep 1 (Step 000005): Train loss 8.111, Val loss 8.339
Every effort moves you,,,,,,,,,,,,.
Ep 2 (Step 000010): Train loss 6.661, Val loss 7.048
Ep 2 (Step 000015): Train loss 5.961, Val loss 6.616
Every effort moves you, and, and, and, and, and, and, and, and, and, and, and,
and, and, and, and, and, and, and, and, and, and,, and, and,
[...] (1)
Ep 9 (Step 000080): Train loss 0.541, Val loss 6.393
Every effort moves you?" "Yes--quite insensible to the irony. She wanted him
vindicated--and by me!" He laughed again, and threw back the window-curtains, I
had the donkey. "There were days when I
Ep 10 (Step 000085): Train loss 0.391, Val loss 6.452
Every effort moves you know," was one of the axioms he laid down across the Sevres
and silver of an exquisitely appointed luncheon-table, when, on a later day, I had
again run over from Monte Carlo; and Mrs. Gis
```

(1) Risultati intermedi rimossi per motivi di spazio.

Come potete vedere, il valore loss di addestramento migliora drasticamente, partendo da un valore `9.781` e convergendo verso `0.391`. Le competenze linguistiche del modello migliorano parecchio. All'inizio, il modello è in grado solo di aggiungere virgole al contesto iniziale (`Every effort moves you,,,,,,,,,,,,`) o ripete la parola `and`. Alla fine dell'addestramento, può generare testo grammaticalmente corretto.

In modo analogo a quanto detto per il valore loss del set di addestramento, possiamo vedere che esso parte alto (`9.933`) e diminuisce durante l'addestramento. Tuttavia, non diventa mai piccolo come il valore loss del set di addestramento, e rimane a `6.452` dopo la decima epoca.

Prima di trattare più in dettaglio il valore loss di validazione, creiamo un semplice grafico che mostra affiancati i valori loss del set

di addestramento e di validazione:

```
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
def plot_losses(epochs_seen, tokens_seen, train_losses, val_losses):
    fig, ax1 = plt.subplots(figsize=(5, 3))
    ax1.plot(epochs_seen, train_losses, label="Training loss")
    ax1.plot(epochs_seen, val_losses, linestyle="-.", label="Validation loss")
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel("Loss")
    ax1.legend(loc="upper right")
    ax1.xaxis.set_major_locator(MaxNLocator(integer=True))
    ax2 = ax1.twiny() (1)
    ax2.plot(tokens_seen, train_losses, alpha=0) (2)
    ax2.set_xlabel("Tokens seen")
    fig.tight_layout()
    plt.show()

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
plot_losses(epochs_tensor, tokens_seen, train_losses, val_losses)
```

(1) Crea un secondo asse x che condivide lo stesso asse y.

(2) Grafico invisibile per allineare le tacche.

Il grafico risultante dei valori loss di addestramento e validazione è rappresentato nella Figura 5.12. Come potete vedere, entrambi i valori iniziano a migliorare già alla prima epoca. Tuttavia, dopo la seconda epoca iniziano a divergere. Questa divergenza e il fatto che il valore loss di validazione sia molto più grande di quello di addestramento indicano che il modello va in overfitting rispetto ai dati di addestramento, si sta adattando a essi. Possiamo confermare che il modello memorizza i dati di addestramento alla lettera cercando i frammenti di testo generati, come `quite insensible to the irony` nel testo di *The Verdict*.

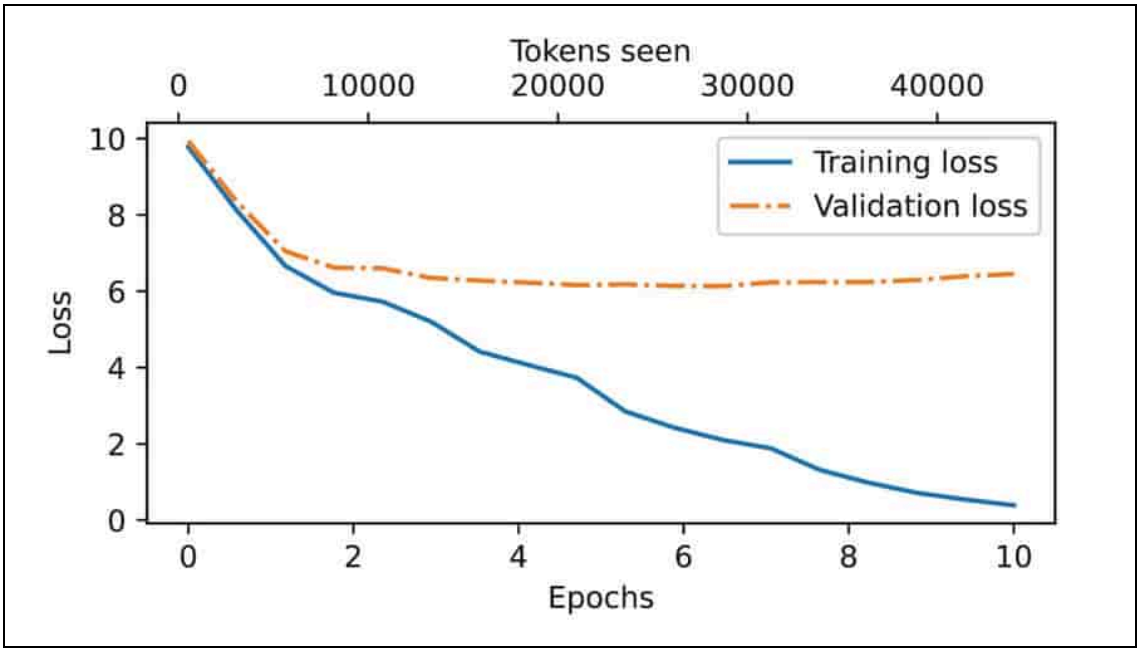


Figura 5.12 All'inizio dell'addestramento, i valori loss del set di addestramento e di validazione diminuiscono drasticamente, il che è un segno che il modello sta imparando. Tuttavia, il valore loss del set di addestramento continua a diminuire oltre la seconda epoca, mentre quello del set di validazione ristagna. Questo è un segno che il modello sta ancora imparando, ma sta andando in overfitting sul set di addestramento oltre l'epoca 2.

Questa memorizzazione è prevista, poiché stiamo lavorando con un set di addestramento molto, molto piccolo e stiamo addestrando il modello per più epoche. Di solito, si addestra un modello su un dataset molto più grande e per una sola epoca.

NOTA

Come accennato in precedenza, i lettori interessati possono provare ad addestrare il modello sui 60.000 libri di pubblico dominio del Progetto Gutenberg, dove questo overfitting non si verifica; vedi l'Appendice B per i dettagli.

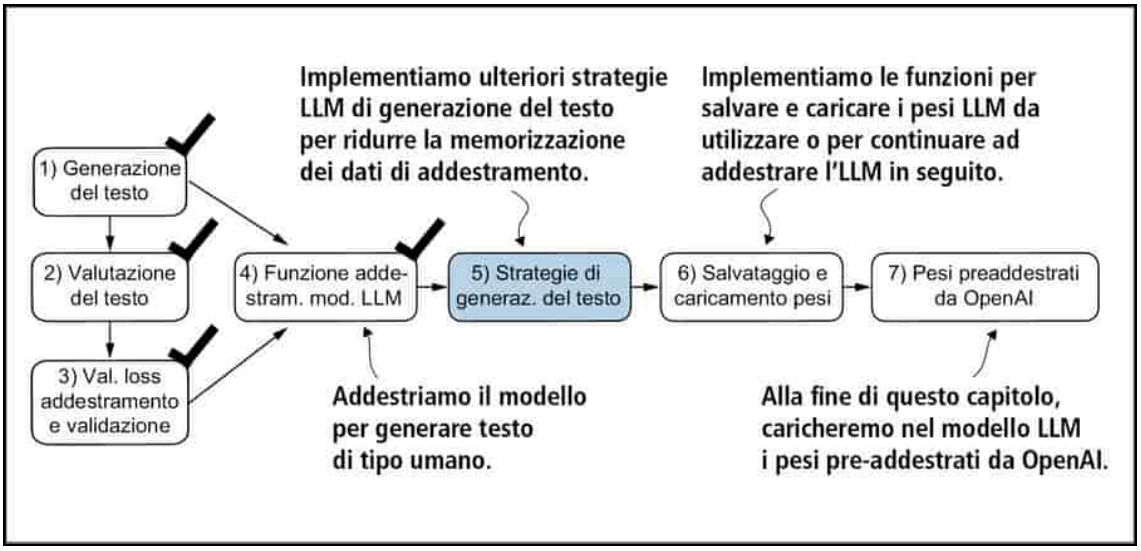


Figura 5.13 Il modello può generare testo coerente dopo aver implementato la funzione di addestramento. Tuttavia, spesso memorizza alla lettera i passi del set di addestramento. Ora, discuteremo le strategie per generare testi di output più diversificati.

Come mostra la Figura 5.13, abbiamo completato quattro dei nostri obiettivi di questo capitolo. Ora, tratteremo le strategie di generazione del testo dei modelli LLM, per ridurre la memorizzazione dei dati di addestramento e aumentare l'originalità del testo generato dal modello LLM prima di trattare il caricamento e il salvataggio dei pesi e il caricamento dei pesi pre-addestrati dal modello GPT di OpenAI.

Strategie di decodifica per controllare la casualità

Esaminiamo ora le strategie di generazione del testo (chiamate anche strategie di decodifica) per generare testo più originale. Innanzitutto, rivedremo brevemente la funzione `generate_text_simple()` che abbiamo usato in precedenza all'interno di

`generate_and_print_sample()`. Poi tratteremo due tecniche, *Temperature Scaling* e *Top-k Sampling*, per migliorare questa funzione.

Iniziamo trasferendo il modello dalla GPU alla CPU, poiché l'inferenza con un modello relativamente piccolo non richiede una GPU. Inoltre, dopo l'addestramento, mettiamo il modello in modalità di valutazione, per disattivare i componenti casuali come il dropout:

```
model.to("cpu")
model.eval()
```

Ora, colleghiamo l'istanza di `GPTModel` (`model`) alla funzione `generate_text_simple()`, che utilizza il modello LLM per generare un token alla volta:

```
tokenizer = tiktoken.get_encoding("gpt2")
token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=25,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

Il testo generato è il seguente:

```
Output text:
Every effort moves you know," was one of the axioms he laid down across the Sevres
and silver of an exquisitely appointed lun
```

Come ho spiegato in precedenza, il token generato a ogni passo è quello corrispondente al punteggio di probabilità più elevato tra tutti i token presenti nel vocabolario. Ciò significa che il modello LLM genererà sempre gli stessi output anche se eseguiamo la funzione `generate_text_simple()` precedente più volte sullo stesso contesto di avvio (`Every effort moves you`).

Temperature Scaling

Diamo ora un'occhiata al *Temperature Scaling*, una tecnica che aggiunge un processo di selezione probabilistica al compito di generazione del token successivo. In precedenza, all'interno della

funzione `generate_text_simple()`, campionavamo sempre il token con la probabilità più elevata come token successivo, utilizzando `torch.argmax()`, con una *decodifica greedy*. Per generare testo con maggiore varietà, possiamo sostituire ad `argmax()` una funzione che campiona da una distribuzione di probabilità (qui, i punteggi di probabilità che il modello LLM genera per ogni elemento del vocabolario a ogni passo di generazione dei token).

Per illustrare il campionamento probabilistico con un esempio concreto, parliamo brevemente del processo di generazione del token successivo utilizzando un vocabolario molto ridotto, a scopo illustrativo:

```
vocab = {
    "closer": 0,
    "every": 1,
    "effort": 2,
    "forward": 3,
    "inches": 4,
    "moves": 5,
    "pizza": 6,
    "toward": 7,
    "you": 8,
}
inverse_vocab = {v: k for k, v in vocab.items()}
```

Quindi, supponiamo che al modello LLM venga assegnato il contesto iniziale "`every effort moves you`" e che esso generi i seguenti logit per il token successivo:

```
next_token_logits = torch.tensor(
    [4.51, 0.89, -1.90, 6.75, 1.63, -1.62, -1.89, 6.28, 1.79]
)
```

Come abbiamo già visto nel Capitolo 4, all'interno di `generate_text_simple()`, convertiamo i logit in probabilità tramite la funzione `softmax()` e otteniamo l'ID corrispondente al token generato tramite la funzione `argmax()`, che possiamo quindi ritrasformare in testo tramite il vocabolario inverso:

```
probas = torch.softmax(next_token_logits, dim=0)
next_token_id = torch.argmax(probas).item()
print(inverse_vocab[next_token_id])
```

Dal momento che il più grande valore logit e, di conseguenza, il più grande punteggio di probabilità softmax è nella quarta posizione (all'indice 3, poiché Python usa l'indicizzazione a 0), la parola generata è "`forward`".

Per implementare un processo di campionamento probabilistico, ora possiamo sostituire ad `argmax()`la funzione `multinomial()` di PyTorch:

```
torch.manual_seed(123)
next_token_id = torch.multinomial(probas, num_samples=1).item()
print(inverse_vocab[next_token_id])
```

L'output prodotto è "`forward`", proprio come prima. Che cosa è successo? La funzione `multinomial()` campiona il token successivo in modo proporzionale al suo punteggio di probabilità. In altre parole, "`forward`" è ancora il token più probabile, e sarà quello selezionato da `multinomial()` la maggior parte delle volte, ma non sempre. Per illustrare questo fatto, implementiamo una funzione che ripete questo campionamento mille volte:

```
def print_sampled_tokens(probas):
    torch.manual_seed(123)
    sample = [torch.multinomial(probas, num_samples=1).item()
        for i in range(1_000)]
    sampled_ids = torch.bincount(torch.tensor(sample))
    for i, freq in enumerate(sampled_ids):
        print(f"{freq} x {inverse_vocab[i]}")

print_sampled_tokens(probas)
```

L'output del campionamento è il seguente:

```
73 x closer
0 x every
0 x effort
582 x forward
2 x inches
0 x moves
0 x pizza
343 x toward
```

Come potete vedere, la parola `forward` è campionata la maggior parte delle volte (582 volte su 1.000), ma qualche volta vengono campionati anche altri token, come `closer`, `inches` e `toward` . Ciò significa

che, se sostituiamo la funzione `argmax()` con la funzione `multinomial()` all'interno della funzione `generate_and_print_sample()`, il modello LLM a volte genererebbe testi come "`every effort moves you toward`", "`every effort moves you inches`" e "`every effort moves you closer`" invece di "`every effort moves you forward`".

Possiamo controllare ulteriormente il processo di distribuzione e selezione tramite un concetto chiamato *Temperature Scaling*, un nome altisonante per chiamare la divisione dei logit per un numero maggiore di 0:

```
def softmax_with_temperature(logits, temperature):
    scaled_logits = logits / temperature
    return torch.softmax(scaled_logits, dim=0)
```

Temperature superiori a 1 fanno sì che le probabilità dei token siano distribuite in modo più uniforme, mentre temperature inferiori a 1 determinano distribuzioni più affidabili (più nette o più precise). Illustriamo questo fatto tracciando le probabilità originali insieme alle probabilità "scalate" con diversi valori di `temperature`:

```
temperatures = [1, 0.1, 5]
scaled_probas = [softmax_with_temperature(next_token_logits, T)
    for T in temperatures]
x = torch.arange(len(vocab))
bar_width = 0.15
fig, ax = plt.subplots(figsize=(5, 3))
for i, T in enumerate(temperatures):
    rects = ax.bar(x + i * bar_width, scaled_probas[i],
                   bar_width, label=f'Temperature = {T}')
ax.set_ylabel('Probability')
ax.set_xticks(x)
ax.set_xticklabels(vocab.keys(), rotation=90)
ax.legend()
plt.tight_layout()
plt.show()
```

Il grafico risultante è rappresentato nella Figura 5.14.

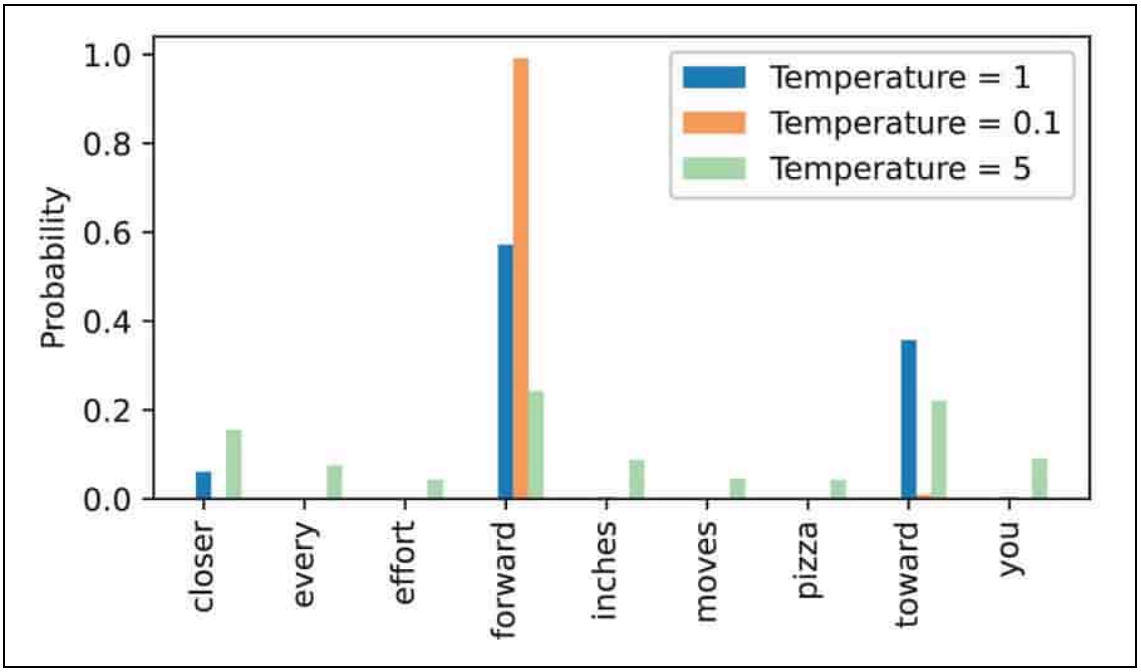


Figura 5.14 Una temperatura pari a 1 rappresenta i punteggi di probabilità non scalati per ogni token presente nel vocabolario. Ridurre la temperatura a 0,1 affina la distribuzione, quindi il token più probabile (qui, "forward") avrà un punteggio di probabilità ancora più elevato. Allo stesso modo, aumentare la temperatura a 5 rende la distribuzione più uniforme.

Una temperatura pari a `1` divide i logit per `1` prima di passarli alla funzione `softmax()` per calcolare i punteggi di probabilità. In altre parole, usare una temperatura di 1 equivale a non usare alcuna scala di temperatura. In questo caso, i token vengono selezionati con una probabilità uguale ai punteggi di probabilità originali `softmax()` tramite la funzione di campionamento `multinomial()` di PyTorch. Per esempio, per l'impostazione della temperatura `1`, il token corrispondente a "`forward`" verrebbe selezionato circa il 60 percento delle volte, come potete vedere nella Figura 5.14.

Inoltre, come potete vedere sempre nella Figura 5.14, l'applicazione di temperature molto piccole, come `0.1`, darà luogo a

distribuzioni più nette, tali che il comportamento della funzione `multinomial()` seleziona il token più probabile (qui, "`forward`") quasi il 100 percento delle volte, avvicinandosi al comportamento della funzione `argmax()`. Allo stesso modo, una temperatura di `5` dà luogo a una distribuzione più uniforme, in cui altri token vengono selezionati più spesso. Ciò può aggiungere più varietà ai testi generati, ma produce più spesso un testo senza senso. Per esempio, utilizzando la temperatura `5` si ottengono testi come `every effort moves you pizza` circa il 4 percento delle volte.

Esercizio 5.1
Utilizzate la funzione `print_sampled_tokens()` per mostrare le frequenze di campionamento delle probabilità softmax scalate con le temperature mostrate nella Figura 5.14. Con quale frequenza viene campionata la parola `pizza` in ciascun caso? Riuscite a pensare a un modo più rapido e accurato per determinare la frequenza con cui viene campionata la parola `pizza`?

Top-k Sampling

Abbiamo implementato un approccio a campionamento probabilistico abbinato al *Temperature Scaling* per aumentare la diversità degli output. Abbiamo visto che valori di temperatura più elevati fanno sì che le probabilità dei token successivi siano distribuite in modo più uniforme, il che determina output più diversificati, poiché riducono la probabilità che il modello selezioni ripetutamente il token più probabile. Questo metodo consente di esplorare percorsi meno probabili ma potenzialmente più interessanti e creativi nel processo di generazione. Tuttavia, uno svantaggio di questo approccio è che a volte porta a output grammaticalmente errati o completamente privi di senso come `every effort moves you pizza`.

Il *Top-k Sampling*, se combinato col campionamento probabilistico e il Temperature Scaling, può migliorare i risultati. Nel Top-k

Sampling, possiamo limitare i token campionati ai k più probabili, ed escludere tutti gli altri token dal processo di selezione mascherando i loro punteggi di probabilità, come mostra la Figura 5.15.

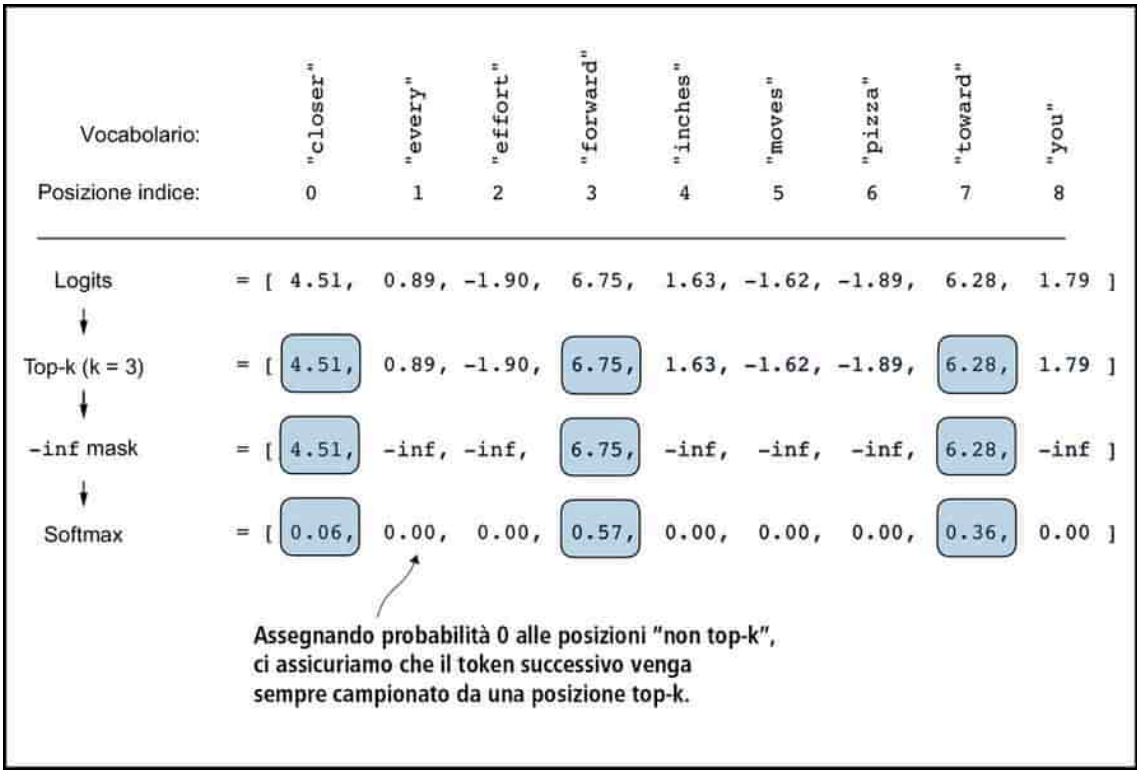


Figura 5.15 Utilizzando il Top-k Sampling con k=3, ci concentriamo sui tre token associati ai logit più elevati e mascheriamo tutti gli altri con un infinito negativo (-inf) prima di applicare la funzione softmax(). Ciò si traduce nella distribuzione di probabilità con valore 0 a tutti i token non top-k. I numeri in questa figura sono troncati a due cifre dopo la virgola decimale per ridurre l'ingombro visivo. I valori nella riga "Softmax" dovrebbero avere somma 1.0.

L'approccio top-k sostituisce tutti i logit non selezionati con un valore infinito negativo (`-inf`), in modo tale che, quando si calcolano i valori softmax, i punteggi di probabilità dei token non top-k siano 0, e le probabilità rimanenti siano a somma 1. I lettori attenti potrebbero ricordare questo trucco di mascheramento dal modulo di attenzione causale che abbiamo implementato nel Capitolo 3.

Nel codice, possiamo implementare la procedura top-k rappresentata nella Figura 5.15 come segue, iniziando con la selezione dei token con i valori logit più grandi:

```
top_k = 3
top_logits, top_pos = torch.topk(next_token_logits, top_k)
print("Top logits:", top_logits)
print("Top positions:", top_pos)
```

I valori dei logit e gli ID dei primi tre token, in ordine decrescente, sono i seguenti:

```
Top logits: tensor([6.7500, 6.2800, 4.5100])
Top positions: tensor([3, 7, 0])
```

Ora, applichiamo la funzione PyTorch `where()` per impostare su meno infinito (`-inf`) i token che sono al di sotto del valore logit più basso all'interno della nostra selezione dei primi tre:

```
new_logits = torch.where(
    condition=next_token_logits < top_logits[-1], (1)
    input=torch.tensor(float('-inf')), (2)
    other=next_token_logits (3)
)
print(new_logits)
```

(1) Identifica i logit inferiori ai top-3.

(2) Assegna `-inf` a questi logit.

(3) Conserva i logit originali per tutti gli altri token.

I logit risultanti per il token successivo nel vocabolario dei nove token sono i seguenti:

```
tensor([4.5100, -inf, -inf, 6.7500, -inf, -inf, -inf, 6.2800, -inf])
```

Infine, applichiamo la funzione `softmax()` per trasformarli in probabilità del token successivo:

```
topk_probas = torch.softmax(new_logits, dim=0)
print(topk_probas)
```

Come potete vedere, il risultato di questo approccio top-3 sono tre soli punteggi di probabilità diversi da zero:

```
tensor([0.0615, 0.0000, 0.0000, 0.5775, 0.0000, 0.0000, 0.0000, 0.3610, 0.0000])
```

Ora possiamo applicare il Temperature Scaling e la funzione `multinomial()` di campionamento probabilistico per selezionare il token

successivo tra questi tre punteggi di probabilità diversi da zero. Lo facciamo modificando la funzione di generazione del testo.

Modifica della funzione di generazione del testo

Ora, combiniamo il Temperature Scaling e il Top-k Sampling per modificare la funzione `generate_text_simple()` che abbiamo utilizzato in precedenza, creando così una nuova funzione `generate()`.

Listato 5.4 Una funzione di generazione del testo modificata per avere maggiore diversità.

```
def generate(model, idx, max_new_tokens, context_size, temperature=0.0,
             top_k=None, eos_id=None):
    for _ in range(max_new_tokens): (1)
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)
        logits = logits[:, -1, :]

        if top_k is not None: (2)
            top_logits, _ = torch.topk(logits, top_k)
            min_val = top_logits[:, -1]
            logits = torch.where(
                logits < min_val,
                torch.tensor(float('-inf')).to(logits.device),
                logits
            )

        if temperature > 0.0: (3)
            logits = logits / temperature
            probs = torch.softmax(logits, dim=-1)
            idx_next = torch.multinomial(probs, num_samples=1)
        else: (4)
            idx_next = torch.argmax(logits, dim=-1, keepdim=True)

        if idx_next == eos_id: (5)
            break
        idx = torch.cat((idx, idx_next), dim=1)
    return idx
```

(1) Il ciclo `for` è lo stesso di prima: ottiene i logit e si concentra solo sull'ultimo time step.

(2) Filtra i logit con Top_k Sampling.

(3) Applica il Temperature Scaling.

(4) Esegue la selezione greedy del token successivo come prima, quando il Temperature Scaling è disabilitato.
(5) Interrompe la generazione anticipata se viene rilevato un token di fine sequenza.

Vediamo ora in azione questa nuova funzione `generate()`:

```
torch.manual_seed(123)
token_ids = generate(
    model=model,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=15,
    context_size=GPT_CONFIG_124M["context_length"],
    top_k=25,
    temperature=1.4
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

Il testo generato è il seguente:

```
Output text:
Every effort moves you stand to work on surprise, a one of us had gone with
random-
```

Come potete vedere, il testo generato è molto diverso da quello che abbiamo generato in precedenza tramite la funzione `generate_simple()` (uno dei risultati era stato "`Every effort moves you know,`"), che era un passo memorizzato dal set di addestramento.

Esercizio 5.2
Provate diverse temperature e impostazioni top-k. Sulla base delle vostre osservazioni, riuscite a immaginare applicazioni in cui sono desiderate temperature e impostazioni top-k più basse? Allo stesso modo, riuscite a pensare ad applicazioni in cui sono preferite temperature e impostazioni top-k più alte? Provate a ripetere questo esercizio alla fine del capitolo dopo aver caricato i pesi pre-addestrati da OpenAI.

Esercizio 5.3
Quali sono le diverse combinazioni di impostazioni per la funzione `generate()` per forzare un comportamento deterministico, ovvero disabilitare il campionamento casuale in modo che produca sempre gli stessi output simili alla funzione `generate_simple()`?

Caricamento e salvataggio dei pesi del modello in PyTorch

Finora abbiamo provato a valutare numericamente i progressi dell'addestramento e a pre-addestrare un modello LLM da zero. Sebbene sia il modello LLM sia il dataset fossero relativamente piccoli, questo esercizio ha mostrato che il pre-addestrare i modelli LLM è computazionalmente costoso. Pertanto, è importante essere in grado di salvare il modello LLM in modo da non dover rieseguire l'addestramento ogni volta che vogliamo utilizzarlo in una nuova sessione.

Pertanto, vediamo ora come salvare e caricare un modello pre-addestrato, come evidenziato nella Figura 5.16. In seguito, caricheremo da OpenAI nella nostra istanza `GPTModel` un modello GPT pre-addestrato più capace.

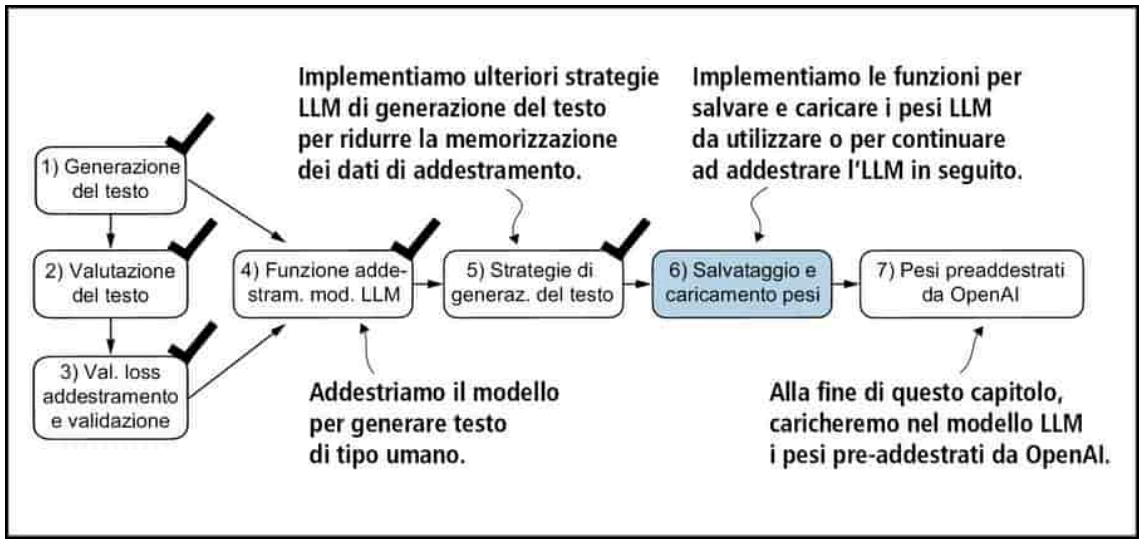


Figura 5.16 Dopo aver addestrato e ispezionato il modello, spesso è utile salvarlo, in modo da poterlo utilizzare o continuare ad addestrarlo in seguito (Passo 6).

Fortunatamente, salvare un modello con PyTorch è relativamente semplice. Il metodo consigliato consiste nel salvare lo `state_dict` di un

modello, un dizionario che mappa ogni layer ai suoi parametri, usando la funzione `torch.save()`:

```
torch.save(model.state_dict(), "model.pth")
```

`"model.pth"` è il nome del file sul quale viene salvato `state_dict`. L'estensione `.pth` è convenzionale per i file di PyTorch, ma tecnicamente potremmo usare qualsiasi estensione di file.

Quindi, dopo aver salvato i pesi del modello tramite `state_dict`, possiamo caricare i pesi del modello in una nuova istanza del modello `GPTModel`:

```
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(torch.load("model.pth", map_location=device))
model.eval()
```

Come abbiamo già visto nel Capitolo 4, il dropout aiuta a evitare che il modello vada in overfitting sui dati di addestramento "eliminando" casualmente alcuni neuroni di un layer durante l'addestramento. Tuttavia, durante l'inferenza, non vorremmo eliminare casualmente una qualsiasi delle informazioni apprese dalla rete. L'uso di `model.eval()` commuta il modello in modalità di valutazione (per l'inferenza), disabilitando i layer di dropout del `model`. Se prevedete di continuare a pre-addestrare un modello in seguito, per esempio utilizzando la funzione `train_model_simple()` definita in precedenza in questo capitolo, vi consiglio di salvare anche lo stato dell'ottimizzatore.

Gli ottimizzatori adattivi, come AdamW, memorizzano dei parametri aggiuntivi per ogni peso del modello. AdamW utilizza dei dati storici per regolare dinamicamente i tassi di apprendimento per ogni parametro del modello. Senza di essi, l'ottimizzatore si reimposterebbe, e il modello potrebbe apprendere in modo subottimale o addirittura non riuscire a convergere correttamente; in altre parole perderà la capacità di generare testo coerente. Utilizzando

`torch.save()`, possiamo salvare sia il modello sia i contenuti dell'ottimizzatore:

```
torch.save({"model_state_dict": model.state_dict(),
            "optimizer_state_dict": optimizer.state_dict(),},
           "model_and_optimizer.pth"
)
```

Possiamo così ripristinare gli stati del modello e dell'ottimizzatore, caricando tramite `torch.load()` i dati salvati e poi utilizzando il metodo `load_state_dict()`:

```
checkpoint = torch.load("model_and_optimizer.pth", map_location=device)
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-4, weight_decay=0.1)
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
model.train();
```

Esercizio 5.4
Dopo aver salvato i pesi, caricate il modello e l'ottimizzatore in una nuova sessione Python o in un Notebook Jupyter e continuate il pre-addestramento per un'altra epoca utilizzando la funzione `train_model_simple()`.

Caricamento di pesi pre-addestrati da OpenAI

In precedenza, abbiamo addestrato un piccolo modello GPT-2 utilizzando un dataset limitato comprendente un libro di racconti brevi. Questo approccio ci ha consentito di concentrarci sui fondamenti senza dover impiegare molto tempo e risorse computazionali.

Fortunatamente, OpenAI ha condiviso i pesi dei propri modelli GPT-2, eliminando quindi la necessità di investire decine o centinaia di migliaia di dollari per riaddestrare il modello su un corpus di grandi dimensioni. Quindi, carichiamo questi pesi nella nostra classe `GPTModel` e utilizziamo il modello per la generazione del testo. Qui, con *pesi* si fa riferimento ai parametri dei pesi immagazzinati negli

attributi `.weight` dei layer `Linear` ed `Embedding` di PyTorch, per esempio. Vi abbiamo fatto accesso in precedenza tramite `model.parameters()` durante l'addestramento del modello. Nel Capitolo 6, riutilizzeremo questi pesi pre-addestrati per eseguire il fine-tuning del modello per un'attività di classificazione del testo e seguiremo istruzioni simili a ChatGPT.

Notate che OpenAI, originariamente, salvava i pesi di GPT-2 tramite TensorFlow, che dobbiamo installare per caricare i pesi in Python. Il seguente codice utilizzerà uno strumento a barra di avanzamento, chiamato `tqdm`, per seguire il processo di download, e dobbiamo installare anche quello.

Potete installare queste librerie con il seguente comando, dal terminale:

```
pip install tensorflow>=2.15.0 tqdm>=4.66
```

Il codice è relativamente lungo, per lo più banale, non molto interessante. Quindi, invece di dedicare spazio prezioso alla discussione del codice Python per il recupero di file da Internet, scarichiamo il modulo Python `gpt_download.py` direttamente dal repository online di questo capitolo:

```
import urllib.request
url = (
    "https://raw.githubusercontent.com/rasbt/"
    "LLMs-from-scratch/main/ch05/"
    "01_main-chapter-code/gpt_download.py"
)
filename = url.split('/')[-1]
urllib.request.urlretrieve(url, filename)
```

Dopo aver scaricato questo file nella directory locale della sessione Python, esaminatene brevemente il contenuto, per assicurarvi che sia stato salvato correttamente e che contenga codice Python valido.

Ora possiamo importare dal file `gpt_download.py` la funzione `download_and_load_gpt2()`, che caricherà nella nostra sessione Python le

impostazioni dell'architettura GPT-2 (`settings`) e i parametri dei pesi (`params`):

```
from gpt_download import download_and_load_gpt2
settings, params = download_and_load_gpt2(
    model_size="124M", models_dir="gpt2"
)
```

L'esecuzione di questo codice scarica i seguenti sette file associati al modello GPT-2 con parametro `124M`:

```
checkpoint: 100%|■■■■■■■>■■■■■■■■■■■■■■■■■■■■■ | 77.0/77.0 [00:00<00:00, 63.9kiB/s]
encoder.json: 100%|■■■■■■■■■■■■■■■■■■■■■■■■■ | 1.04M/1.04M [00:00<00:00,
2.20MiB/s]
hprams.json: 100%|■■■■■■■■■■■■■■■■■■■■■■■■■ | 90.0/90.0 [00:00<00:00, 78.3kiB/s]
model.ckpt.data-00000-of-00001: 100%|■■■■■■■■ | 498M/498M [01:09<00:00, 7.16MiB/s]
model.ckpt.index: 100%|■■■■■■■■■■>■■■■■■■■■■■ | 5.21k/5.21k [00:00<00:00,
3.24MiB/s]
model.ckpt.meta: 100%|■■■■■■■■>■■■■■■■■■■■■■■ | 471k/471k [00:00<00:00, 2.46MiB/s]
vocab.bpe: 100%|■■■■■>■■■■■■■■■■■■■■■■■■■■■■ | 456k/456k [00:00<00:00, 1.70MiB/s]
```

NOTA

Se il codice di download non vi funziona, ciò potrebbe essere dovuto a una connessione Internet scadente, a problemi al server o a cambiamenti nel modo in cui OpenAI condivide i pesi del modello open source GPT-2. In questo caso, visitate il repository del codice di questo capitolo per ottenere istruzioni alternative aggiornate, su https://github.com/rasbt/LLMs-from-scratch.

Supponendo che l'esecuzione del codice precedente sia stata completata, esaminiamo il contenuto di `settings` e `params`:

```
print("Settings:", settings)
print("Parameter dictionary keys:", params.keys())
```

Il contenuto è il seguente:

```
Settings: {'n_vocab': 50257, 'n_ctx': 1024, 'n_embd': 768, 'n_head': 12,
'n_layer': 12}
Parameter dictionary keys: dict_keys(['blocks', 'b', 'g', 'wpe', 'wte'])
```

Sia `settings` sia `params` sono dizionari Python. Il dizionario `settings` memorizza le impostazioni dell'architettura del modello LLM in modo simile alle nostre impostazioni `GPT_CONFIG_124M` definite manualmente. Il dizionario `params` contiene i tensori dei pesi. Notate che abbiamo mostrato solo le chiavi del dizionario, perché stampare i pesi occuperebbe troppo spazio; tuttavia, potete ispezionare questi

tensori dei pesi con `print(params)` o selezionare i singoli tensori tramite le rispettive chiavi del dizionario, per esempio, i pesi del layer di embedding:

```
print(params["wte"])
print("Token embedding weight tensor dimensions:", params["wte"].shape)
```

I pesi del layer di embedding dei token sono i seguenti:

```
[[-0.11010301 ... -0.1363697 0.01506208 0.04531523]
 [ 0.04034033 ... 0.08605453 0.00253983 0.04318958]
 [-0.12746179 ... 0.08991534 -0.12972379 -0.08785918]
 ...
 [-0.04453601 ... 0.10435229 0.09783269 -0.06952604]
 [ 0.1860082 ... -0.09625227 0.07847701 -0.02245961]
 [ 0.05135201 ... 0.00704835 0.15519823 0.12067825]]
Token embedding weight tensor dimensions: (50257, 768)
```

Abbiamo scaricato e caricato i pesi del modello GPT-2 più piccolo tramite `download_and_load_gpt2(model_size="124M", ...)`. OpenAI condivide anche i pesi di modelli più grandi: `355M`, `774M` e `1558M`. L'architettura complessiva di questi altri modelli GPT è la stessa, come mostra la Figura 5.17, tranne per il fatto che alcuni elementi dell'architettura vengono ripetuti un numero differente di volte e le dimensioni dell'embedding sono differenti. Il codice rimanente in questo capitolo è compatibile anche con questi modelli più grandi.

Dopo aver caricato i pesi del modello GPT-2 in Python, dobbiamo ancora trasferirli dai dizionari `settings` e `params` alla nostra istanza `GPTModel`. Innanzitutto, creiamo un dizionario che elenca le differenze tra le diverse dimensioni del modello GPT nella Figura 5.17:

```
model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12},
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16},
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20},
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}
```

Supponiamo di essere interessati a caricare il modello più piccolo, `"gpt2-small (124M)"`. Possiamo utilizzare le impostazioni corrispondenti dalla tabella `model_configs` per aggiornare il nostro `GPT_CONFIG_124M` completo che abbiamo definito e utilizzato in precedenza:

```
model_name = "gpt2-small (124M)"
NEW_CONFIG = GPT_CONFIG_124M.copy()
NEW_CONFIG.update(model_configs[model_name])
```

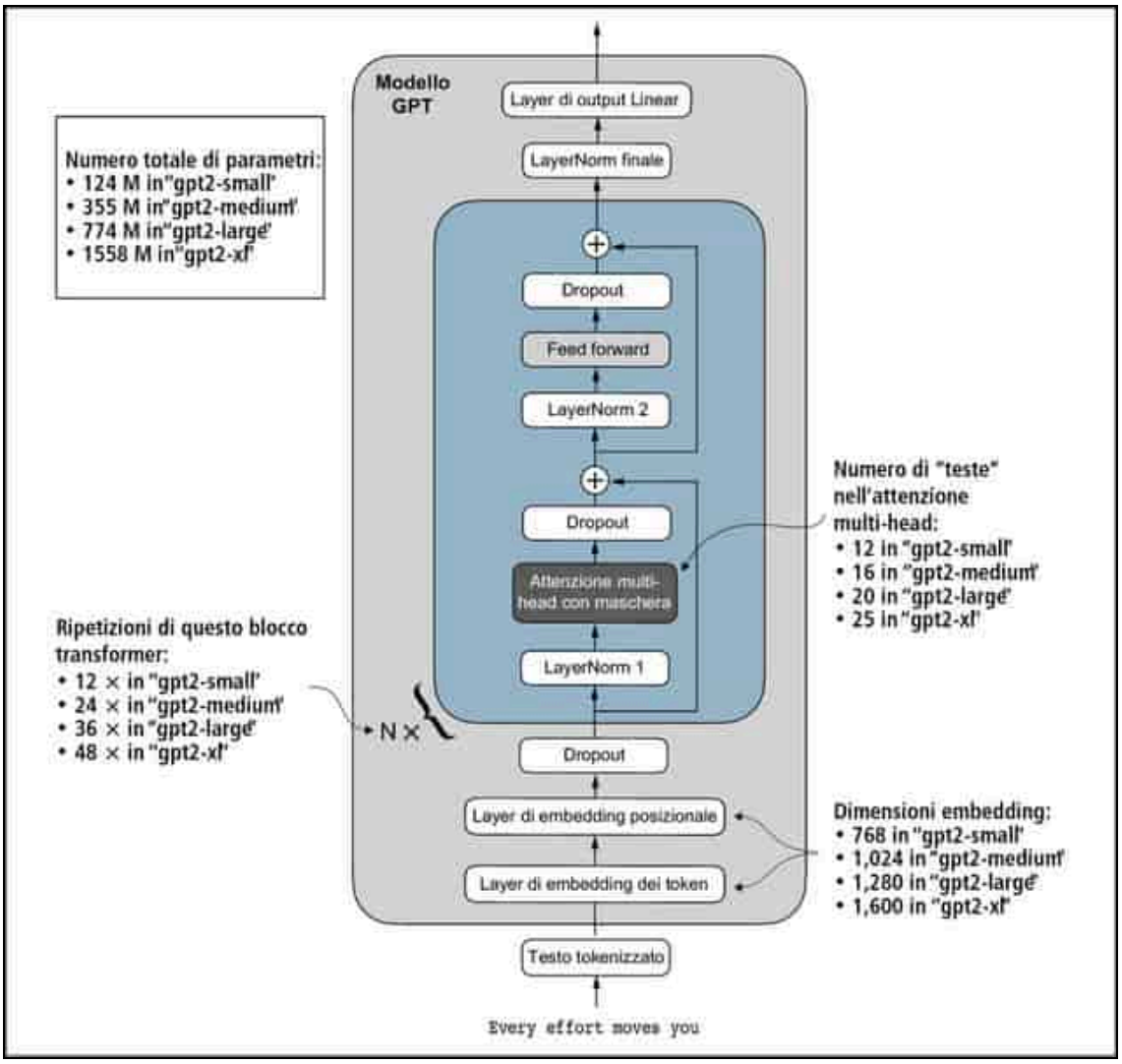


Figura 5.17 I modelli LLM GPT-2 sono disponibili in diverse dimensioni, da 124 milioni a 1.558 milioni di parametri. L'architettura di base è la stessa, con l'unica differenza nelle dimensioni dell'embedding e nel numero di volte in cui vengono ripetuti singoli componenti come le "teste di attenzione" e i blocchi transformer.

I lettori attenti ricorderanno che in precedenza abbiamo utilizzato una lunghezza di 256 token, ma i modelli GPT-2 originali di OpenAI

sono stati addestrati con una lunghezza di 1024 token, quindi dobbiamo aggiornare di conseguenza `NEW_CONFIG`:

```
NEW_CONFIG.update({"context_length": 1024})
```

Inoltre, OpenAI ha utilizzato vettori di bias nei layer Linear del modulo di attenzione multi-head per implementare i calcoli di query, chiavi e matrici dei valori. I vettori di bias non sono più comunemente utilizzati nei modelli LLM, poiché non migliorano le prestazioni di modellazione e sono quindi inutili. Tuttavia, poiché stiamo lavorando con pesi pre-addestrati, dobbiamo abbinare le impostazioni per la coerenza e abilitare questi vettori di bias:

```
NEW_CONFIG.update({"qkv_bias": True})
```

Ora possiamo utilizzare il dizionario `NEW_CONFIG` aggiornato per inizializzare una nuova istanza di `GPTModel`:

```
gpt = GPTModel(NEW_CONFIG)
gpt.eval()
```

Per default, l'istanza `GPTModel` viene inizializzata con pesi casuali per il pre-addestramento. L'ultimo passo per utilizzare i pesi del modello di OpenAI è quello di sovrascrivere questi pesi casuali con i pesi che abbiamo caricato nel dizionario `params`. Per questo, definiremo prima una piccola funzione di servizio `assign()` che controlla se due tensori o array (`left` e `right`) hanno le stesse dimensioni o forma e restituisce il tensore `right` come parametri addestrabili:

```
def assign(left, right):
    if left.shape != right.shape:
        raise ValueError(f"Shape mismatch. Left: {left.shape}, "
                         "Right: {right.shape}"
        )
    return torch.nn.Parameter(torch.tensor(right))
```

Ora, definiamo una funzione `load_weights_into_gpt()` che carica i pesi dal dizionario `params` in un'istanza `gpt` di `GPTModel`.

Listato 5.5 Caricamento dei pesi di OpenAI nel codice del modello GPT.

```
import numpy as np

def load_weights_into_gpt(gpt, params): (1)
    gpt.pos_emb.weight = assign(gpt.pos_emb.weight, params['wpe'])
    gpt.tok_emb.weight = assign(gpt.tok_emb.weight, params['wte'])

    for b in range(len(params["blocks"])): (2)
        q_w, k_w, v_w = np.split( (3)
            (params["blocks"][b]["attn"]["c_attn"])["w"], 3, axis=-1)
        gpt.trf_blocks[b].att.W_query.weight = assign(
            gpt.trf_blocks[b].att.W_query.weight, q_w.T)
        gpt.trf_blocks[b].att.W_key.weight = assign(
            gpt.trf_blocks[b].att.W_key.weight, k_w.T)
        gpt.trf_blocks[b].att.W_value.weight = assign(
            gpt.trf_blocks[b].att.W_value.weight, v_w.T)

        q_b, k_b, v_b = np.split(
            (params["blocks"][b]["attn"]["c_attn"])["b"], 3, axis=-1)
        gpt.trf_blocks[b].att.W_query.bias = assign(
            gpt.trf_blocks[b].att.W_query.bias, q_b)
        gpt.trf_blocks[b].att.W_key.bias = assign(
            gpt.trf_blocks[b].att.W_key.bias, k_b)
        gpt.trf_blocks[b].att.W_value.bias = assign(
            gpt.trf_blocks[b].att.W_value.bias, v_b)

        gpt.trf_blocks[b].att.out_proj.weight = assign(
            gpt.trf_blocks[b].att.out_proj.weight,
            params["blocks"][b]["attn"]["c_proj"]["w"].T)
        gpt.trf_blocks[b].att.out_proj.bias = assign(
            gpt.trf_blocks[b].att.out_proj.bias,
            params["blocks"][b]["attn"]["c_proj"]["b"])

        gpt.trf_blocks[b].ff.layers[0].weight = assign(
            gpt.trf_blocks[b].ff.layers[0].weight,
            params["blocks"][b]["mlp"]["c_fc"]["w"].T)
        gpt.trf_blocks[b].ff.layers[0].bias = assign(
            gpt.trf_blocks[b].ff.layers[0].bias,
            params["blocks"][b]["mlp"]["c_fc"]["b"])
        gpt.trf_blocks[b].ff.layers[2].weight = assign(
            gpt.trf_blocks[b].ff.layers[2].weight,
            params["blocks"][b]["mlp"]["c_proj"]["w"].T)
        gpt.trf_blocks[b].ff.layers[2].bias = assign(
            gpt.trf_blocks[b].ff.layers[2].bias,
            params["blocks"][b]["mlp"]["c_proj"]["b"])

        gpt.trf_blocks[b].norm1.scale = assign(
            gpt.trf_blocks[b].norm1.scale,
            params["blocks"][b]["ln_1"]["g"])
        gpt.trf_blocks[b].norm1.shift = assign(
            gpt.trf_blocks[b].norm1.shift,
            params["blocks"][b]["ln_1"]["b"])
        gpt.trf_blocks[b].norm2.scale = assign(
            gpt.trf_blocks[b].norm2.scale,
            params["blocks"][b]["ln_2"]["g"])
        gpt.trf_blocks[b].norm2.shift = assign(
            gpt.trf_blocks[b].norm2.shift,
```

```
            params["blocks"][b]["ln_2"]["b"])

    gpt.final_norm.scale = assign(gpt.final_norm.scale, params["g"])
    gpt.final_norm.shift = assign(gpt.final_norm.shift, params["b"])
    gpt.out_head.weight = assign(gpt.out_head.weight, params["wte"]) (4)
```

(1) Imposta i pesi di embedding posizionale e dei token del modello su quelli specificati in `params`.

(2) Esegue l'iterazione su ogni blocco transformer del modello.

(3) La funzione `np.split()` viene utilizzata per dividere i pesi di attenzione e bias in tre parti uguali per i componenti query, chiave e valore.

(4) Il modello GPT-2 originale di OpenAI ha riutilizzato i pesi di embedding dei token nel layer di output per ridurre il numero totale di parametri, un concetto noto come *weight tying*.

Nella funzione `load_weights_into_gpt()`, abbiniamo con cura i pesi dell'implementazione di OpenAI con la nostra implementazione `GPTModel`. Per scegliere uno specifico esempio, OpenAI ha memorizzato il tensore dei pesi per il layer di proiezione di output del primo blocco transformer come `params["blocks"][0]["attn"]["c_proj"]["w"]`. Nella nostra implementazione, questo tensore dei pesi corrisponde a `gpt.trf_blocks[b].att.out_proj.weight`, dove `gpt` è un'istanza di `GPTModel`.

Lo sviluppo della funzione `load_weights_into_gpt()` ha richiesto molte congetture, poiché OpenAI ha utilizzato un convenzione di denominazione leggermente diversa dalla nostra. Tuttavia, la funzione `assign()` ci avviserebbe se provassimo a far corrispondere due tensori di dimensioni differenti. Inoltre, se commettessimo un errore in questa funzione, lo noteremmo, poiché il modello GPT risultante non sarebbe in grado di produrre testo coerente.

Proviamo ora a mettere all'opera `load_weights_into_gpt()` e carichiamo i pesi del modello OpenAI nella nostra istanza di `GPTModel`, `gpt`:

```
load_weights_into_gpt(gpt, params)
gpt.to(device)
```

Se il modello è caricato correttamente, ora possiamo utilizzarlo per generare nuovo testo utilizzando la nostra precedente funzione `generate()`:

```
torch.manual_seed(123)
token_ids = generate(
    model=gpt,
    idx=text_to_token_ids("Every effort moves you", tokenizer).to(device),
    max_new_tokens=25,
    context_size=NEW_CONFIG["context_length"],
    top_k=50,
    temperature=1.5
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

Il testo risultante è il seguente:

```
Output text:
Every effort moves you toward finding an ideal new way to practice something!
What makes us want to be on top of that?
```

Possiamo essere certi di aver caricato correttamente i pesi del modello perché il modello riesce a produrre testo coerente. Un piccolo errore in questo processo causerebbe il fallimento del modello. Nei prossimi capitoli, elaboreremo ulteriormente questo modello pre-addestrato e lo perfezioneremo per classificare il testo e seguire istruzioni.

Esercizio 5.5
Calcolate i valori loss del set di addestramento e validazione del `GPTModel` con i pesi pre-addestrati da OpenAI sul dataset *The Verdict*.

Esercizio 5.6
Sperimentate con modelli GPT-2 di diverse dimensioni, per esempio il modello più grande da 1.558 milioni di parametri, e confrontate il testo generato col modello da 124 milioni.

Riepilogo

- Quando i modelli LLM generano testo, restituiscono un token alla volta.

- Per default, il token successivo viene generato convertendo gli output del modello in punteggi di probabilità e selezionando dal vocabolario il token che corrisponde al punteggio di probabilità più elevato, noto come “greedy decoding”.
- Utilizzando il campionamento probabilistico e il Temperature Scaling, possiamo influenzare la diversità e la coerenza del testo generato.
- I valori loss dei set di addestramento e di validazione possono essere utilizzati per valutare la qualità del testo generato dal modello LLM durante l’addestramento.
- Il pre-addestramento di un modello LLM prevede la modifica dei pesi per minimizzare il valore loss di addestramento.
- Il ciclo di addestramento dei modelli LLM è una procedura standard nel deep learning, che utilizza un valore loss a entropia incrociata convenzionale e un ottimizzatore AdamW.
- Il pre-addestramento di un modello LLM su un ampio corpus di testo richiede molto tempo e risorse; pertanto, come alternativa al pre-addestramento del modello su un ampio dataset, possiamo caricare pesi di pubblico dominio.

Capitolo 6

Fine-tuning per attività di classificazione

Finora abbiamo programmato l'architettura LLM, l'abbiamo pre-addestrata e abbiamo imparato a importare nel modello dei pesi pre-addestrati da una fonte esterna, OpenAI. Ora raccoglieremo i frutti del nostro lavoro perfezionando il modello LLM su un'attività specifica: la classificazione del testo. L'esempio concreto che esaminiamo è la classificazione degli sms come "spam" o "non spam". La Figura 6.1 evidenzia i due modi per eseguire il fine-tuning di un modello LLM: perfezionamento per la classificazione (Passo 8) e perfezionamento per rispondere a istruzioni (Passo 9).

Diverse categorie di fine-tuning

I modi più comuni per mettere a punto i modelli linguistici sono il *fine-tuning per rispondere a istruzioni* e il *fine-tuning per attività di classificazione*. Il fine-tuning per rispondere a istruzioni comporta l'addestramento di un modello linguistico su una serie di attività utilizzando istruzioni specifiche, per migliorare la sua capacità di comprendere ed eseguire compiti descritti tramite prompt in linguaggio naturale, come mostra la Figura 6.2.

Nel fine-tuning per attività di classificazione, un concetto che potreste già conoscere se avete un background nel machine learning,

il modello viene addestrato a riconoscere un set specifico di etichette delle classi, come “spam” e “non spam”.

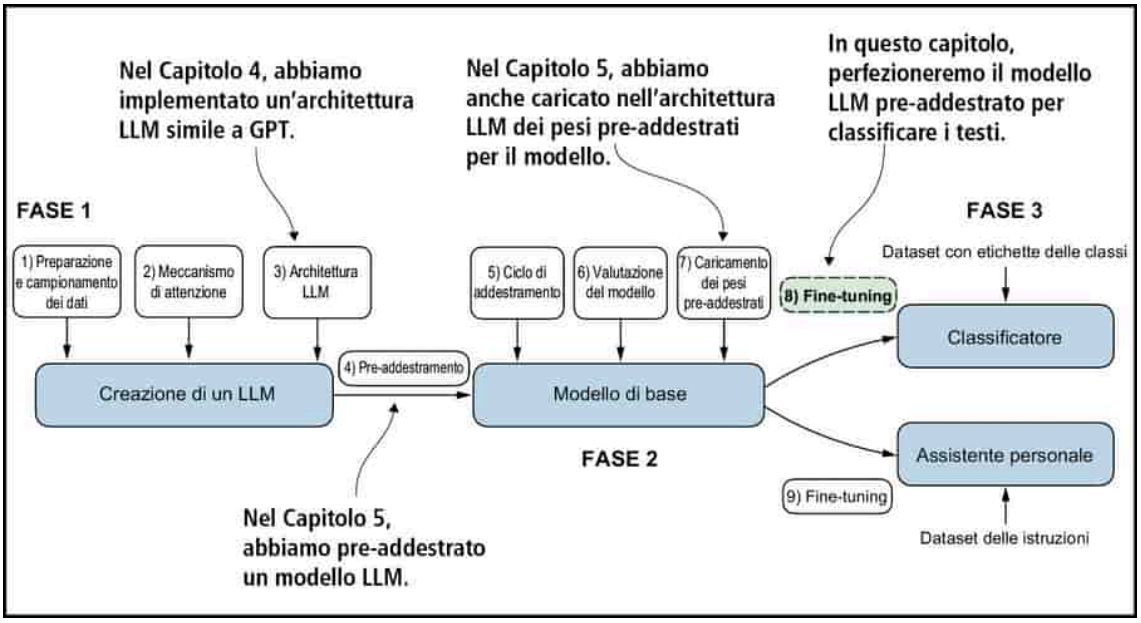


Figura 6.1 Le tre fasi principali della programmazione di un modello LLM. Questo capitolo si concentra sulla Fase 3 (Passo 8): fine-tuning di un modello LLM pre-addestrato come classificatore.

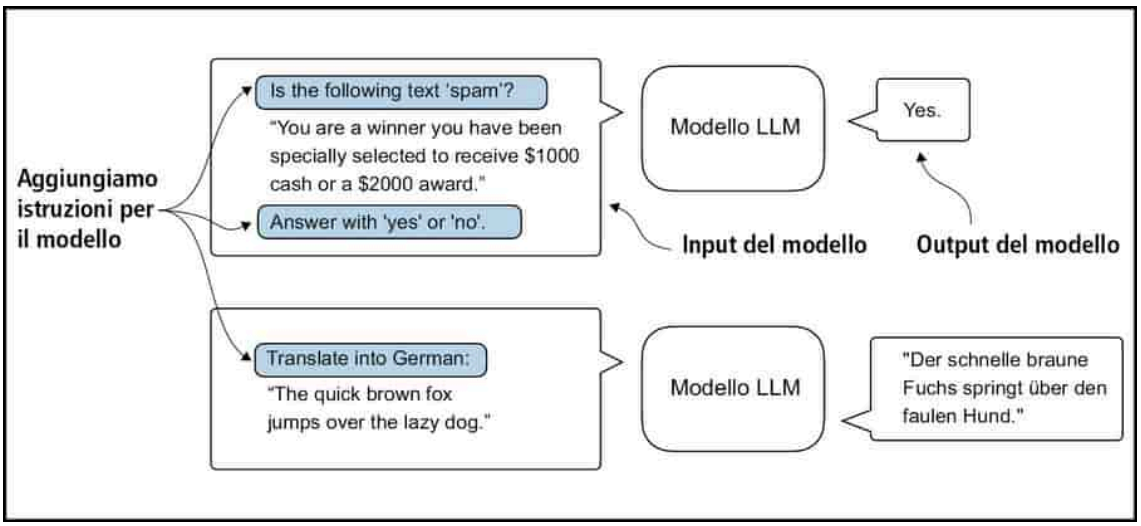


Figura 6.2 Due diversi scenari di fine-tuning per rispondere a istruzioni. In alto, il modello ha il compito di determinare se un dato testo è spam. In basso, al modello viene data

un’istruzione su come tradurre una frase dall’inglese al tedesco.

Le attività di classificazione vanno ben oltre i modelli LLM e il filtraggio delle e-mail: includono l’identificazione di diverse specie di piante dalle immagini; la catalogazione di articoli di giornale in argomenti come sport, politica e tecnologia; la distinzione tra tumori benigni e maligni dall’imaging medico.

Il punto chiave è che un modello di classificazione, pur ottimizzato, si limita a prevedere le classi che ha incontrato durante il suo addestramento. Per esempio, può determinare se qualcosa è “spam” o “non spam”, come mostra la Figura 6.3, ma non può dire nient’altro sul testo di input.

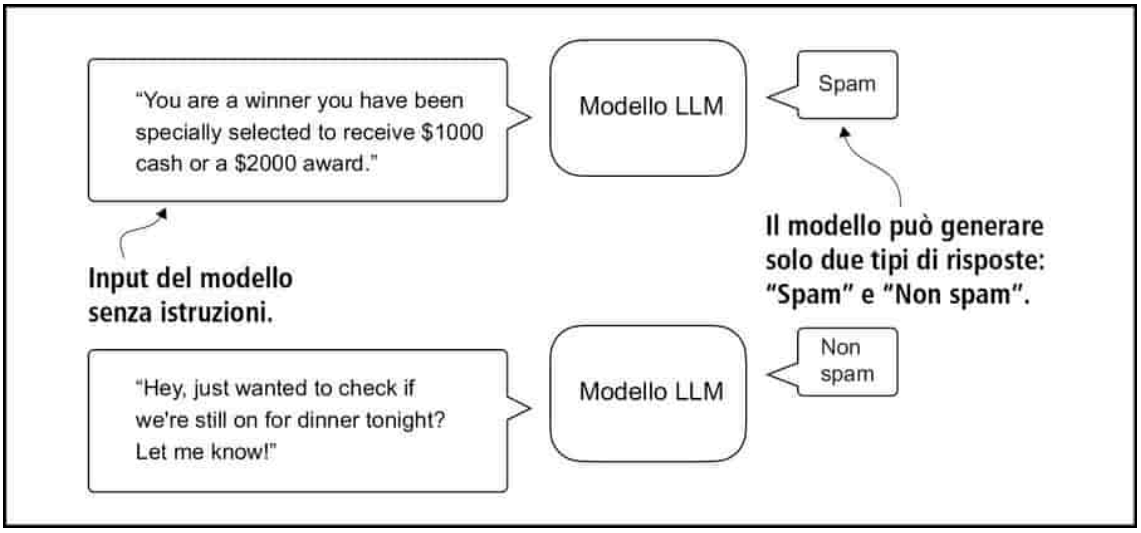


Figura 6.3 Uno scenario di classificazione del testo che utilizza un modello LLM. Un modello ottimizzato per la classificazione dello spam non richiede ulteriori istruzioni insieme all’input. A differenza di un modello a istruzioni ottimizzato, può rispondere solo “spam” o “non spam”.

Contrariamente al modello di classificazione ottimizzato raffigurato nella Figura 6.3, tipicamente un modello a istruzioni ottimizzato può intraprendere una gamma più ampia di attività. Un modello di classificazione ottimizzato è altamente specializzato e, in

genere, è più facile da sviluppare rispetto a un modello generalista che funzioni bene in più attività.

Scegliere l'approccio giusto

Il fine-tuning per rispondere a istruzioni migliora la capacità di un modello di comprendere e generare risposte in base a istruzioni specifiche dell'utente. Il fine-tuning per rispondere a istruzioni è più adatto a modelli che devono saper gestire varie attività in base a istruzioni complesse dell'utente, migliorando la flessibilità e la qualità dell'interazione. Il fine-tuning delle classificazioni è ideale per progetti che richiedono una catalogazione precisa dei dati in classi predefinite, come l'analisi del sentiment o il rilevamento dello spam.

Il fine-tuning per rispondere a istruzioni è più versatile, ma richiede dataset più grandi e maggiori risorse computazionali per sviluppare modelli competenti in vari compiti. Al contrario, il fine-tuning per attività di classificazione richiede meno dati e potenza di calcolo, ma il suo utilizzo si limita alle classi specifiche sulle quali è stato addestrato il modello.

Preparazione del dataset

Modificheremo e ottimizzeremo per la classificazione il modello GPT che abbiamo implementato e pre-addestrato in precedenza. Iniziamo scaricando e preparando il dataset, come evidenziato nella Figura 6.4. Per fornire un esempio intuitivo e utile di ottimizzazione per la classificazione, lavoreremo con un dataset di sms costituito da messaggi spam e non spam.

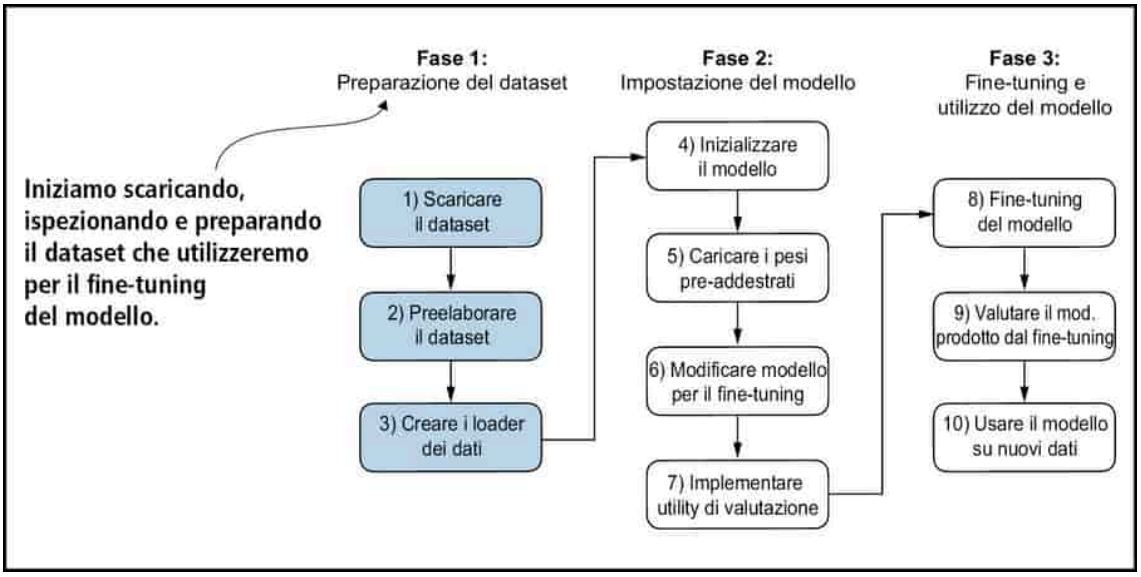


Figura 6.4 Il processo a tre fasi per il fine-tuning di un modello LLM per attività di classificazione. La Fase 1 riguarda la preparazione del dataset. La Fase 2 si concentra sulla configurazione del modello. La Fase 3 riguarda il fine-tuning e la valutazione del modello.

NOTA

Gli sms vengono solitamente inviati tramite smartphone, non tramite e-mail. Tuttavia, questi stessi passi si applicano anche alla classificazione delle e-mail. I lettori interessati possono trovare nell'Appendice B i link ai dataset di classificazione delle e-mail di spam.

Il primo passo consiste nello scaricare il dataset.

Listato 6.1 Download ed espansione del dataset.

```
import urllib.request
import zipfile
import os
from pathlib import Path

url = "https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip"
zip_path = "sms_spam_collection.zip"
extracted_path = "sms_spam_collection"
data_file_path = Path(extracted_path) / "SMSSpamCollection.tsv"

def download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_path):
    if data_file_path.exists():
        print(f"{data_file_path} already exists. Skipping download and
extraction.")
        return
```

```
    with urllib.request.urlopen(url) as response: (1)
        with open(zip_path, "wb") as out_file:
            out_file.write(response.read())

    with zipfile.ZipFile(zip_path, "r") as zip_ref: (2)
        zip_ref.extractall(extracted_path)

    original_file_path = Path(extracted_path) / "SMSSpamCollection"
    os.rename(original_file_path, data_file_path) (3)
    print(f"File downloaded and saved as {data_file_path}")

download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_path)
```

(1) Scarica il file.

(2) Espande il file.

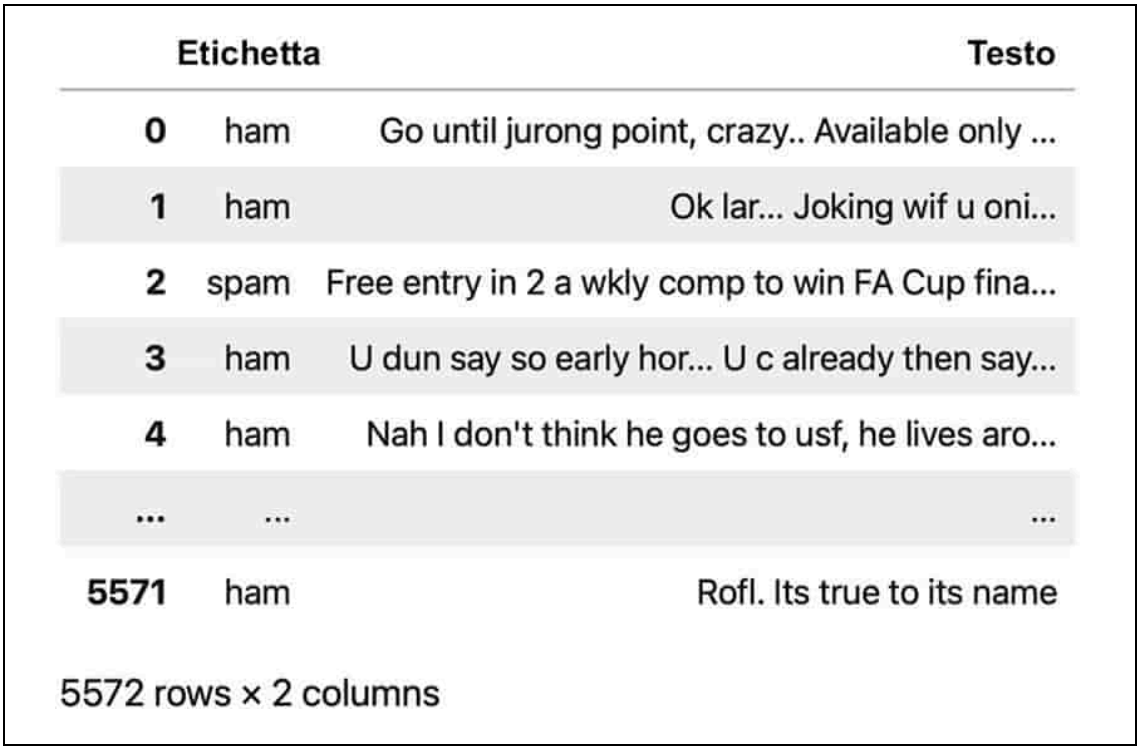
(3) Aggiunge l'estensione `.tsv`.

Dopo aver eseguito il codice precedente, il dataset viene salvato come un file di testo, separato da tabulazioni (*tab-separated values*), `SMSSpamCollection.tsv`, nella cartella `sms_spam_collection`. Possiamo caricarlo in un `DataFrame` pandas come segue:

```
import pandas as pd
df = pd.read_csv(
data_file_path, sep="\t", header=None, names=["Label", "Text"]
)
df (1)
```

(1) Esegue il rendering del data frame in un notebook Jupyter. In alternativa, usate `print(df)`.

La Figura 6.5 mostra il data frame risultante del dataset dello spam.



	Etichetta	Testo
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...
...	...	...
5571	ham	Rofl. Its true to its name

5572 rows × 2 columns

Figura 6.5 Anteprima del dataset SMSSpamCollection in un DataFrame pandas, che mostra le etichette delle classi (“ham” o “spam”) e gli sms corrispondenti. Il dataset è composto da 5.572 righe (sms ed etichette).

Esaminiamo la distribuzione delle etichette delle classi:

```
print(df ["Label"].value_counts())
```

Eseguendo il codice precedente, scopriamo che i dati contengono molti più messaggi “ham” (cioè, “ciccia”, “non spam”) che “spam”:

```
Label
ham     4825
spam     747
Name: count, dtype: int64
```

Per semplicità, e poiché preferiamo un dataset di piccole dimensioni (che faciliterà un più rapido fine-tuning del modello LLM), abbiamo scelto di sub-campionare il dataset per includere 747 istanze da ciascuna classe.

NOTA

Esistono molti altri metodi per gestire gli squilibri fra le classi, ma vanno oltre gli scopi di questo libro. I lettori interessati a esplorare i metodi per gestire lo sbilanciamento nei dati possono trovare ulteriori informazioni nell'Appendice B.

Possiamo utilizzare il codice riportato nel Listato 6.2 per sub-campionare i dati e creare un dataset bilanciato.

Listato 6.2 Creazione di un dataset bilanciato.

```
def create_balanced_dataset(df):
    num_spam = df[df["Label"] == "spam"].shape[0] (1)
    ham_subset = df[df["Label"] == "ham"].sample(num_spam, random_state=123) (2)
    balanced_df = pd.concat([ham_subset, df[df["Label"] == "spam"]]) (3)
    return balanced_df

balanced_df = create_balanced_dataset(df)
print(balanced_df["Label"].value_counts())
```

(1) Conta le istanze di "spam".

(2) Campiona casualmente le istanze di "ham" per farle corrispondere al numero di istanze di "spam".

(3) Combina il sottoinsieme "ham" con "spam".

Dopo aver eseguito il codice precedente per bilanciare il dataset, possiamo vedere che ora abbiamo la stessa quantità di messaggi spam e non spam:

```
Label
ham      747
spam     747
Name: count, dtype: int64
```

Ora, convertiamo le etichette della classe di stringhe `"ham"` e `"spam"` nelle etichette intere `0` e `1`, rispettivamente:

```
balanced_df["Label"] = balanced_df["Label"].map({"ham": 0, "spam": 1})
```

Questo processo è simile alla conversione del testo in token-ID. Tuttavia, invece di usare il vocabolario GPT, costituito da oltre 50.000 parole, abbiamo a che fare con due soli "token-ID": `0` e `1`.

Ora, creiamo una funzione `random_split()` per dividere il dataset in tre parti: il 70 percento per l'addestramento, il 10 percento per la

validazione e il 20 percento per i test. Questi rapporti sono comuni nel machine learning.

Listato 6.3 Suddivisione del dataset.

```
def random_split(df, train_frac, validation_frac):
    df = df.sample(frac=1, random_state=123).reset_index(drop=True) (1)
    train_end = int(len(df) * train_frac) (2)
    validation_end = train_end + int(len(df) * validation_frac)
    (4)
    train_df = df[:train_end]
    validation_df = df[train_end:validation_end]
    test_df = df[validation_end:]

    return train_df, validation_df, test_df

train_df, validation_df, test_df = random_split(balanced_df, 0.7, 0.1) (4)
```

(1) Mescola l'intero DataFrame.

(2) Calcola gli indici delle suddivisioni.

(3) Divide il DataFrame.

(4) La dimensione del test implicita è `0.2`, il "resto".

Salviamo il dataset come un file CSV (*comma separated values*) in modo da poterlo riutilizzare in seguito:

```
train_df.to_csv("train.csv", index=None)
validation_df.to_csv("validation.csv", index=None)
test_df.to_csv("test.csv", index=None)
```

Finora abbiamo scaricato il dataset, lo abbiamo bilanciato e lo abbiamo diviso in sottoinsiemi di addestramento e valutazione. Ora imposteremo i data loader PyTorch che useremo per addestrare il modello.

Creazione dei data loader

Svilupperemo dei data loader PyTorch concettualmente simili a quelli che abbiamo implementato mentre lavoravamo con dati testuali. In precedenza, abbiamo utilizzato una tecnica a finestra scorrevole per generare blocchi di testo di dimensioni uniformi, che abbiamo poi raggruppato in batch per un addestramento più efficiente

del modello. Ogni blocco funzionava come una singola istanza di addestramento. Tuttavia, ora stiamo lavorando con un dataset di spam che contiene sms di lunghezze variabili. Per raggruppare questi messaggi come abbiamo fatto con i blocchi di testo, abbiamo sostanzialmente due opzioni:

- troncare tutti i messaggi alla lunghezza del messaggio più breve del dataset o del batch,
- estendere con un “padding” tutti i messaggi fino a far loro raggiungere la lunghezza del messaggio più lungo del dataset o del batch.

La prima opzione è computazionalmente più economica, ma potrebbe causare una perdita significativa di informazioni, se i messaggi più corti sono molto più piccoli della media o dei messaggi più lunghi, riducendo potenzialmente le prestazioni del modello. Quindi, optiamo per la seconda opzione, che preserva l’intero contenuto di tutti i messaggi.

Per implementare la soluzione in cui tutti i messaggi vengono estesi fino alla lunghezza del messaggio più lungo del dataset, aggiungiamo token di padding a tutti i messaggi più corti. A questo scopo, utilizziamo `"<|endoftext|>"` come token di padding.

Tuttavia, invece di aggiungere la stringa `"<|endoftext|>"` direttamente a ciascuno degli sms, possiamo aggiungere il token-ID corrispondente a `"<|endoftext|>"` agli sms codificati, come mostra la Figura 6.6. Il token-ID di padding `"<|endoftext|>"` è `50256`. Possiamo verificare se l’ID è corretto codificando `"<|endoftext|>"` utilizzando il *tokenizer GPT-2* del pacchetto `tiktoken` che abbiamo utilizzato in precedenza:

```
import tiktoken
tokenizer = tiktoken.get_encoding("gpt2")
print(tokenizer.encode("<|endoftext|>", allowed_special={"<|endoftext|>"}))
```

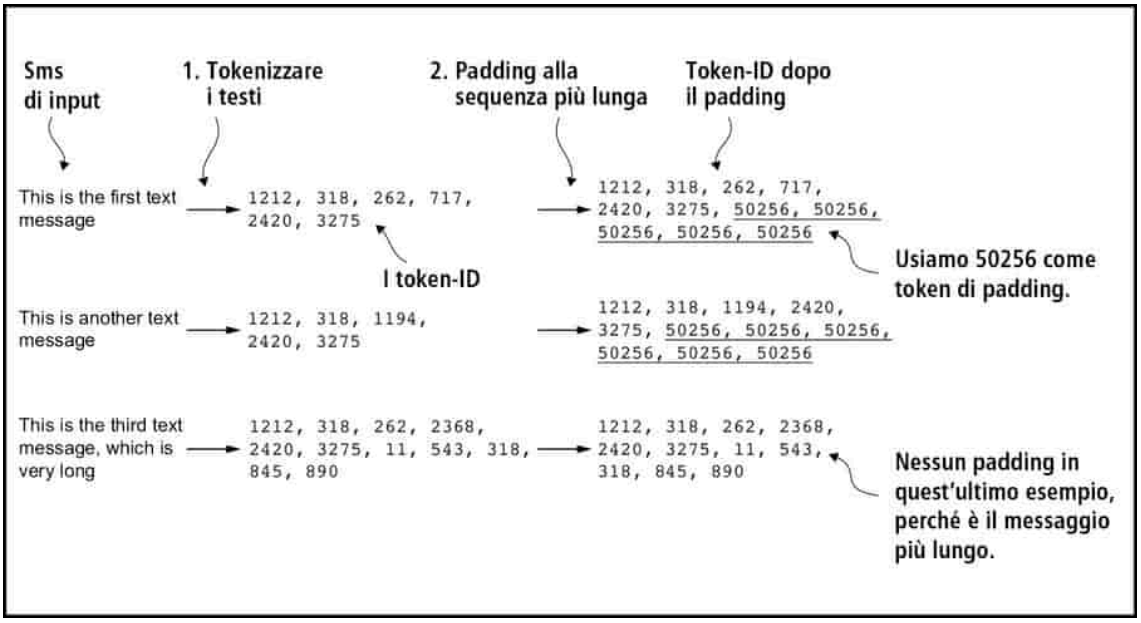


Figura 6.6 Il processo di preparazione del testo di input. Innanzitutto, ogni sms di input viene convertito in una sequenza di token-ID. Quindi, per garantire lunghezze uniformi, alle sequenze più corte viene aggiunto un token di padding (in questo caso, con token-ID 50256) per corrispondere alla lunghezza della sequenza più lunga.

Infatti, l'esecuzione del codice precedente restituisce `[50256]`.

Per prima cosa dobbiamo implementare un `Dataset` PyTorch, che specifica come i dati vengono caricati ed elaborati prima di poter istanziare i data loader. A questo scopo, definiamo la classe `SpamDataset`, che implementa i concetti rappresentati nella Figura 6.6. Questa classe gestisce diverse attività chiave: identifica la sequenza più lunga nel set di addestramento, codifica gli sms e assicura che tutte le altre sequenze siano riempite con un *token di padding* fino alla lunghezza della sequenza più lunga.

Listato 6.4 Impostazione di una classe Dataset con Pytorch.

```
import torch
from torch.utils.data import Dataset

class SpamDataset(Dataset):
    def __init__(self, csv_file, tokenizer, max_length=None, pad_token_id=50256):
```

```
        self.data = pd.read_csv(csv_file)

        self.encoded_texts = [tokenizer.encode(text) for text in
self.data["Text"]] (1)

        if max_length is None:
            self.max_length = self._longest_encoded_length()
        else:
            self.max_length = max_length
            self.encoded_texts = [ (2)
                encoded_text[:self.max_length]
                for encoded_text in self.encoded_texts
            ]

        self.encoded_texts = [ (3)
            encoded_text + [pad_token_id] * (self.max_length - len(encoded_text))
            for encoded_text in self.encoded_texts
        ]

    def __getitem__(self, index):
        encoded = self.encoded_texts[index]
        label = self.data.iloc[index]["Label"]
        return (
            torch.tensor(encoded, dtype=torch.long),
            torch.tensor(label, dtype=torch.long)
        )

    def __len__(self):
        return len(self.data)

    def _longest_encoded_length(self):
        max_length = 0
        for encoded_text in self.encoded_texts:
            encoded_length = len(encoded_text)
            if encoded_length > max_length:
                max_length = encoded_length
        return max_length
```

(1) Pre-tokenizza i testi.

(2) Tronca le sequenze se sono più lunghe di `max_length`.

(3) Estende (padding) le sequenze alla sequenza più lunga.

La classe `SpamDataset` carica i dati dai file CSV creati in precedenza, tokenizza il testo utilizzando il tokenizer GPT-2 di `tiktoken` e ci consente di *estendere* o *troncare* le sequenze a una lunghezza uniforme, determinata dalla sequenza più lunga o da una lunghezza massima predefinita. Ciò garantisce che ogni tensore di input abbia le stesse dimensioni, il che è necessario per creare i batch nel data loader di addestramento che implementiamo in seguito:

```
train_dataset = SpamDataset(
    csv_file="train.csv",
    max_length=None,
    tokenizer=tokenizer
)
```

La lunghezza della sequenza più lunga viene memorizzata nell'attributo `max_length` del dataset. Se siete curiosi di conoscere il numero di token nella sequenza più lunga, potete usare la seguente riga di codice:

```
print(train_dataset.max_length)
```

Il codice restituisce `120`, mostrando che la sequenza più lunga contiene non più di 120 token, una lunghezza comune per gli sms. Il modello può gestire sequenze fino a 1024 token, data la lunghezza limite del contesto. Se il vostro dataset include testi più lunghi, potete passare `max_length=1024` durante la creazione del set di addestramento nel codice precedente, per garantire che i dati non superino la lunghezza di input (il contesto) supportata dal modello.

Ora, eseguiamo il padding ai set di validazione e di test per farli corrispondere alla lunghezza della sequenza di addestramento più lunga. È importante notare che tutti i campioni dei set di validazione e di test che superano la lunghezza dell'esempio di addestramento più lungo vengono troncati usando `encoded_text[:self.max_length]` nel codice di `SpamDataset` che abbiamo definito in precedenza. Questo troncamento è facoltativo; potete impostare `max_length=None` per i set di validazione e di test, a condizione che non contengano sequenze che superano 1024 token:

```
val_dataset = SpamDataset(
    csv_file="validation.csv",
    max_length=train_dataset.max_length,
    tokenizer=tokenizer
)

test_dataset = SpamDataset(
    csv_file="test.csv",
    max_length=train_dataset.max_length,
```

```
    tokenizer=tokenizer
)
```

Esercizio 6.1 Aumento della lunghezza del contesto
Eseguite il padding dell'input al numero massimo di token supportati dal modello e osservate come ciò influisce sulle prestazioni delle predizioni.

Utilizzando i dataset come input, possiamo istanziare i data loader in modo simile a quando lavoravamo con dati testuali. Tuttavia, in questo caso, i target rappresentano le etichette delle classi anziché i token successivi nel testo. Per esempio, se scegliamo una dimensione del batch pari a 8, ogni batch sarà composto da otto esempi di addestramento di lunghezza 120 e dall'etichetta della classe corrispondente a ogni esempio, come mostra la Figura 6.7.

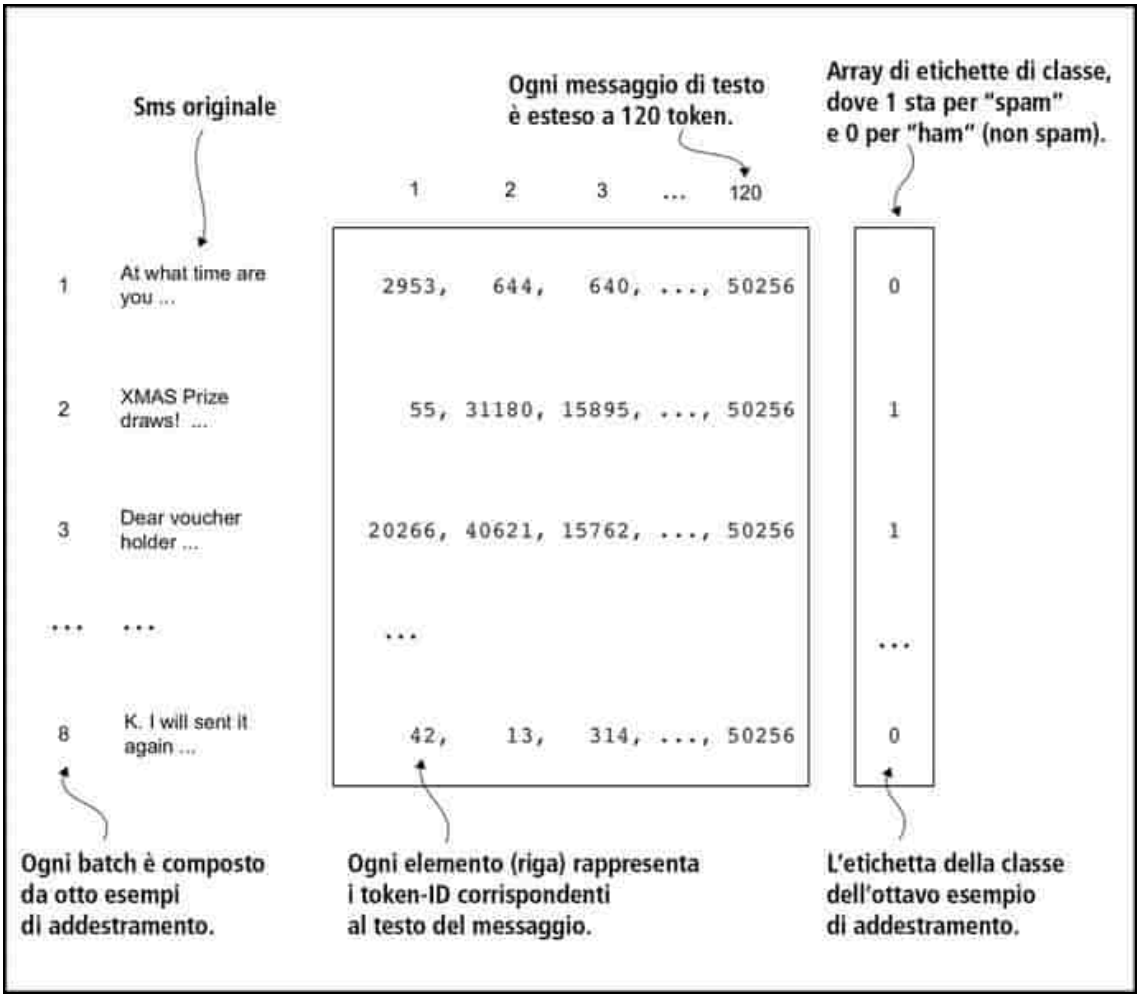


Figura 6.7 Un singolo batch di addestramento composto da otto sms rappresentati come token-ID. Ogni messaggio di testo è composto da 120 token-ID. Un array di etichette delle classi memorizza le otto etichette delle classi corrispondenti agli sms, che possono essere 0 ("non spam") o 1 ("spam").

Il codice riportato nel Listato 6.5 crea i data loader dei set di addestramento, validazione e test che caricano gli sms e le etichette in batch di dimensioni 8.

Listato 6.5 Creazione dei data loader PyTorch.

```
from torch.utils.data import DataLoader

num_workers = 0 (1)
batch_size = 8
```

```
torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True,
    num_workers=num_workers,
    drop_last=True,
)

val_loader = DataLoader(
    dataset=val_dataset,
    batch_size=batch_size,
    num_workers=num_workers,
    drop_last=False,
)

test_loader = DataLoader(
    dataset=test_dataset,
    batch_size=batch_size,
    num_workers=num_workers,
    drop_last=False,
)
```

(1) Questa impostazione è per motivi di compatibilità.

Per garantire che i data loader funzionino e restituiscano effettivamente batch delle dimensioni previste, eseguiamo un'iterazione sul loader di addestramento e visualizziamo le dimensioni del tensore dell'ultimo batch:

```
for input_batch, target_batch in train_loader:
    pass
print("Input batch dimensions:", input_batch.shape)
print("Label batch dimensions", target_batch.shape)
```

L'output è il seguente:

```
Input batch dimensions: torch.Size([8, 120])
Label batch dimensions torch.Size([8])
```

Come potete vedere, i batch di input sono composti da otto esempi di addestramento di 120 token ciascuno, come previsto. Il tensore memorizza le etichette delle classi corrispondenti agli otto esempi di addestramento.

Infine, per avere un'idea delle dimensioni del dataset, visualizziamo il numero totale di batch in ciascun dataset:

```
print(f"{len(train_loader)} training batches")
print(f"{len(val_loader)} validation batches")
print(f"{len(test_loader)} test batches")
```

Il numero di batch in ogni dataset è il seguente:

```
130 training batches
19 validation batches
38 test batches
```

Ora che abbiamo preparato i dati, dobbiamo preparare il modello per il fine-tuning.

Inizializzazione di un modello con pesi pre-addestrati

Dobbiamo preparare il modello per eseguire il fine-tuning per attività di classificazione con lo scopo di identificare i messaggi spam. Iniziamo inizializzando il modello pre-addestrato, come evidenziato nella Figura 6.8.

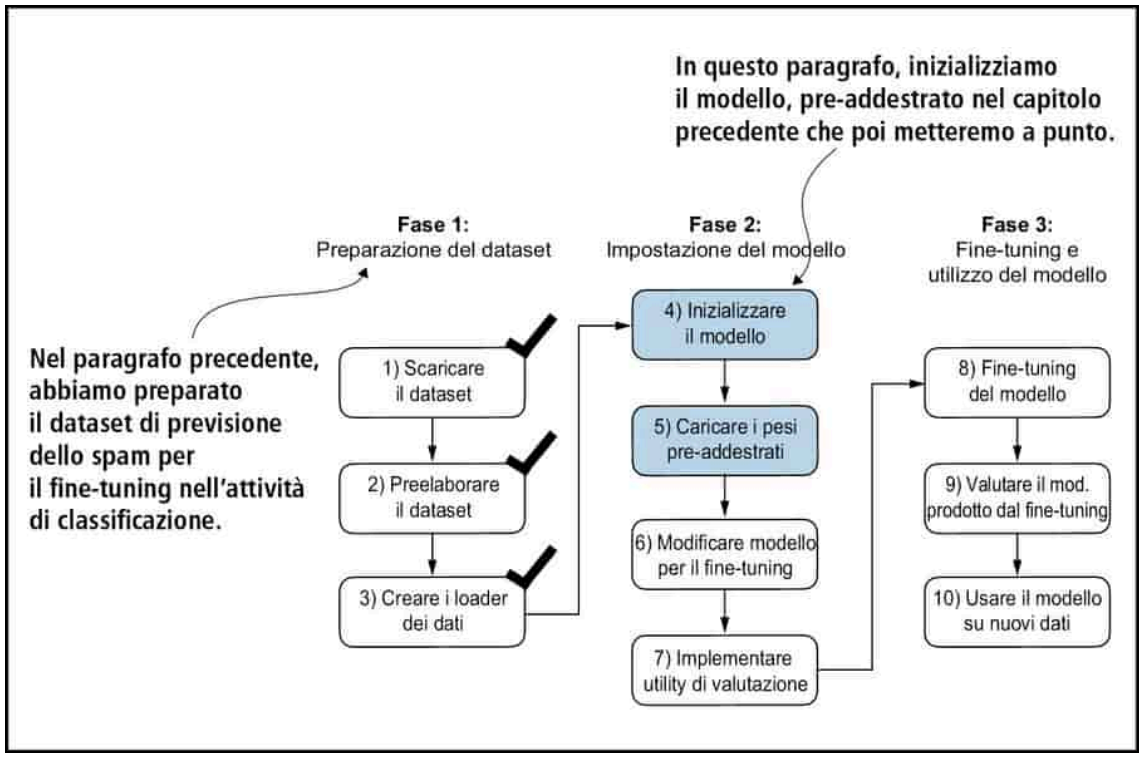


Figura 6.8 Il processo a tre fasi per il fine-tuning per scopi di classificazione del modello LLM. Dopo aver completato la Fase 1, preparando il dataset, ora dobbiamo inizializzare il

modello LLM, che poi ottimizzeremo per classificare i messaggi di spam.

Per iniziare il processo di preparazione del modello, utilizziamo le stesse configurazioni utilizzate per pre-addestrare i dati non etichettati:

```
CHOOSE_MODEL = "gpt2-small (124M)"
INPUT_PROMPT = "Every effort moves"

BASE_CONFIG = {
    "vocab_size": 50257, (1)
    "context_length": 1024, (2)
    "drop_rate": 0.0, (3)
    "qkv_bias": True (4)
}

model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12},
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16},
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20},
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}

BASE_CONFIG.update(model_configs[CHOOSE_MODEL])
```

(1) Dimensioni del vocabolario.

(2) Lunghezza del contesto.

(3) Tasso di dropout.

(4) Bias query-chiave-valore.

Ora, importiamo la funzione `download_and_load_gpt2()` dal file `gpt_download.py` e riutilizziamo la classe `GPTModel` e la funzione `load_weights_into_gpt()` dal pre-addestramento (vedi il Capitolo 5) per caricare nel modello GPT i pesi scaricati.

Listato 6.6 Caricamento di un modello GPT pre-addestrato.

```
from gpt_download import download_and_load_gpt2
from chapter05 import GPTModel, load_weights_into_gpt

model_size = CHOOSE_MODEL.split(" ")[-1].lstrip("(").rstrip(")")
settings, params = download_and_load_gpt2(model_size=model_size,
models_dir="gpt2")

model = GPTModel(BASE_CONFIG)
load_weights_into_gpt(model, params)
model.eval()
```

Dopo aver caricato in `GPTModel` i pesi del modello, riutilizziamo la funzione di servizio di generazione del testo dei Capitoli 4 e 5 per garantire che il modello generi un testo coerente:

```
from chapter04 import generate_text_simple
from chapter05 import text_to_token_ids, token_ids_to_text

text_1 = "Every effort moves you"
token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(text_1, tokenizer),
    max_new_tokens=15,
    context_size=BASE_CONFIG["context_length"]
)
print(token_ids_to_text(token_ids, tokenizer))
```

L’output seguente mostra che il modello genera un testo coerente, il che indica che i pesi del modello sono stati caricati correttamente:

```
Every effort moves you forward.
The first step is to understand the importance of your work
```

Prima di iniziare a eseguire il fine-tuning del modello come classificatore dello spam, vediamo se il modello classifica già i messaggi di spam fornendogli istruzioni:

```
text_2 = (
    "Is the following text 'spam'? Answer with 'yes' or 'no':"
    " 'You are a winner you have been specially"
    " selected to receive $1000 cash or a $2000 award.'"
)
token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(text_2, tokenizer),
    max_new_tokens=23,
    context_size=BASE_CONFIG["context_length"]
)
print(token_ids_to_text(token_ids, tokenizer))
```

L’output del modello è il seguente:

```
Is the following text 'spam'? Answer with 'yes' or 'no': 'You are a winner you
have been specially selected to receive $1000 cash or a $2000 award.'
The following text 'spam'? Answer with 'yes' or 'no': 'You are a winner
```

In base all’output, è evidente che il modello fatica a seguire le istruzioni. Questo risultato è previsto, poiché ha subito un solo pre-addestramento e non ha un fine-tuning per rispondere a istruzioni. Quindi, prepariamo il modello per il fine-tuning per attività di classificazione.

Aggiunta di una “testa” per la classificazione

Dobbiamo modificare il modello LLM pre-addestrato per prepararlo al fine-tuning per attività di classificazione. Per farlo, sostituiamo il layer di output originale, che mappa la rappresentazione nascosta su un vocabolario di 50257 elementi, con un layer di output più piccolo, che mappa su due sole classi: `0` (“non spam”) e `1` (“spam”), come mostra la Figura 6.9. Utilizziamo lo stesso modello di prima, tranne per il nuovo layer di output.

Nodi del layer di output

Tecnicamente potevamo utilizzare un singolo nodo di output, poiché abbiamo a che fare con un compito di classificazione binario. Tuttavia, ciò avrebbe richiesto la modifica della funzione `loss`, come ho trattato in *Losses Learned – Optimizing Negative Log-Likelihood and Cross-Entropy in PyTorch* (`https://mng.bz/NRZ2`). Pertanto, scegliamo un approccio più generale, in cui il numero di nodi di output corrisponde al numero delle classi. Per esempio, per un problema a tre classi, come la classificazione di articoli di giornale come “Tecnologia”, “Sport” o “Politica”, utilizzeremmo tre nodi di output, e così via.

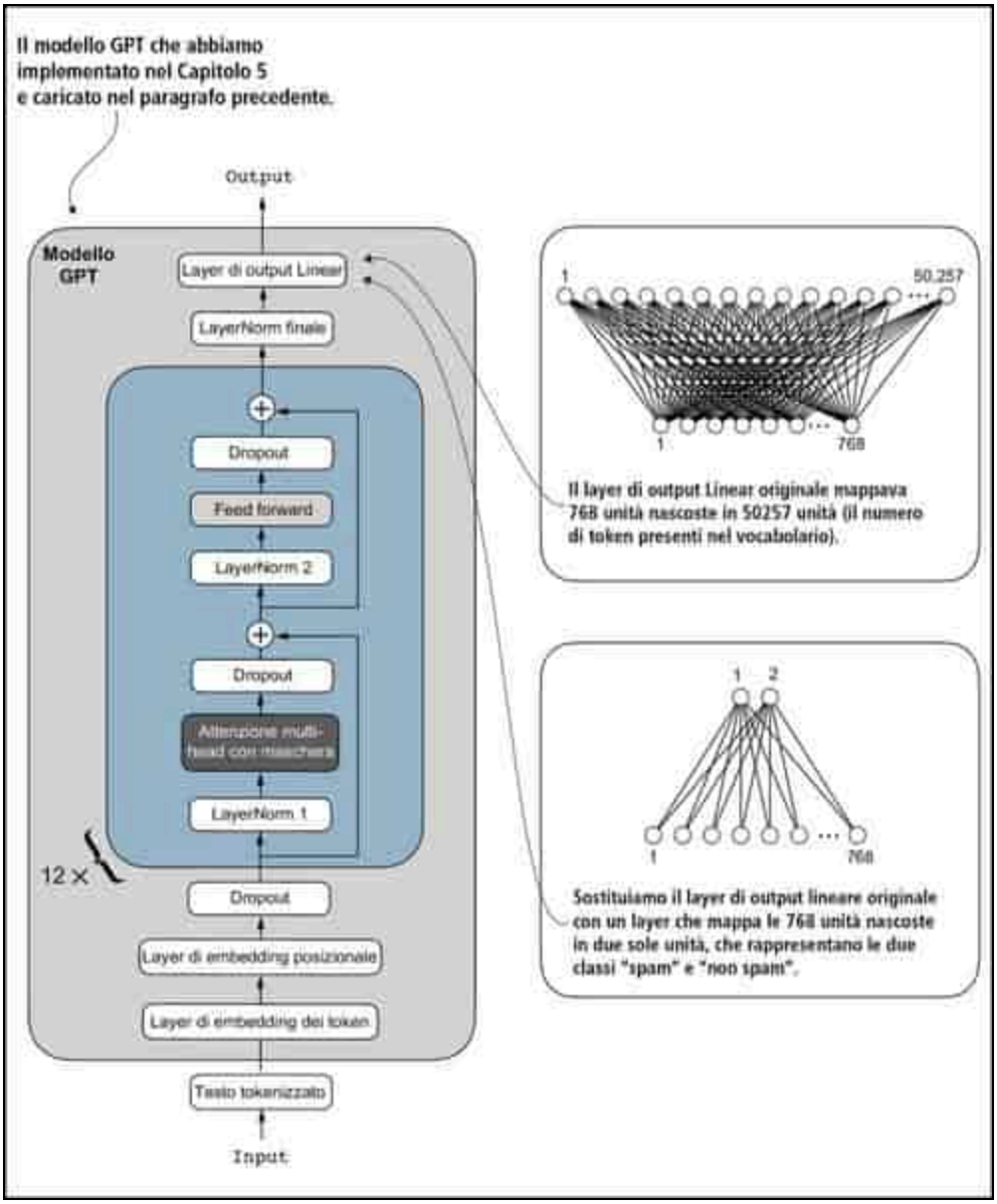


Figura 6.9 Adattamento di un modello GPT per la classificazione dello spam modificandone l'architettura. Inizialmente, il layer di output lineare del modello ha mappato 768 unità nascoste su un vocabolario di 50257 token. Per rilevare lo spam, sostituiamo questo layer con un nuovo layer di output che mappa le stesse 768 unità nascoste su due sole classi: "spam" e "non spam".

Prima di tentare la modifica rappresentata nella Figura 6.9, visualizziamo l'architettura del modello tramite `print(model)`:

```
GPTModel(
    (tok_emb): Embedding(50257, 768)
    (pos_emb): Embedding(1024, 768)
    (drop_emb): Dropout(p=0.0, inplace=False)
    (trf_blocks): Sequential(
...
        (11): TransformerBlock(
            (att): MultiHeadAttention(
                (W_query): Linear(in_features=768, out_features=768, bias=True)
                (W_key): Linear(in_features=768, out_features=768, bias=True)
                (W_value): Linear(in_features=768, out_features=768, bias=True)
                (out_proj): Linear(in_features=768, out_features=768, bias=True)
                (dropout): Dropout(p=0.0, inplace=False)
            )
            (ff): FeedForward(
                (layers): Sequential(
                    (0): Linear(in_features=768, out_features=3072, bias=True)
                    (1): GELU()
                    (2): Linear(in_features=3072, out_features=768, bias=True)
                )
            )
            (norm1): LayerNorm()
            (norm2): LayerNorm()
            (drop_resid): Dropout(p=0.0, inplace=False)
        )
    )
    (final_norm): LayerNorm()
    (out_head): Linear(in_features=768, out_features=50257, bias=False)
)
```

Questo output mostra ordinatamente l'architettura che abbiamo visto nel Capitolo 4. Come abbiamo precedentemente discusso, il `GPTModel` è costituito dai layer di embedding seguiti da 12 *blocchi transformer* identici (per brevità viene mostrato solo l'ultimo blocco), seguiti da un ultimo layer `LayerNorm` e dal layer di output, `out_head`. Ora, sostituiamo `out_head` con un nuovo layer di output (vedi Figura 6.9) che poi perfezioneremo.

Fine-tuning di determinati layer vs di tutti i layer

Poiché partiamo da un modello pre-addestrato, non è necessario eseguire il fine-tuning di tutti i layer del modello. Nei modelli linguistici basati su reti neurali, i layer inferiori generalmente catturano le strutture linguistiche di base e gli aspetti semantici applicabili a un'ampia varietà di attività e dataset. Quindi, il fine-tuning solo degli ultimi layer (vale a dire, quelli più vicini all'output), che sono più specifici per i modelli linguistici e per le caratteristiche specifiche dell'attività, spesso è sufficiente per adattare il modello a nuove attività. Un piacevole effetto collaterale è che è computazionalmente più efficiente eseguire il fine-tuning solo di un piccolo

numero di layer. I lettori interessati possono trovare maggiori informazioni, inclusi esperimenti, sui layer da sottoporre a fine-tuning nell'Appendice B.

Per preparare il modello al fine-tuning per attività di classificazione, per prima cosa *congeliamo* il modello, ovvero rendiamo tutti i livelli non addestrabili:

```
for param in model.parameters():
    param.requires_grad = False
```

Poi, sostituiamo il layer di output (`model.out_head`), che originariamente mappa gli input del layer su 50257 dimensioni, le dimensioni del vocabolario (vedi Figura 6.9).

Listato 6.7 Aggiunta di un layer di classificazione.

```
torch.manual_seed(123)
num_classes = 2
model.out_head = torch.nn.Linear(
    in_features=BASE_CONFIG["emb_dim"],
    out_features=num_classes
)
```

Per mantenere il codice più generale, utilizziamo `BASE_CONFIG["emb_dim"]`, che è uguale a 768 nel modello `"gpt2-small (124M)"`. Successivamente, possiamo anche usare lo stesso codice per lavorare con le varianti più grandi del modello GPT-2.

Questo nuovo layer di output, `model.out_head`, ha l'attributo `requires_grad` impostato su `True`, di default, il che significa che è l'unico layer del modello che verrà aggiornato durante l'addestramento. Tecnicamente, l'addestramento del layer di output che abbiamo appena aggiunto è sufficiente. Tuttavia, come ho scoperto negli esperimenti, il fine-tuning anche di altri layer può migliorare notevolmente le prestazioni delle predizioni del modello (per maggiori dettagli, fate riferimento all'Appendice B). Configuriamo anche l'ultimo blocco transformer e il modulo `LayerNorm` finale, che collega questo blocco al layer di output, in modo che siano addestrabili, come mostra la Figura 6.10.

Per rendere addestrabili il `LayerNorm` finale e l'ultimo blocco transformer, impostiamo i rispettivi `requires_grad` su `True`:

```
for param in model.trf_blocks[-1].parameters():
    param.requires_grad = True
for param in model.final_norm.parameters():
    param.requires_grad = True
```

Esercizio 6.2 Ottimizzazione dell'intero modello

Invece di perfezionare solo l'ultimo blocco transformer, eseguite il fine-tuning dell'intero modello e valutate l'effetto sulle prestazioni delle predizioni.

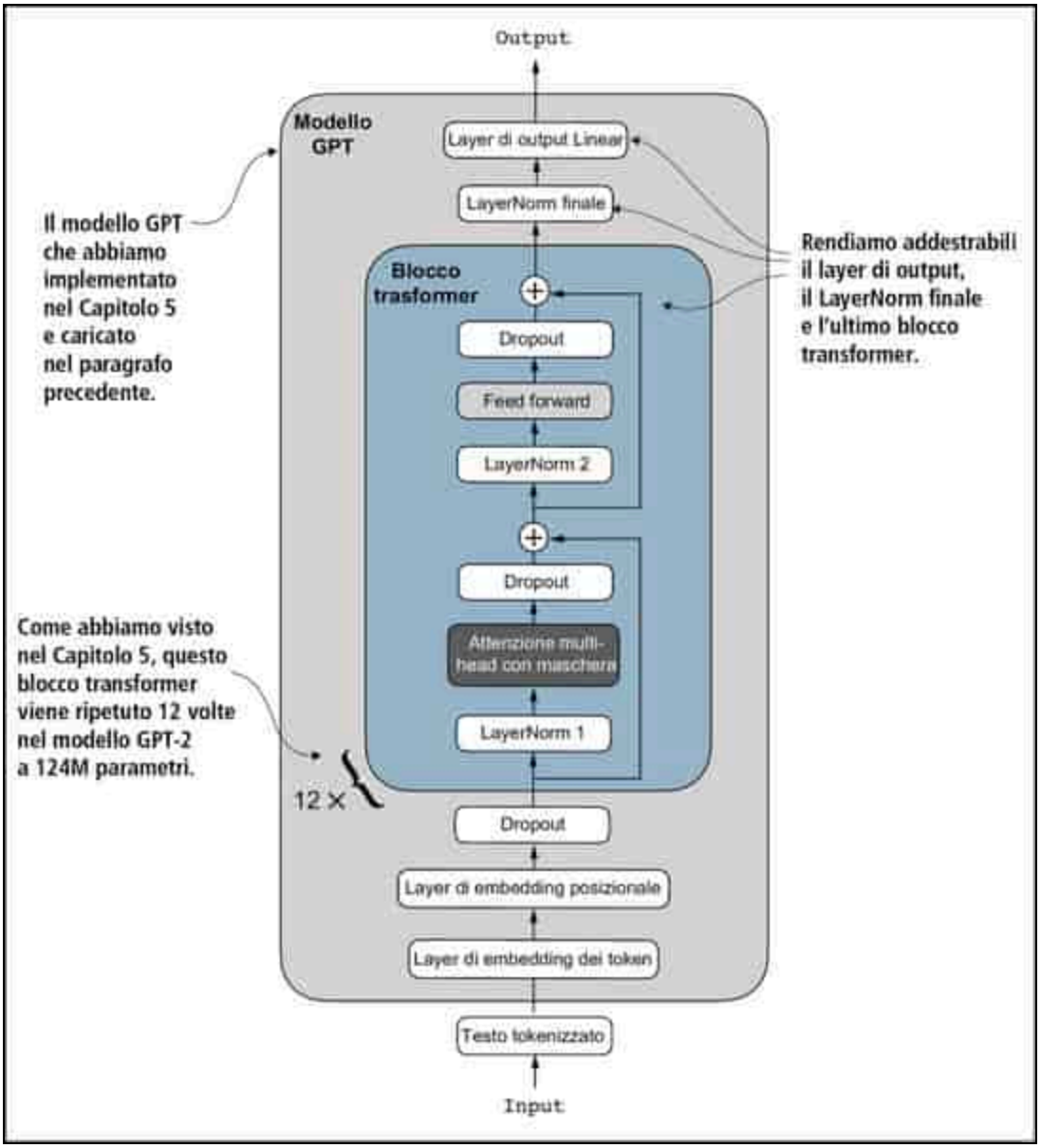


Figura 6.10 Il modello GPT include 12 blocchi transformer ripetuti. Accanto al layer di output, impostiamo il LayerNorm finale e l'ultimo blocco transformer come addestrabili. I restanti 11 blocchi transformer e i layer di embedding vengono mantenuti non addestrabili.

Anche se abbiamo aggiunto un nuovo layer di output e contrassegnato alcuni layer come addestrabili o non addestrabili, possiamo comunque usare questo modello in modo simile a come

abbiamo fatto in precedenza. Per esempio, possiamo fornirgli un testo campione identico a quello che abbiamo usato in precedenza:

```
inputs = tokenizer.encode("Do you have time")
inputs = torch.tensor(inputs).unsqueeze(0)
print("Inputs:", inputs)
print("Inputs dimensions:", inputs.shape) (1)
```

(1) `shape`: `(batch_size, num_tokens)`.

L’output mostra che il codice precedente codifica gli input in un tensore costituito da quattro token di input:

```
Inputs: tensor([[5211, 345, 423, 640]])
Inputs dimensions: torch.Size([1, 4])
```

Possiamo quindi passare al modello i token-ID codificati, come di consueto:

```
with torch.no_grad():
    outputs = model(inputs)
print("Outputs:\n", outputs)
print("Outputs dimensions:", outputs.shape)
```

Il tensore di output ha il seguente aspetto:

```
Outputs:
 tensor([[[-1.5854, 0.9904],
          [-3.7235, 7.4548],
          [-2.2661, 6.6049],
          [-3.5983, 3.9902]]])
Outputs dimensions: torch.Size([1, 4, 2])
```

Un input simile, precedentemente avrebbe prodotto un tensore di output `[1, 4, 50257]`, dove `50257` rappresenta le dimensioni del vocabolario. Il numero di righe di output corrisponde al numero di token di input (in questo caso, `4`). Tuttavia, le dimensioni dell’embedding di ogni output (il numero di colonne) ora è `2` anziché `50257`, poiché abbiamo sostituito il layer di output del modello.

Ricordate che siamo interessati a eseguire il fine-tuning di questo modello per restituire l’etichetta di una classe, che indica se l’input del modello è “spam” o “non spam”. Non abbiamo bisogno di eseguire il fine-tuning di tutte e quattro le righe di output; invece, possiamo concentrarci su un singolo token di output. In particolare, ci

concentreremo sull’ultima riga, corrispondente all’ultimo token di output, come mostra la Figura 6.11.

Per estrarre l’ultimo token di output fornito dal tensore di output, utilizziamo la seguente riga di codice:

```
print("Last output token:", outputs[:, -1, :])
```

Questo codice produce il seguente output:

```
Last output token: tensor([[-3.5983, 3.9902]])
```

Dobbiamo ancora convertire i valori nella previsione di una classe. Ma prima, cerchiamo di capire perché siamo così interessati solo all’ultimo token dell’output.

Abbiamo già esplorato il meccanismo di attenzione, che stabilisce una relazione tra ogni token di input e ogni altro token di input, e il concetto di *maschera di attenzione causale*, comunemente usata nei modelli di tipo GPT (vedi il Capitolo 3). Questa maschera limita il focus di un token alla sua posizione corrente e a quelle precedenti, assicurando che ogni token possa essere influenzato solo da se stesso e dai token precedenti, come mostra la Figura 6.12.

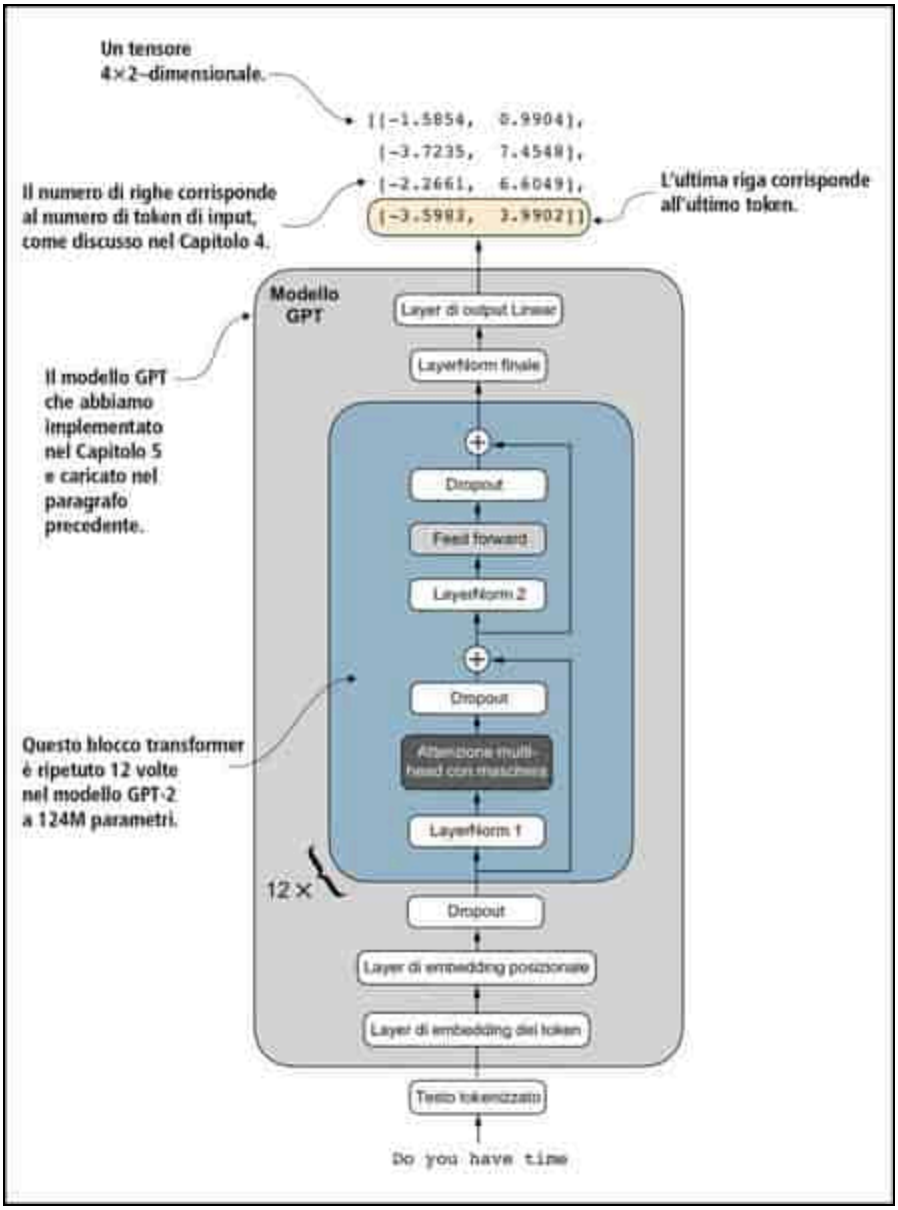


Figura 6.11 Il modello GPT con un input e un output di esempio a 4 token. Il tensore di output è costituito da 2 colonne grazie al layer di output modificato. Siamo interessati solo all'ultima riga, corrispondente all'ultimo token, quando eseguiamo il fine-tuning del modello per la classificazione dello spam.

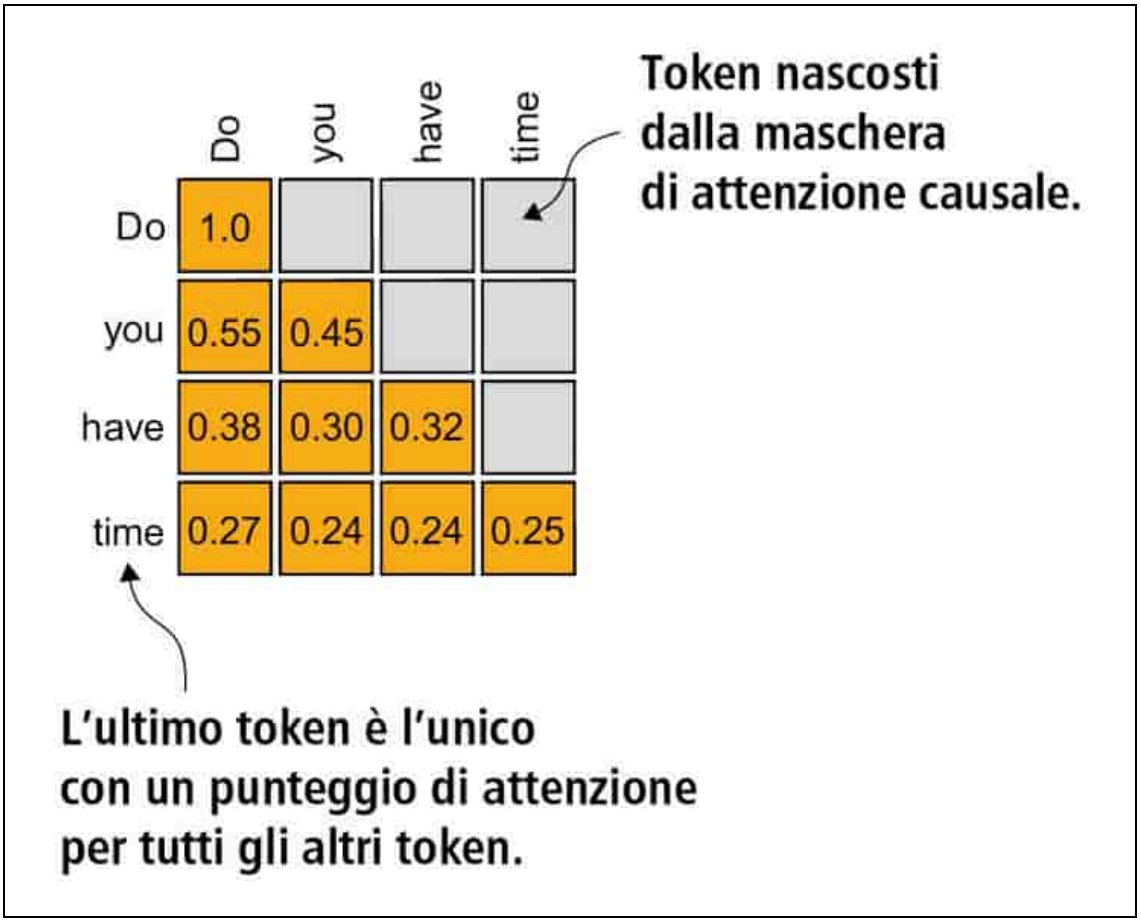


Figura 6.12 Il meccanismo di attenzione causale, in cui i punteggi di attenzione tra i token di input vengono visualizzati in un formato a matrice. Le celle vuote indicano le posizioni occultate dalla maschera di attenzione causale, per impedire ai token di considerare i token successivi. I valori nelle celle rappresentano i punteggi di attenzione; l'ultimo token, time, è l'unico che calcola i punteggi di attenzione per tutti i token precedenti.

Data la configurazione della maschera di attenzione causale nella Figura 6.12, l'ultimo token di una sequenza accumula la maggior parte delle informazioni, poiché è l'unico con accesso ai dati di tutti i token precedenti. Pertanto, nel nostro compito di classificazione dello spam, ci concentriamo su quest'ultimo token durante il processo di fine-tuning.

Ora siamo pronti a trasformare l'ultimo token in predizioni di etichette delle classi e a calcolare l'accuratezza della predizione iniziale del modello. Ora, perfezioneremo il modello per l'attività di classificazione dello spam.

Esercizio 6.3 Ottimizzazione del primo vs dell'ultimo token
Provate a eseguire il fine-tuning del primo token di output. Prendete nota dei cambiamenti nelle prestazioni delle predizioni rispetto al fine-tuning dell'ultimo token di output.

Calcolo del valore loss e dell'accuratezza della classificazione

Resta solo un piccolo compito prima del fine-tuning del modello: dobbiamo implementare le funzioni di valutazione del modello da utilizzare per il fine-tuning, come mostra la Figura 6.13.

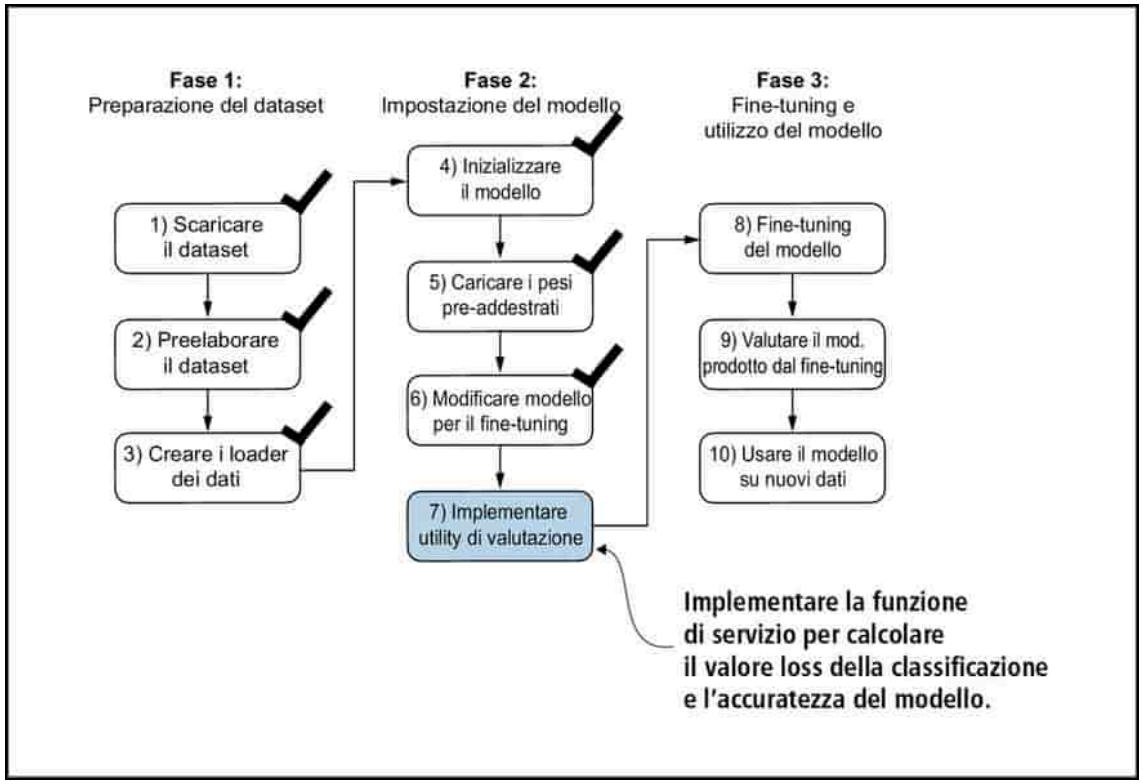


Figura 6.13 Il processo a tre fasi per il fine-tuning per scopi di classificazione del modello LLM. Abbiamo completato i primi sei passi. Ora siamo pronti per intraprendere l'ultimo passo della Fase 2: implementare le funzioni per valutare le prestazioni del modello di classificazione dei messaggi di spam prima, durante e dopo il fine-tuning.

Prima di implementare le utility di valutazione, vediamo brevemente come convertiamo gli output del modello in previsioni di etichette delle classi. In precedenza, abbiamo calcolato il token-ID successivo generato dal modello LLM convertendo i 50257 output in probabilità tramite la funzione `softmax()` e quindi restituendo la posizione della probabilità più elevata tramite la funzione `argmax()`. Adottiamo lo stesso approccio per calcolare se il modello genera una previsione "spam" o "non spam" per un dato input, come mostra la Figura 6.14. L'unica differenza è che lavoriamo con output bidimensionali anziché 50257-dimensionali.

Consideriamo l’ultimo output del token utilizzando un esempio concreto:

```
print("Last output token:", outputs[:, -1, :])
```

I valori del tensore corrispondente all’ultimo token sono i seguenti:

```
Last output token: tensor([[-3.5983, 3.9902]])
```

Possiamo ottenere l’etichetta della classe:

```
probas = torch.softmax(outputs[:, -1, :], dim=-1)
label = torch.argmax(probas)
print("Class label:", label.item())
```

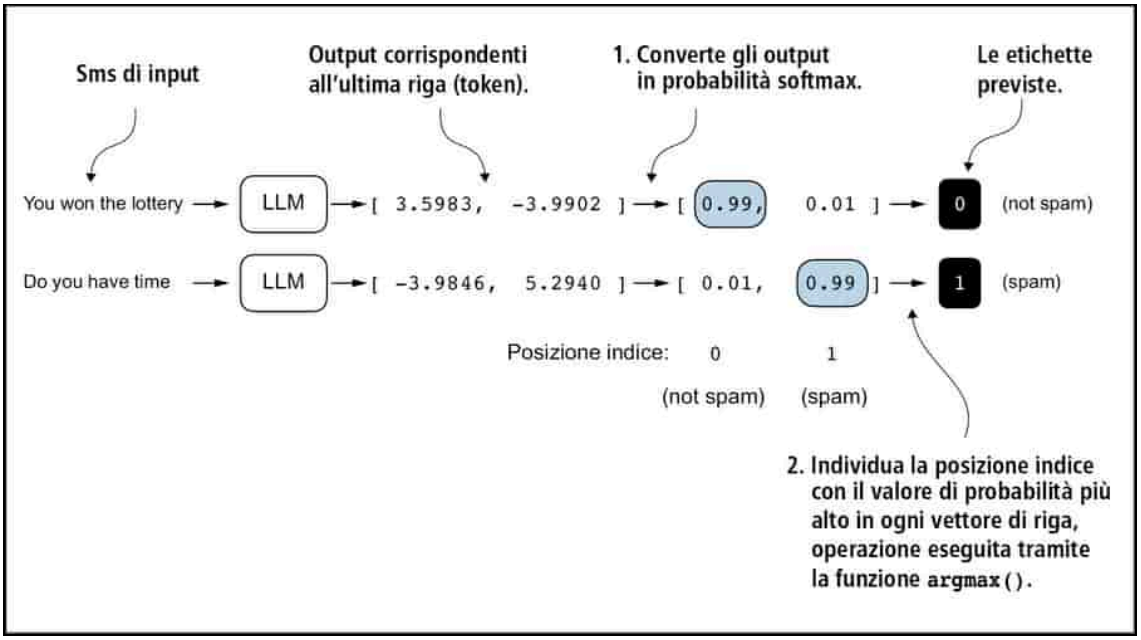


Figura 6.14 Gli output del modello corrispondenti all’ultimo token vengono convertiti in punteggi di probabilità per ogni testo di input. Le etichette delle classi vengono ottenute cercando l’indice del punteggio di probabilità più elevato. Il modello prevede le etichette spam in modo errato, perché non è ancora stato addestrato.

In questo caso, il codice restituisce `1`, il che significa che il modello prevede che il testo di input sia “spam”. L’utilizzo della funzione `softmax()` qui è facoltativo, perché gli output più grandi corrispondono direttamente ai punteggi di probabilità più elevati. Quindi, possiamo semplificare il codice evitando di utilizzare `softmax()`:

```
logits = outputs[:, -1, :]
label = torch.argmax(logits)
print("Class label:", label.item())
```

Questo concetto può essere utilizzato per calcolare l'accuratezza della classificazione, che misura la percentuale di previsioni corrette su un dataset.

Per determinare l'accuratezza della classificazione, applichiamo il codice di previsione basato su `argmax()` a tutti gli esempi del dataset e calcoliamo la percentuale di previsioni corrette definendo una funzione `calc_accuracy_loader()`.

Listato 6.8 Calcolo dell'accuratezza della classificazione.

```
def calc_accuracy_loader(data_loader, model, device, num_batches=None):
    model.eval()
    correct_predictions, num_examples = 0, 0

    if num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            input_batch = input_batch.to(device)
            target_batch = target_batch.to(device)

            with torch.no_grad():
                logits = model(input_batch)[:, -1, :] (1)
            predicted_labels = torch.argmax(logits, dim=-1)

            num_examples += predicted_labels.shape[0]
            correct_predictions += ((predicted_labels ==
target_batch).sum().item())
        else:
            break
    return correct_predictions / num_examples
```

(1) Logit dell'ultimo token di output.

Utilizziamo la funzione per determinare l'accuratezza della classificazione tra vari dataset stimati da 10 batch, per motivi di efficienza:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

torch.manual_seed(123)

train_accuracy = calc_accuracy_loader(train_loader, model, device, num_batches=10)
```

```
val_accuracy = calc_accuracy_loader(val_loader, model, device, num_batches=10)
test_accuracy = calc_accuracy_loader(test_loader, model, device, num_batches=10)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

Tramite l'impostazione `device`, il modello viene eseguito automaticamente su una GPU (se è disponibile una GPU con supporto Nvidia CUDA), altrimenti viene eseguito su una CPU. L'output è il seguente:

```
Training accuracy: 46.25%
Validation accuracy: 45.00%
Test accuracy: 48.75%
```

Come potete vedere, le accuratezze delle previsioni sono vicine a una previsione casuale, che sarebbe del 50 percento in questo caso. Per migliorare le accuratezze delle previsioni, dobbiamo eseguire il fine-tuning del modello.

Tuttavia, prima di iniziare a eseguire il fine-tuning del modello, dobbiamo definire la funzione `loss` che vogliamo ottimizzare durante l'addestramento. Il nostro obiettivo è quello di massimizzare l'accuratezza del modello nella classificazione dello spam, il che significa che il codice precedente dovrebbe restituire le etichette delle classi corrette: `0` per non spam e `1` per spam.

Poiché l'accuratezza della classificazione non è una funzione differenziabile, utilizziamo il valore loss a entropia incrociata per massimizzare l'accuratezza. Di conseguenza, la funzione `calc_loss_batch()` rimane la stessa, con una modifica: ci concentriamo sull'ottimizzazione solo dell'ultimo token, `model(input_batch)[:, -1, :]`, invece che su tutti i token, `model(input_batch)`:

```
def calc_loss_batch(input_batch, target_batch, model, device):
    input_batch = input_batch.to(device)
    target_batch = target_batch.to(device)
    logits = model(input_batch)[:, -1, :] (1)
    loss = torch.nn.functional.cross_entropy(logits, target_batch)
    return loss
```

(1) Logit dell'ultimo token di output.

Utilizziamo la funzione `calc_loss_batch()` per calcolare il valore loss per un singolo batch ottenuto dai data loader definiti in precedenza. Per calcolare il valore loss per tutti i batch di un data loader, definiamo la funzione `calc_loss_loader()` come prima.

Listato 6.9 Calcolo del valore loss per la classificazione.

```
def calc_loss_loader(data_loader, model, device, num_batches=None):
    total_loss = 0.
    if len(data_loader) == 0:
        return float("nan")
    elif num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader)) (1)
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            loss = calc_loss_batch(input_batch, target_batch, model, device)
            total_loss += loss.item()
        else:
            break
    return total_loss / num_batches
```

(1) Assicura che il numero di batch non superi i batch nel data loader.

In modo analogo al calcolo dell'accuratezza dell'addestramento, ora calcoliamo il valore loss iniziale per ciascun dataset:

```
with torch.no_grad(): (1)
    train_loss = calc_loss_loader(train_loader, model, device, num_batches=5)
    val_loss = calc_loss_loader(val_loader, model, device, num_batches=5)
    test_loss = calc_loss_loader(test_loader, model, device, num_batches=5)

print(f"Training loss: {train_loss:.3f}")
print(f"Validation loss: {val_loss:.3f}")
print(f"Test loss: {test_loss:.3f}")
```

(1) Disabilita il tracciamento del gradiente per motivi di efficienza, perché non stiamo ancora eseguendo l'addestramento.

I valori di loss iniziali sono i seguenti:

```
Training loss: 2.453
Validation loss: 2.583
Test loss: 2.322
```

Ora, implementeremo una funzione di addestramento per il fine-tuning del modello, il che significa regolare il modello per minimizzare il valore loss del set di addestramento. Minimizzare il

valore loss del set di addestramento aiuterà ad aumentare l’accuratezza della classificazione, che è il nostro obiettivo generale.

Fine-tuning del modello su dati supervisionati

Dobbiamo definire e usare la funzione di addestramento per il fine-tuning del modello LLM pre-addestrato e migliorare la sua accuratezza nella classificazione dello spam. Il ciclo di addestramento, rappresentato nella Figura 6.15, è lo stesso che abbiamo usato per il pre-addestramento; l’unica differenza è che per valutare il modello calcoliamo l’accuratezza della classificazione anziché generare un testo campione.

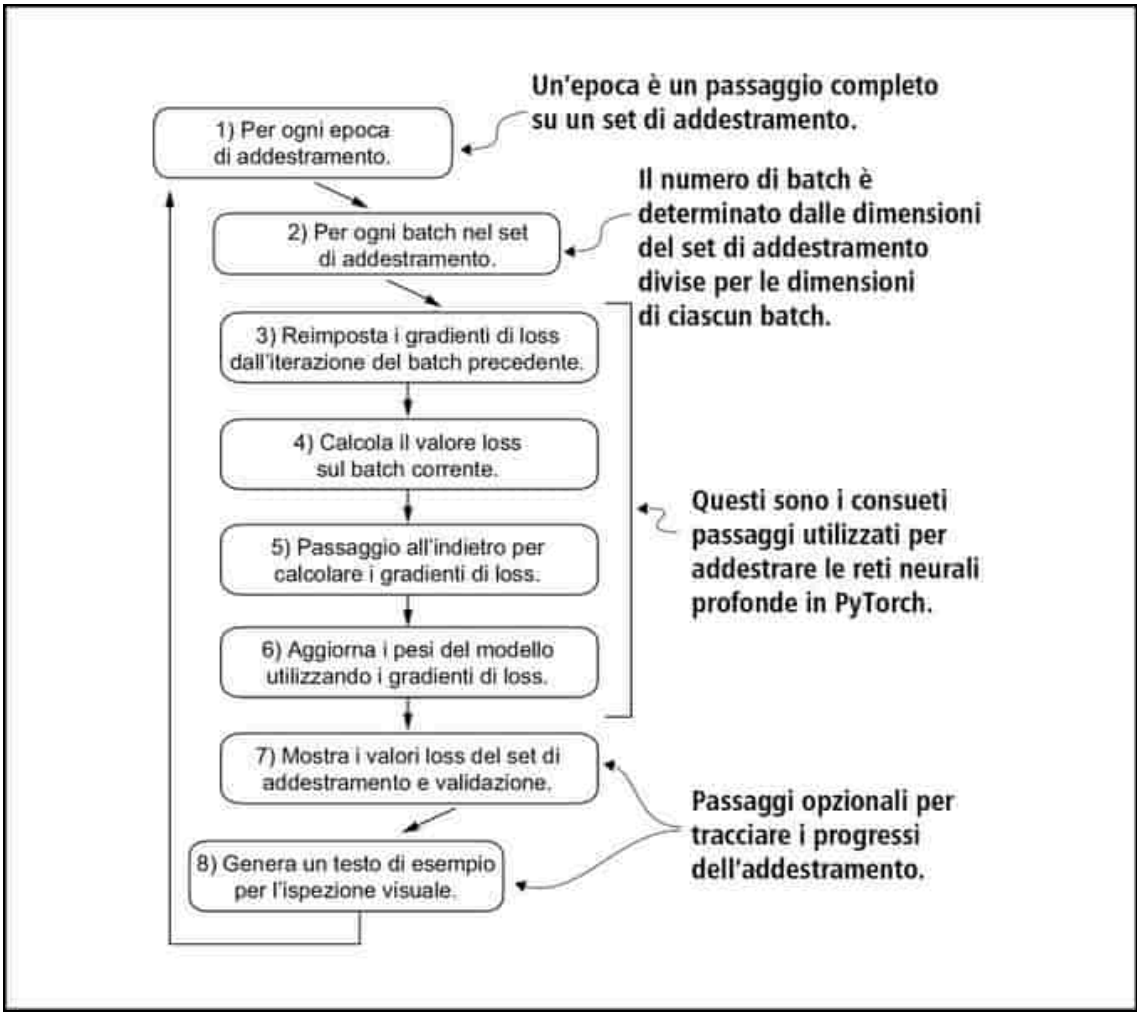


Figura 6.15 Un tipico ciclo per l'addestramento delle reti neurali profonde in PyTorch è costituito da diversi passi, che iterano sui batch del set di addestramento per diverse epoche. A ogni ciclo, calcoliamo il valore loss per ogni batch del set di addestramento, per determinare i gradienti del valore loss, che utilizziamo per aggiornare i pesi del modello con lo scopo di minimizzare il valore loss del set di addestramento.

La funzione di addestramento che implementa i concetti mostrati nella Figura 6.15 rispecchia da vicino la funzione `train_model_simple()` utilizzata per il pre-addestramento del modello. Le uniche due differenze sono che ora tracciamo il numero di esempi di addestramento visti (`examples_seen`) invece del numero di token, e che

ora calcoliamo l’accuratezza dopo ogni epoca invece di mostrare un testo campione.

Listato 6.10 Fine-tuning del modello per classificare lo spam.

```
def train_classifier_simple(model, train_loader, val_loader, optimizer, device,
num_epochs,
                            eval_freq, eval_iter):
    train_losses, val_losses, train_accs, val_accs = [], [], [], [] (1)
    examples_seen, global_step = 0, -1

    for epoch in range(num_epochs): (2)
        model.train() (3)

        for input_batch, target_batch in train_loader:
            optimizer.zero_grad() (4)
            loss = calc_loss_batch(input_batch, target_batch, model, device)
            loss.backward() (5)
            optimizer.step() (6)
            examples_seen += input_batch.shape[0] (7)
            global_step += 1

            if global_step % eval_freq == 0: (8)
                train_loss, val_loss = evaluate_model(model, train_loader,
val_loader,
                                                      device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, "
                      f"Val loss {val_loss:.3f}")

        train_accuracy = calc_accuracy_loader(train_loader, model, device, (9)
                                              num_batches=eval_iter)
        val_accuracy = calc_accuracy_loader(val_loader, model, device,
                                            num_batches=eval_iter)

        print(f"Training accuracy: {train_accuracy*100:.2f}% | ", end="")
        print(f"Validation accuracy: {val_accuracy*100:.2f}%")
        train_accs.append(train_accuracy)
        val_accs.append(val_accuracy)

    return train_losses, val_losses, train_accs, val_accs, examples_seen
```

(1) Inizializza le liste per tracciare i valori loss e gli esempi visti.

(2) Ciclo di addestramento principale.

(3) Imposta il modello in modalità di addestramento.

(4) Reimposta i gradienti di loss dall’iterazione del batch precedente.

(5) Calcola i gradienti di loss.

(6) Aggiorna i pesi del modello utilizzando i gradienti di loss.

(7) Nuovo: traccia gli esempi anziché i token.

(8) Fase di valutazione facoltativa.

(9) Calcola l'accuratezza dopo ogni epoca.

La funzione `evaluate_model()` è identica a quella che abbiamo utilizzato per il pre-addestramento:

```
def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with torch.no_grad():
        train_loss = calc_loss_loader(train_loader, model, device,
num_batches=eval_iter)
        val_loss = calc_loss_loader(val_loader, model, device,
num_batches=eval_iter)
    model.train()
    return train_loss, val_loss
```

Ora, inizializziamo l'ottimizzatore, impostiamo il numero di epoche di addestramento e avviamo l'addestramento utilizzando la funzione `train_classifier_simple()`. L'addestramento dura circa 6 minuti su un computer portatile M3 MacBook Air e meno di mezzo minuto su una GPU V100 o A100:

```
import time

start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5, weight_decay=0.1)
num_epochs = 5

train_losses, val_losses, train_accs, val_accs, examples_seen = \
    train_classifier_simple(model, train_loader, val_loader, optimizer, device,
                            num_epochs=num_epochs, eval_freq=50, eval_iter=5)

end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Training completed in {execution_time_minutes:.2f} minutes.")
```

L'output durante l'addestramento è il seguente:

```
Ep 1 (Step 000000): Train loss 2.153, Val loss 2.392
Ep 1 (Step 000050): Train loss 0.617, Val loss 0.637
Ep 1 (Step 000100): Train loss 0.523, Val loss 0.557
Training accuracy: 70.00% | Validation accuracy: 72.50%
Ep 2 (Step 000150): Train loss 0.561, Val loss 0.489
Ep 2 (Step 000200): Train loss 0.419, Val loss 0.397
Ep 2 (Step 000250): Train loss 0.409, Val loss 0.353
Training accuracy: 82.50% | Validation accuracy: 85.00%
Ep 3 (Step 000300): Train loss 0.333, Val loss 0.320
Ep 3 (Step 000350): Train loss 0.340, Val loss 0.306
Training accuracy: 90.00% | Validation accuracy: 90.00%
Ep 4 (Step 000400): Train loss 0.136, Val loss 0.200
Ep 4 (Step 000450): Train loss 0.153, Val loss 0.132
Ep 4 (Step 000500): Train loss 0.222, Val loss 0.137
```

```
Training accuracy: 100.00% | Validation accuracy: 97.50%
Ep 5 (Step 000550): Train loss 0.207, Val loss 0.143
Ep 5 (Step 000600): Train loss 0.083, Val loss 0.074
Training accuracy: 100.00% | Validation accuracy: 97.50%
Training completed in 5.65 minutes.
```

Ora utilizziamo Matplotlib per tracciare la funzione `loss` per i set di addestramento e validazione.

Listato 6.11 Tracciamento del valore loss della classificazione.

```
import matplotlib.pyplot as plt

def plot_values(epochs_seen, examples_seen, train_values, val_values,
label="loss"):
    fig, ax1 = plt.subplots(figsize=(5, 3))

    ax1.plot(epochs_seen, train_values, label=f"Training {label}") (1)
    ax1.plot(epochs_seen, val_values, linestyle="-.", label=f"Validation {label}")
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel(label.capitalize())
    ax1.legend()

    ax2 = ax1.twiny() (2)
    ax2.plot(examples_seen, train_values, alpha=0) (3)
    ax2.set_xlabel("Examples seen")

    fig.tight_layout() (4)
    plt.savefig(f"{label}-plot.pdf")
    plt.show()

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_losses))

plot_values(epochs_tensor, examples_seen_tensor, train_losses, val_losses)
```

(1) Traccia il valore loss di addestramento e validazione rispetto alle epoche.

(2) Crea un secondo asse x per gli esempi visti.

(3) Invisibile, per allineare i tick.

(4) Regola il layout per fare spazio.

La Figura 6.16 traccia le curve risultanti.

Come potete vedere dalla forte pendenza negativa nella Figura 6.16, il modello apprende bene dai dati di addestramento e non vi sono quasi indicazioni di overfitting; vale a dire, non vi è alcun divario evidente tra i valori loss del set di addestramento e del set di validazione.

Scelta del numero di epoche

In precedenza, quando abbiamo avviato l'addestramento, abbiamo impostato il numero di epoche su 5. Il numero di epoche dipende dal dataset e dalla difficoltà del compito, e non esiste una soluzione o un valore universale, sebbene un numero di epoche pari a 5 sia solitamente un buon punto di partenza. Se il modello va in overfitting dopo le prime epoche rappresentate nel grafico (vedi Figura 6.16), potrebbe essere necessario ridurre il numero di epoche. Al contrario, se la linea di tendenza suggerisce che il valore loss di validazione potrebbe migliorare con un ulteriore addestramento, è necessario aumentare il numero di epoche. In questo caso concreto, cinque epoche sono un numero ragionevole, poiché non ci sono segni di overfitting precoce e il valore loss di validazione è prossimo a 0.

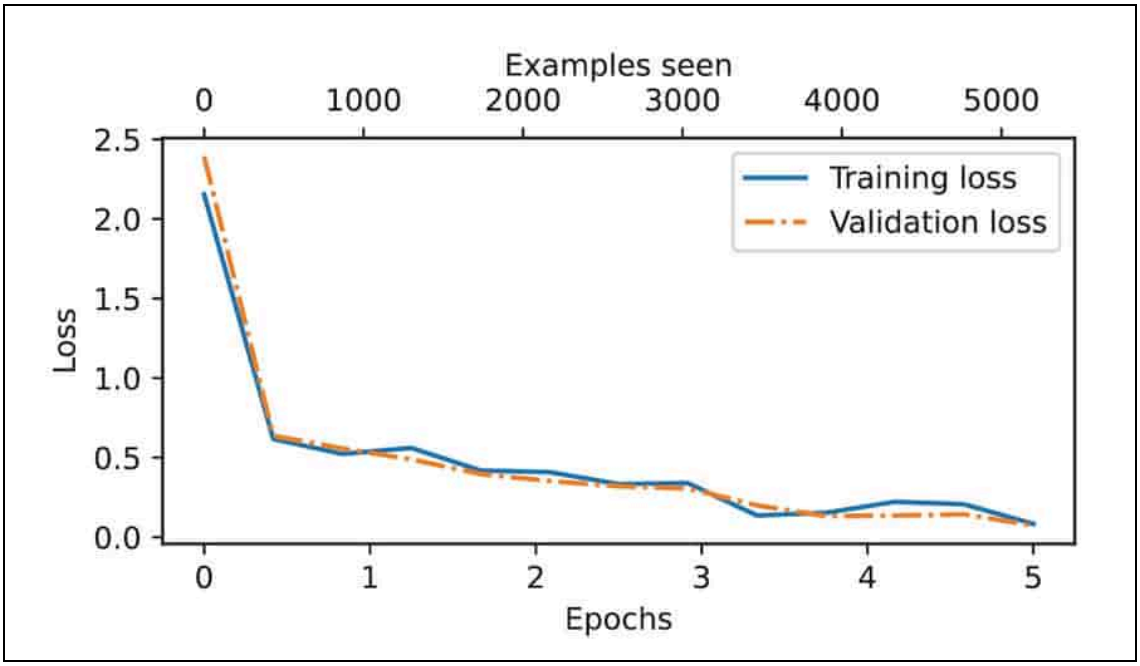


Figura 6.16 Il valore loss di addestramento e validazione del modello nelle cinque epoche di addestramento. Sia il valore loss di addestramento, rappresentato dalla linea continua, sia il valore loss di validazione, rappresentato dalla linea tratteggiata, diminuiscono bruscamente nella prima epoca e si stabilizzano gradualmente verso la quinta epoca. Questo schema indica un buon progresso di apprendimento e suggerisce che il modello abbia imparato dai dati di addestramento, generalizzando bene ai dati di validazione, sconosciuti.

Utilizzando la stessa funzione `plot_values()`, rappresentiamo graficamente le accuratezze della classificazione:

```
epochs_tensor = torch.linspace(0, num_epochs, len(train_accs))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_accs))

plot_values(epochs_tensor, examples_seen_tensor, train_accs, val_accs,
label="accuracy")
```

La Figura 6.17 rappresenta graficamente l'accuratezza risultante. Il modello raggiunge un'accuratezza di addestramento e di validazione relativamente elevata dopo le epoche 4 e 5. È importante notare che in precedenza abbiamo impostato `eval_iter=5` nell'utilizzare la funzione `train_classifier_simple()`, il che significa che le nostre stime delle prestazioni dell'addestramento e della validazione si basano solo su cinque batch, per garantire l'efficienza dell'addestramento.

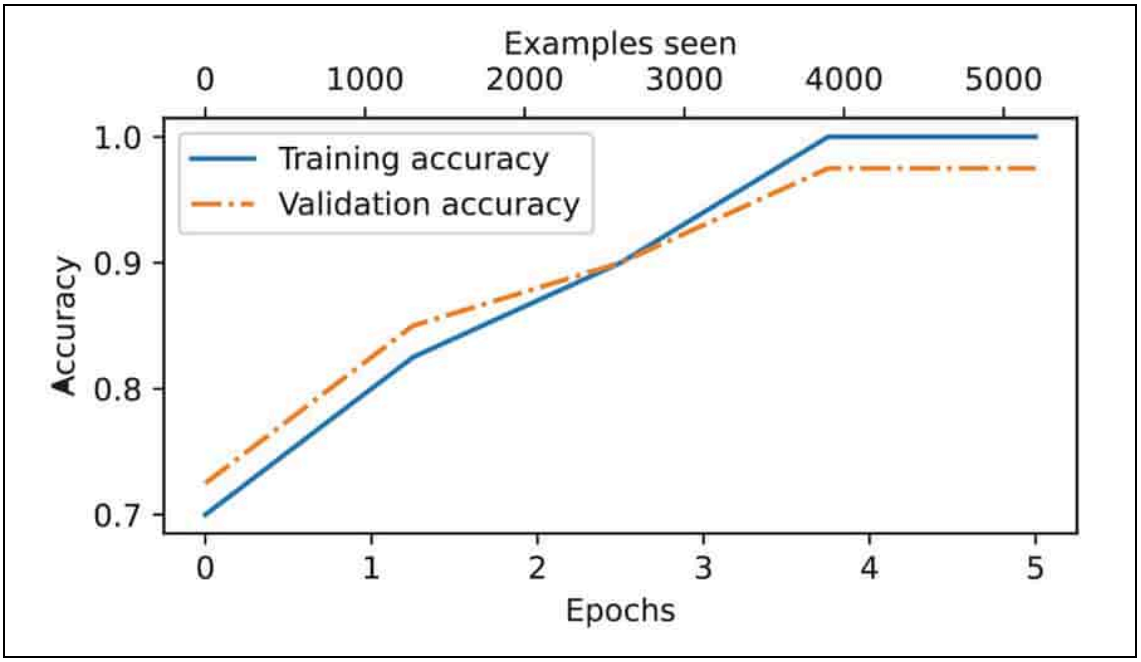


Figura 6.17 Sia l'accuratezza dell'addestramento (linea continua) sia l'accuratezza della validazione (linea tratteggiata) aumentano sostanzialmente nelle prime epoche e poi si stabilizzano, raggiungendo punteggi di accuratezza quasi perfetti: 1.0. La stretta vicinanza

delle due linee durante le epoche suggerisce che il modello non va molto in overfitting sui dati di addestramento.

Ora dobbiamo calcolare le metriche prestazionali per i set di addestramento, validazione e test nell'intero dataset eseguendo il seguente codice, questa volta senza definire il valore `eval_iter`:

```
train_accuracy = calc_accuracy_loader(train_loader, model, device)
val_accuracy = calc_accuracy_loader(val_loader, model, device)
test_accuracy = calc_accuracy_loader(test_loader, model, device)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

I valori di accuratezza risultanti sono i seguenti:

```
Training accuracy: 97.21%
Validation accuracy: 97.32%
Test accuracy: 95.67%
```

Le prestazioni sul set di addestramento e sul set di test sono pressoché identiche. La lieve discrepanza fra i set di addestramento e di test suggerisce un minimo overfitting sui dati di addestramento. In genere, l'accuratezza del set di validazione è leggermente superiore a quella del set di test, perché lo sviluppo del modello spesso comporta il fine-tuning degli iperparametri per ottenere buone prestazioni sul set di validazione, che potrebbe non generalizzare in modo efficace il set di test. Questa situazione è comune, ma il divario potrebbe essere ridotto al minimo regolando le impostazioni del modello, per esempio aumentando il tasso di dropout (`drop_rate`) o il parametro `weight_decay` nella configurazione dell'ottimizzatore.

Utilizzo del modello LLM come classificatore dello spam

Dopo aver messo a punto e valutato il modello, siamo pronti per classificare i messaggi di spam (vedi Figura 6.18) usando il modello

di classificazione dello spam basato su GPT. La seguente funzione `classify_review()` segue i passi di pre-elaborazione dei dati simili a quelli che abbiamo usato nello `SpamDataset` implementato in precedenza. Quindi, dopo aver elaborato il testo in token-ID, la funzione usa il modello per prevedere un'etichetta della classe (un numero intero), simile a quella che abbiamo implementato in precedenza, e poi restituisce il nome della classe corrispondente.

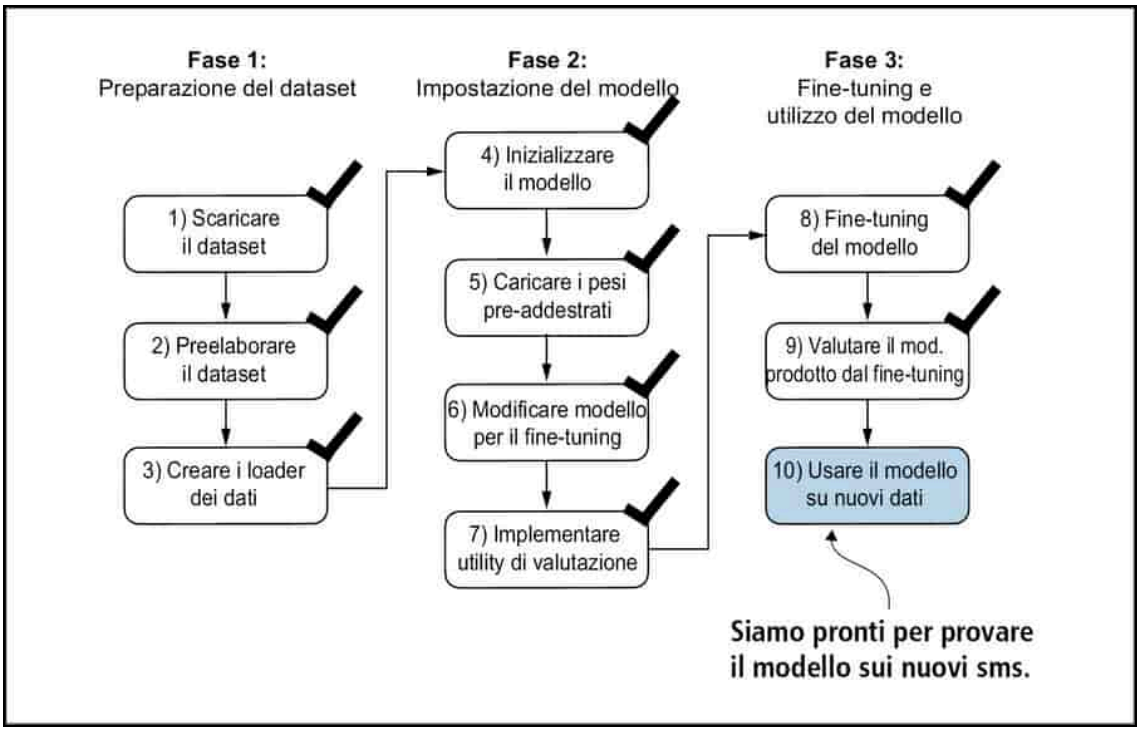


Figura 6.18 Il processo a tre fasi di fine-tuning per attività di classificazione per il modello LLM. Il Passo 10 è il passo finale della Fase 3: utilizzare il modello ottimizzato per classificare i nuovi messaggi di spam.

Listato 6.12 Utilizzo del modello per classificare nuovi messaggi.

```
def classify_review(text, model, tokenizer, device, max_length=None,
pad_token_id=50256):
    model.eval()

    input_ids = tokenizer.encode(text) (1)
    supported_context_length = model.pos_emb.weight.shape[1]
```

```
    input_ids = input_ids[:min(max_length, supported_context_length)] (2)

    input_ids += [pad_token_id] * (max_length - len(input_ids)) (3)

    input_tensor = torch.tensor(input_ids, device=device).unsqueeze(0) (4)

    with torch.no_grad(): (5)
        logits = model(input_tensor)[:, -1, :] (6)
    predicted_label = torch.argmax(logits, dim=-1).item()

    return "spam" if predicted_label == 1 else "not spam" (7)
```

(1) Prepara gli input per il modello.

(2) Tronca le sequenze se sono troppo lunghe.

(3) Padding delle sequenze alla sequenza più lunga.

(4) Aggiunge la dimensione del batch.

(5) Modella l'inferenza senza il tracciamento del gradiente.

(6) Logit dell'ultimo token di output.

(7) Restituisce il risultato della classificazione.

Proviamo questa funzione `classify_review()` su un testo campione:

```
text_1 = (
    "You are a winner you have been specially"
    " selected to receive $1000 cash or a $2000 award."
)

print(classify_review(text_1, model, tokenizer, device,
                      max_length=train_dataset.max_length))
```

Il modello risultante prevede correttamente `"spam"`. Proviamo con un altro esempio:

```
text_2 = (
    "Hey, just wanted to check if we're still on"
    " for dinner tonight? Let me know!"
)

print(classify_review(text_2, model, tokenizer, device,
                      max_length=train_dataset.max_length))
```

Ancora una volta il modello effettua una previsione corretta e restituisce l'etichetta "non spam".

Infine, salviamo il modello nel caso in cui volessimo riutilizzarlo in seguito senza doverlo riaddestrare. Possiamo usare il metodo `torch.save()`:

```
torch.save(model.state_dict(), "review_classifier.pth")
```

Una volta salvato, il modello può essere ricaricato:

```
model_state_dict = torch.load("review_classifier.pth, map_location=device")
model.load_state_dict(model_state_dict)
```

Riepilogo

- Esistono diverse strategie per il fine-tuning dei modelli LLM, tra cui il fine-tuning per attività di classificazione e il fine-tuning per rispondere a istruzioni.
- Il fine-tuning per attività di classificazione comporta la sostituzione del layer di output di un modello LLM tramite un piccolo layer di classificazione.
- Nel caso della classificazione di sms come “spam” o “non spam”, il nuovo layer di classificazione è costituito da due soli nodi di output. In precedenza, abbiamo utilizzato un numero di nodi di output pari al numero di token univoci presenti nel vocabolario (per esempio, 50256).
- Invece di prevedere il token successivo nel testo come nel pre-addestramento, il fine-tuning per attività di classificazione addestra il modello allo scopo di produrre l’etichetta di una classe, per esempio “spam” o “non spam”.
- L’input del modello per il fine-tuning è un testo convertito in token-ID, simile a quanto visto nel pre-addestramento.
- Prima di eseguire il fine-tuning di un modello LLM, carichiamo il modello pre-addestrato come modello di partenza.
- La valutazione di un modello di classificazione implica il calcolo dell’accuratezza della classificazione (la percentuale di previsioni corrette).

- Il fine-tuning di un modello di classificazione utilizza la stessa funzione loss a entropia incrociata utilizzata durante il pre-addestramento del modello LLM.

Capitolo 7

Fine-tuning per rispondere a istruzioni

Finora, abbiamo implementato l'architettura LLM, abbiamo eseguito il pre-addestramento e abbiamo importato pesi pre-addestrati da fonti esterne. Quindi, ci siamo concentrati sul fine-tuning del modello LLM per una specifica attività di classificazione: distinguere tra sms spam e non spam. Ora implementeremo il processo di fine-tuning di un modello LLM per rispondere a istruzioni, come mostra la Figura 7.1. Il fine-tuning per rispondere a istruzioni è una delle principali tecniche per lo sviluppo di modelli LLM per applicazioni chatbot, assistenti personali e altre attività conversazionali.

La Figura 7.1 mostra due modi per eseguire il fine-tuning di un modello LLM: perfezionamento di un modello LLM per attività di classificazione (Passo 8) e perfezionamento per rispondere a istruzioni (Passo 9). Abbiamo implementato il Passo 8 nel Capitolo 6. Ora perfezioneremo un modello LLM utilizzando un *dataset delle istruzioni*.

Introduzione al fine-tuning per rispondere a istruzioni

Ora sappiamo che il pre-addestramento di un modello LLM comporta una procedura di addestramento in cui esso impara a generare una parola alla volta. Il modello LLM pre-addestrato risultante è in grado di *completare un testo*, il che significa che può finire frasi o scrivere paragrafi di testo dato un frammento come input.

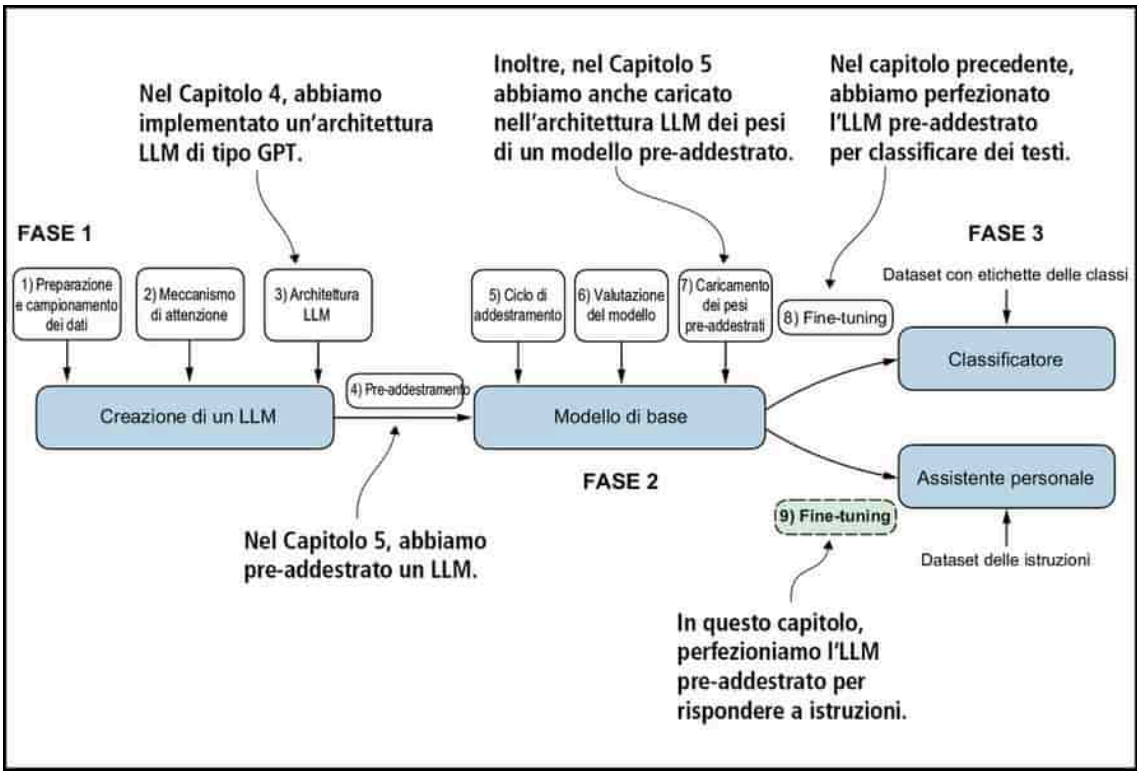


Figura 7.1 Le tre fasi principali della programmazione di un modello LLM. Questo capitolo si concentra sul Passo 9 della Fase 3: fine-tuning di un modello LLM pre-addestrato per rispondere a istruzioni.

Tuttavia, i modelli LLM pre-addestrati spesso hanno difficoltà a rispondere a istruzioni specifiche, come "Correggi la grammatica in questo testo" o "Converti questo testo in forma passiva". Più avanti,

esamineremo un esempio concreto in cui carichiamo il modello LLM pre-addestrato come base per il fine-tuning per rispondere a istruzioni, il cosiddetto *supervised instruction fine-tuning*.

Qui, ci concentriamo sul miglioramento della capacità del modello LLM di seguire istruzioni e generare la risposta desiderata, come mostra la Figura 7.2. La preparazione del dataset è un aspetto chiave di questo tipo di fine-tuning. Quindi completeremo tutti i passi delle tre fasi del processo di fine-tuning per rispondere a istruzioni, iniziando con la preparazione del dataset, come mostra la Figura 7.3.

Preparazione di un dataset per il fine-tuning

Scarichiamo e formattiamo il dataset delle istruzioni per eseguire il fine-tuning di un modello LLM pre-addestrato, per consentirgli di rispondere a istruzioni. Il dataset è composto da 1100 *coppie istruzione-risposta* simili a quelle rappresentate nella Figura 7.2. Questo dataset è stato creato appositamente per questo libro, ma il lettore interessato può trovare dataset alternativi e di pubblico dominio nell'Appendice B.

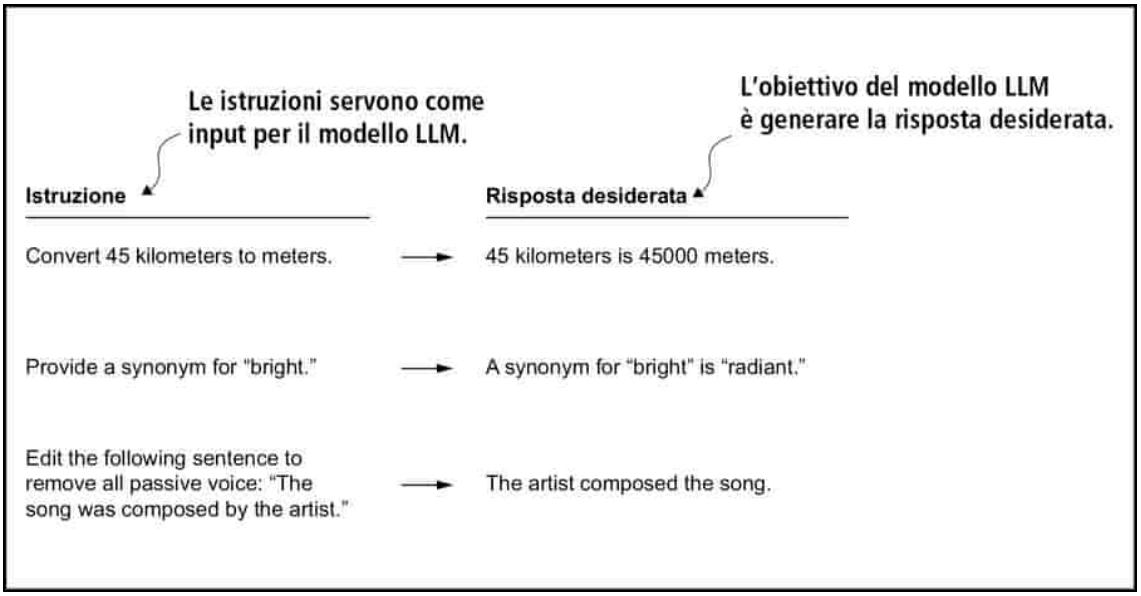


Figura 7.2 Esempi di istruzioni elaborate da un modello LLM per generare le risposte desiderate.

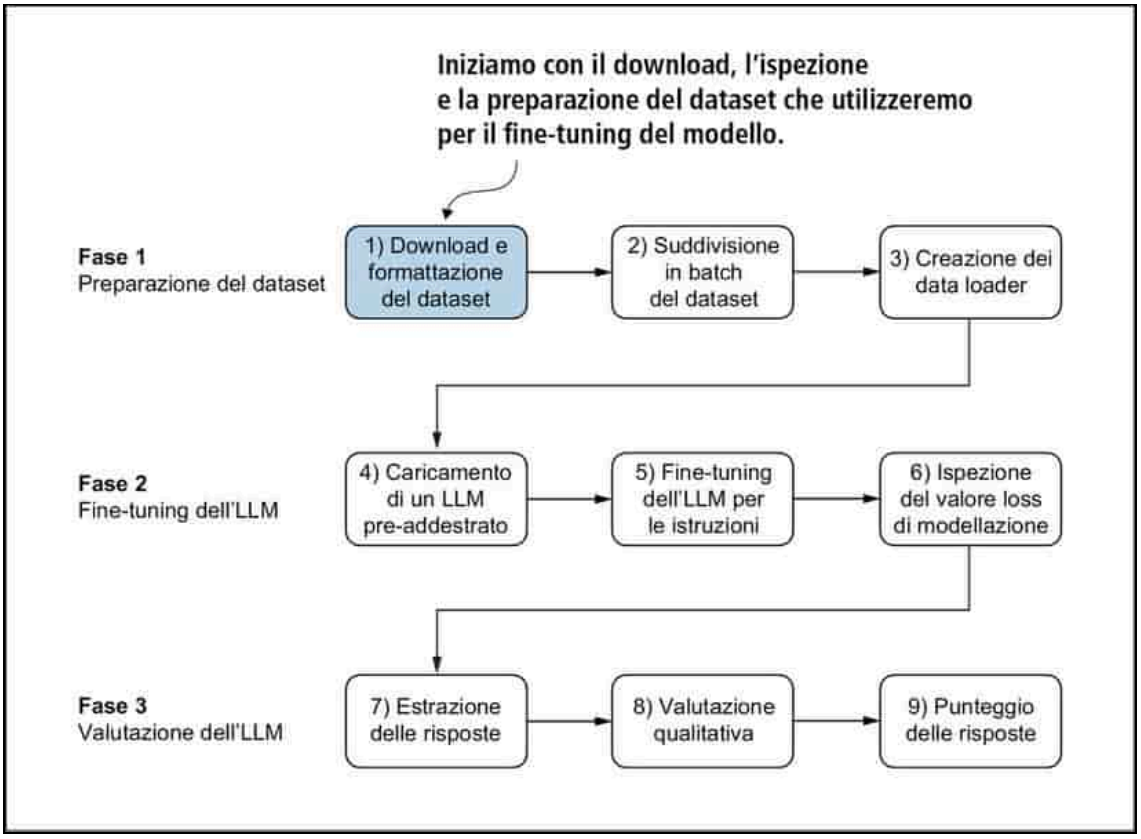


Figura 7.3 Il processo a tre fasi per il fine-tuning di un modello LLM per rispondere a istruzioni. La Fase 1 riguarda la preparazione del dataset; la Fase 2 si concentra sulla configurazione e il fine-tuning del modello e la Fase 3 riguarda la valutazione del modello. Inizieremo con il Passo 1 della Fase 1: download e formattazione del dataset.

Il seguente codice implementa ed esegue una funzione per scaricare questo dataset, che è un file relativamente piccolo (solo 204 KB) in JSON (*JavaScript Object Notation*), un formato che rispecchia la struttura dei dizionari Python, fornendo una struttura semplice per l'interscambio di dati che da un lato è leggibile e dall'altro adatto alle macchine.

Listato 7.1 Download del dataset.

```
import json
import os
import urllib
```

```
def download_and_load_file(file_path, url):
    if not os.path.exists(file_path):
        with urllib.request.urlopen(url) as response:
            text_data = response.read().decode("utf-8")
        with open(file_path, "w", encoding="utf-8") as file:
            file.write(text_data)
    else: (1)
        with open(file_path, "r", encoding="utf-8") as file:
            text_data = file.read()
    with open(file_path, "r") as file:
        data = json.load(file)
    return data

file_path = "instruction-data.json"
url = ("https://raw.githubusercontent.com/rasbt/LLMs-from-scratch"
       "/main/ch07/01_main-chapter-code/instruction-data.json"
)

data = download_and_load_file(file_path, url)
print("Number of entries:", len(data))
```

(1) Salta il download se il file è già stato scaricato.

L’output è il seguente:

```
Number of entries: 1100
```

La lista `data` che abbiamo caricato dal file JSON contiene i 1100 elementi del dataset delle istruzioni. Visualizziamo uno degli elementi per vedere come è strutturato:

```
print("Example entry:\n", data[50])
```

L’aspetto dell’elemento è il seguente:

```
Example entry:
 {'instruction': 'Identify the correct spelling of the following word.',
  'input': 'Ocassion',
  'output': "The correct spelling is 'Occasion.'"
 }
```

Come potete vedere, gli elementi sono oggetti a dizionario Python contenenti un’istruzione, un `input` e un `output`. Esaminiamo ora un altro esempio:

```
print("Another example entry:\n", data[999])
```

In base al contenuto di questo elemento, il campo `"input"` potrebbe occasionalmente essere vuoto:

```
Another example entry:
 {'instruction': "What is an antonym of 'complicated'?",
  'input': '',
  'output': "An antonym of 'complicated' is 'simple'."}
```

Il fine-tuning per rispondere a istruzioni comporta l'addestramento di un modello su un dataset in cui le coppie input-output, come quelle che abbiamo estratto dal file JSON, vengono fornite esplicitamente. Esistono vari metodi per formattare questi elementi per i modelli LLM. La Figura 7.4 illustra due diversi formati di esempi, o *stili di prompt*, utilizzati per l'addestramento di modelli LLM notevoli come Alpaca e Phi-3.

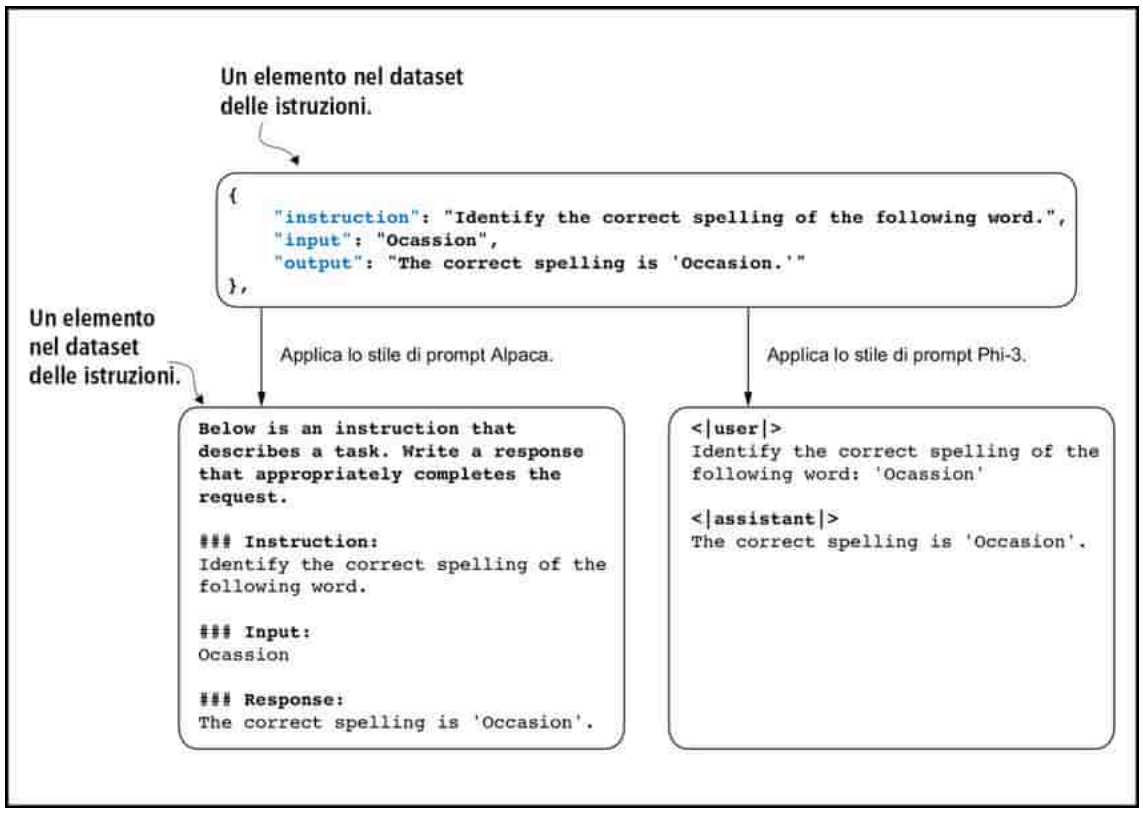


Figura 7.4 Confronto degli stili di prompt per il fine-tuning per rispondere a istruzioni nei modelli LLM. Lo stile Alpaca (a sinistra) utilizza un formato strutturato con sezioni definite per istruzione, input e risposta, mentre lo stile Phi-3 (a destra) impiega un formato più semplice con i token <|user|> e <|assistant|>.

Alpaca è stato uno dei primi modelli LLM a divulgare il suo processo di fine-tuning per rispondere a istruzioni. Phi-3, sviluppato da Microsoft, è stato incluso per dimostrare la diversità negli stili di

prompt. Il resto di questo capitolo usa lo stile di prompt Alpaca, poiché è uno dei più popolari, in gran parte perché ha contribuito a definire l'approccio al fine-tuning.

Esercizio 7.1 Cambiare stile dei prompt
Dopo aver perfezionato il modello con lo stile di prompt Alpaca, provate lo stile di prompt Phi-3 rappresentato nella Figura 7.4 e osservate se influisce sulla qualità della risposta del modello.

Definiamo una funzione `format_input()` che possiamo utilizzare per convertire gli elementi della lista `data` nel formato di input Alpaca.

Listato 7.2 Implementazione della funzione di formattazione del prompt.

```
def format_input(entry):
    instruction_text = (f"Below is an instruction that describes a task. "
                        f"Write a response that appropriately completes the
request."
                        f"\n\n### Instruction:\n{entry['instruction']}"
    )

    input_text = (f"\n\n### Input:\n{entry['input']}" if entry["input"] else "")

    return instruction_text + input_text
```

Questa funzione `format_input()` accetta come input un elemento del dizionario e costruisce una stringa formattata. Proviamola con l'elemento `data[50]` del dataset, che abbiamo esaminato in precedenza:

```
model_input = format_input(data[50])
desired_response = f"\n\n### Response:\n{data[50]['output']}"
print(model_input + desired_response)
```

L'input formattato ha il seguente aspetto:

```
Below is an instruction that describes a task. Write a response that
appropriately completes the request.

### Instruction:
Identify the correct spelling of the following word.

### Input:
Ocassion

### Response:
The correct spelling is 'Occasion.'
```

Notate che `format_input()` salta la sezione opzionale `### Input:` se il campo `'input'` è vuoto, cosa che possiamo verificare applicando la funzione `format_input()` all'elemento `data[999]` che abbiamo esaminato in precedenza:

```
model_input = format_input(data[999])
desired_response = f"\n\n### Response:\n{data[999]['output']}"
print(model_input + desired_response)
```

L'output mostra che gli elementi con un campo `'input'` vuoto non contengono una sezione `### Input:` nell'input formattato:

```
Below is an instruction that describes a task. Write a response that
appropriately completes the request.

### Instruction:
What is an antonym of 'complicated'?

### Response:
An antonym of 'complicated' is 'simple'.
```

Prima di passare alla configurazione dei data loader PyTorch, nel prossimo paragrafo, dividiamo il dataset nei set di addestramento, validazione e test analogamente a quanto abbiamo fatto col dataset di classificazione dello spam nel capitolo precedente. Il Listato 7.3 mostra come calcoliamo le porzioni.

Listato 7.3 Partizionamento del dataset.

```
train_portion = int(len(data) * 0.85) (1)
test_portion = int(len(data) * 0.1) (2)
val_portion = len(data) - train_portion - test_portion (3)

train_data = data[:train_portion]
test_data = data[train_portion:train_portion + test_portion]
val_data = data[train_portion + test_portion:]

print("Training set length:", len(train_data))
print("Validation set length:", len(val_data))
print("Test set length:", len(test_data))
```

(1) Usa l'85% dei dati per l'addestramento.

(2) Usa il 10% dei dati per i test.

(3) Il rimanente 5% per la validazione.

Questa suddivisione produce le seguenti suddivisioni del dataset:

```
Training set length: 935
Validation set length: 55
Test set length: 110
```

Dopo aver scaricato e partizionato il dataset e aver acquisito una chiara comprensione della formattazione del prompt del dataset, siamo pronti per l'implementazione del processo di fine-tuning per rispondere a istruzioni. Concentriamoci inizialmente sullo sviluppo del metodo per la costruzione dei batch di addestramento per il fine-tuning del modello LLM.

Organizzazione dei dati in batch di addestramento

Avanzando nella fase di implementazione del processo di fine-tuning per rispondere a istruzioni, il passo successivo, rappresentato nella Figura 7.5, si concentra sulla costruzione dei batch di addestramento. Ciò comporta la definizione di un metodo che garantisca che il modello riceva, durante il processo di fine-tuning, i dati di addestramento formattati.

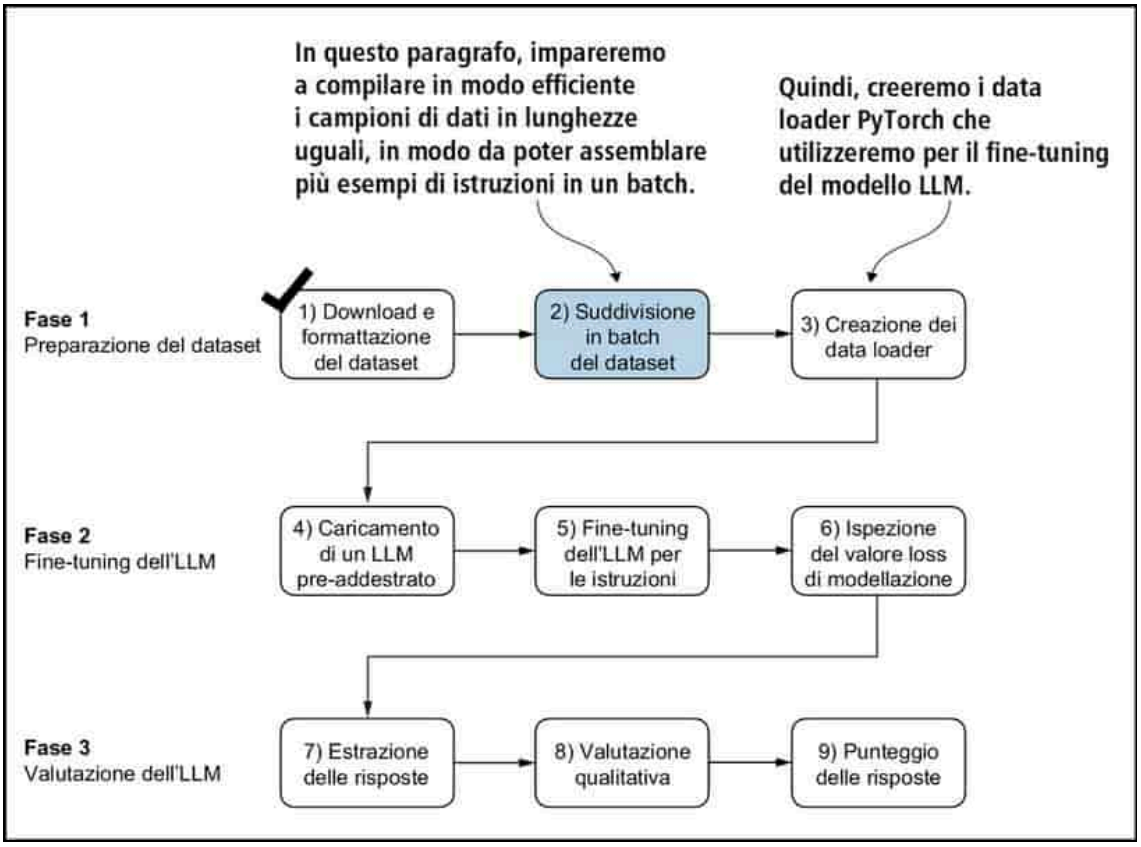


Figura 7.5 Il processo a tre fasi per il fine-tuning di un modello LLM per rispondere a istruzioni. Di seguito, esaminiamo il Passo 2 della Fase 1: assemblaggio dei batch di addestramento.

Nel capitolo precedente, i batch di addestramento sono stati creati automaticamente dalla classe PyTorch `DataLoader`, che impiega una funzione di *collazione* di default per combinare le liste di campioni in "lotti", *batch*. La funzione di collazione prende una lista di singoli campioni di dati e li unisce in un singolo batch che può essere elaborato in modo più efficiente dal modello, durante l'addestramento.

Tuttavia, il processo di batching per il fine-tuning di un modello che deve rispondere a istruzioni è un po' più complesso e richiede la creazione di una specifica funzione di collazione che collegheremo in

seguito a `DataLoader`. Implementiamo allora questa funzione di collazione specifica.

Affrontiamo il *processo di batching* in diversi passi, inclusa la programmazione della funzione di collazione personalizzata, come mostra la Figura 7.6. Innanzitutto, per implementare i Passi 2.1 e 2.2, predisponiamo una classe `InstructionDataset` che applica `format_input()` e *pre-tokenizza* tutti gli input del dataset, in modo simile a quanto fatto da `SpamDataset` nel Capitolo 6. Questo processo in due passi, dettagliato nella Figura 7.7, è implementato nel metodo costruttore `__init__`di `InstructionDataset`.

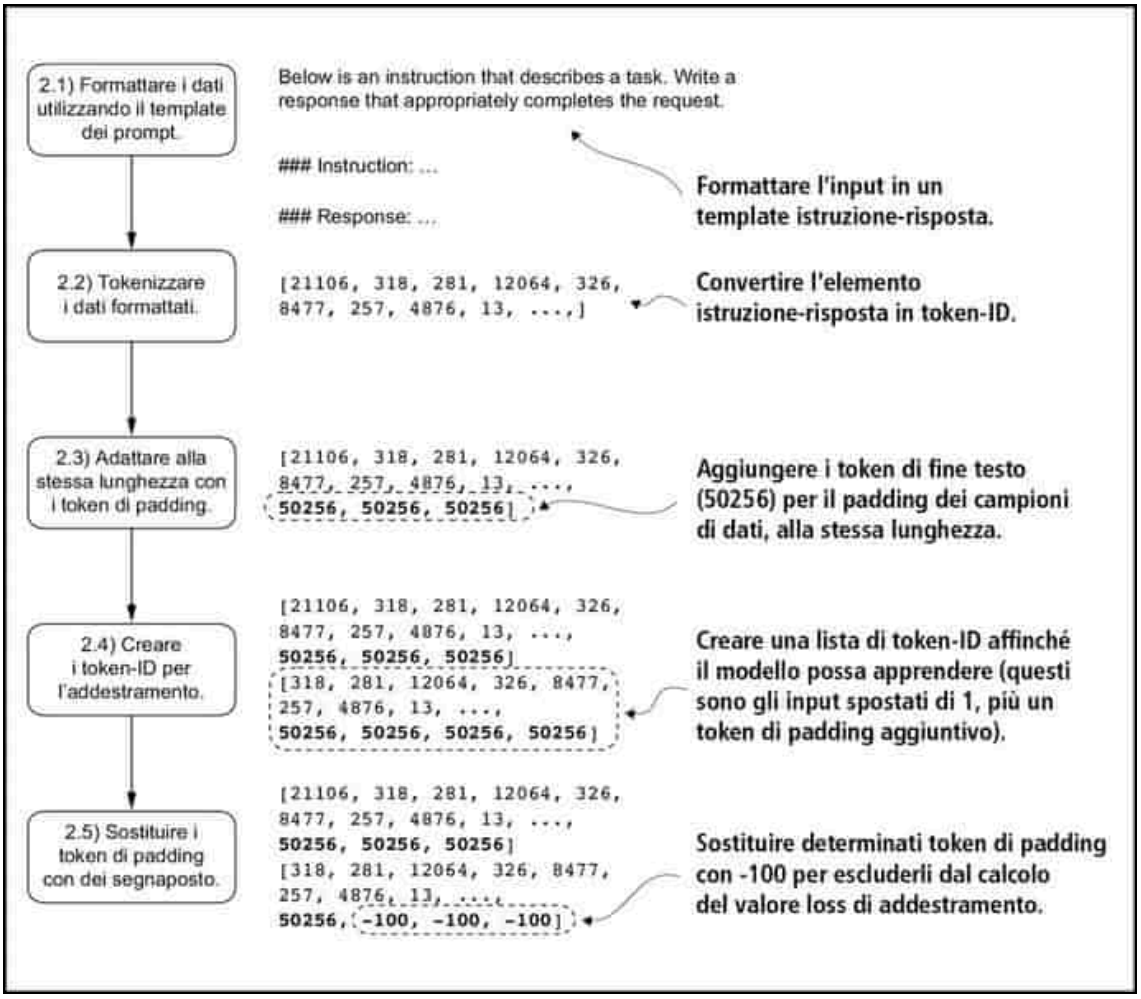


Figura 7.6 I cinque sub-passi coinvolti nell'implementazione del processo di batching: (2.1) applicazione del template dei prompt, (2.2) utilizzo della tokenizzazione dai capitoli precedenti, (2.3) aggiunta dei token di padding, (2.4) creazione dei token-ID e (2.5) sostituzione dei token segnaposto -100 per mascherare i token di padding nella funzione loss.

Listato 7.4 Implementazione di una InstructionDataset.

```
import torch
from torch.utils.data import Dataset

class InstructionDataset(Dataset):
    def __init__(self, data, tokenizer):
        self.data = data
        self.encoded_texts = []
        for entry in data: (1)
```

```
            instruction_plus_input = format_input(entry)
            response_text = f"\n\n### Response:\n{entry['output']}"
            full_text = instruction_plus_input + response_text
            self.encoded_texts.append(tokenizer.encode(full_text))

    def __getitem__(self, index):
        return self.encoded_texts[index]

    def __len__(self):
        return len(self.data)
```

(1) Pre-tokenizza i testi.

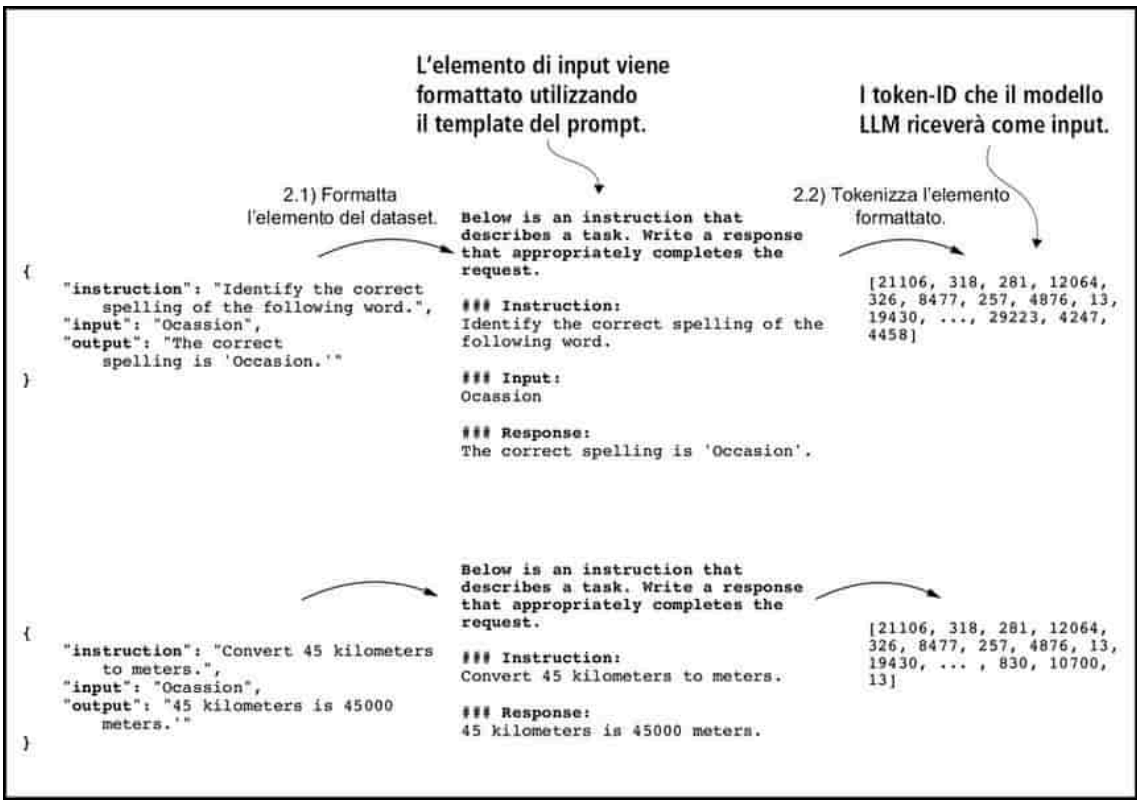


Figura 7.7 I primi due passi dell'implementazione del processo di batching. Gli elementi vengono prima formattati utilizzando uno specifico template dei prompt (2.1) e poi tokenizzati (2.2), con conseguente creazione di una sequenza di token-ID che il modello può elaborare.

In modo analogo a quanto detto per l'approccio utilizzato nell'eseguire il fine-tuning per attività di classificazione, vogliamo accelerare l'addestramento raccogliendo più esempi di addestramento in un batch, il che richiede di eseguire il padding di tutti gli input a

una lunghezza uniforme. Come per il fine-tuning per attività di classificazione, utilizziamo `<|endoftext|>` come token di padding.

Invece di aggiungere i token `<|endoftext|>` agli input di testo, possiamo aggiungere direttamente il token-ID corrispondente a `<|endoftext|>` agli input pre-tokenizzati. Possiamo usare il metodo `.encode()` del tokenizer su un token `<|endoftext|>` per ricordarci quale token-ID dovremmo usare:

```
import tiktoken
tokenizer = tiktoken.get_encoding("gpt2")
print(tokenizer.encode("<|endoftext|>", allowed_special={"<|endoftext|>"}))
```

Il token-ID risultante è `50256`.

Procedendo al Passo 2.3 del processo (vedi Figura 7.6), adottiamo un approccio più sofisticato, sviluppando una funzione di collazione personalizzata che possiamo passare al data loader. Questa funzione esegue il padding degli esempi di addestramento in ogni batch alla stessa lunghezza, consentendo al contempo a batch differenti di avere lunghezze differenti, come mostra la Figura 7.8. Questo approccio riduce al minimo il padding non necessario, estendendo le sequenze in modo che corrispondano a quella più lunga di ogni batch, non dell’intero dataset.

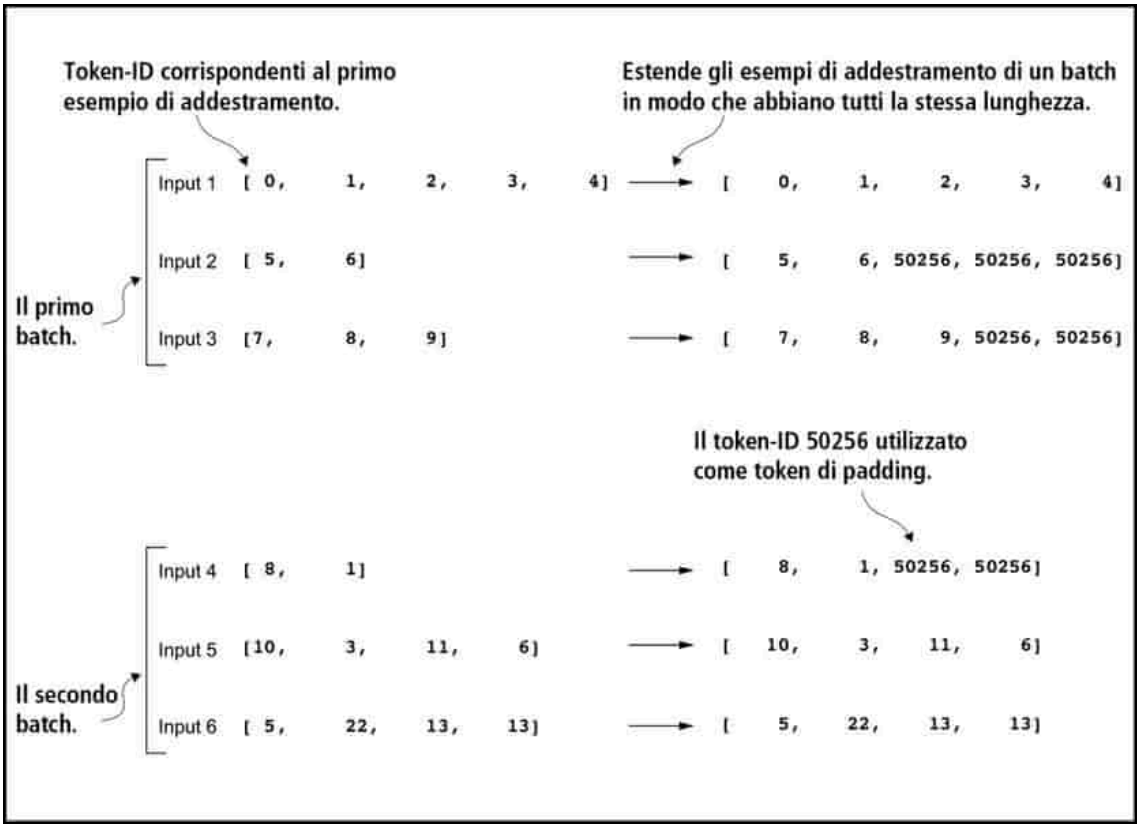


Figura 7.8 Il padding degli esempi di addestramento in batch usando il token-ID 50256 per garantire una lunghezza uniforme all'interno di ogni batch. Ogni batch può avere lunghezze differenti, come mostrato dal primo e dal secondo.

Possiamo implementare il processo di padding con una funzione di collazione personalizzata:

```
def custom_collate_draft_1(batch,
                           pad_token_id=50256,
                           device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch) (1)
    inputs_lst = []

    for item in batch: (2)
        new_item = item.copy()
        new_item += [pad_token_id]

    padded = (new_item + [pad_token_id] * (batch_max_length - len(new_item)))
    inputs = torch.tensor(padded[:-1]) (3)
    inputs_lst.append(inputs)

    inputs_tensor = torch.stack(inputs_lst).to(device) (4)
    return inputs_tensor
```

(1) Trova la sequenza più lunga nel batch.
(2) Esegue il padding e prepara gli input.
(3) Rimuove il token di padding extra aggiunto in precedenza.
(4) Converte la lista di input in un tensore e lo trasferisce al dispositivo di destinazione.

La funzione `custom_collate_draft_1()` che abbiamo implementato è progettata per essere integrata in un `DataLoader` PyTorch, ma può anche funzionare come uno strumento autonomo. Qui, la usiamo in modo indipendente per verificare che si comporti come previsto. Proviamola su tre diversi input che vogliamo assemblare in un batch, dove ogni esempio viene esteso (col padding) alla stessa lunghezza:

```
inputs_1 = [0, 1, 2, 3, 4]
inputs_2 = [5, 6]
inputs_3 = [7, 8, 9]
batch = (inputs_1,
         inputs_2,
         inputs_3
)
print(custom_collate_draft_1(batch))
```

Il batch risultante ha il seguente aspetto:

```
tensor([[    0,     1,     2,     3,     4],
        [    5,     6, 50256, 50256, 50256],
        [    7,     8,     9, 50256, 50256]]
)
```

Questo output mostra che tutti gli input sono stati portati alla lunghezza della lista di input più lunga, `inputs_1`, contenente cinque token-ID.

Abbiamo implementato la funzione di collazione per creare batch da liste di input. Tuttavia, come abbiamo appreso in precedenza, dobbiamo anche creare i batch con i token-ID target corrispondenti al batch dei token-ID di input. Questi ID target, come mostra la Figura 7.9, sono cruciali, perché rappresentano ciò che vogliamo che il modello generi e ciò di cui abbiamo bisogno durante l’addestramento per calcolare il valore loss per gli aggiornamenti dei pesi. Vale a dire,

modifichiamo la nostra funzione di collazione personalizzata per restituire i token-ID target oltre ai token-ID di input.

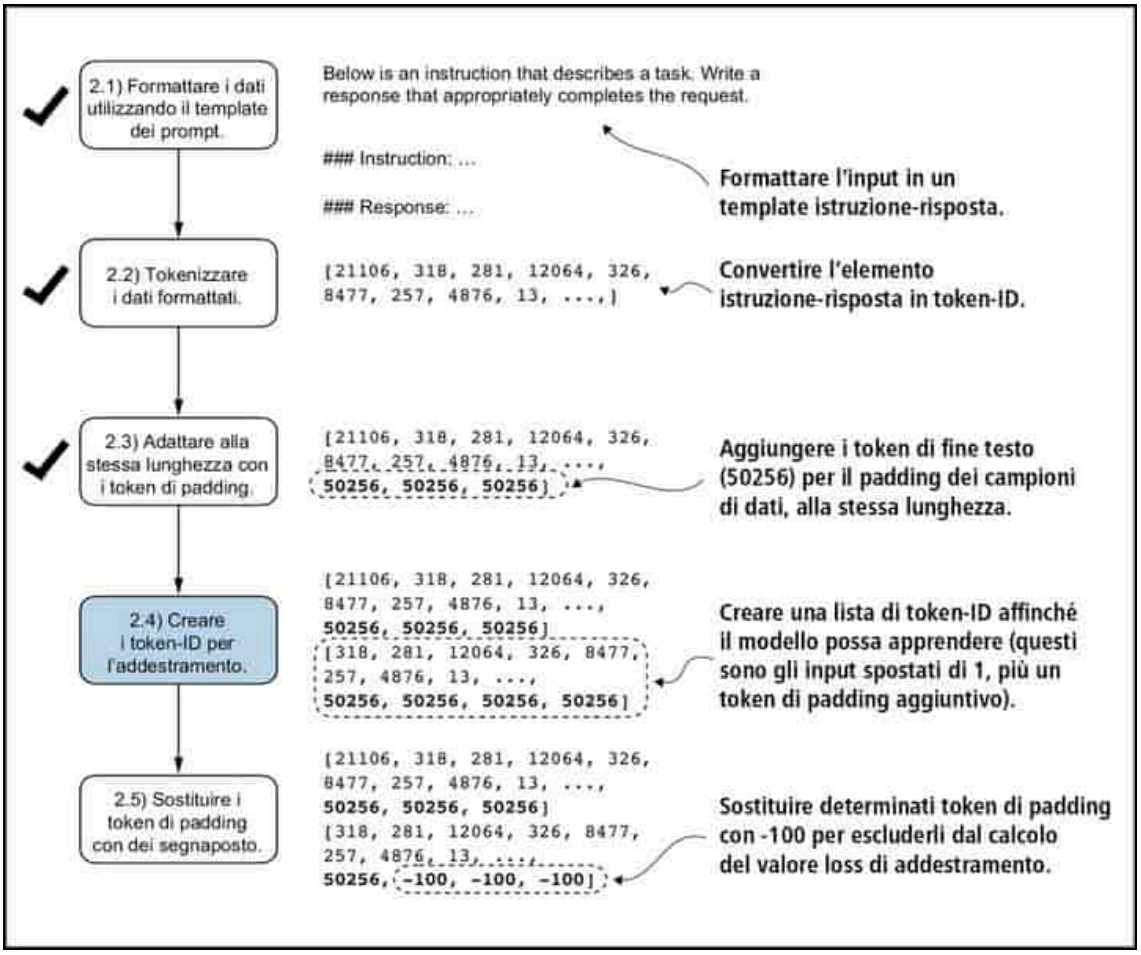


Figura 7.9 I cinque sub-passi coinvolti nell'implementazione del processo di batching. Ora ci stiamo concentrando sul Passo 2.4, la creazione dei token-ID target. Questo passo è essenziale, in quanto consente al modello di apprendere e prevedere i token che deve generare.

Come abbiamo fatto quando abbiamo pre-addestrato un modello LLM, i token-ID target corrispondono ai token-ID di input ma spostati di una posizione a destra. Questa configurazione, come mostra la Figura 7.10, consente al modello LLM di imparare a predire il token successivo in una sequenza.

La seguente funzione di collazione aggiornata genera i token-ID target dai token-ID di input:

```
def custom_collate_draft_2(batch,
                           pad_token_id=50256,
                           device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst, targets_lst = [], []

    for item in batch:
        new_item = item.copy()
        new_item += [pad_token_id]
        padded = (new_item + [pad_token_id] * (batch_max_length - len(new_item)))
        inputs = torch.tensor(padded[:-1]) (1)
        targets = torch.tensor(padded[1:]) (2)
        inputs_lst.append(inputs)
        targets_lst.append(targets)

    inputs_tensor = torch.stack(inputs_lst).to(device)
    targets_tensor = torch.stack(targets_lst).to(device)
    return inputs_tensor, targets_tensor

inputs, targets = custom_collate_draft_2(batch)
print(inputs)
print(targets)
```

(1) Tronca l’ultimo token degli input.

(2) Sposta di una posizione a destra i target.

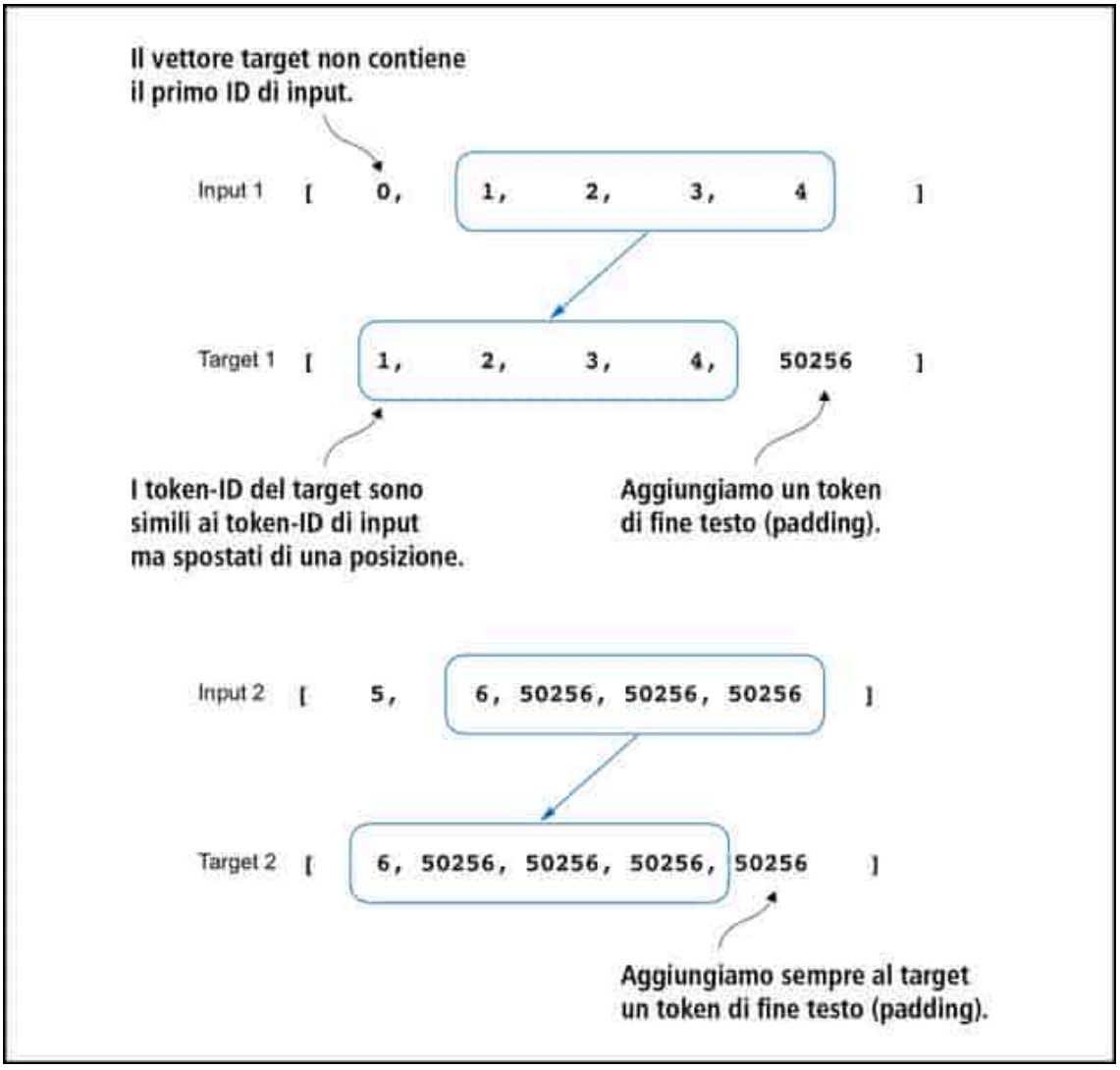


Figura 7.10 L'allineamento dei token di input e target utilizzato nel processo di fine-tuning di un modello LLM per rispondere a istruzioni. Per ogni sequenza di input, la sequenza target corrispondente viene creata spostando i token-ID di una posizione a destra, omettendo il primo token dell'input e aggiungendo un token di fine testo.

Applicata al `batch` di esempio costituito da tre liste di input definite in precedenza, la nuova funzione `custom_collate_draft_2()` restituisce i batch di input e target:

```
tensor([[    0,     1,     2,     3,     4], (1)
        [    5,     6, 50256, 50256, 50256],
        [    7,     8,     9, 50256, 50256]])
tensor([[    1,     2,     3,     4, 50256], (2)
```

```
[    6, 50256, 50256, 50256, 50256],
[    8,     9, 50256, 50256, 50256]])
```

(1) Il primo tensore rappresenta gli input.

(2) Il secondo tensore rappresenta i target.

Nel passo successivo, assegniamo il valore segnaposto `-100` a tutti i token di padding, come evidenziato nella Figura 7.11. Questo valore speciale ci consente di evitare che questi token di padding contribuiscano al calcolo del valore loss di addestramento, assicurando che solo i dati significativi influenzino l'apprendimento del modello. Discuteremo questo processo in modo più dettagliato dopo aver implementato questa modifica (quando abbiamo eseguito il fine-tuning per l'attività di classificazione, non abbiamo dovuto preoccuparcene, poiché abbiamo addestrato il modello solo in base all'ultimo token di output).

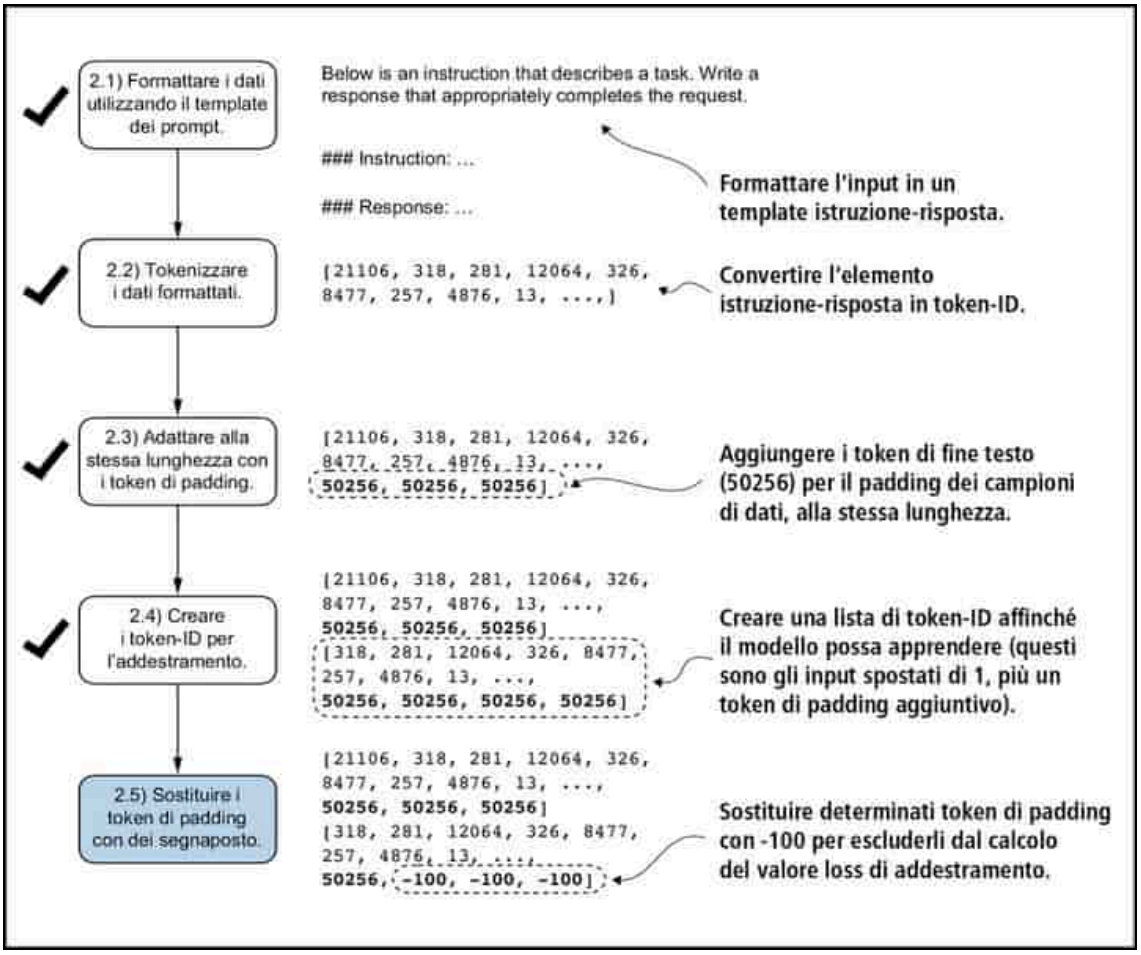


Figura 7.11 I cinque sub-passi coinvolti nell'implementazione del processo di batching. Dopo aver creato la sequenza target spostando i token-ID di una posizione a destra e aggiungendo un token di fine testo, nel Passo 2.5 sostituiamo i token di padding di fine testo con un valore segnaposto (-100).

Tuttavia, notate che manteniamo un token di fine testo, ID `50256`, nella lista target, come mostra la Figura 7.12. Ciò consente al modello LLM di scoprire quando generare un token di fine testo in risposta alle istruzioni, che utilizziamo come indicatore che la risposta generata è completa.

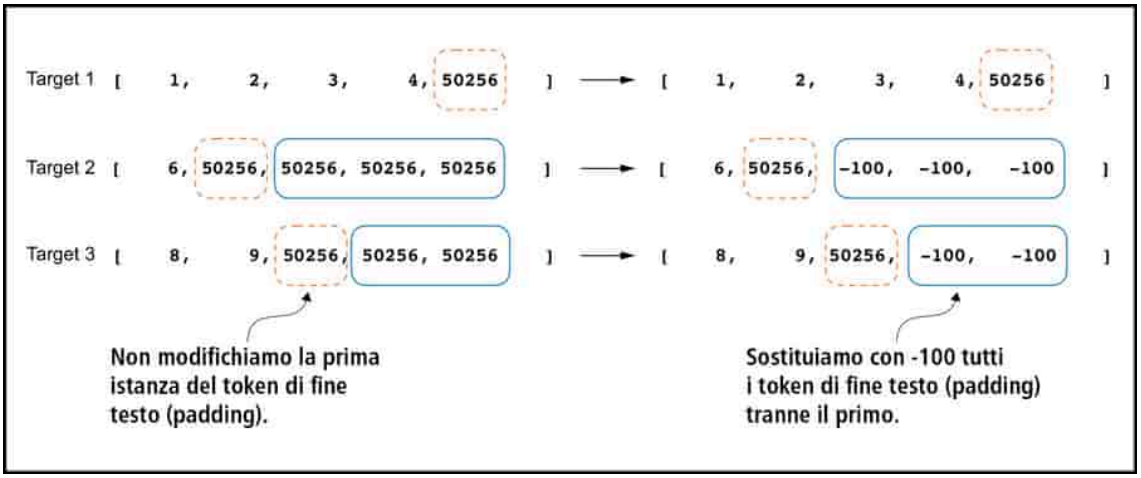


Figura 7.12 La Fase 2.4 nel processo di sostituzione del token nel batch target per la preparazione dei dati di addestramento. Sostituiamo tutti tranne la prima istanza del token di fine testo, che utilizziamo come padding, col valore segnaposto -100, mantenendo il token di fine testo iniziale in ogni sequenza target.

Nel Listato 7.5, modifichiamo la funzione di collazione personalizzata per sostituire i token aventi ID `50256` con `-100` nelle liste target. Inoltre, introduciamo un parametro `allowed_max_length` per limitare facoltativamente la lunghezza dei campioni. Questa modifica sarà utile se prevedete di lavorare con dataset che superano le dimensioni del contesto (1024 token) supportate dal modello GPT-2.

Listato 7.5 Implementazione di una funzione di collazione batch personalizzata.

```
def custom_collate_fn(batch,
                      pad_token_id=50256,
                      ignore_index=-100,
                      allowed_max_length=None,
                      device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst, targets_lst = [], []

    for item in batch:
        new_item = item.copy()
        new_item += [pad_token_id]

        padded = (new_item + [pad_token_id] * (batch_max_length - len(new_item)))
(1)
        inputs = torch.tensor(padded[:-1]) (2)
```

```
        targets = torch.tensor(padded[1:]) (3)

        mask = targets == pad_token_id          (4)
        indices = torch.nonzero(mask).squeeze() (4)
        if indices.numel() > 1:                 (4)
            targets[indices[1:]] = ignore_index (4)

        if allowed_max_length is not None:         (5)
            inputs = inputs[:allowed_max_length]   (5)
            targets = targets[:allowed_max_length] (5)

        inputs_lst.append(inputs)
        targets_lst.append(targets)

    inputs_tensor = torch.stack(inputs_lst).to(device)
    targets_tensor = torch.stack(targets_lst).to(device)
    return inputs_tensor, targets_tensor
```

(1) Padding delle sequenze a `max_length`.

(2) Tronca l'ultimo token per gli input.

(3) Sposta +1 a destra per i target.

(4) Nei target sostituisce tutti i token di padding tranne il primo con `ignore_index`.

(5) Tronca facoltativamente alla lunghezza massima della sequenza.

Proviamo di nuovo la funzione di collazione sul batch di esempio che abbiamo creato in precedenza, per verificare che si comporti come previsto:

```
inputs, targets = custom_collate_fn(batch)
print(inputs)
print(targets)
```

I risultati sono i seguenti, dove il primo tensore rappresenta gli input e il secondo tensore rappresenta i target:

```
tensor([[    0,     1,     2,     3,     4],
        [    5,     6, 50256, 50256, 50256],
        [    7,     8,     9, 50256, 50256]])
tensor([[    1,     2,     3,     4, 50256],
        [    6, 50256,  -100,  -100,  -100],
        [    8,     9, 50256,  -100,  -100]])
```

La funzione di collazione modificata si comporta come previsto, modificando la lista target inserendo il token-ID `-100`. Qual è il significato di questa modifica?

A scopo dimostrativo, considerate il seguente esempio, in cui ogni logit di output corrisponde a un potenziale token del vocabolario del modello. Ecco come potremmo calcolare il valore loss a entropia incrociata (introdotto nel Capitolo 5) durante l'addestramento quando il modello prevede una sequenza di token, che è simile a ciò che abbiamo fatto quando abbiamo pre-addestrato il modello e ne abbiamo eseguito il fine-tuning per attività di classificazione:

```
logits_1 = torch.tensor([[-1.0, 1.0], (1)
                         [-0.5, 1.5]] (2)
)
targets_1 = torch.tensor([0, 1]) # Correct token indices to generate
loss_1 = torch.nn.functional.cross_entropy(logits_1, targets_1)
print(loss_1)
```

(1) Predizioni per il primo token.

(2) Predizioni per il secondo token.

Il valore loss calcolato dal codice precedente è `1.1269`:

```
tensor(1.1269)
```

Come ci aspettiamo, l'aggiunta di un ulteriore token-ID influisce sul calcolo del valore loss:

```
logits_2 = torch.tensor([[-1.0, 1.0],
                         [-0.5, 1.5],
                         [-0.5, 1.5]] (1)
)
targets_2 = torch.tensor([0, 1, 1])
loss_2 = torch.nn.functional.cross_entropy(logits_2, targets_2)
print(loss_2)
```

(1) Nuovo terzo token.

Dopo aver aggiunto il terzo token, il valore loss è `0.7936`.

Finora abbiamo eseguito alcuni semplici calcoli più o meno ovvi utilizzando la funzione loss a entropia incrociata di PyTorch, la stessa funzione loss che abbiamo utilizzato nelle funzioni per il pre-addestramento e il fine-tuning per attività di classificazione. Ora passiamo alla parte interessante, e vediamo che cosa succede se sostituiamo il terzo token-ID target con `-100`:

```
targets_3 = torch.tensor([0, 1, -100])
loss_3 = torch.nn.functional.cross_entropy(logits_2, targets_3)
```

```
print(loss_3)
print("loss_1 == loss_3:", loss_1 == loss_3)
```

L’output risultante è il seguente:

```
tensor(1.1269)
loss_1 == loss_3: tensor(True)
```

Il valore loss risultante in questi tre esempi di addestramento è identica al valore loss che abbiamo calcolato in precedenza dai due esempi di addestramento. In altre parole, la funzione loss a entropia incrociata ha ignorato il terzo elemento nel vettore `targets_3`, il token-ID corrispondente a `-100` (il lettore interessato può provare a sostituire il valore `-100` con un altro token-ID che non sia `0` o `1`, ma ciò causerà un errore).

Quindi che cosa c’è di così speciale in `-100` da essere ignorato dal valore loss a entropia incrociata? L’impostazione di default della funzione entropia incrociata in PyTorch è `cross_entropy(..., ignore_index=-100)`. Ciò significa che ignora i target etichettati con `-100`. Sfruttiamo questo `ignore_index` per ignorare i token di fine testo (padding) aggiuntivi che abbiamo utilizzato per riempire gli esempi di addestramento in modo che abbiano la stessa lunghezza in ogni batch. Tuttavia, vogliamo mantenere un token-ID `50256` (end-of-text) nei target, perché aiuta il modello LLM a imparare a generare token di fine testo, che possiamo utilizzare come indicatore del completamento di una risposta.

Oltre a mascherare i token di padding, è pratica comune anche mascherare i token-ID target che corrispondono all’istruzione, come mostra la Figura 7.13. Mascherando i token-ID target del modello LLM corrispondenti all’istruzione, il valore loss a entropia incrociata viene calcolato solo per gli ID dei target di risposta generati. Pertanto, il modello viene addestrato a concentrarsi sulla generazione di risposte accurate anziché sulla memorizzazione di istruzioni, il che può aiutare a ridurre l’overfitting.

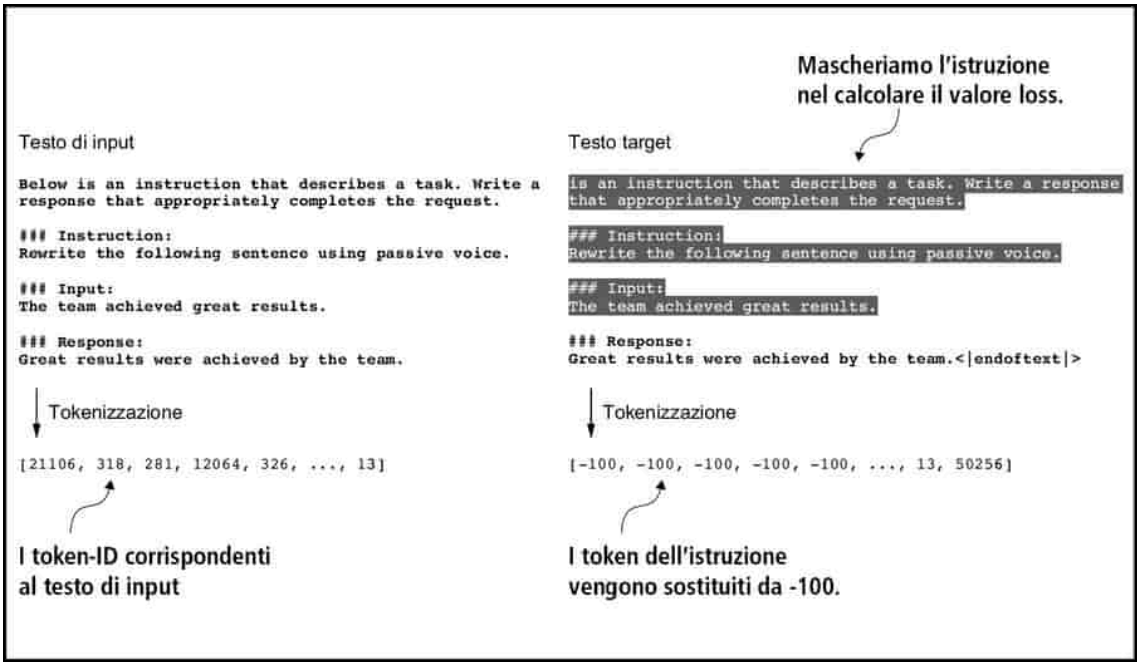


Figura 7.13 A sinistra: il testo di input formattato che tokenizziamo e poi forniamo al modello LLM durante l'addestramento. A destra: il testo target che prepariamo per il modello LLM, dove possiamo facoltativamente mascherare la sezione di istruzioni, il che significa sostituire i token-ID corrispondenti con il valore -100, ignore_index.

Al momento della stesura, i ricercatori sono divisi sul fatto che mascherare le istruzioni sia universalmente vantaggioso durante il fine-tuning per rispondere a istruzioni. Per esempio, il documento del 2024 di Shi et al., *Instruction Tuning With Loss Over Instructions* (`https://arxiv.org/abs/2405.14394`), ha dimostrato che non mascherando le istruzioni migliorano le prestazioni del modello LLM (vedi l'Appendice B per maggiori dettagli). Qui, non applicheremo il mascheramento e lo lasceremo come esercizio facoltativo per i lettori interessati.

Esercizio 7.2 Mascheramento delle istruzioni e degli input

Dopo aver completato il capitolo e aver perfezionato il modello con `InstructionDataset`, sostituite i token delle istruzioni e di input con la maschera `-100`

per utilizzare il metodo di mascheramento delle istruzioni rappresentato nella Figura 7.13. Quindi valutate se ciò ha un effetto positivo sulle prestazioni del modello.

Creazione di data loader per un dataset delle istruzioni

Abbiamo completato diverse fasi per implementare una classe `InstructionDataset` e una funzione `custom_collate_fn()` per il dataset delle istruzioni. Come mostra la Figura 7.14, siamo pronti a raccogliere i frutti del nostro lavoro inserendo sia gli oggetti `InstructionDataset` sia la funzione `custom_collate_fn()` nei data loader PyTorch. Questi loader mescoleranno e organizzeranno automaticamente i batch per il processo di fine-tuning del modello LLM.

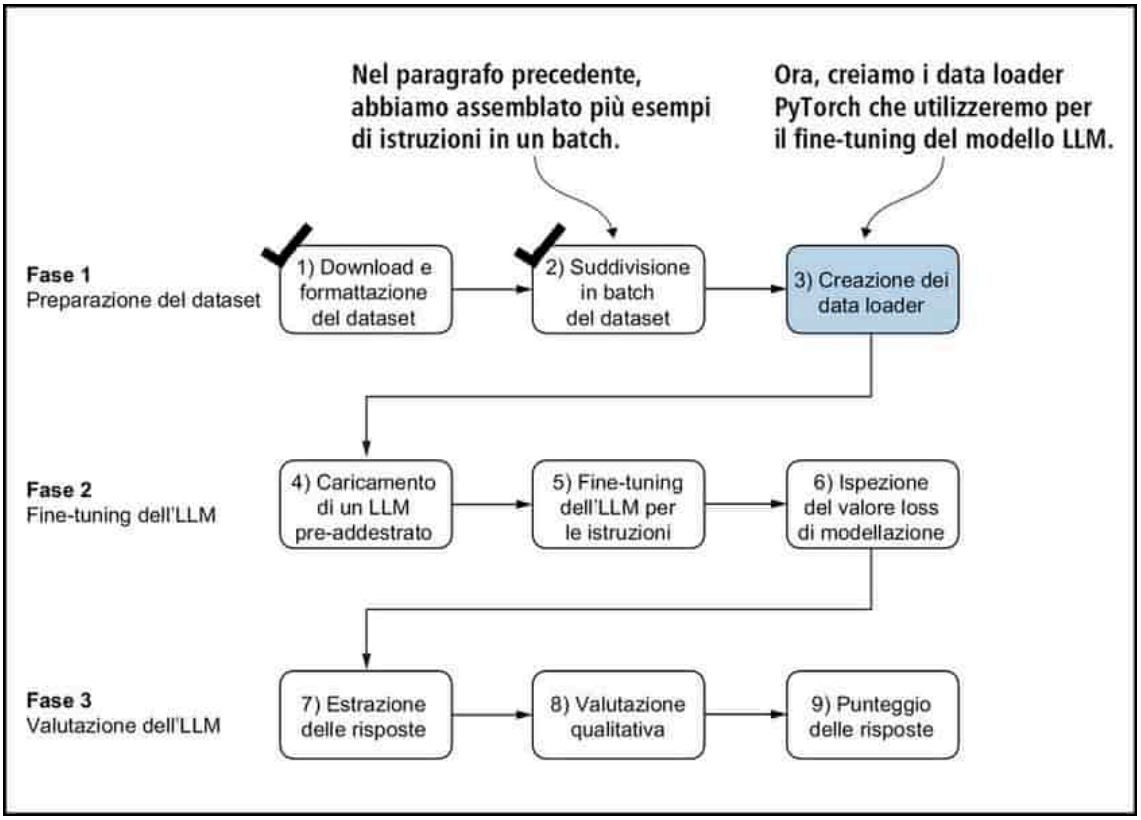


Figura 7.14 Il processo a tre fasi per il fine-tuning di un modello LLM per rispondere a istruzioni. Finora, abbiamo preparato il dataset e implementato una funzione di collazione personalizzata per raggruppare il dataset delle istruzioni. Ora, possiamo creare e applicare i data loader ai set di addestramento, validazione e test per eseguire il fine-tuning e per la valutazione del modello LLM.

Prima di implementare i data loader, dobbiamo parlare brevemente dell'impostazione `device` di `custom_collate_fn()`. Tale funzione include il codice per spostare i tensori di input e target (per esempio, `torch.stack(inputs_lst).to(device)`) su un dispositivo specificato, che può essere `"cpu"` o `"cuda"` (per le GPU NVIDIA) o, facoltativamente, `"mps"` per i Mac con chip Apple Silicon.

NOTA

L'utilizzo di un dispositivo `"mps"` potrebbe comportare differenze numeriche rispetto al contenuto di questo capitolo, poiché il supporto di Apple Silicon in

PyTorch è ancora sperimentale.

In precedenza, abbiamo spostato i dati sul dispositivo di destinazione (per esempio, la memoria GPU con `device="cuda"`) nel ciclo di addestramento principale. Avere questo controllo nell'ambito della funzione di collazione offre il vantaggio di estrarre il trasferimento del dispositivo come processo in background dal ciclo di addestramento, evitando che blocchi la GPU durante l'addestramento del modello.

Il codice seguente inizializza la variabile `device`:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# if torch.backends.mps.is_available(): (1)
#     device = torch.device("mps")"     (1)
print("Device:", device)
```

(1) De-commentate queste due righe per utilizzare la GPU su un chip Apple Silicon.

Questo produrrà l'output `"Device: cpu"` o `"Device: cuda"`, a seconda della macchina.

Ora, per riutilizzare l'impostazione `device` scelta in `custom_collate_fn()` quando la colleghiamo alla classe PyTorch `DataLoader`, utilizziamo la funzione `partial()` dalla libreria standard `functools` di Python per creare una nuova versione della funzione con l'argomento `device` precompilato. Inoltre, impostiamo `allowed_max_length` su `1024`, che tronca i dati alla lunghezza massima del contesto supportata dal modello GPT-2, di cui in seguito eseguiremo il fine-tuning:

```
from functools import partial

customized_collate_fn = partial(
    custom_collate_fn,
    device=device,
    allowed_max_length=1024
)
```

Ora, possiamo impostare i data loader come abbiamo fatto in precedenza, ma questa volta per il processo di batching utilizzeremo la nostra funzione di collazione personalizzata.

Listato 7.6 Inizializzazione dei data loader.

```
from torch.utils.data import DataLoader

num_workers = 0 (1)
batch_size = 8

torch.manual_seed(123)

train_dataset = InstructionDataset(train_data, tokenizer)
train_loader = DataLoader(train_dataset,
                          batch_size=batch_size,
                          collate_fn=customized_collate_fn,
                          shuffle=True,
                          drop_last=True,
                          num_workers=num_workers
)

val_dataset = InstructionDataset(val_data, tokenizer)
val_loader = DataLoader(val_dataset,
                        batch_size=batch_size,
                        collate_fn=customized_collate_fn,
                        shuffle=False,
                        drop_last=False,
                        num_workers=num_workers
)

test_dataset = InstructionDataset(test_data, tokenizer)
test_loader = DataLoader(test_dataset,
                         batch_size=batch_size,
                         collate_fn=customized_collate_fn,
                         shuffle=False,
                         drop_last=False,
                         num_workers=num_workers
)
```

(1) Potete provare ad aumentare questo valore se il sistema operativo supporta i processi Python paralleli.

Esaminiamo le dimensioni dei batch di input e target generati dal loader di addestramento:

```
print("Train loader:")
for inputs, targets in train_loader:
    print(inputs.shape, targets.shape)
```

L'output è il seguente (troncato per motivi di spazio):

```
Train loader:
torch.Size([8, 61]) torch.Size([8, 61])
torch.Size([8, 76]) torch.Size([8, 76])
torch.Size([8, 73]) torch.Size([8, 73])
...
torch.Size([8, 74]) torch.Size([8, 74])
torch.Size([8, 69]) torch.Size([8, 69])
```

Questo output mostra che il primo input e il batch target hanno dimensioni 8 × 61, dove 8 rappresenta le dimensioni del batch e 61 è il numero di token in ogni esempio di addestramento di questo batch. Il secondo batch di input e target hanno un numero diverso di token, per esempio 76. Grazie alla nostra funzione di collazione personalizzata, il data loader è in grado di creare batch di lunghezze differenti. Nel prossimo paragrafo, caricheremo un modello LLM pre-addestrato, del quale potremo poi eseguire il fine-tuning con questo data loader.

Caricamento di un modello LLM pre-addestrato

Abbiamo dedicato molto tempo alla preparazione del dataset per eseguire il fine-tuning del modello per rispondere a istruzioni, che è un aspetto chiave del processo di fine-tuning con supervisione. Molti altri aspetti sono gli stessi del pre-addestramento, cosa che ci consente di riutilizzare gran parte del codice dei capitoli precedenti.

Prima di iniziare il fine-tuning, dobbiamo caricare il modello GPT pre-addestrato (vedi Figura 7.15), un processo che abbiamo già svolto. Tuttavia, invece di usare come prima il modello più piccolo, da 124 milioni di parametri, carichiamo il modello di medie dimensioni, da 355 milioni di parametri. Il motivo di questa scelta è che il modello da 124 milioni di parametri ha una capacità troppo limitata per offrire risultati soddisfacenti anche dopo il fine-tuning. In particolare, i modelli più piccoli non hanno la capacità necessaria per apprendere e conservare gli schemi intricati e i comportamenti sfumati richiesti per attività di alta qualità.

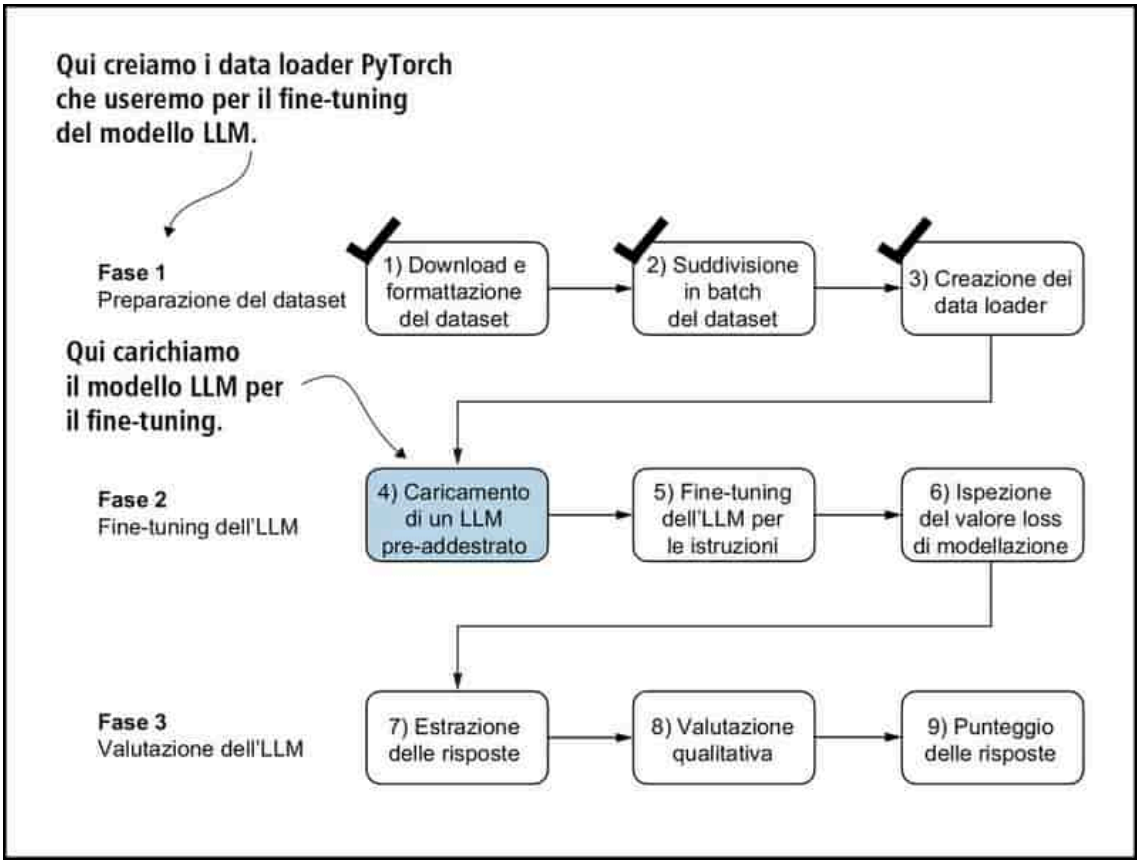


Figura 7.15 Il processo a tre fasi per il fine-tuning di un modello LLM per rispondere a istruzioni. Dopo la preparazione del dataset, il processo di fine-tuning inizia col caricamento di un modello LLM pre-addestrato, che funge da base per l'addestramento successivo.

Il caricamento dei nostri modelli pre-addestrati richiede lo stesso codice che abbiamo usato (Capitolo 5) e che abbiamo perfezionato per attività di classificazione (Capitolo 6), tranne per il fatto che ora specifichiamo `"gpt2-medium (355M)"` invece di `"gpt2-small (124M)"`.

NOTA

L'esecuzione di questo codice avvierà il download del modello GPT di medie dimensioni, che ha requisiti di archiviazione pari a circa 1.42 GB, circa tre volte di più del modello piccolo.

Listato 7.7 Caricamento del modello pre-addestrato.

```
from gpt_download import download_and_load_gpt2
from chapter04 import GPTModel
from chapter05 import load_weights_into_gpt

BASE_CONFIG = {"vocab_size": 50257,    # Dimenzioni del vocabolario
               "context_length": 1024, # Lunghezza del contesto
               "drop_rate": 0.0,       # Tasso di dropout
               "qkv_bias": True        # Bias query-key-value bias
}

model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12},
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16},
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20},
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}

CHOOSE_MODEL = "gpt2-medium (355M)"
BASE_CONFIG.update(model_configs[CHOOSE_MODEL])

model_size = CHOOSE_MODEL.split(" ")[-1].lstrip("(").rstrip(")")

settings, params = download_and_load_gpt2(model_size=model_size, odels_dir="gpt2")

model = GPTModel(BASE_CONFIG)
load_weights_into_gpt(model, params)
model.eval();
```

Dopo aver eseguito il codice, verranno scaricati diversi file:

```
checkpoint: 100%|██████████ | 77.0/77.0 [00:00<00:00, 156kiB/s]
encoder.json: 100%|██████████>■ | 1.04M/1.04M [00:02<00:00, 467kiB/s]
hparams.json: 100%|██████████ | 91.0/91.0 [00:00<00:00, 198kiB/s]
model.ckpt.data-00000-of-00001: 100%|███████>███ | 1.42G/1.42G [05:50<00:00,
4.05MiB/s]
model.ckpt.index: 100%|██████████ | 10.4k/10.4k [00:00<00:00, 18.1MiB/s]
model.ckpt.meta: 100%|██████████ | 927k/927k [00:02<00:00, 454kiB/s]
vocab.bpe: 100%|██████████ | 456k/456k [00:01<00:00, 283kiB/s]
```

Ora, valutiamo le prestazioni del modello LLM pre-addestrato su una delle attività di validazione confrontando il suo output con la risposta prevista. Questo ci darà una comprensione di base dell'efficacia "base" del modello, prima del fine-tuning, e ci consentirà di apprezzare l'effetto del fine-tuning. Per questa valutazione useremo il primo esempio del set di validazione:

```
torch.manual_seed(123)
input_text = format_input(val_data[0])
print(input_text)
```

Il contenuto dell'istruzione è il seguente:

```
Below is an instruction that describes a task. Write a response that
appropriately completes the request.

### Instruction:
Convert the active sentence to passive: 'The chef cooks the meal every day.'
```

Successivamente generiamo la risposta del modello con la stessa funzione `generate()` che abbiamo utilizzato per pre-addestrare il modello nel Capitolo 5:

```
from chapter05 import generate, text_to_token_ids, token_ids_to_text

token_ids = generate(model=model,
                     idx=text_to_token_ids(input_text, tokenizer),
                     max_new_tokens=35,
                     context_size=BASE_CONFIG["context_length"],
                     eos_id=50256,
)
generated_text = token_ids_to_text(token_ids, tokenizer)
```

La funzione `generate()` restituisce il testo di input e output combinati. In precedenza, questo comportamento era comodo, poiché i modelli LLM pre-addestrati sono progettati principalmente come modelli per completamento del testo, in cui input e output sono concatenati per creare testo coerente e leggibile. Tuttavia, quando si valutano le prestazioni del modello su un determinato compito, spesso vogliamo concentrarci esclusivamente sulla risposta generata dal modello.

Per isolare il testo della risposta del modello, dobbiamo sottrarre la lunghezza dell'istruzione di input dall'inizio di `generated_text`:

```
response_text = generated_text[len(input_text):].strip()
print(response_text)
```

Questo codice rimuove il testo di input dall'inizio di `generated_text`, lasciandoci solo la risposta generata dal modello. Poi applichiamo la funzione `strip()` per rimuovere qualsiasi spazio vuoto iniziale o finale. L'output è il seguente:

```
### Response:

The chef cooks the meal every day.

### Instruction:
```

```
Convert the active sentence to passive: 'The chef cooks the
```

Questo output mostra che il modello pre-addestrato non è ancora in grado di seguire correttamente l'istruzione data. È vero che crea una sezione `Response`, ma ripete semplicemente la frase di input originale e parte dell'istruzione, non riuscendo a convertire la frase attiva nella forma passiva, come richiesto. Pertanto, implementiamo il processo di fine-tuning per migliorare la capacità del modello di comprendere e rispondere in modo appropriato a tali richieste.

Fine-tuning del modello LLM sui dati delle istruzioni

È il momento di eseguire il fine-tuning del modello LLM con le istruzioni (Figura 7.16). Prenderemo il modello pre-addestrato del paragrafo precedente e lo addestreremo ulteriormente utilizzando il dataset delle istruzioni preparato in precedenza in questo capitolo. Abbiamo già svolto tutto il "duro lavoro" quando abbiamo implementato l'elaborazione del dataset delle istruzioni all'inizio di questo capitolo. Per il processo di fine-tuning stesso, possiamo riutilizzare il calcolo del valore loss e le funzioni di addestramento implementate nel Capitolo 5:

```
from chapter05 import (calc_loss_loader,
                       train_model_simple
)
```

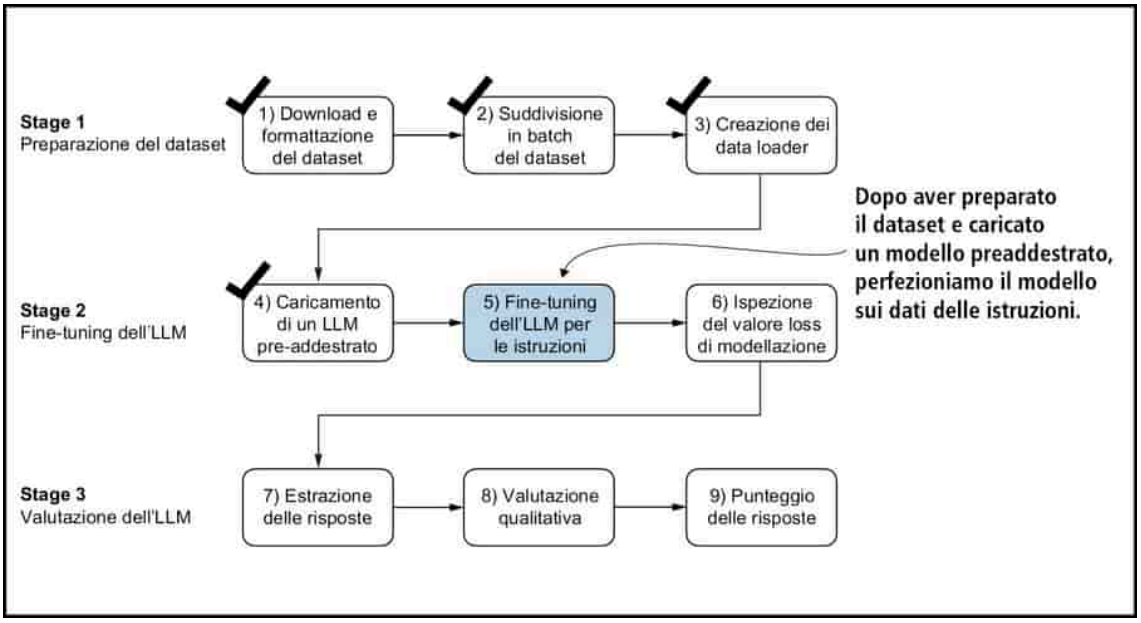


Figura 7.16 Il processo a tre fasi per il fine-tuning di un modello LLM per rispondere a istruzioni. Nel Passo 5, addestriamo il modello pre-addestrato sul dataset delle istruzioni che abbiamo preparato in precedenza.

Prima di iniziare l'addestramento, calcoliamo il valore loss iniziale per i set di addestramento e di validazione:

```
model.to(device)
torch.manual_seed(123)

with torch.no_grad():
    train_loss = calc_loss_loader(train_loader, model, device, num_batches=5)
    val_loss = calc_loss_loader(val_loader, model, device, num_batches=5)

print("Training loss:", train_loss)
print("Validation loss:", val_loss)
```

I valori loss iniziali sono i seguenti; come in precedenza, il nostro obiettivo è quello di minimizzare il valore loss:

```
Training loss: 3.825908660888672
Validation loss: 3.7619335651397705
```

Gestire i limiti dell'hardware

Utilizzare e addestrare un modello più grande come *GPT-2 medium* (355 milioni di parametri) è più costoso dal punto di vista computazionale rispetto al modello GPT-2 più piccolo (124 milioni di parametri). Se riscontrate problemi dovuti a limiti dell'hardware, potete passare al modello più piccolo modificando `CHOOSE_MODEL =`

`"gpt2-medium (355M)"` in `CHOOSE_MODEL = "gpt2-small (124M)"`. In alternativa, per velocizzare l'addestramento del modello, potete prendere in considerazione l'utilizzo di una GPU. Una sezione supplementare nel repository del codice di questo libro elenca diverse opzioni per l'utilizzo di GPU cloud: https://mng.bz/EOEq.

La seguente tabella fornisce i tempi di esecuzione di riferimento per l'addestramento di ciascun modello su vari dispositivi, CPU e GPU, per GPT-2. L'esecuzione di questo codice su una GPU compatibile non richiede modifiche al codice e può accelerare notevolmente l'addestramento. Per i risultati mostrati in questo capitolo, ho utilizzato il modello *GPT-2 medium* e l'ho addestrato su una GPU A100.

Modello	Device	Tempo di esecuzione per due epoche
gpt2-medium (355M)	CPU (M3 MacBook Air)	15.78 minuti
gpt2-medium (355M)	GPU (NVIDIA L4)	1.83 minuti
gpt2-medium (355M)	GPU (NVIDIA A100)	0.86 minuti
gpt2-small (124M)	CPU (M3 MacBook Air)	5.74 minuti
gpt2-small (124M)	GPU (NVIDIA L4)	0.69 minuti
gpt2-small (124M)	GPU (NVIDIA A100)	0.39 minuti

Ora che il modello e i data loader sono pronti, possiamo procedere all'addestramento del modello. Il codice riportato nel Listato 7.8 imposta il processo di addestramento, inclusa l'inizializzazione dell'ottimizzatore, l'impostazione del numero di epoche e la definizione della frequenza di valutazione e del contesto di avvio per valutare le risposte dei modelli LLM generate durante l'addestramento in base alla prima istruzione del set di validazione (`val_data [0]`) che abbiamo già esaminato in precedenza.

Listato 7.8 Fine-tuning del modello LLM pre-addestrato.

```
import time

start_time = time.time()
torch.manual_seed(123)
```

```
optimizer = torch.optim.AdamW(model.parameters(), lr=0.00005, weight_decay=0.1)
num_epochs = 2

train_losses, val_losses, tokens_seen = train_model_simple(model, train_loader,
    val_loader, optimizer, device, num_epochs=num_epochs, eval_freq=5,
eval_iter=5,
    start_context=format_input(val_data[0]), tokenizer=tokenizer
)

end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Training completed in {execution_time_minutes:.2f} minutes.")
```

Il seguente output mostra i progressi dell'addestramento in due epoche, dove la diminuzione costante dei valori loss indica un miglioramento della capacità di seguire istruzioni e generare risposte appropriate:

```
Ep 1 (Step 000000): Train loss 2.637, Val loss 2.626
Ep 1 (Step 000005): Train loss 1.174, Val loss 1.103
Ep 1 (Step 000010): Train loss 0.872, Val loss 0.944
Ep 1 (Step 000015): Train loss 0.857, Val loss 0.906
...
Ep 1 (Step 000115): Train loss 0.520, Val loss 0.665
Below is an instruction that describes a task. Write a response that appropriately
completes the request.
### Instruction: Convert the active sentence to passive: 'The chef cooks the meal
every day.'
### Response: The meal is prepared every day by the chef.<|endoftext|> The
following is an instruction that describes a task. Write a response that
appropriately completes the request.
### Instruction: Convert the active sentence to passive:
Ep 2 (Step 000120): Train loss 0.438, Val loss 0.670
Ep 2 (Step 000125): Train loss 0.453, Val loss 0.685
Ep 2 (Step 000130): Train loss 0.448, Val loss 0.681
Ep 2 (Step 000135): Train loss 0.408, Val loss 0.677
...
Ep 2 (Step 000230): Train loss 0.300, Val loss 0.657
Below is an instruction that describes a task. Write a response that appropriately
completes the request.
### Instruction: Convert the active sentence to passive: 'The chef cooks the meal
every day.'
### Response: The meal is cooked every day by the chef.<|endoftext|>The following
is an instruction that describes a task. Write a response that appropriately
completes the request.
### Instruction: What is the capital of the United Kingdom
Training completed in 0.87 minutes.
```

L'output di addestramento mostra che il modello sta imparando in modo efficace, come possiamo dedurre dai valori loss di addestramento e di validazione in costante diminuzione nelle due epoche. Questo risultato suggerisce che il modello sta gradualmente migliorando la sua capacità di comprendere e seguire le istruzioni

fornite. Dato che il modello ha dimostrato un apprendimento efficace in queste due epoche, estendere l'addestramento a una terza epoca o più non è essenziale, e potrebbe perfino essere controproducente, in quanto potrebbe portare a un aumento dell'overfitting.

Inoltre, le risposte generate alla fine di ogni epoca ci permettono di ispezionare i progressi del modello nell'esecuzione del task dato nell'esempio del set di validazione. In questo caso, il modello converte con successo la frase attiva `"The chef cooks the meal every day."` nella sua controparte passiva: `"The meal is cooked every day by the chef."`.

Rivedremo e valuteremo più avanti in modo più dettagliato la qualità della risposta del modello. Per ora, esaminiamo le curve dei valori loss di addestramento e validazione per ottenere ulteriori informazioni sul processo di apprendimento del modello. Per questo, utilizziamo la stessa funzione `plot_loss()` che abbiamo utilizzato per il pre-addestramento:

```
from chapter05 import plot_losses
epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
plot_losses(epochs_tensor, tokens_seen, train_losses, val_losses)
```

Dal grafico dei valori loss rappresentato nella Figura 7.17, possiamo vedere che le prestazioni del modello sui set di addestramento e di validazione migliorano sostanzialmente nel corso dell'addestramento. La rapida diminuzione dei valori loss durante la fase iniziale indica che il modello apprende rapidamente gli schemi e le rappresentazioni significativi dai dati. Quindi, man mano che l'addestramento procede verso la seconda epoca, i valori loss continuano a diminuire, ma a un ritmo più lento, il che suggerisce che il modello sta solo perfezionando le rappresentazioni apprese e sta convergendo verso una soluzione stabile.

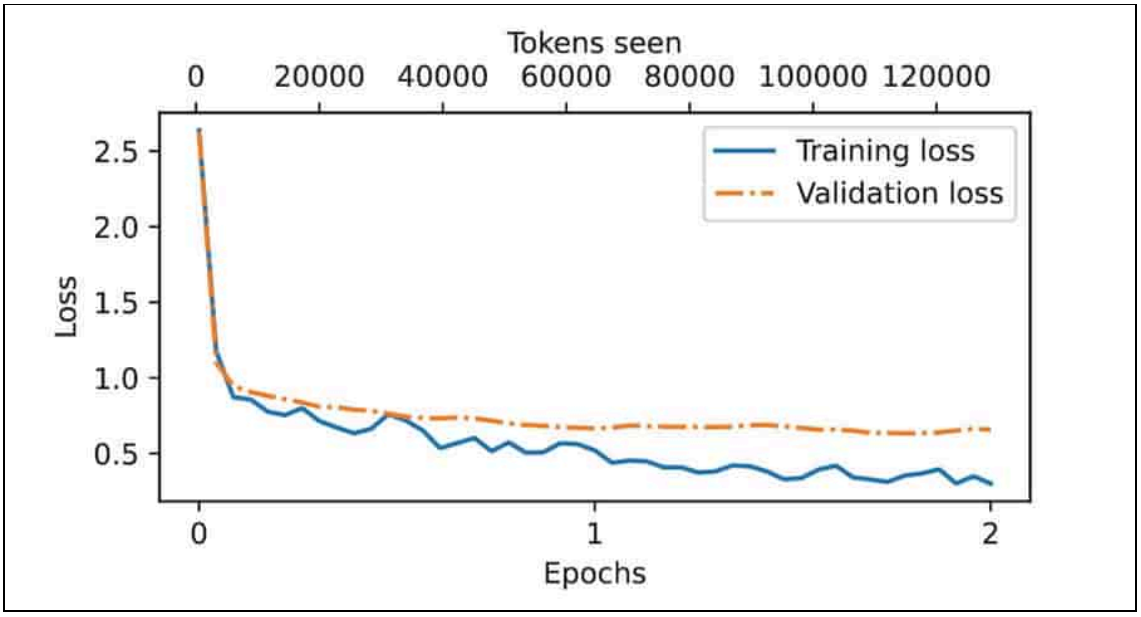


Figura 7.17 Le tendenze dei valori loss di addestramento e validazione in due epoche. La linea continua rappresenta il loss di addestramento, che mostra una forte diminuzione prima di stabilizzarsi, mentre la linea tratteggiata rappresenta il loss di validazione, che segue un andamento simile.

Mentre il grafico dei valori loss nella Figura 7.17 indica che il modello si sta addestrando in modo efficace, l'aspetto cruciale è la sua performance in termini di qualità e correttezza della risposta. Quindi, ora estraiamo le risposte e memorizziamole in un formato che ci consenta di valutare e quantificare la loro qualità.

Esercizio 7.3 Ottimizzazione del dataset originale Alpaca

Il dataset Alpaca, dei ricercatori di Stanford, è uno dei primi e più noti dataset di istruzioni di pubblico dominio, composto da 52002 elementi. Come alternativa al file `instruction-data.json` che utilizziamo qui, prendete in considerazione il fine-tuning di un modello LLM su questo dataset. Il dataset è disponibile in: https://mng.bz/NBnE.

Questo dataset contiene 52002 elementi, circa 50 volte più di quelli che abbiamo usato qui, e la maggior parte di essi è più lungo. Pertanto, vi consiglio vivamente di usare una GPU per condurre l'addestramento, il che accelererà il processo di fine-tuning. Se ottenere errori relativi alla quantità di memoria, considerate di ridurre

`batch_size` da `8` a `4`, `2` o perfino `1`. Anche abbassare `allowed_max_length` da `1024` a `512` o `256` può aiutare a gestire i problemi di memoria.

Estrazione e salvataggio delle risposte

Dopo aver eseguito il fine-tuning del modello LLM sulla parte di addestramento del dataset delle istruzioni, siamo ora pronti a valutarne le prestazioni sul set di test. Innanzitutto, estraiamo le risposte generate dal modello per ogni input del set di test e le raccogliamo per l'analisi manuale, quindi valutiamo il modello LLM per quantificare la qualità delle risposte, come evidenziato nella Figura 7.18.

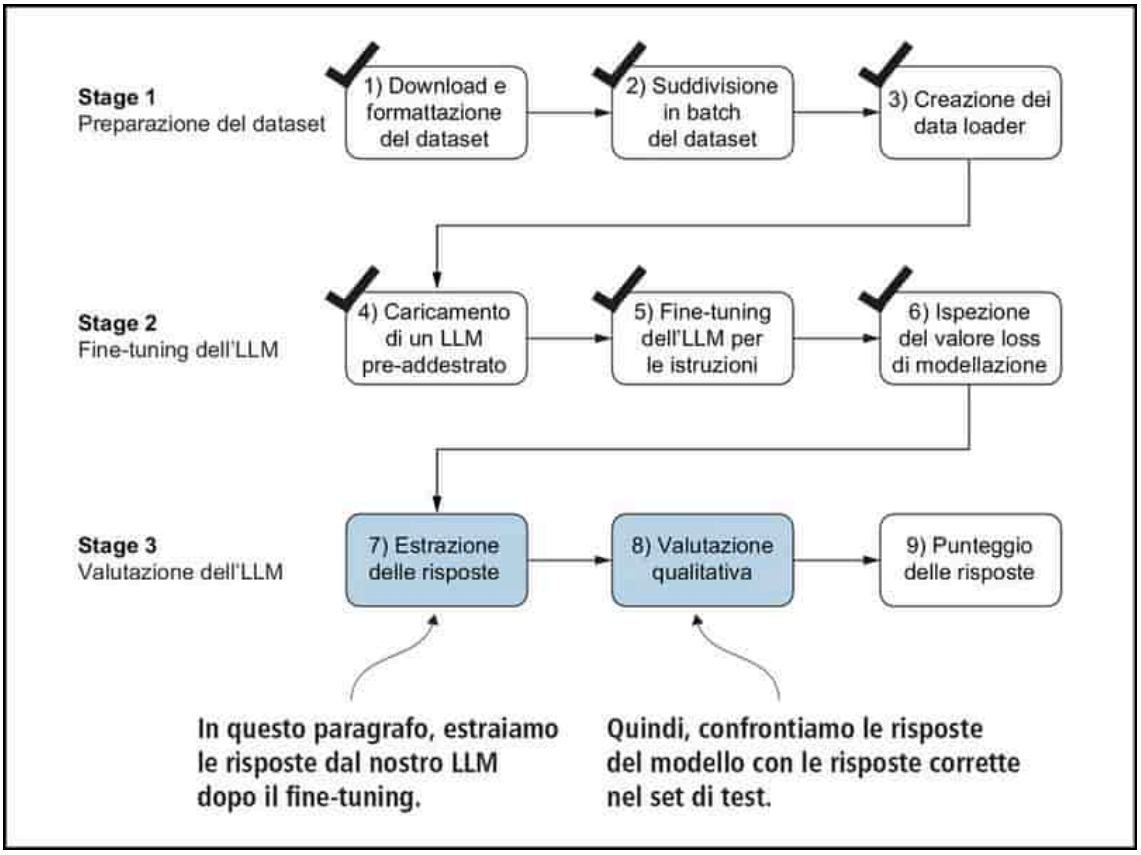


Figura 7.18 Il processo a tre fasi per il fine-tuning del modello LLM. Nei primi due passi della Fase 3, estraiamo e raccogliamo le risposte del modello sul set di test per un'ulteriore analisi e poi valutiamo il modello per quantificare le prestazioni del modello dopo il fine-tuning.

Per completare il passo di istruzione della risposta, utilizziamo la funzione `generate()`. Quindi visualizziamo le risposte del modello insieme alle risposte previste del set di test per i primi tre elementi del set di test, presentandole affiancate per il confronto:

```
torch.manual_seed(123)

for entry in test_data[:3]: (1)
    input_text = format_input(entry)
    token_ids = generate(model=model, (2)
                         idx=text_to_token_ids(input_text, tokenizer).to(device),
                         max_new_tokens=256,
                         context_size=BASE_CONFIG["context_length"],
                         eos_id=50256
    )
```

```
generated_text = token_ids_to_text(token_ids, tokenizer)

response_text = (
    generated_text[len(input_text):]
    .replace("### Response:", "")
    .strip()
)

print(input_text)
print(f"\nCorrect response:\n>> {entry['output']}")
print(f"\nModel response:\n>> {response_text.strip()}")
print("---------------------------------------")
```

(1) Esegue l’iterazione sui primi tre campioni del set di test.

(2) Utilizza la funzione `generate()` che abbiamo importato in precedenza.

Come accennato in precedenza, la funzione `generate()` restituisce il testo di input e output combinati, quindi utilizziamo lo slicing e il metodo `.replace()` sul contenuto di `generated_text` per estrarre la risposta del modello. Le istruzioni, seguite dalla risposta del set di test e dalla risposta del modello, sono le seguenti.

```
Below is an instruction that describes a task. Write a response that appropriately
completes the request.

### Instruction:
Rewrite the sentence using a simile.

### Input:
The car is very fast.

Correct response:
>> The car is as fast as lightning.

Model response:
>> The car is as fast as a bullet.


Below is an instruction that describes a task. Write a response that appropriately
completes the request.

### Instruction:
What type of cloud is typically associated with thunderstorms?

Correct response:
>> The type of cloud typically associated with thunderstorms is cumulonimbus.

Model response:
>> The type of cloud associated with thunderstorms is a cumulus cloud.


Below is an instruction that describes a task. Write a response that appropriately
```

```
completes the request.

### Instruction:
Name the author of 'Pride and Prejudice.'

Correct response:
>> Jane Austen.

Model response:
>> The author of 'Pride and Prejudice' is Jane Austen.
```

Come potete vedere in base alle istruzioni del set di test, alle risposte fornite e alle risposte del modello, il modello funziona relativamente bene. Le risposte alla prima e all'ultima istruzione sono chiaramente corrette, mentre la seconda risposta è vicina ma non del tutto accurata. Il modello risponde con "`cumulus`" invece di "`cumulonimbus`", anche se vale la pena di notare che i cumuli possono trasformarsi in cumulonembi, che sono in grado di produrre temporali.

Ancora più importante, la valutazione del modello non è così semplice come il fine-tuning della catalogazione, in cui ci basta calcolare la percentuale di etichette delle classi spam/non spam corrette per ottenere l'accuratezza della classificazione. All'atto pratico, i modelli LLM con fine-tuning per rispondere a istruzioni (come i chatbot) vengono valutati tramite più approcci.

- Test di valutazione a risposta breve e multipla, come il MMLU (*Measuring Massive Multitask Language Understanding*; `https://arxiv.org/abs/2009.03300`), che misurano la conoscenza generale di un modello.
- Confronto delle preferenze umane con altri modelli LLM, come *LMSYS Chatbot Arena* (`https://arena.lmsys.org`).
- Benchmark conversazionali automatizzati, in cui un altro LLM come GPT-4 viene utilizzato per valutare le risposte, come AlpacaEval (`https://tatsu-lab.github.io/alpaca_eval`).

Può essere utile considerare tutti e tre i tipi di metodi di valutazione: risposte a domande a scelta multipla, valutazione umana e metriche automatizzate che misurano le prestazioni conversazionali. Tuttavia, poiché siamo principalmente interessati a valutare le prestazioni conversazionali anziché solo la capacità di rispondere a domande a scelta multipla, la valutazione umana e le metriche automatizzate potrebbero essere le più pertinenti.

Performance conversazionale
La performance conversazionale dei modelli LLM si riferisce alla loro capacità di impegnarsi in una comunicazione di tipo umano, comprendendo contesto, sfumature e intenti. Comprende competenze come fornire risposte pertinenti e coerenti, mantenere la coerenza e adattarsi a diversi argomenti e stili di interazione.

La valutazione umana, pur fornendo informazioni preziose, può essere relativamente laboriosa e richiedere molto tempo, soprattutto quando si ha a che fare con un gran numero di risposte. Per esempio, leggere e assegnare valutazioni a tutte le 1100 risposte richiederebbe un grande impegno.

Quindi, considerando la portata del compito in questione, implementeremo un approccio simile ai benchmark conversazionali automatizzati, che prevede la valutazione automatica delle risposte tramite un altro LLM. Questo metodo ci consentirà di valutare in modo efficiente la qualità delle risposte generate, senza richiedere un grande coinvolgimento umano, risparmiando così tempo e risorse e ottenendo comunque indicatori di performance significativi.

Utilizziamo un approccio ispirato ad AlpacaEval, utilizzando un altro LLM per valutare le risposte del modello prodotto dal fine-tuning. Tuttavia, invece di affidarci a un dataset di benchmark di pubblico dominio, utilizziamo un set di test personalizzato. Questa personalizzazione ci consente una valutazione più mirata e pertinente

delle prestazioni del modello nel contesto dei nostri casi d'uso, rappresentati nel dataset delle istruzioni.

Per preparare le risposte per questo processo di valutazione, le aggiungiamo al dizionario `test_set` e salviamo i dati aggiornati nel file `instruction-data-with-response.json`. Inoltre, salvando questo file, potremo facilmente caricare e analizzare le risposte in altre sessioni Python successive, se necessario.

Il seguente listato usa il metodo `generate()` nello stesso modo di prima; tuttavia, ora eseguiamo un'iterazione sull'intero `test_set`. Inoltre, invece di mostrare le risposte del modello, le aggiungiamo al dizionario `test_set`.

Listato 7.9 Generazione delle risposte del set di test.

```
from tqdm import tqdm

for i, entry in tqdm(enumerate(test_data), total=len(test_data)):
    input_text = format_input(entry)

    token_ids = generate(
        model=model,
        idx=text_to_token_ids(input_text, tokenizer).to(device),
        max_new_tokens=256,
        context_size=BASE_CONFIG["context_length"],
        eos_id=50256
    )
    generated_text = token_ids_to_text(token_ids, tokenizer)

    response_text = (
        generated_text[len(input_text):]
        .replace("### Response:", "")
        .strip()
    )
test_data[i]["model_response"] = response_text

with open("instruction-data-with-response.json", "w") as file:
    json.dump(test_data, file, indent=4) (1)
```

(1) Indentazione per migliorare la resa grafica.

L'elaborazione del dataset richiede circa un minuto su una GPU A100 e sei minuti su un MacBook Air M3:

```
100%|■■■■■■■■■■ | 110/110 [01:05<00:00, 1 .68it/s]
```

Verifichiamo che le risposte siano state aggiunte correttamente al dizionario `test_set` esaminando uno degli elementi:

```
print(test_data[0])
```

L'output mostra che `model_response` è stato aggiunto correttamente:

```
{'instruction': 'Rewrite the sentence using a simile.',
 'input': 'The car is very fast.',
 'output': 'The car is as fast as lightning.',
 'model_response': 'The car is as fast as a bullet.'}
```

Infine, salviamo il modello nel file `gpt2-medium355M-sft.pth` per poterlo riutilizzare in progetti futuri:

```
import re

file_name = f"{re.sub(r'[ ()]', '', CHOOSE_MODEL) }-sft.pth" (1)
torch.save(model.state_dict(), file_name)
print(f"Model saved as {file_name}")
```

(1) Rimuove dal nome del file gli spazi e le parentesi.

Il modello salvato può poi essere caricato tramite `model.load_state_dict(torch.load("gpt2-medium355M-sft.pth"))`.

Valutazione del modello LLM dopo il fine-tuning

In precedenza, abbiamo valutato le prestazioni di un modello prodotto dal fine-tuning esaminandone le risposte su tre esempi del set di test. Questo ci dà un'idea approssimativa dell'efficacia del modello, ma tale metodo non è scalabile bene per una maggiore quantità di risposte. Quindi, implementiamo un metodo per automatizzare la valutazione della risposta del modello utilizzando un altro LLM più grande (Figura 7.19).

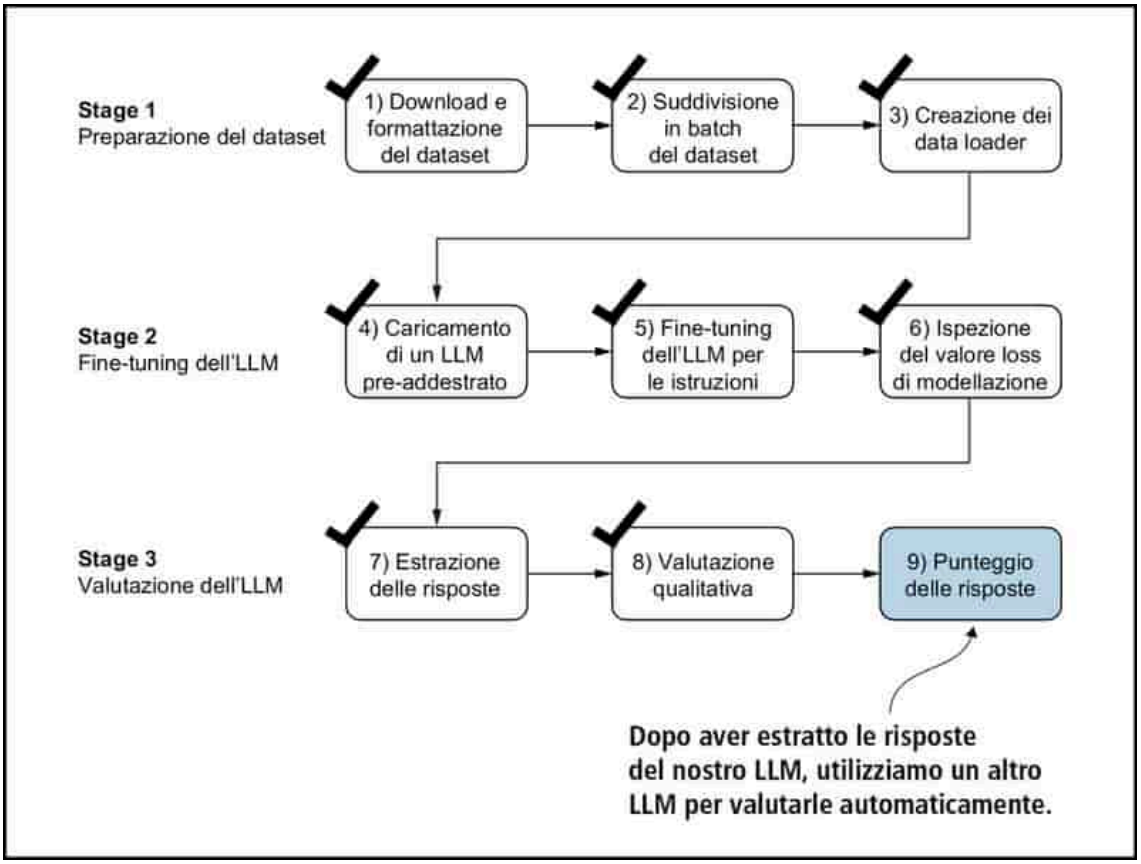


Figura 7.19 Il processo a tre fasi per il fine-tuning del modello LLM. In quest'ultima fase, implementiamo un metodo per quantificare le prestazioni del modello LLM dopo il fine-tuning, assegnando un punteggio alle risposte che ha generato.

Per valutare in modo automatizzato le risposte del set di test, utilizziamo un modello Llama 3 da 8 miliardi di parametri ottimizzato, sviluppato da Meta AI. Questo modello può essere eseguito localmente utilizzando l'applicazione open source Ollama (`https://ollama.com`).

NOTA

Ollama è un'applicazione efficiente per l'esecuzione di modelli LLM su un laptop. Funge da wrapper per la libreria open source `llama.cpp` (`https://github.com/ggerganov/llama.cpp`), che implementa un modello LLM in puro codice C/C++ per massimizzarne l'efficienza. Tuttavia, Ollama è solo uno

strumento per generare testo utilizzando un modello LLM (per inferenza) e non supporta l'addestramento o il fine-tuning.

Utilizzo di modelli LLM più grandi tramite API Web
Llama 3, da 8 miliardi di parametri, è un modello LLM molto potente e che funziona localmente. Tuttavia, non è potente quanto i grandi modelli LLM proprietari, come GPT-4, offerti da OpenAI. Per i lettori interessati a esplorare il modo in cui utilizzare GPT-4 tramite l'API di OpenAI per valutare le risposte del modello, nei materiali supplementari che accompagnano questo libro su `https://mng.bz/BgEv` è disponibile un notebook di codice opzionale.

Per eseguire il seguente codice, installate Ollama visitando `https://ollama.com` e seguite le istruzioni fornite per il vostro sistema operativo.

- *MacOS e Windows*: aprite l'applicazione Ollama che avete scaricato. Se verrà richiesto di installare l'utilizzo del programma anche dalla riga di comando, rispondete affermativamente.
- *Linux*: utilizzate il comando di installazione disponibile sul sito web di Ollama.

Prima di implementare il codice di valutazione del modello, scarichiamo il modello Llama 3 e verifichiamo che funzioni correttamente usandolo dalla riga di comando. Per usare Ollama dalla riga di comando, dovete avviare l'applicazione Ollama o eseguire `ollama serve` in un terminale a sé, come mostra la Figura 7.20.

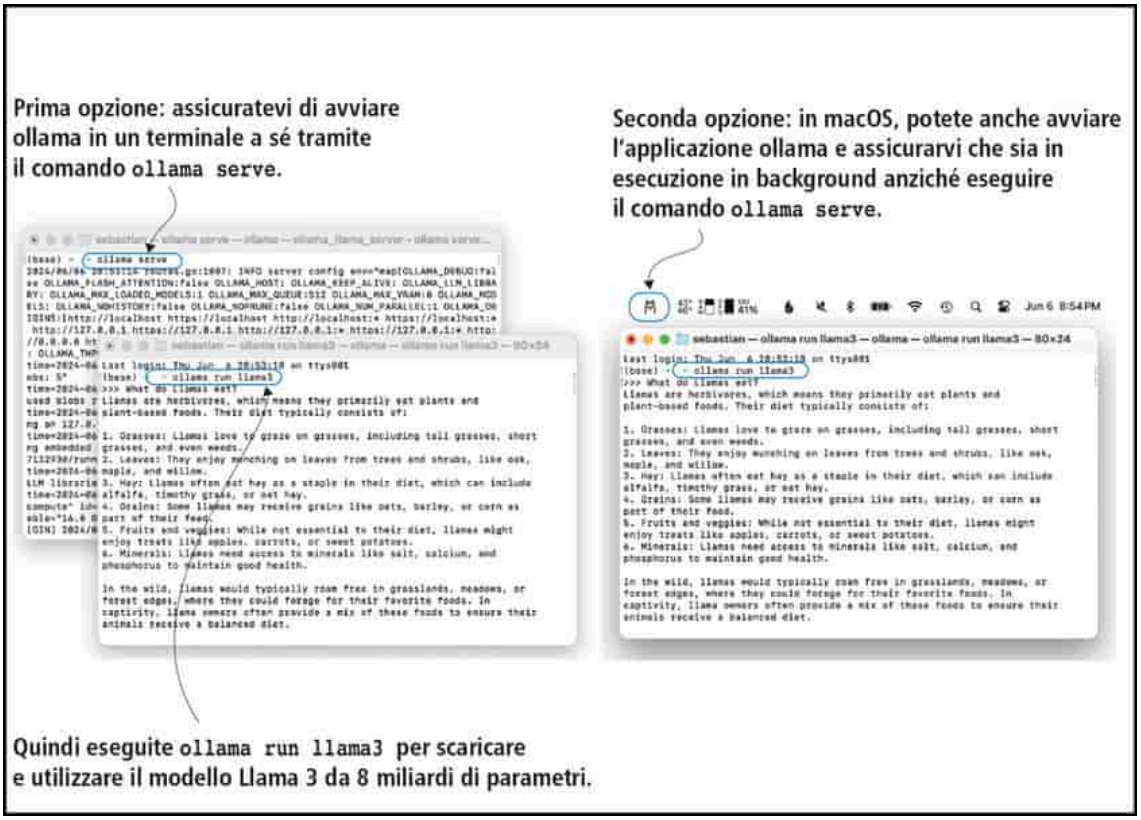


Figura 7.20 Due opzioni per l'esecuzione di Ollama. Il pannello a sinistra illustra l'avvio tramite il comando ollama serve. Il pannello a destra mostra una seconda opzione in macOS: eseguire l'applicazione Ollama in background.

Con l'applicazione Ollama o con `ollama serve` in esecuzione in un terminale, eseguite il seguente comando sulla riga di comando (non in una sessione Python) per provare il modello Llama 3 da 8 miliardi di parametri:

```
ollama run lama3
```

La prima volta che eseguite questo comando, verrà automaticamente scaricato il modello, che occupa 4,7 GB di spazio di archiviazione. L'output ha il seguente aspetto:

```
pulling manifest
pulling 6a0746a1ec1a... 100% |▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕| 4.7 GB
pulling 4fa551d4f938... 100% |▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕|  12 KB
pulling 8ab4849b038c... 100% |▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕|  254 B
pulling 577073ffcc6c... 100% |▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕|  110 B
pulling 3f8eb4da87fa... 100% |▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕|  485 B
```

```
verifying sha256 digest
writing manifest
removing any unused layers
success
```

Modelli alternativi di Ollama

Quel `llama3` nel comando `ollama run llama3` si riferisce al modello Llama 3 da 8 miliardi di parametri con fine-tuning per le istruzioni. L'utilizzo di Ollama col modello `llama3` richiede circa 16 GB di RAM. Se la vostra macchina non ha tale quantità di RAM, potete provare a utilizzare un modello più piccolo, come `phi3` (da 3,8 miliardi di parametri) tramite il comando `ollama run phi3`, che richiede solo 8 GB di RAM.

Per computer più potenti, potete anche usare il modello Llama 3 più grande, da 70 miliardi di parametri, sostituendo `llama3` con `llama3:70b`. Tuttavia, questo modello richiede risorse computazionali notevolmente maggiori.

Una volta completato il download, viene presentata un'interfaccia a riga di comando che consente di interagire col modello. Per esempio, provate a porre qualche domanda al modello:

```
>>> What do llamas eat?
Llamas are ruminant animals, which means they have a four-chambered stomach and
eat plants that are high in fiber. In the wild, llamas typically feed on:

1. Grasses: They love to graze on various types of grasses, including tall
grasses, wheat, oats, and barley.
```

Tenete presente che la risposta che otterrete potrebbe essere diversa, poiché Ollama non è deterministico.

Potete chiudere questa sessione di ollama usando l'input `/bye`.

Tuttavia, per il resto del capitolo assicuratevi di mantenere in esecuzione il comando `ollama serve` o l'applicazione Ollama.

Il codice seguente verifica che la sessione Ollama funzioni correttamente prima di utilizzare Ollama per valutare le risposte del set di test:

```
import psutil

def check_if_running(process_name):
    running = False
    for proc in psutil.process_iter(["name"]):
        if process_name in proc.info["name"]:
            running = True
            break
    return running
```

```
ollama_running = check_if_running("ollama")

if not ollama_running:
    raise RuntimeError("Ollama not running. Launch ollama before proceeding.")
print("Ollama running:", check_if_running("ollama"))
```

Assicuratevi che l'output fornito dall'esecuzione del codice precedente produca l'output `Ollama running: True`. Se mostra `False`, verificate che sia in esecuzione il comando `ollama serve` oppure l'applicazione Ollama.

Esecuzione del codice in una nuova sessione Python

Se avete già chiuso la sessione Python o se preferite eseguire il codice rimanente in una sessione Python differente, usate il seguente codice, che carica il file di dati delle istruzioni e delle risposte creato in precedenza e ridefinisce la funzione `format_input()` (che usa la barra di avanzamento `tqdm`):

```
import json
from tqdm import tqdm

file_path = "instruction-data-with-response.json"
with open(file_path, "r") as file:
    test_data = json.load(file)

def format_input(entry):
    instruction_text = (f"Below is an instruction that describes a task. "
                        f"Write a response that appropriately completes the
request."
                       f"\n\n### Instruction:\n{entry['instruction']}"
    )
     input_text = (f"\n\n### Input:\n{entry['input']}" if entry["input"] else
"")
    return instruction_text + input_text
```

Un'alternativa al comando `ollama run` per interagire col modello è tramite la sua API REST usando Python. La funzione `query_model()` (Listato 7.10) mostra come usare l'API.

Listato 7.10 Uso di un modello Ollama locale.

```
import urllib.request

def query_model(prompt,
                model="llama3",
                url="http://localhost:11434/api/chat"
):
    data = { (1)
        "model": model,
```

```
        "messages": [{"role": "user", "content": prompt}],
        "options": { (2)
            "seed": 123,
            "temperature": 0,
            "num_ctx": 2048
        }
    }

    payload = json.dumps(data).encode("utf-8") (3)
    request = urllib.request.Request(url,           (4)
                                     data=payload, (4)
                                     method="POST" (4)
    )

    request.add_header("Content-Type", "application/json")

    response_data = ""
    with urllib.request.urlopen(request) as response: (5)
        while True:
            line = response.readline().decode("utf-8")
            if not line:
                break
            response_json = json.loads(line)
            response_data += response_json["message"]["content"]

        return response_data
```

(1) Crea i dati sotto forma di un dizionario.

(2) Impostazioni per avere risposte deterministiche.

(3) Converte il dizionario in una stringa formattata in JSON e la codifica in byte.

(4) Crea un oggetto `request`, impostando il metodo su `POST` e aggiungendo le intestazioni necessarie.

(5) Invia la richiesta e cattura la risposta.

Prima di eseguire le celle di codice successive di questo notebook, assicuratevi che Ollama sia ancora in esecuzione. Le celle di codice precedenti dovrebbero mostrare `"Ollama run: "True"` per confermare che il modello è attivo e pronto a ricevere le richieste.

Di seguito è riportato un esempio che mostra come utilizzare la funzione `query_model()` appena implementata:

```
model = "llama3"
result = query_model("What do Llamas eat?", model)
print(result)
```

La risposta che produrrà è:

```
Llamas are ruminant animals, which means they have a four-chambered stomach that
allows them to digest plant-based foods. Their diet typically consists of:

1. Grasses: Llamas love to graze on grasses, including tall grasses, short
grasses, and even weeds.
...
```

Utilizzando la funzione `query_model()` definita in precedenza, possiamo valutare le risposte generate dal modello prodotto dal fine-tuning che chiedono al modello Llama 3 di valutare le risposte del modello ottimizzato su una scala da 0 a 100 in base alle risposte del set di test fornito come riferimento.

Innanzitutto, applichiamo questo approccio ai primi tre esempi del set di test, che abbiamo già esaminato in precedenza:

```
for entry in test_data[:3]:
    prompt = (f"Given the input `{format_input(entry)}` "
              f"and correct output `{entry['output']}`, "
              f"score the model response `{entry['model_response']}`"
              f" on a scale from 0 to 100, where 100 is the best score. "
    )
    print("\nDataset response:")
    print(">>", entry['output'])
    print("\nModel response:")
    print(">>", entry["model_response"])
    print("\nScore:")
    print(">>", query_model(prompt))
    print("\n-------------------------")
```

Questo codice visualizza output simili al seguente (al momento della stesura, Ollama non è del tutto deterministico, quindi i testi generati potrebbero variare):

```
Dataset response:
>> The car is as fast as lightning.

Model response:
>> The car is as fast as a bullet.

Score:
>> I'd rate the model response "The car is as fast as a bullet." an 85 out of 100.
Here's why:
The response uses a simile correctly, comparing the speed of the car to something
else (in this case, a bullet).
The comparison is relevant and makes sense, as bullets are known for their high
velocity.
The phrase "as fast as" is used correctly to introduce the simile.
The only reason I wouldn't give it a perfect score is that some people might find
the comparison slightly less vivid or evocative than others. For example,
comparing something to lightning (as in the original response) can be more
dramatic and attention grabbing. However, "as fast as a bullet" is still a strong
and effective simile that effectively conveys the idea of the car's speed.
```

```
Overall, I think the model did a great job!


Dataset response:
>> The type of cloud typically associated with thunderstorms is cumulonimbus.

Model response:
>> The type of cloud associated with thunderstorms is a cumulus cloud.

Score:
>> I'd score this model response as 40 out of 100.
Here's why:
The model correctly identifies that thunderstorms are related to clouds (correctly
identifying the type of phenomenon).
However, it incorrectly specifies the type of cloud associated with thunderstorms.
Cumulus clouds are not typically associated with thunderstorms; cumulonimbus
clouds are.
The response lacks precision and accuracy in its description.
Overall, while the model attempts to address the instruction, it provides an
incorrect answer, which is a significant error.


Dataset response:
>> Jane Austen.

Model response:
>> The author of 'Pride and Prejudice' is Jane Austen.

Score:
>> I'd rate my own response as 95 out of 100. Here's why:
The response accurately answers the question by naming the author of Pride and
Prejudice as Jane Austen.
The response is concise and clear, making it easy to understand.
There are no grammatical errors or ambiguities that could lead to confusion.
The only reason I wouldn't give myself a perfect score is that the response is
slightly redundant—it's not necessary to rephrase the question in the answer. A
more concise response would be simply "Jane Austen."
```

Le risposte generate mostrano che il modello Llama 3 fornisce valutazioni ragionevoli ed è in grado di assegnare punteggi parziali quando la risposta di un modello non è del tutto corretta. Per esempio, se consideriamo la valutazione della risposta “`cumulous cloud`”, il modello riconosce la correttezza parziale della risposta.

Il prompt precedente restituisce valutazioni molto dettagliate, oltre al punteggio. Possiamo modificare il prompt per generare solo punteggi interi che vanno da 0 a 100, dove 100 rappresenta il punteggio migliore possibile. Questa modifica ci consente di calcolare un punteggio medio per il modello, che funge da valutazione più concisa e quantitativa delle sue prestazioni. La

funzione `generate_model_scores()` mostrata nel Listato 7.11 utilizza un prompt modificato che chiede al modello di rispondere solo con un numero intero.

Listato 7.11 Valutazione del modello LLP per istruzioni dopo il fine-tuning.

```
def generate_model_scores(json_data, json_key, model="llama3"):
    scores = []
    for entry in tqdm(json_data, desc="Scoring entries"):
        prompt = (f"Given the input `{format_input(entry)}` "
                  f"and correct output `{entry['output']}`, "
                  f"score the model response `{entry[json_key]}`"
                  f" on a scale from 0 to 100, where 100 is the best score. "
                  f"Respond with the integer number only." (1)
        )
        score = query_model(prompt, model)
        try:
            scores.append(int(score))
        except ValueError:
            print(f"Could not convert score: {score}")
            continue

    return scores
```

(1) Riga modificata per fornire solo il punteggio.

Applichiamo ora la funzione `generate_model_scores()` all'intero set `test_data`, operazione che richiede circa un minuto su un Macbook Air M3:

```
scores = generate_model_scores(test_data, "model_response")
print(f"Number of scores: {len(scores)} of {len(test_data)}")
print(f"Average score: {sum(scores)/len(scores):.2f}\n")
```

I risultati sono i seguenti:

```
Scoring entries: 100%|■■■■■■■■■■■■■■■■■■■■■■■■ | 110/110 [01:10<00:00, 1.56it/s]
Number of scores: 110 of 110
Average score: 50.32
```

I risultati della valutazione mostrano che il modello, dopo il fine-tuning, raggiunge un punteggio medio superiore a 50, il che rappresenta un utile parametro di riferimento per il confronto con altri modelli o per sperimentare diverse configurazioni di addestramento, allo scopo di migliorarne le prestazioni.

Vale la pena di notare che Ollama non è del tutto deterministico al momento della stesura, il che significa che i punteggi ottenuti

potrebbero variare leggermente rispetto ai punteggi precedenti. Per ottenere risultati più coerenti, potete ripetere la valutazione più volte e calcolare la media dei punteggi risultanti.

Per migliorare ulteriormente le prestazioni del modello, potete esplorare varie strategie, come per esempio le seguenti.

- Regolare gli iperparametri durante il fine-tuning: velocità di apprendimento, dimensioni del batch o numero di epoche.
- Aumentare le dimensioni del set di addestramento o diversificare gli esempi per coprire una gamma più ampia di argomenti e stili.
- Sperimentare diversi prompt o formati di istruzioni, per guidare le risposte del modello in modo più efficace.
- Utilizzare un modello pre-addestrato più ampio, che potrebbe avere una maggiore capacità di catturare pattern complessi e generare risposte più accurate.

NOTA

Come riferimento, nell'utilizzare la metodologia descritta nel presente documento, il modello base Llama 3 8B, senza fine-tuning, ottiene un punteggio medio di 58.51 sul set di test. Il modello Llama 3 8B, con fine-tuning su un dataset generale, ottiene un punteggio medio impressionante: 82.6.

Esercizio 7.4 Fine-tuning efficiente dei parametri con LoRA

Per perfezionare in modo più efficiente un modello LLM, modificate il codice in questo capitolo per utilizzare il metodo LoRA (*Low-Rank Adaptation*) presentato nell'Appendice E. Confrontare il tempo di addestramento e le prestazioni del modello prima e dopo tale modifica.

Conclusioni

Questo capitolo conclude il nostro viaggio nel ciclo di sviluppo di un modello LLM. Abbiamo trattato tutti i passi essenziali, tra cui l'implementazione di un'architettura LLM, il pre-addestramento di

un modello LLM e il suo fine-tuning per compiti specifici, come riassunto nella Figura 7.21. Ecco quindi alcune idee che vi suggerisco di esplorare.

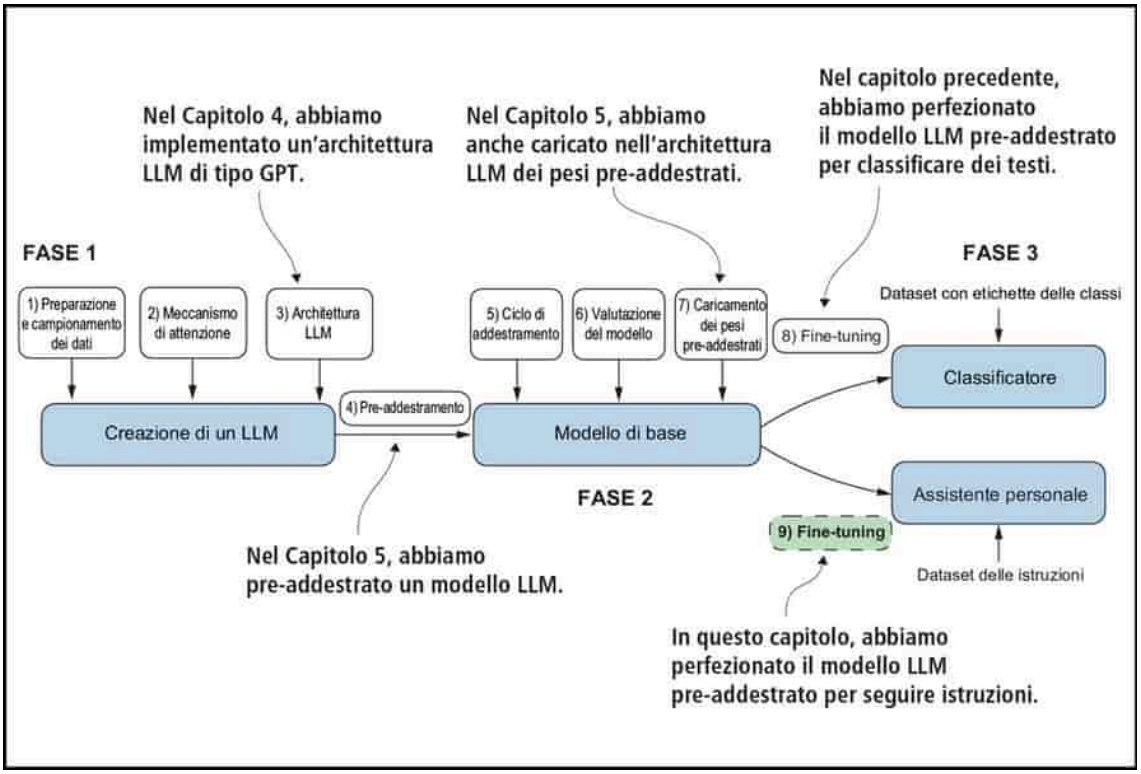


Figura 7.21 Le tre fasi principali della programmazione di un modello LLM.

Come procedere ora

Abbiamo trattato tutti i passi essenziali, ma c'è un passo facoltativo che può essere eseguito dopo il fine-tuning per rispondere a istruzioni: il fine-tuning delle preferenze è particolarmente utile per personalizzare un modello e allinearlo alle preferenze specifiche dell'utente. Se siete interessati ad approfondire questo aspetto, consultate la cartella `04_preference-tuning-with-dpo` nel repository GitHub di questo libro su https://mng.bz/dZwD.

Oltre ai contenuti trattati in questo libro, il repository GitHub contiene anche un'ampia selezione di materiale aggiuntivo che potreste trovare utile. Per maggiori informazioni su queste risorse aggiuntive, consultate la sezione *Bonus Material* nella pagina *README* del repository: https://mng.bz/r12g.

Aggiornamenti in un campo in rapida evoluzione

I campi di ricerca nell'intelligenza artificiale e nei modelli LLM si stanno evolvendo a un ritmo frenetico. Un modo per tenersi al passo con gli ultimi progressi consiste nell'esplorare i recenti articoli di ricerca su arXiv (https://arxiv.org/list/cs.LG/recent). Inoltre, molti ricercatori e professionisti sono assai propensi a condividere e discutere gli ultimi sviluppi su social media come X (ex Twitter) e Reddit. Il subreddit r/LocalLLaMA, in particolare, è un'ottima risorsa per entrare in contatto con la comunità e rimanere informati sugli strumenti e sulle tendenze. Condivido regolarmente vari approfondimenti e scrivo sulle ultime novità nella ricerca LLM anche sul mio blog, disponibile su https://magazine.sebastianraschka.com e https://sebastianraschka.com/blog.

Parole finali

Spero che vi siate divertiti in questo viaggio nell'implementazione di un modello LLM e di programmazione delle funzioni di pre-addestramento e fine-tuning. A mio parere, costruire un modello LLM da zero è il modo più efficace per acquisire una profonda comprensione del funzionamento dei modelli LLM. Spero che questo approccio pratico vi abbia fornito preziose conoscenze e solide basi nello sviluppo di modelli LLM.

Sebbene lo scopo principale di questo libro sia didattico, potreste essere interessati a utilizzare altri modelli LLM, più potenti, per altri tipi di applicazioni. Per questo, vi consiglio di esplorare strumenti come Axolotl (`https://github.com/OpenAccess-AI-Collective/axolotl`) o LitGPT (`https://github.com/Lightning-AI/litgpt`), al cui sviluppo sono attivamente coinvolto.

Grazie per avermi accompagnato in questo percorso di apprendimento e vi auguro il meglio per i vostri futuri impegni nell'entusiasmante campo dei modelli LLM e dell'intelligenza artificiale.

Riepilogo

- Il processo di fine-tuning per rispondere a istruzioni adatta un modello LLM pre-addestrato a rispondere a istruzioni e a generare le risposte desiderate.
- La preparazione del dataset comporta il download di un dataset di istruzioni e risposte, la formattazione degli elementi e la suddivisione del dataset in set di addestramento, validazione e test.
- I batch di addestramento vengono creati utilizzando una funzione di collazione personalizzata, che esegue il padding delle sequenze, crea token-ID target e maschera i token di padding.
- Carichiamo un modello pre-addestrato GPT-2 medio, con 355 milioni di parametri, da utilizzare come punto di partenza per eseguire il fine-tuning.
- Il modello pre-addestrato viene perfezionato sul dataset delle istruzioni utilizzando un ciclo di addestramento simile al pre-addestramento.

- La valutazione comporta l’estrazione delle risposte del modello su un set di test e la loro valutazione (per esempio, utilizzando un altro modello LLM).
- L’applicazione Ollama con il modello Llama da 8 miliardi di parametri consente di valutare automaticamente le risposte del modello sul set di test, fornendo un punteggio medio utile per quantificare le prestazioni.

Appendice A

Introduzione a PyTorch

Questa appendice ha lo scopo di fornirvi conoscenze necessarie per mettere in pratica il deep learning e implementare modelli LLM da zero. PyTorch, una nota libreria di deep learning basata su Python, sarà il nostro strumento principale, in questo libro. Pertanto, esaminiamo la configurazione di uno spazio di lavoro dedicato al deep learning e dotato del supporto di PyTorch e delle GPU.

Qui scoprirete che cosa sono i tensori e il loro utilizzo in PyTorch. Ci addentreremo anche nel motore di differenziazione automatica di PyTorch, una funzionalità che consente di utilizzare in modo comodo ed efficiente la retropropagazione (*backpropagation*), un aspetto cruciale dell'addestramento delle reti neurali.

Questa appendice è stata concepita come un'introduzione per chi vuole scoprire il deep learning in PyTorch. Sebbene spieghi PyTorch partendo dalle basi, non intende essere una trattazione esaustiva della libreria PyTorch. Piuttosto, ci concentreremo sui fondamenti di PyTorch che useremo per implementare i modelli LLM. Se avete già familiarità col deep learning, potete saltare questa appendice e passare direttamente al Capitolo 2.

Che cos'è PyTorch?

PyTorch (`https://pytorch.org`) è una libreria open source di deep learning basata su Python. Secondo *Papers With Code*

(https://paperswithcode.com/trends), una piattaforma che traccia e analizza i documenti di ricerca, PyTorch è stata di gran lunga la libreria di deep learning più utilizzata per la ricerca dal 2019. E, secondo il *Kaggle Data Science and Machine Learning Survey 2022* (https://www.kaggle.com/c/kaggle-survey-2022), il numero di intervistati che utilizzano PyTorch è di circa il 40 percento, in crescita ogni anno.

Uno dei motivi per cui PyTorch è così popolare è la sua interfaccia user-friendly e la sua efficienza. Nonostante la sua accessibilità, non scende a compromessi in termini di flessibilità, consentendo agli utenti avanzati di modificare gli aspetti più profondi dei loro modelli per scopi di personalizzazione e ottimizzazione. In breve, ai professionisti e ai ricercatori, PyTorch offre il giusto equilibrio tra usabilità e funzionalità.

I tre componenti principali di PyTorch

PyTorch è una libreria relativamente completa e un modo per affrontarla consiste nel concentrarsi sui suoi tre componenti principali, riassunti nella Figura A.1.

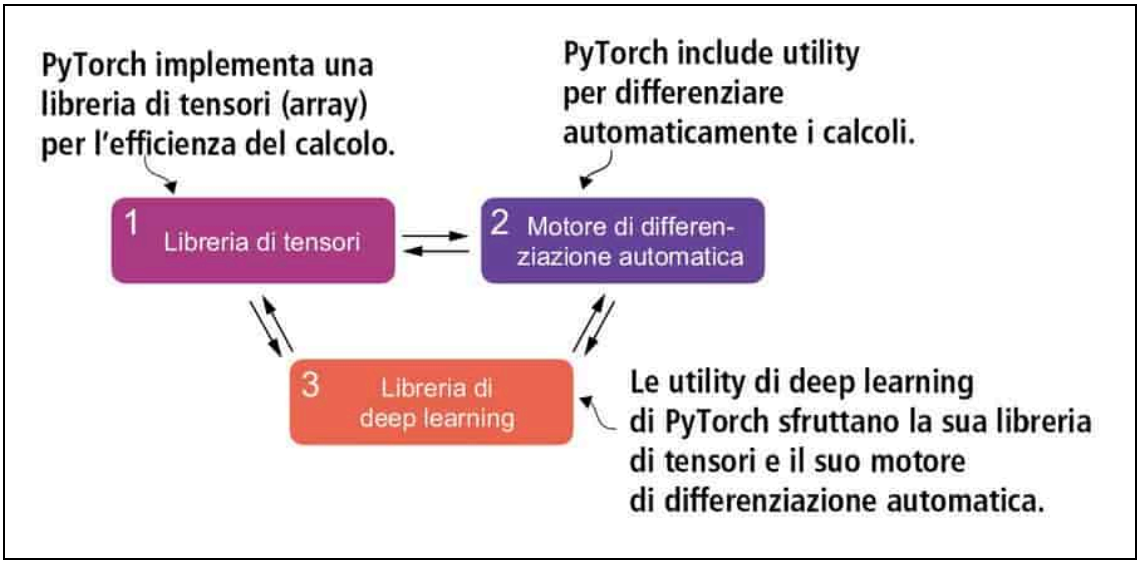


Figura A.1 I tre componenti principali di PyTorch includono una libreria di tensori, come elemento fondamentale per l'elaborazione; la differenziazione automatica per il fine-tuning del modello; funzioni di servizio di deep learning, che semplificano l'implementazione e l'addestramento di modelli a reti neurali profonde.

Innanzitutto, PyTorch è una *libreria di tensori* che estende il concetto della libreria di programmazione array-oriented NumPy con funzionalità aggiuntive che accelerano i calcoli su GPU, fornendo così un passaggio fluido tra CPU e GPU. In secondo luogo, PyTorch è un *motore di differenziazione automatica*, noto anche come "autograd", che consente il calcolo automatico dei gradienti per le operazioni sui tensori, semplificando la retropropagazione e il fine-tuning dei modelli. Infine, PyTorch è una *libreria di deep learning*. Offre blocchi costitutivi modulari, flessibili ed efficienti, che comprendono modelli pre-addestrati, funzioni loss e ottimizzatori, per la progettazione e l'addestramento di un'ampia gamma di modelli di deep learning, rivolti sia ai ricercatori sia agli sviluppatori.

Che cosa si intende per deep learning

Negli articoli di giornale, i modelli LLM spesso sono definiti “intelligenza artificiale”. Tuttavia, i modelli LLM sono anche un tipo di rete neurale profonda e PyTorch è una libreria di deep learning. Cerchiamo allora di riassumere la relazione tra questi termini, prima di procedere.

L’*intelligenza artificiale* consiste fondamentalmente nel creare sistemi di calcolo in grado di svolgere compiti che solitamente richiedono un’intelligenza umana. Questi compiti includono la comprensione del linguaggio naturale, il riconoscimento di schemi e la capacità di prendere decisioni. Nonostante i progressi significativi, l’intelligenza artificiale è ancora ben lontana dal raggiungere questo livello di intelligenza generale.

Il *machine learning* rappresenta un sottocampo dell’intelligenza artificiale (Figura A.2) che si concentra sullo sviluppo e sul miglioramento degli algoritmi di apprendimento. L’idea che è alla base del machine learning consiste nel consentire ai computer di apprendere dai dati e di fare previsioni o prendere decisioni senza essere programmati esplicitamente per svolgere tale attività. Ciò comporta lo sviluppo di algoritmi in grado di identificare schemi, apprendere dai dati storici e migliorare le proprie prestazioni nel tempo grazie a nuovi dati e ai feedback.

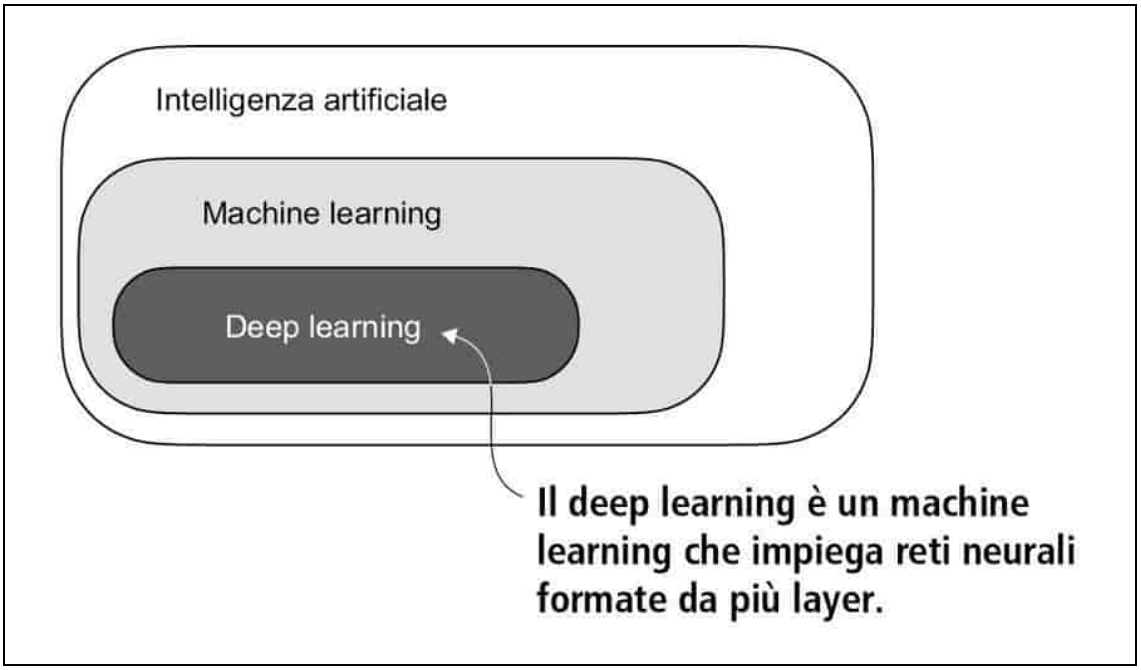


Figura A.2 Il deep learning è una sottocategoria del machine learning focalizzata sull'implementazione di reti neurali profonde. Il machine learning è una sottocategoria dell'intelligenza artificiale che si occupa di algoritmi che apprendono dai dati. L'intelligenza artificiale è il concetto più ampio: macchine in grado di eseguire attività che in genere richiedono un'intelligenza umana.

Il machine learning è stato parte integrante dell'evoluzione dell'intelligenza artificiale, alimentando molti dei progressi che vediamo oggi, inclusi i modelli LLM. Il machine learning è anche alla base di tecnologie come i sistemi di suggerimento utilizzati dai rivenditori online e dai servizi di streaming, il filtraggio dello spam via e-mail, il riconoscimento vocale negli assistenti virtuali e perfino le auto a guida autonoma. L'introduzione e poi gli avanzamenti del machine learning hanno notevolmente esteso le capacità dell'intelligenza artificiale, consentendole di andare oltre i rigidi sistemi basati su regole e di adattarsi a nuovi input o ad ambienti in evoluzione.

Il *deep learning* è una sottocategoria del machine learning che si concentra sull'addestramento e l'impiego di reti neurali profonde. Queste reti sono state originariamente ispirate dal funzionamento del cervello umano, in particolare dalle interconnessioni tra i neuroni. La qualifica "deep" del deep learning si riferisce ai molti layer di neuroni o "nodi artificiali" che consentono di modellare relazioni complesse e non lineari presenti nei dati. A differenza delle tecniche tradizionali di machine learning, che eccellono nel semplice riconoscimento di schemi, *pattern*, il deep learning è particolarmente efficace nella gestione di dati non strutturati, come immagini, audio o testi, quindi è particolarmente adatto ai modelli LLM.

La Figura A.3 riassume il tipico flusso di lavoro di modellazione predittiva (o *apprendimento con supervisione*) nel machine learning e nel deep learning.

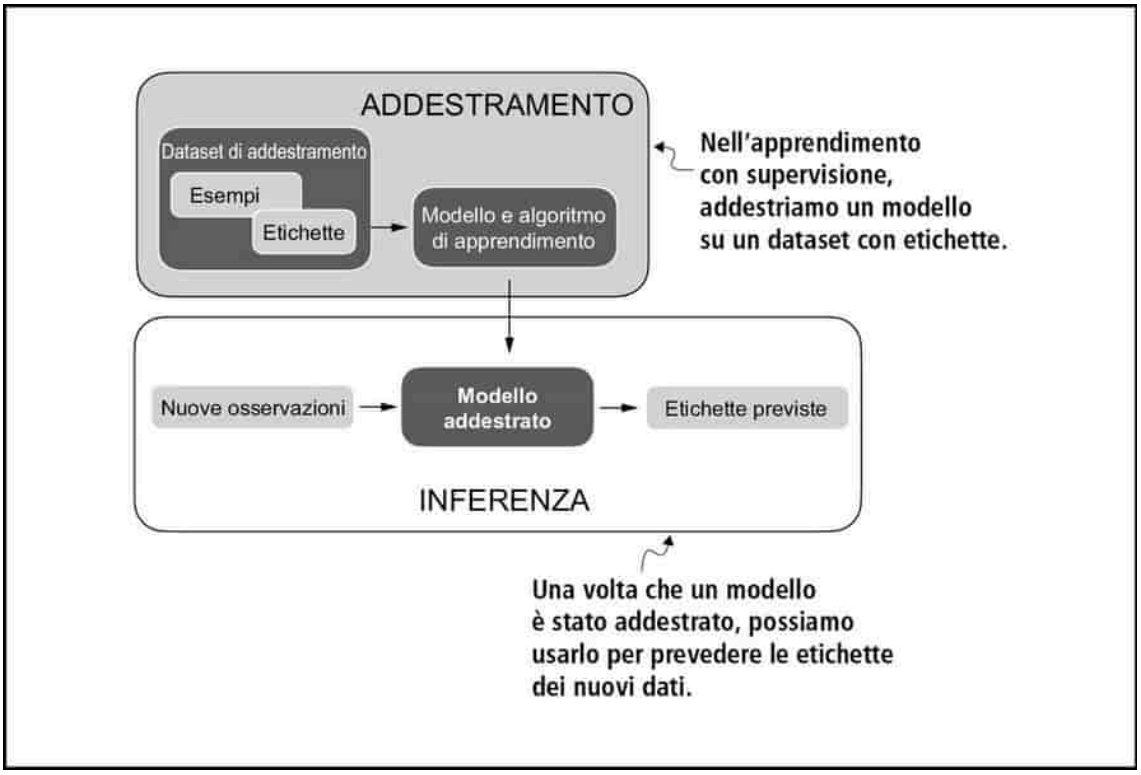


Figura A.3 Il flusso di lavoro di apprendimento con supervisione per la modellazione predittiva consiste in una fase di addestramento, in cui un modello studia esempi pre-etichettati di un set di addestramento. Il modello addestrato potrà poi essere utilizzato per prevedere le etichette di nuove osservazioni.

Utilizzando un algoritmo di apprendimento, un modello viene addestrato su un dataset costituito da esempi e dalle relative etichette, pre-compilate. Nel caso di un classificatore dello spam via e-mail, per esempio, il dataset di addestramento è costituito da e-mail e dalle relative etichette "spam" e "non spam" pre-identificate da un essere umano. Successivamente, il modello addestrato può essere applicato a nuove osservazioni (per esempio, a nuove e-mail) per prevedere la loro etichetta, ora sconosciuta: "spam" o "non spam". Naturalmente, vogliamo avere anche una valutazione del modello tra le fasi di addestramento e inferenza, per garantire che il modello stesso

soddisfi i criteri previsti, prima di impiegarlo in un'applicazione reale.

Se addestriamo i modelli LLM a classificare dei testi, il flusso di lavoro per il loro addestramento e utilizzo diviene simile a quello rappresentato nella Figura A.3, che vale anche se siamo interessati ad addestrare modelli LLM per generare testi. In questo caso, le etichette del pre-addestramento possono derivare dal testo stesso (il compito di previsione della parola successiva introdotto nel Capitolo 1). Durante l'inferenza, il modello LLM genererà un testo completamente nuovo (invece di prevedere le etichette), dato un prompt di input.

Installazione di PyTorch

PyTorch può essere installata come qualsiasi altra libreria o pacchetto Python. Tuttavia, poiché PyTorch è una libreria compatibile con le CPU e le GPU, l'installazione potrebbe richiedere qualche spiegazione in più.

> **Versione di Python**
> Molte librerie di calcolo scientifico non supportano la versione più recente di Python. Pertanto, quando installate PyTorch, vi consiglio di utilizzare una versione di Python che sia di una o due release "arretrata". Per esempio, se "oggi" l'ultima versione di Python è 3.13, è consigliabile utilizzare Python 3.11 o 3.12.

Per esempio, ci sono due versioni di PyTorch: una più snella che supporta solo il calcolo su CPU e una completa che supporta il calcolo sia su CPU sia su GPU. Se la vostra macchina ha una GPU compatibile CUDA che può essere utilizzata per il deep learning (idealmente, una NVIDIA T4, una RTX 2080 Ti o una scheda più recente), vi consiglio di installare la versione GPU. In ogni caso, il comando standard per installare PyTorch da terminale è:

```
pip install torch
```

Supponiamo che il vostro computer sia dotato di una GPU compatibile CUDA. In tal caso, vorrete installare una versione di PyTorch che ne supporti l'accelerazione, presupponendo che l'ambiente Python che state impiegando sia dotato di tutte le dipendenze necessarie (come `pip`).

NOTA

Al momento della stesura, PyTorch ha aggiunto anche il supporto sperimentale per le GPU AMD, tramite ROCm. Consultate `https://pytorch.org` per ulteriori istruzioni.

Per installare esplicitamente la versione di PyTorch compatibile CUDA, spesso è meglio specificare il CUDA in questione. Il sito web ufficiale di PyTorch (`https://pytorch.org`) fornisce i comandi necessari per installare PyTorch con supporto CUDA per diversi sistemi operativi. La Figura A.4 mostra un comando che installa PyTorch e le librerie `torchvision` e `torchaudio`, che sono opzionali per questo libro.

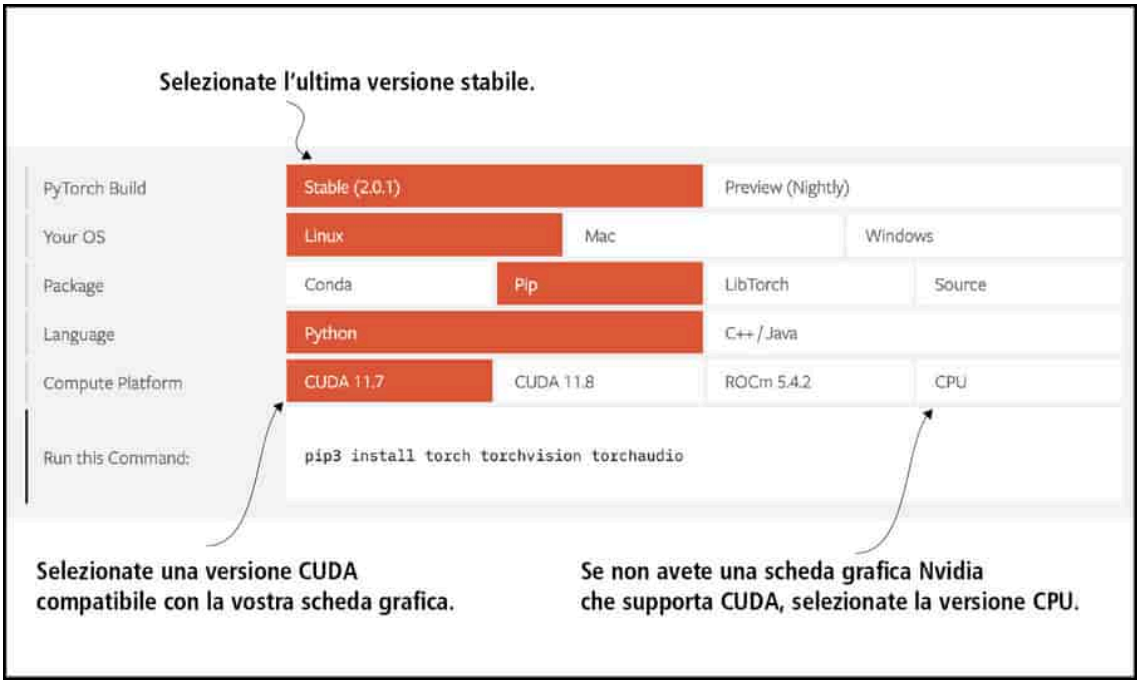


Figura A.4 Suggerimenti di installazione di PyTorch su https://pytorch.org per

personalizzare e selezionare il comando di installazione per il vostro sistema.

Per gli esempi, ho utilizzato PyTorch 2.4.0, quindi vi consiglio di utilizzare il seguente comando per installare la versione che garantisce la piena la compatibilità con questo libro:

```
pip install torch==2.4.0
```

Tuttavia, come ho già detto, in base al sistema operativo che impiegate, il comando di installazione potrebbe differire leggermente da questo. Quindi, vi consiglio di visitare `https://pytorch.org` e di usare il menu di installazione (Figura A.4) per selezionare il comando di installazione specifico per il vostro sistema operativo. Ricordatevi di sostituire `torch` con `torch==2.4.0` nel comando.

Per verificare la versione di PyTorch installata, eseguite il seguente codice in PyTorch:

```
import torch
torch.__version__
```

Questo codice produrrà il seguente output:

```
'2.4.0'
```

PyTorch e Torch

La libreria Python si chiama PyTorch principalmente perché è un seguito della libreria Torch adattata per Python. Torch è un framework di calcolo scientifico con un ampio supporto per gli algoritmi di machine learning, inizialmente creato utilizzando il linguaggio di programmazione Lua.

Per ulteriori consigli e istruzioni per configurare il vostro ambiente Python o installare le altre librerie utilizzate in questo libro, visitate il repository GitHub di questo libro all'indirizzo `https://github.com/rasbt/LLMs-from-scratch`.

Dopo aver installato PyTorch, potete verificare se l'installazione riconosce la GPU NVIDIA eseguendo il seguente codice in Python:

```
import torch
torch.cuda.is_available()
```

Otterrete:

```
True
```

Se il comando restituisce `True`, tutto è a posto. Se il comando restituisce `False`, il vostro computer potrebbe non avere una GPU compatibile, oppure PyTorch non la riconosce. Sebbene non sia necessario utilizzare una GPU per i capitoli iniziali di questo libro, che sono focalizzati sull'implementazione di modelli LLM per scopi didattici, e pur vero che essa può accelerare significativamente i calcoli correlati al deep learning.

Se non avete una GPU, ci sono diversi provider di cloud computing che offrono la possibilità di eseguire calcoli su GPU a pagamento. Un ambiente simile a Jupyter Notebook molto diffuso è Google Colab (`https://colab.research.google.com`), che (al momento della stesura) fornisce un accesso limitato alle GPU. Utilizzando il menu *Runtime*, potete selezionare una GPU, come mostrato nello screenshot rappresentato nella Figura A.5.

Figura A.5 Selezionate una GPU per Google Colab nel menu Runtime/Change Runtime Type.

PyTorch su Apple Silicon

Se avete un Apple Mac con chip Apple Silicon (come i modelli M1, M2, M3 o più recenti), potete usare le sue capacità per accelerare l'esecuzione del codice PyTorch. Per usare il chip Apple Silicon in PyTorch, dovete prima installare PyTorch come fareste normalmente. Poi, per verificare se il vostro Mac supporta l'accelerazione di PyTorch col chip Apple Silicon, potete eseguire questo semplice comando Python:

```
print(torch.backends.mps.is_available())
```

Se ottenete `True`, significa che il vostro Mac è dotato di un chip Apple Silicon che può essere utilizzato per accelerare il codice PyTorch.

Esercizio A.1

Installate e configurate PyTorch sul vostro computer.

Esercizio A.2

Eseguite il codice supplementare su `https://mng.bz/o05v`, che verifica se il vostro ambiente è configurato correttamente.

Che cosa sono i tensori

I tensori rappresentano un concetto matematico che generalizza i vettori e le matrici a dimensioni potenzialmente molto superiori. In altre parole, i tensori sono oggetti matematici che possono essere caratterizzati dal loro ordine (o rango), che fornisce il numero delle dimensioni. Per esempio, uno scalare (un numero) è un tensore di rango 0, un vettore è un tensore di rango 1 e una matrice è un tensore di rango 2, come mostra la Figura A.6.

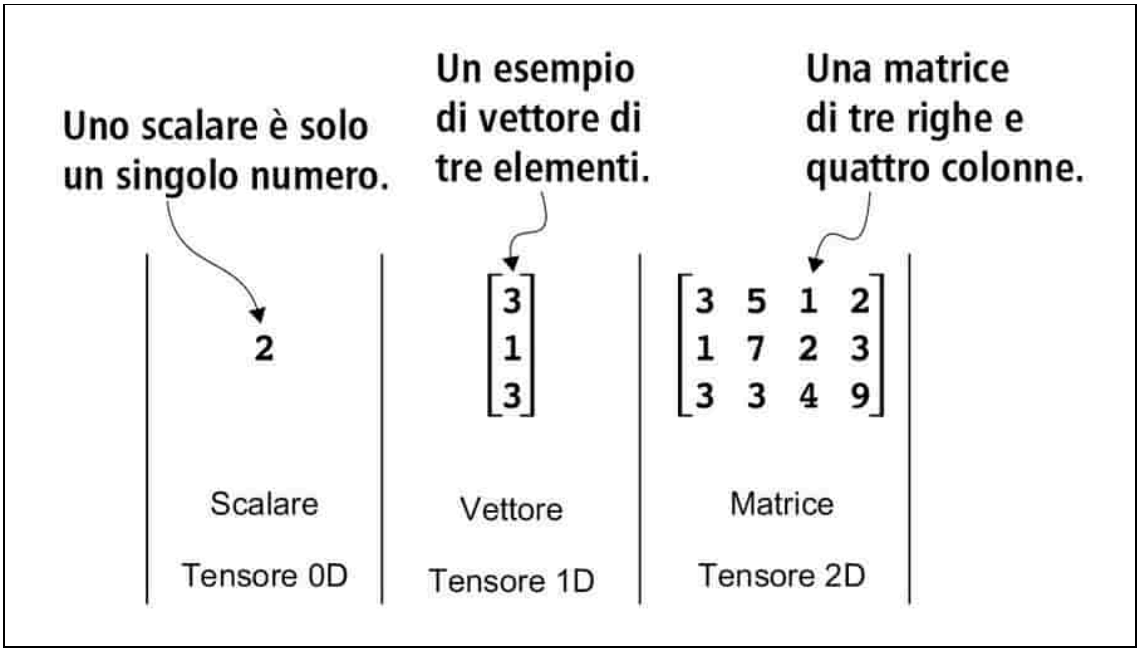


Figura A.6 Tensori di diversi ranghi. Qui 0D corrisponde al rango 0, 1D al rango 1 e 2D al rango 2. Un vettore di lunghezza 3, ovvero costituito da tre elementi, è sempre un tensore di rango 1.

Da una prospettiva computazionale, i tensori sono “contenitori di dati”. Per esempio, contengono dati multidimensionali, in cui ogni dimensione rappresenta una caratteristica differente. Le librerie di tensori come PyTorch possono creare, manipolare e calcolare questi array in modo efficiente. In questo contesto, una libreria di tensori funziona come una libreria di array.

I tensori di PyTorch sono simili agli array NumPy, ma hanno diverse funzionalità in più, che sono importanti per il deep learning. Per esempio, PyTorch aggiunge un motore di differenziazione automatica, semplificando il *calcolo dei gradienti* (come vedremo più avanti in questa appendice). I tensori di PyTorch supportano anche i calcoli su GPU per accelerare l’addestramento delle reti neurali profonde.

PyTorch con un'API di tipo NumPy

PyTorch adotta la maggior parte dell'API e della sintassi di NumPy per le sue operazioni sui tensori. Se non conoscete NumPy, potete trovare una breve panoramica dei concetti più rilevanti sul mio articolo *Scientific Computing in Python: Introduction to NumPy and Matplotlib* (https://sebastianraschka.com/blog/2020/numpy-intro.html).

Scalari, vettori, matrici e tensori

Come abbiamo visto, i tensori di PyTorch sono contenitori di dati per strutture simili ad array. Uno scalare è un tensore di dimensione zero (è solo un numero), un vettore è un tensore unidimensionale e una matrice è un tensore bidimensionale. Non esiste un termine specifico per i tensori di dimensioni superiori, quindi in genere ci riferiamo a un tensore tridimensionale come a un tensore 3D e così via. Possiamo creare oggetti della classe PyTorch `Tensor` utilizzando la funzione `torch.tensor()` come mostrato nel Listato A.1.

Listato A.1 Creazione di tensori di PyTorch.

```
import torch

tensor0d = torch.tensor(1) (1)

tensor1d = torch.tensor([1, 2, 3]) (2)

tensor2d = torch.tensor([[1, 2],
                         [3, 4]]
) (3)

tensor3d = torch.tensor([[[1, 2], [3, 4]],
                         [[5, 6], [7, 8]]]
) (4)
```

(1) Crea un tensore zero-dimensionale (scalare) da un intero Python.

(2) Crea un tensore unidimensionale (vettore) da una lista Python.

(3) Crea un tensore bidimensionale da una lista Python annidata.

(4) Crea un tensore tridimensionale da una lista Python annidata.

Tipi di dati dei tensori

PyTorch adotta il tipo di dati standard intero a 64 bit di Python. Possiamo accedere al tipo di dati di un tensore tramite l'attributo `.dtype` di un tensore:

```
tensor1d = torch.tensor([1, 2, 3])
print(tensor1d.dtype)
```

Questo codice produce il seguente output:

```
torch.int64
```

Se creiamo tensori usando float Python, PyTorch crea per default tensori con una precisione di 32 bit:

```
floatvec = torch.tensor([1.0, 2.0, 3.0])
print(floatvec.dtype)
```

L'output è il seguente:

```
torch.float32
```

Questa scelta è dovuta principalmente a un equilibrio tra precisione ed efficienza computazionale. Un numero in virgola mobile a 32 bit offre un'accuratezza sufficiente per la maggior parte delle attività di deep learning, consumando meno memoria e risorse computazionali rispetto a un numero in virgola mobile a 64 bit. Inoltre, le architetture GPU sono ottimizzate per i calcoli a 32 bit e l'utilizzo di questo tipo di dati può accelerare significativamente l'addestramento e le inferenze del modello.

Inoltre, potete modificare l'accuratezza utilizzando il metodo `.to` dei tensori. Il seguente codice ne mostra l'uso, cambiando un tensore intero a 64 bit in un tensore float a 32 bit:

```
floatvec = tensor1d.to(torch.float32)
print(floatvec.dtype)
```

Questo produce il seguente output:

```
torch.float32
```

Per maggiori informazioni sui diversi tipi di dati dei tensori disponibili in PyTorch, consultate la documentazione ufficiale su

https://pytorch.org/docs/stable/tensors.html.

Operazioni sui tensori PyTorch

Una trattazione esaustiva di tutte le operazioni e di tutti i comandi sui tensori PyTorch esula dagli scopi di questo libro. Tuttavia, descriverò brevemente le operazioni rilevanti per il libro.

Abbiamo già introdotto la funzione `torch.tensor()` per creare nuovi tensori:

```
tensor2d = torch.tensor([[1, 2, 3],
                         [4, 5, 6]]
)
print(tensor2d)
```

Questo codice produce il seguente output:

```
tensor([[1, 2, 3],
        [4, 5, 6]]
)
```

L'attributo `.shape` ci consente di accedere alla forma di un tensore:

```
print(tensor2d.shape)
```

L'output è il seguente:

```
torch.Size([2, 3])
```

Come potete vedere, `.shape` restituisce `[2, 3]`, il che significa che il tensore è formato da due righe e tre colonne. Per rimodellare il tensore in un tensore 3 × 2, possiamo usare il metodo `.reshape()`:

```
print(tensor2d.reshape(3, 2))
```

Questo codice produce il seguente output:

```
tensor([[1, 2],
        [3, 4],
        [5, 6]]
)
```

Tuttavia, il comando più comune per rimodellare i tensori in PyTorch è `.view()`:

```
print(tensor2d.view(3, 2))
```

L'output è il seguente:

```
tensor([[1, 2],
        [3, 4],
        [5, 6]]
)
```

Come abbiamo visto per `.reshape()` e `.view()`, in diversi casi PyTorch offre più forme sintattiche per eseguire lo stesso calcolo. Inizialmente, PyTorch ha seguito la convenzione sintattica Lua Torch, ma poi, cedendo alle richieste, è stata aggiunta la sintassi per renderla più simile a NumPy. La sottile differenza fra `.view()` e `.reshape()` in PyTorch risiede nella loro gestione della memoria: `.view()` richiede che i dati originali siano contigui, mentre `.reshape()` funziona sempre, copiando i dati se necessario per garantire la forma desiderata.

Ora, possiamo usare `.T` per trasporre un tensore, il che significa ribaltarlo lungo la diagonale. Notate che l'operazione è simile a rimodellare un tensore, come potete vedere dal seguente risultato:

```
print(tensor2d.T)
```

L'output è il seguente:

```
tensor([[1, 4],
        [2, 5],
        [3, 6]]
)
```

Infine, per moltiplicare due matrici in PyTorch si usa il metodo `.matmul()`:

```
print(tensor2d.matmul(tensor2d.T))
```

L'output è il seguente:

```
tensor([[14, 32],
        [32, 77]]
)
```

Tuttavia, possiamo anche impiegare l'operatore `@`, che realizza la stessa cosa in modo più compatto:

```
print(tensor2d @ tensor2d.T)
```

Questo codice produce il seguente output:

```
tensor([[14, 32],
        [32, 77]]
)
```

A coloro che desiderino esaminare tutte le diverse operazioni sui tensori disponibili in PyTorch (ma non ne abbiamo bisogno, in questo libro), consiglio di consultare la documentazione ufficiale su https://pytorch.org/docs/stable/tensors.html.

I modelli come grafi di calcoli

Ora esaminiamo il motore di differenziazione automatica di PyTorch, noto anche come *autograd*. Il sistema autograd di PyTorch fornisce funzioni per calcolare automaticamente i gradienti nei grafi di calcolo dinamici.

Un grafo di calcolo è un grafo orientato che consente di esprimere e visualizzare espressioni matematiche. Nel contesto del deep learning, un grafo di calcolo stabilisce la sequenza di calcoli necessari per calcolare l'output di una rete neurale: ne avremo bisogno per calcolare i gradienti richiesti per la retropropagazione, il principale algoritmo di addestramento per le reti neurali.

Esaminiamo ora un esempio concreto per illustrare il concetto di grafo di calcolo. Il codice riportato nel Listato A.2 implementa il passo forward (fase di previsione) di un semplice classificatore di regressione logistica, che può essere visto come una rete neurale mono-layer. Restituisce un punteggio tra 0 e 1, che va confrontato con l'etichetta della classe (0 o 1) nel calcolo del valore loss.

Listato A.2 Un passo forward nella regressione logistica.

```
import torch.nn.functional as F (1)

y = torch.tensor([1.0]) (2)
x1 = torch.tensor([1.1]) (3)
w1 = torch.tensor([2.2]) (4)
b = torch.tensor([0.0]) (5)
z = x1 * w1 + b (6)
a = torch.sigmoid(z) (7)

loss = F.binary_cross_entropy(a, y)
```

(1) Istruzione di importazione: una convenzione comune in PyTorch per evitare lunghe righe di codice.
(2) Etichetta vera.
(3) Caratteristica di input.
(4) Parametro del peso.
(5) Unità di bias.
(6) Input netto.
(7) Attivazione e output.

Se non tutti i componenti del codice precedente vi sembrano avere senso, non preoccupatevi. Lo scopo di questo esempio non consiste nell'implementare un classificatore di regressione logistica, ma piuttosto nell'illustrare come possiamo considerare una sequenza di calcoli come un grafo di calcolo (Figura A.7).

In effetti, PyTorch costruisce in background un grafo di calcolo di questo tipo, e possiamo usarlo per calcolare i gradienti di una funzione loss rispetto ai parametri del modello (in questo caso w_1 e b) per addestrare il modello.

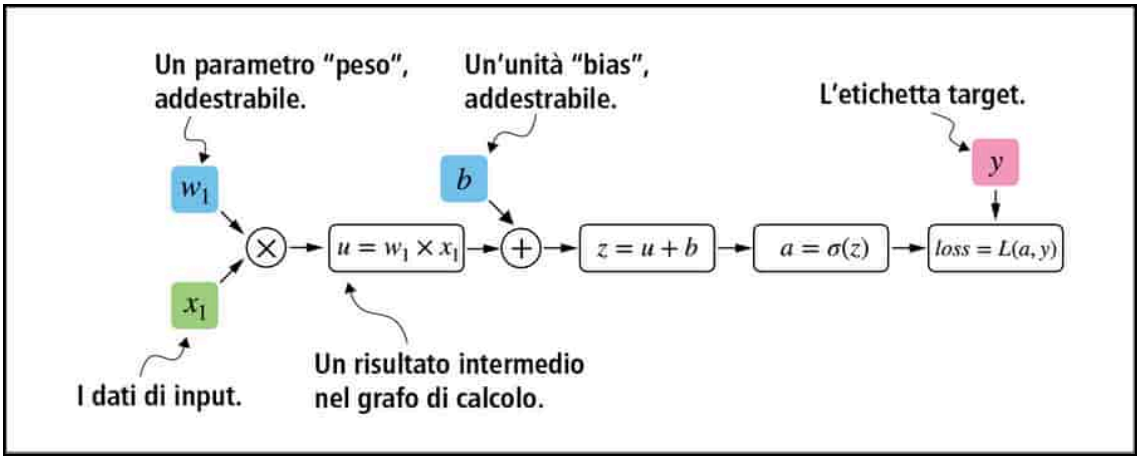


Figura A.7 Un passo forward di regressione logistica come grafo di calcolo. La caratteristica di input x_1 viene moltiplicata per un peso w_1 e fatta passare attraverso una

funzione di attivazione σ dopo aver aggiunto il bias. Il valore loss viene calcolato confrontando l'output del modello a con una data etichetta y.

L'ABC della differenziazione automatica

Se eseguiamo i calcoli in PyTorch, questo costruirà un grafo di calcolo, se uno dei suoi nodi terminali ha l'attributo `requires_grad` impostato su True. Ciò è utile se vogliamo calcolare i gradienti. I gradienti sono necessari quando si addestra una rete neurale tramite il noto algoritmo di retropropagazione (*backpropagation*), che può essere considerato un'implementazione della *regola della catena*, dal calcolo per reti neurali, illustrata nella Figura A.8.

Derivate parziali e gradienti

La Figura A.8 mostra le derivate parziali, che misurano la velocità con cui una funzione cambia rispetto a una delle sue variabili. Un *gradiente* è un vettore contenente tutte le derivate parziali di una funzione multivariata, ovvero una funzione che ha più di una variabile di input.

Se non avete familiarità con le derivate parziali, i gradienti o la regola della catena, non preoccupatevi. Per quanto ci riguarda, tutto ciò che dovete sapere è che la regola della catena è un modo per calcolare i gradienti di una funzione loss dati i parametri del modello in un grafo di calcolo. Ciò fornisce le informazioni necessarie per aggiornare ogni parametro per minimizzare la funzione loss, che è un mezzo per misurare le prestazioni del modello utilizzando un metodo come la discesa del gradiente. Rivedremo più avanti in questa stessa

appendice l'implementazione computazionale di questo ciclo di addestramento in PyTorch.

Come si è correlato tutto questo al motore di differenziazione automatica (autograd), il secondo componente della libreria PyTorch menzionato in precedenza? Il motore autograd di PyTorch costruisce un grafo di calcolo tracciando ogni operazione eseguita sui tensori. Quindi, richiamando la funzione `grad()`, possiamo calcolare il gradiente del valore loss relativo al parametro `w1` del modello, come mostrato nel Listato A.3.

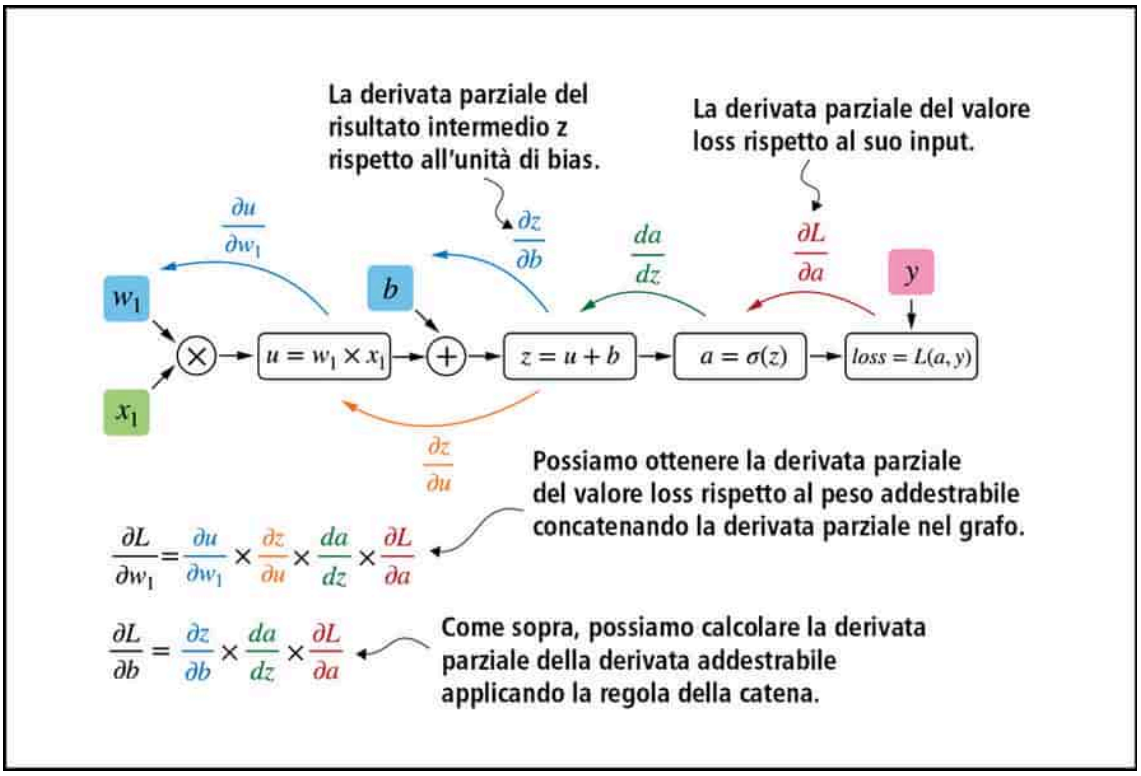


Figura A.8 Il modo più comune per calcolare i gradienti del valore loss in un grafo di calcolo comporta l'applicazione della regola della catena, da destra a sinistra, detta anche differenziazione automatica inversa del modello o retropropagazione. Partiamo dal layer di output (o dal valore loss) e procediamo a ritroso nella rete fino al layer di input. Lo facciamo per calcolare il gradiente del valore loss rispetto a ciascun parametro (pesi e bias) della rete, che informa l'aggiornamento di questi parametri durante l'addestramento.

Listato A.3 Calcolo dei gradienti tramite autograd.

```
import torch.nn.functional as F
from torch.autograd import grad

y = torch.tensor([1.0])
x1 = torch.tensor([1.1])
w1 = torch.tensor([2.2], requires_grad=True)
b = torch.tensor([0.0], requires_grad=True)

z = x1 * w1 + b
a = torch.sigmoid(z)

loss = F.binary_cross_entropy(a, y)

grad_L_w1 = grad(loss, w1, retain_graph=True) (1)
grad_L_b = grad(loss, b, retain_graph=True)
```

(1) Per default, per liberare memoria, PyTorch distrugge il grafo di calcolo dopo aver calcolato i gradienti. Tuttavia, poiché a breve riutilizzeremo questo grafo di calcolo, impostiamo `retain_graph=True`, in modo che rimanga in memoria.

I valori risultanti del valore loss dati i parametri del modello sono i seguenti:

```
print(grad_L_w1)
print(grad_L_b)
```

Questo codice produce il seguente output:

```
(tensor([-0,0898]),)
(tensor([-0,0817]),)
```

Qui, abbiamo utilizzato manualmente la funzione `grad()`, che può essere utile per sperimentare, per eseguire il debug e per la dimostrazione di concetti. Ma, all'atto pratico, PyTorch fornisce strumenti di livello più elevato per automatizzare questo processo. Per esempio, possiamo richiamare `.backward()` sul valore loss, e PyTorch calcolerà i gradienti di tutti i nodi foglia del grafo, che saranno memorizzati tramite gli attributi `.grad` dei tensori:

```
loss.backward()
print(w1.grad)
print(b.grad)
```

Gli output sono i seguenti:

```
(tensor([-0,0898]),)
(tensor([-0,0817]),)
```

Ho presentato molti concetti e potreste sentirvi sopraffatti, ma non preoccupatevi. Questo gergo è un mezzo per spiegare l'uso di autograd, ma tutto ciò che dovete sapere è che PyTorch si occupa di tutti i calcoli tramite il metodo `.backward()`: non avremo bisogno di calcolare manualmente nessuna derivata o gradiente.

Implementazione delle reti neurali multi-layer

Ora, ci concentreremo sull'uso della libreria PyTorch per implementare reti neurali profonde. Per partire da un esempio concreto, esaminiamo un perceptron multi-layer, una rete neurale completamente connessa, rappresentata nella Figura A.9.

Quando implementiamo una rete neurale in PyTorch, possiamo derivare una sottoclasse della classe `torch.nn.Module` per definire la nostra architettura di rete personalizzata. Questa classe base `Module` fornisce molte funzionalità, aiutando nella costruzione e nell'addestramento dei modelli. Per esempio, ci consente di incapsulare i layer e le operazioni e di tenere traccia dei parametri del modello.

All'interno di questa sottoclasse, definiamo i layer di rete nel costruttore `__init__` e specifichiamo come interagiscono i layer nel metodo `forward()`. Il metodo `forward()` descrive come i dati di input passano attraverso la rete e si uniscono a formare un grafo. Al contrario, il metodo `backward()`, che in genere non dobbiamo implementare noi stessi, viene utilizzato durante l'addestramento per calcolare i gradienti della funzione loss dati i parametri del modello. Il codice riportato nel Listato A.4 implementa un classico perceptron

multi-layer con due layer nascosti per illustrare un utilizzo tipico della classe `Module`.

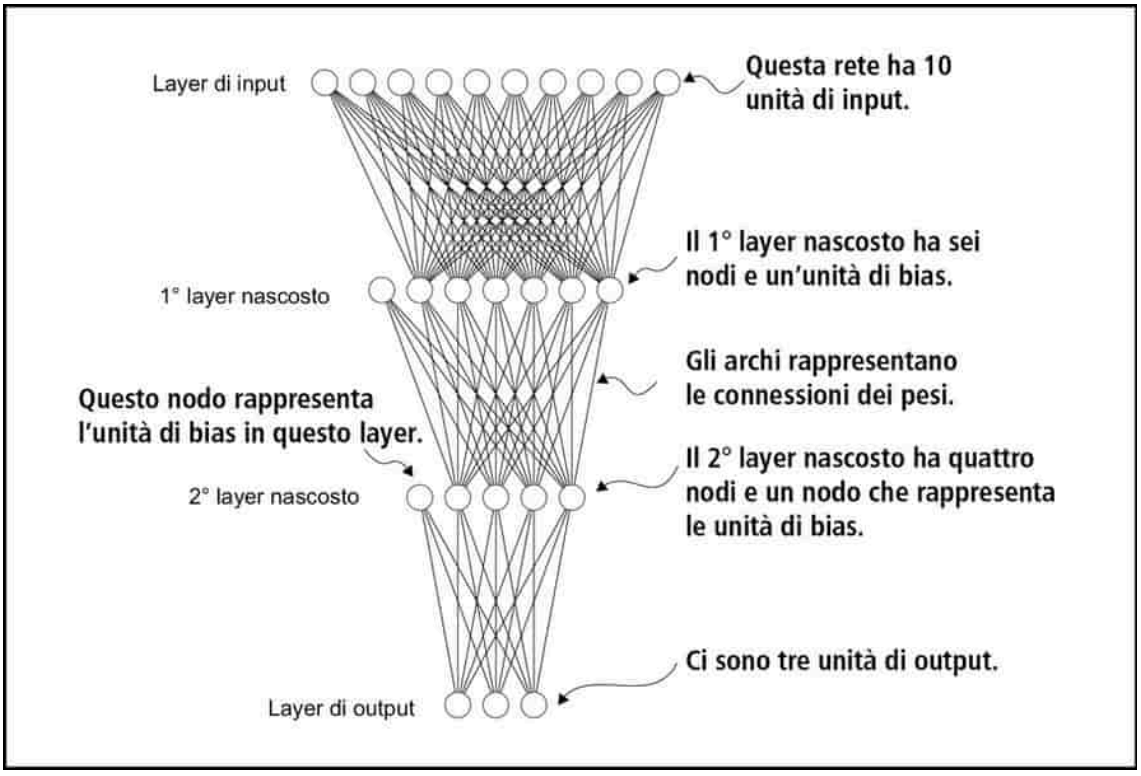


Figura A.9 Un perceptron multi-layer con due layer nascosti. Ogni nodo rappresenta un'unità nel rispettivo layer. A scopo illustrativo, ogni layer ha un numero molto piccolo di nodi.

Listato A.4 Un perceptron multi-layer con due layer nascosti.

```
class NeuralNetwork(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs): (1)
        super().__init__()

        self.layers = torch.nn.Sequential(

            # 1° layer nascosto
            torch.nn.Linear(num_inputs, 30), (2)
            torch.nn.ReLU(), (3)

            # 2nd layer nascosto
            torch.nn.Linear(30, 20), (4)
            torch.nn.ReLU(),

            # layer di output
            torch.nn.Linear(20, num_outputs),
```

```
        )

    def forward(self, x):
        logits = self.layers(x)
        return logits (5)
```

(1) Il fatto di codificare il numero di input e output come variabili ci consente di riutilizzare lo stesso codice per dataset con caratteristiche e classi in numero differente.

(2) Il layer lineare accetta come argomenti il numero di nodi di input e output.

(3) Le funzioni di attivazione non lineari sono posizionate tra i layer nascosti.

(4) Il numero di nodi di output di un layer nascosto deve corrispondere al numero di input del layer successivo.

(5) Gli output dell'ultimo layer sono chiamati *logit*.

Possiamo pertanto istanziare un nuovo oggetto rete neurale come segue:

```
model = NeuralNetwork(50, 3)
```

Prima di utilizzare questo nuovo oggetto `model`, possiamo chiamare `print()` sul modello per vedere un riepilogo della sua struttura:

```
print(model)
```

Questo codice produce il seguente output:

```
NeuralNetwork(
  (layers): Sequential(
     (0): Linear(in_features=50, out_features=30, bias=True)
     (1): ReLU()
     (2): Linear(in_features=30, out_features=20, bias=True)
     (3): ReLU()
     (4): Linear(in_features=20, out_features=3, bias=True)
  )
)
```

Notate che utilizziamo la classe `Sequential` quando implementiamo la classe `NeuralNetwork`. `Sequential` non è obbligatorio, ma può semplificare le cose se abbiamo una serie di layer che vogliamo eseguire in un ordine ben preciso, come in questo caso. In questo modo, dopo aver istanziato `self.layers = Sequential(...)` nel costruttore

`__init__`, dobbiamo solo richiamare `self.layers` invece di chiamare individualmente ciascun layer nel metodo `forward()` di `NeuralNetwork`.

Ora controlliamo il numero totale di parametri addestrabili di questo modello:

```
num_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print("Total number of trainable model parameters:", num_params)
```

Questo codice produce il seguente output:

```
Total number of trainable model parameters: 2213
```

Ogni parametro per il quale `requires_grad=True` viene considerato un parametro addestrabile e verrà aggiornato durante l'addestramento.

Nel caso del modello a rete neurale precedente, con due layer nascosti, questi parametri addestrabili sono contenuti nei layer `torch.nn.Linear`. Un layer `Linear` moltiplica gli input con una matrice dei pesi e aggiunge un vettore di bias. Questo a volte è definito layer *feedforward* o *completamente connesso*.

Dalla chiamata di `print(model)` che abbiamo eseguito, possiamo vedere che il primo layer `Linear` ha indice `0`. Possiamo accedere alla matrice dei parametri dei pesi corrispondente come segue:

```
print(model.layers[0].weight)
```

Questo codice produce il seguente output:

```
Parameter containing:
tensor([[ 0.1174, -0.1350, -0.1227, ...,  0.0275, -0.0520, -0.0192],
        [-0.0169,  0.1265,  0.0255, ..., -0.1247,  0.1191, -0.0698],
        [-0.0973, -0.0974, -0.0739, ..., -0.0068, -0.0892,  0.1070],
       ...,
        [-0.0681,  0.1058, -0.0315, ..., -0.1081, -0.0290, -0.1374],
        [-0.0159,  0.0587, -0.0916, ..., -0.1153,  0.0700,  0.0770],
        [-0.1019,  0.1345, -0.0176, ...,  0.0114, -0.0559, -0.0088]],
      requires_grad=True
)
```

Questa grande matrice non è mostrata nella sua interezza, quindi utilizziamo l'attributo `.shape` per mostrarne le dimensioni:

```
print(model.layers[0].weight.shape)
```

Il risultato è il seguente:

```
torch.Size([30, 50])
```

Allo stesso modo, potreste accedere al vettore di bias tramite `model.layers[0].bias`.

Qui la matrice dei pesi ha forma 30 × 50 e possiamo vedere che `requires_grad` è `True`, il che significa che i suoi elementi sono addestrabili: questa è l'impostazione di default per pesi e bias in `torch.nn.Linear`.

Se eseguite il codice precedente sul vostro computer, i numeri nella matrice dei pesi probabilmente saranno diversi da quelli mostrati. I pesi del modello sono inizializzati con piccoli numeri casuali, che differiscono ogni volta che istanziamo la rete. Nel deep learning, l'inizializzazione dei pesi del modello con piccoli numeri casuali ha lo scopo di evitare la simmetria durante l'addestramento. Altrimenti, i nodi eseguirebbero sempre le stesse operazioni e gli stessi aggiornamenti durante la retropropagazione, il che non consentirebbe alla rete di apprendere le mappature complesse dagli input agli output.

Tuttavia, se da un lato vogliamo continuare a utilizzare piccoli numeri casuali come valori iniziali per i pesi dei layer, possiamo rendere riproducibile l'inizializzazione dei numeri casuali eseguendo il seeding del generatore di numeri casuali di PyTorch tramite `manual_seed`:

```
torch.manual_seed(123)
model = NeuralNetwork(50, 3)
print(model.layers[0].weight)
```

Il risultato è il seguente:

```
Parameter containing:
tensor([[-0.0577,  0.0047, -0.0702, ...,  0.0222,  0.1260,  0.0865],
        [ 0.0502,  0.0307,  0.0333, ...,  0.0951,  0.1134, -0.0297],
        [ 0.1077, -0.1108,  0.0122, ...,  0.0108, -0.1049, -0.1063],
        ...,
        [-0.0787,  0.1259,  0.0803, ...,  0.1218,  0.1303, -0.1351],
        [ 0.1359,  0.0175, -0.0673, ...,  0.0674,  0.0676,  0.1058],
        [ 0.0790,  0.1343, -0.0293, ...,  0.0344, -0.0971, -0.0509]],
       requires_grad=True
)
```

Ora che abbiamo dedicato del tempo all'ispezione dell'istanza `NeuralNetwork`, vediamo brevemente come viene utilizzata tramite il passo forward:

```
torch.manual_seed(123)
X = torch.rand((1, 50))
out = model(X)
print(out)
```

Il risultato è il seguente:

```
tensor([[-0.1262,  0.1080, -0.1792]], grad_fn=<AddmmBackward0>)
```

Nel codice precedente, abbiamo generato un singolo esempio di addestramento casuale `x` come "input giocattolo" (notate che la nostra rete si aspetta vettori di caratteristiche 50-dimensionali) e lo abbiamo immesso nel modello, ottenendo tre punteggi. Quando richiamiamo `model(x)`, eseguiamo automaticamente il passo forward del modello.

Il passo forward si riferisce al calcolo dei tensori di output fornito dai tensori di input. Ciò comporta il passaggio dei dati di input attraverso tutti i layer della rete neurale, a partire dal layer di input, attraverso i layer nascosti e infine al layer di output.

I tre numeri restituiti corrispondono a un punteggio assegnato a ciascuno dei tre nodi di output. Notate che anche il tensore di output include un valore `grad_fn`.

Qui, `grad_fn=<AddmmBackward0>` rappresenta l'ultima funzione utilizzata per calcolare una variabile nel grafo di calcolo. In particolare, `grad_fn=<AddmmBackward0>` significa che il tensore che stiamo ispezionando è stato creato tramite un'operazione di moltiplicazione e somma di matrici. PyTorch utilizzerà queste informazioni per calcolare i gradienti durante la retropropagazione. La parte `<AddmmBackward0>` di `grad_fn=<AddmmBackward0>` specifica l'operazione eseguita. In questo caso, si tratta di un'operazione `Addmm`, che sta per una moltiplicazione (`mm`) seguita da un'addizione (`Add`) di matrici.

Se vogliamo usare semplicemente una rete, senza addestramento o retropropagazione (per esempio se la usiamo per la previsione dopo l'addestramento), costruire questo grafo di calcolo per la retropropagazione può essere uno spreco, in quanto esegue calcoli non necessari e consuma memoria. Quindi, quando usiamo un modello per l'inferenza (ovvero, per fare previsioni) anziché per l'addestramento, la best practice consiste nell'usare il gestore di contesto `torch.no_grad()`, che dice a PyTorch che non ha bisogno di tenere traccia dei gradienti, il che può comportare risparmi significativi in termini di memoria e tempo di calcolo:

```
with torch.no_grad():
    out = model(X)
print(out)
```

Il risultato è il seguente:

```
tensor([[-0.1262,  0.1080, -0.1792]])
```

In PyTorch, è prassi comune programmare i modelli in modo che restituiscano gli output dell'ultimo layer (`logits`) senza passarli a una funzione di attivazione non lineare. Questo perché le funzioni loss comunemente utilizzate da PyTorch combinano in una singola classe l'operazione `softmax()` (o `sigmoid()` per la classificazione binaria) con il valore loss di verosimiglianza logaritmica negativa. I motivi sono due: efficienza numerica e stabilità. Quindi, se vogliamo calcolare le probabilità di appartenenza a una classe, dobbiamo richiamare esplicitamente la funzione `softmax()`:

```
with torch.no_grad():
    out = torch.softmax(model(X), dim=1)
print(out)
```

Questo codice produce il seguente output:

```
tensor([[0.3113, 0.3934, 0.2952]]))
```

Ora i valori possono essere interpretati come probabilità di appartenenza alle classi (a somma 1). I valori sono più o meno uguali

per questo input casuale, il che è normale per un modello inizializzato in modo casuale e senza addestramento.

Impostazione di data loader efficienti

Prima di poter addestrare un modello, dobbiamo parlare della creazione di data loader efficienti in PyTorch, che itereremo durante l’addestramento. La Figura A.10 illustra l’’idea generale alla base del caricamento dei dati in PyTorch.

Seguendo la Figura A.10, implementeremo una classe `Dataset` personalizzata, che utilizzeremo per creare un set di addestramento e un set di test che utilizzeremo poi per creare i data loader. Iniziamo creando un semplice set giocattolo di addestramento di cinque esempi con due caratteristiche ciascuno. Insieme agli esempi di addestramento, creiamo anche un tensore contenente le etichette delle classi corrispondenti: tre esempi appartengono alla classe 0 e due appartengono alla classe 1. Inoltre, creiamo un set di test composto da due elementi.

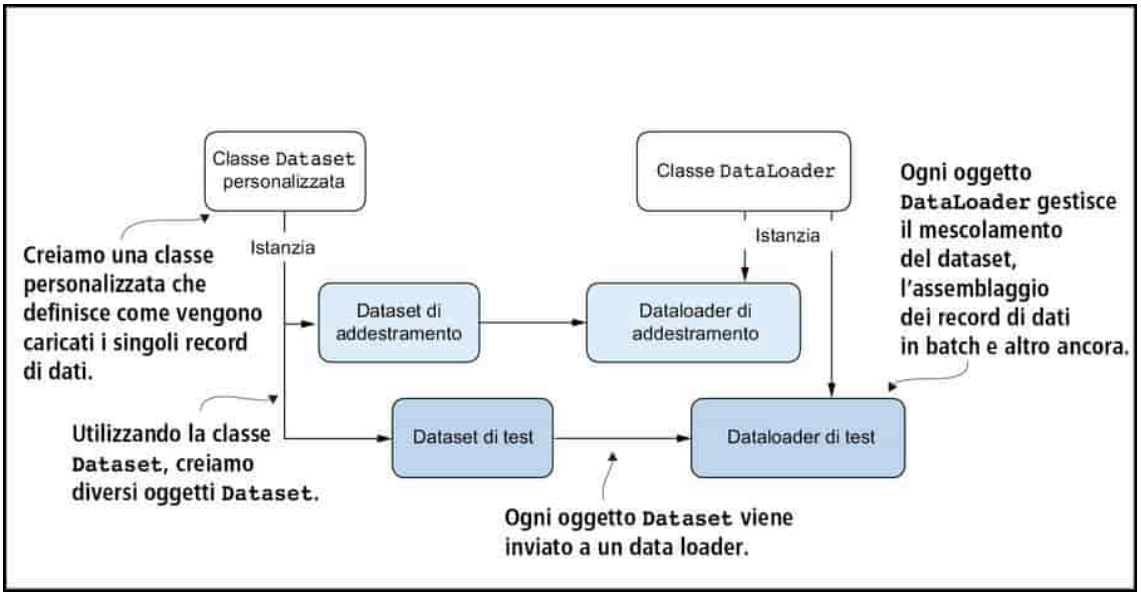


Figura A.10 PyTorch implementa una classe Dataset (utilizzata per istanziare oggetti che definiscono come viene caricato ogni record di dati) e una classe DataLoader (che gestisce come i dati vengono mescolati e assemblati in batch).

Il codice per creare questo dataset è rappresentato nel Listato A.5.

Listato A.5 Creazione di un piccolo dataset giocattolo.

```
X_train = torch.tensor([[-1.2, 3.1],
                        [-0.9, 2.9],
                        [-0.5, 2.6],
                        [2.3, -1.1],
                        [2.7, -1.5]]
)
y_train = torch.tensor([0, 0, 0, 1, 1])

X_test = torch.tensor([[-0.8, 2.8],
                       [2.6, -1.6],]
)
y_test = torch.tensor([0, 1])
```

NOTA

PyTorch richiede che le etichette delle classi inizino con l'etichetta 0 e che la più grande etichetta della classe non superi il numero di nodi di output meno 1 (poiché in Python il conteggio degli indici parte da zero). Quindi, se abbiamo classi con etichette 0, 1, 2, 3 e 4, il layer di output della rete neurale dovrebbe essere costituito da cinque nodi.

Ora, creiamo una classe per il dataset, `ToyDataset`, derivandola dalla classe PyTorch `Dataset`, come mostrato nel Listato A.6.

Listato A.6 Definizione di una classe Dataset personalizzata

```
from torch.utils.data import Dataset

class ToyDataset(Dataset):
    def __init__(self, X, y):
        self.features = X
        self.labels = y

    def __getitem__(self, index): (1)
        one_x = self.features[index]
        one_y = self.labels[index]
        return one_x, one_y

    def __len__(self):
        return self.labels.shape[0] (2)

train_ds = ToyDataset(X_train, y_train)
test_ds = ToyDataset(X_test, y_test)
```

(1) Istruzioni per recuperare un record di dati e l'etichetta corrispondente.

(2) Istruzioni per restituire la lunghezza totale del dataset.

Lo scopo di questa classe `ToyDataset` personalizzata è quello di istanziare un `DataLoader` PyTorch. Ma prima di arrivare a questo passo, ripercorriamo brevemente la struttura generale del codice di `ToyDataset`.

In PyTorch, i tre componenti principali di una classe `Dataset` personalizzata sono il costruttore `__init__`, il metodo `__getitem__` e il metodo `__len__` (vedi Listato A.6). Nel metodo `__init__`, impostiamo gli attributi cui potremo accedere più avanti dai metodi `__getitem__` e `__len__` metodi. Questi potrebbero essere percorsi di file, oggetti file, connettori a database e così via. Poiché abbiamo creato un dataset di tensori in memoria, ci basta assegnare `X` e `y` a questi attributi, che sono segnaposto per i nostri oggetti tensori.

Nel metodo `__getitem__`, definiamo le istruzioni per restituire esattamente un elemento dal dataset, tramite un indice. Ciò si

riferisce alle caratteristiche e all'etichetta della classe corrispondente a un singolo esempio di addestramento o istanza di test. Sarà il data loader a fornire questo indice, come vedremo tra poco.

Infine, il metodo `__len__` contiene istruzioni per recuperare la lunghezza del dataset. Qui, utilizziamo l'attributo `.shape` di un tensore per restituire il numero di righe dell'array delle caratteristiche. Nel caso del set di addestramento, abbiamo cinque righe, come possiamo verificare:

```
print(len(train_ds))
```

Il risultato è il seguente:

```
5
```

Ora che abbiamo definito una classe PyTorch `Dataset` che possiamo usare per il dataset giocattolo, possiamo usare la classe PyTorch `DataLoader` per trarvi campioni, come mostrato nel Listato A.7.

Listato A.7 Istanziazione dei data loader.

```
from torch.utils.data import DataLoader

torch.manual_seed(123)

train_loader = DataLoader(dataset=train_ds,
                          batch_size=2,
                          shuffle=True,
                          num_workers=0
)

test_loader = DataLoader(dataset=test_ds,
                         batch_size=2,
                         shuffle=False,
                         num_workers=0
)
```

Dopo aver istanziato il data loader di addestramento, possiamo iterare su di esso. L'iterazione su `test_loader` funziona in modo simile, ma la ometto per brevità:

```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}:", x, y)
```

Il risultato è il seguente:

```
Batch 1: tensor([[-1.2000, 3.1000],
                 [-0.5000, 2.6000]]) tensor([0, 0])
Batch 2: tensor([[ 2.3000, -1.1000],
                 [-0.9000, 2.9000]]) tensor([1, 0])
Batch 3: tensor([[ 2.7000, -1.5000]]) tensor([1])
```

Come potete vedere in base all'output precedente, `train_loader` esegue l'iterazione sul set di addestramento, visitando ogni esempio esattamente una volta. Questo è noto come un'*epoca* di addestramento. Poiché abbiamo eseguito il seeding del generatore di numeri casuali usando `torch.manual_seed(123)`, dovreste ottenere esattamente lo stesso ordine di mescolamento degli esempi di addestramento. Tuttavia, se eseguirete una seconda volta un'iterazione sul dataset, vedrete che l'ordine di mescolamento cambierà. Ciò è auspicabile, per evitare che le reti neurali profonde finiscano in cicli di aggiornamento ripetitivi, durante l'addestramento.

Qui abbiamo specificato dimensioni del batch pari a 2, ma il terzo batch contiene un singolo esempio (perché abbiamo cinque esempi di addestramento e 5 non è divisibile per 2). All'atto pratico, il fatto di avere un batch sostanzialmente più piccolo come ultimo batch in un'epoca di addestramento può disturbare la convergenza. Per evitarlo, impostate `drop_last=True`, che eliminerà l'ultimo batch in ogni epoca, come mostrato nel Listato A.8.

Listato A.8 Un data loader di addestramento che scarta l'ultimo batch.

```
train_loader = DataLoader(dataset=train_ds,
                          batch_size=2,
                          shuffle=True,
                          num_workers=0,
                          drop_last=True
)
```

Ora, iterando sul loader di addestramento, possiamo vedere che l'ultimo batch è stato omesso:

```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}:", x, y)
```

Il risultato è il seguente:

```
Batch 1: tensor([[-0.9000, 2.9000],
                 [ 2.3000, -1.1000]]) tensor([0, 1])
Batch 2: tensor([[ 2.7000, -1.5000],
                 [-0.5000, 2.6000]]) tensor([1, 0])
```

Infine, parliamo dell'impostazione `num_workers=0` in `DataLoader`. Questo parametro della funzione `DataLoader` di PyTorch è fondamentale per parallelizzare il caricamento e la pre-elaborazione dei dati. Quando `num_workers` è impostato a `0`, il caricamento dei dati verrà eseguito nel processo principale e non in processi worker separati. Questo potrebbe sembrare poco problematico, ma può portare a rallentamenti significativi durante l'addestramento del modello quando alleniamo reti più grandi su una GPU. Invece di concentrarsi esclusivamente sull'elaborazione del modello di deep learning, la CPU deve impiegare del tempo anche per caricare e pre-elaborare i dati. Di conseguenza, la GPU può rimanere inattiva in attesa che la CPU completi queste attività. Al contrario, quando `num_workers` è impostato su un numero maggiore di `0`, si avviano più processi worker per caricare i dati in parallelo, liberando il processo principale che potrà concentrarsi sull'addestramento del modello e utilizzare al meglio le risorse del sistema (Figura A.11).

Tuttavia, se lavoriamo con dataset molto piccoli, impostare `num_workers` su `1` o più potrebbe non essere necessario, poiché il tempo di addestramento totale richiede comunque solo una frazione di secondo. Quindi, se lavorate con dataset minuscoli o ambienti interattivi come i notebook Jupyter, l'aumento di `num_workers` potrebbe non fornirvi alcuna accelerazione. Potrebbe, in realtà, causare anche alcuni problemi. Un potenziale problema è il sovraccarico del fatto di dover avviare più processi worker, che potrebbe richiedere più tempo del caricamento dei dati quando il dataset è molto piccolo.

Inoltre, per i notebook Jupyter, impostando `num_workers` su un valore maggiore di `0` potreste riscontare problemi relativi alla condivisione delle risorse fra i processi, con conseguenti errori o crash del notebook. Pertanto, è essenziale scegliere un compromesso e prendere una decisione oculata sull'impostazione del parametro `num_workers`. Se usato correttamente, può essere uno strumento utile, ma deve essere adattato alle dimensioni specifiche del dataset e alle caratteristiche dell'ambiente di calcolo.

Per esperienza, l'impostazione `num_workers=4` solitamente offre prestazioni ottimali su molti dataset reali, ma tutto dipende dall'hardware e dal codice utilizzato per caricare l'esempio definito nella classe `Dataset`.

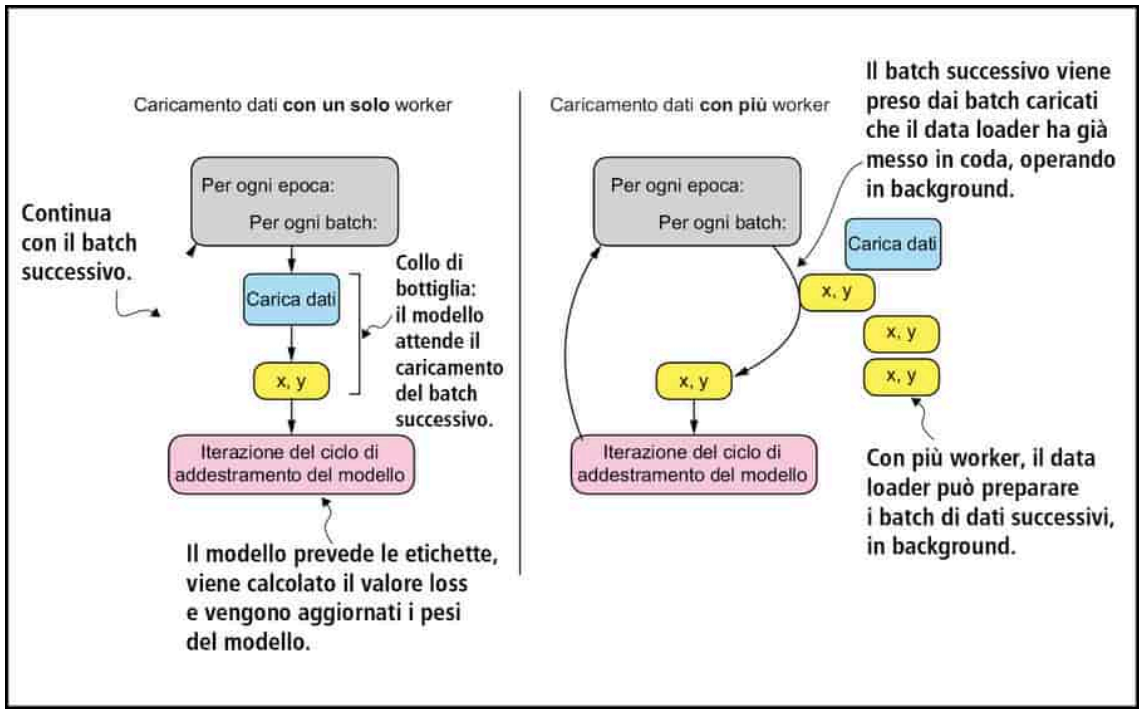


Figura A.11 Il caricamento dei dati senza altri worker (impostazione num_workers=0) crea un collo di bottiglia nel caricamento dei dati, in quanto il modello rimane inattivo finché

non viene caricato il batch successivo (a sinistra). Se vengono abilitati più worker, il data loader può mettere in coda il batch successivo (a destra).

Un tipico ciclo di addestramento

Ora addestriamo una rete neurale sul dataset giocattolo. Il seguente listato mostra il codice per l'addestramento.

Listato A.9 Addestramento della rete neurale in PyTorch.

```
import torch.nn.functional as F

torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2) (1)
optimizer = torch.optim.SGD(model.parameters(), lr=0.5) (2)

num_epochs = 3
for epoch in range(num_epochs):
    model.train()

    for batch_idx, (features, labels) in enumerate(train_loader):
        logits = model(features)

        loss = F.cross_entropy(logits, labels)

        optimizer.zero_grad() (3)
        loss.backward() (4)
        optimizer.step() (5)

        ### LOGGING
        print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
              f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
              f" | Train Loss: {loss:.2f}")
    model.eval()
    # Qui collocate il codice opzionale per la valutazione del modello
```

(1) Il dataset ha due caratteristiche e due classi.

(2) L'ottimizzatore deve sapere quali parametri ottimizzare.

(3) Imposta a `0` i gradienti del round precedente, per evitare un accumulo indesiderato di gradienti.

(4) Calcola i gradienti del valore loss, in base ai parametri del modello.

(5) L'ottimizzatore utilizza i gradienti per aggiornare i parametri del modello.

L’esecuzione di questo codice produce i seguenti output:

```
Epoch: 001/003 | Batch 000/002 | Train Loss: 0.75
Epoch: 001/003 | Batch 001/002 | Train Loss: 0.65
Epoch: 002/003 | Batch 000/002 | Train Loss: 0.44
Epoch: 002/003 | Batch 001/002 | Train Loss: 0.13
Epoch: 003/003 | Batch 000/002 | Train Loss: 0.03
Epoch: 003/003 | Batch 001/002 | Train Loss: 0.00
```

Come potete vedere, il valore loss raggiunge lo `0` dopo tre epoche, segno che il modello è riuscito a convergere sul set di addestramento. Qui, inizializziamo un modello con due input e due output, perché il dataset giocattolo ha due caratteristiche di input e due classi da prevedere. Abbiamo utilizzato un ottimizzatore a discesa stocastica del gradiente (`SGD`) con un tasso di apprendimento (`lr`) di `0.5`. Il tasso di apprendimento è un iperparametro, il che significa che è un’impostazione regolabile con cui dobbiamo sperimentare in base all’osservazione del valore loss. Idealmente, vogliamo scegliere un tasso di apprendimento tale che il valore loss converga entro un certo numero di epoche: il numero di epoche è un altro iperparametro da scegliere.

Esercizio A.3
Quanti parametri ha la rete neurale prodotta nel Listato A.9?

All’atto pratico, per trovare le impostazioni ottimali degli iperparametri utilizziamo spesso un terzo set, di validazione. Un set di validazione è simile a un set di test. Tuttavia, mentre un set di test si utilizza una sola volta per evitare di distorcere la valutazione, di solito il set di validazione viene utilizzato più volte, per modificare le impostazioni del modello.

Abbiamo anche introdotto le nuove impostazioni `model.train()` e `model.eval()`,utilizzate per mettere il modello in modalità di addestramento e di valutazione. Ciò è necessario per i componenti che si comportano diversamente durante l’addestramento e l’inferenza, come i layer di *dropout* o di *normalizzazione del batch*.

Dato che nella classe `NeuralNetwork` non abbiamo il dropout o altri componenti interessati da queste impostazioni, l'utilizzo di `model.train()` e `model.eval()` è ridondante nel codice precedente. Tuttavia, è buona norma includerli comunque, per evitare comportamenti imprevisti quando dovessimo modificare l'architettura del modello o riutilizzare il codice per addestrare un modello differente.

Come abbiamo già visto in precedenza, passiamo i `logits` direttamente alla funzione loss `cross_entropy()`, che applicherà la funzione `softmax()` internamente per motivi di efficienza e di stabilità numerica. Poi, richiamando `loss.backward()`, calcoleremo i gradienti nel grafo di calcolo costruito in background da PyTorch. Il metodo `optimizer.step()` utilizzerà poi i gradienti per aggiornare i parametri del modello, con lo scopo di minimizzare il valore loss. Nel caso dell'ottimizzatore `SGD`, ciò significa moltiplicare i gradienti per il tasso di apprendimento e aggiungere ai parametri il gradiente negativo scalato.

NOTA

Per evitare un accumulo di gradienti, è importante includere una chiamata `optimizer.zero_grad()` in ogni round di aggiornamento, per reimpostare i gradienti a `0`. In caso contrario, i gradienti si accumuleranno, il che potrebbe essere indesiderato.

Dopo aver addestrato il modello, possiamo utilizzarlo per fare previsioni:

```
model.eval()
with torch.no_grad():
    outputs = model(X_train)
print(outputs)
```

I risultati sono i seguenti:

```
tensor([[ 2.8569, -4.1618],
        [ 2.5382, -3.7548],
        [ 2.0944, -3.1820],
        [-1.4814,  1.4816],
        [-1.7176,  1.7342]]
)
```

Per ottenere le probabilità di appartenenza alle classi, possiamo utilizzare la funzione PyTorch `softmax()`:

```
torch.set_printoptions(sci_mode=False)
probas = torch.softmax(outputs, dim=1)
print(probas)
```

Questo produce il seguente output:

```
tensor([[0,9991, 0,0009],
        [0,9982, 0,0018],
        [0,9949, 0,0051],
        [0,0491, 0,9509],
        [0,0307, 0,9693]]
)
```

Consideriamo la prima riga nell'output precedente. Qui, il primo valore (la prima colonna) dice che l'esempio ha una probabilità del 99.91 percento di appartenere alla classe 0 e una probabilità dello 0.09 percento di appartenere alla classe 1. Ho usato `set_printoptions()` per ottenere output più leggibili.

Possiamo convertire questi valori in previsioni di etichette delle classi utilizzando la funzione PyTorch `argmax()`, che restituisce l'indice con il valore più elevato in ogni riga (se impostiamo `dim=1`; impostando `dim=0` restituirebbe invece il valore più elevato in ogni colonna):

```
predictions = torch.argmax(probas, dim=1)
print(predictions)
```

Questo codice produce il seguente output:

```
tensor([0, 0, 0, 1, 1])
```

Notate che non è necessario calcolare le probabilità `softmax()` per ottenere le etichette delle classi. Potremmo applicare la funzione `argmax()` anche direttamente ai `logits` (`outputs`):

```
predictions = torch.argmax(outputs, dim=1)
print(predictions)
```

L'output è il seguente:

```
tensor([0, 0, 0, 1, 1])
```

Qui, abbiamo calcolato le etichette previste per il set di addestramento; essendo questo relativamente piccolo, potremmo confrontarlo con le vere etichette di addestramento a occhio, e vedere che il modello è corretto al 100 percento. Possiamo ricontrollarlo usando l'operatore di confronto `==`:

```
predictions == y_train
```

I risultati sono i seguenti:

```
tensor([True, True, True, True, True])
```

Utilizzando `torch.sum()`, possiamo contare il numero di previsioni corrette:

```
torch.sum(predictions == y_train)
```

L'output è il seguente:

```
5
```

Poiché il dataset è composto da cinque esempi di addestramento, abbiamo cinque previsioni corrette su cinque, con un'accuratezza pari a 5 / 5 × 100 percento = 100 percento.

Per generalizzare il calcolo dell'accuratezza della previsione, implementiamo una funzione `compute_accuracy()`, mostrata nel Listato A.10.

Listato A.10 Una funzione per calcolare l'accuratezza della previsione.

```
def compute_accuracy(model, dataloader):

    model = model.eval()
    correct = 0.0
    total_examples = 0

    for idx, (features, labels) in enumerate(dataloader):
        with torch.no_grad():
            logits = model(features)

        predictions = torch.argmax(logits, dim=1)
        compare = labels == predictions (1)
        correct += torch.sum(compare) (2)
        total_examples += len(compare) (3)

    return (correct / total_examples).item()
```

(1) Restituisce un tensore di valori `True`/`False` a seconda del fatto che le etichette corrispondano.
(2) La somma conta il numero di valori `True`.
(3) La frazione delle previsioni corrette, un valore compreso tra `0` e `1`. `.item()` restituisce il valore del tensore come un float Python.

Il codice esegue iterazioni su un data loader per calcolare il numero e la frazione delle previsioni corrette. Quando lavoriamo con grandi dataset, in genere possiamo richiamare il modello solo su una piccola parte del dataset, a causa dei limiti della memoria. Questa funzione `compute_accuracy()` è un metodo generale che si adatta a dataset di dimensioni arbitrarie, poiché a ogni iterazione, il blocco del dataset ricevuto dal modello ha le stesse dimensioni del batch visualizzato durante l'addestramento. La funzione `compute_accuracy()` è simile a quella che abbiamo usato quando abbiamo convertito i `logits` in etichette di classi.

Possiamo quindi applicare la funzione all'addestramento:

```
print(compute_accuracy(model, train_loader))
```

Il risultato è il seguente:

```
1.0
```

Allo stesso modo, possiamo applicare la funzione al set di test:

```
print(compute_accuracy(model, test_loader))
```

Questo codice produce il seguente output:

```
1.0
```

Salvataggio e caricamento dei modelli

Ora che abbiamo addestrato il modello, vediamo come salvarlo in modo da poterlo riutilizzare in seguito. Ecco il modo consigliato per

salvare e caricare i modelli in PyTorch:

```
torch.save(model.state_dict(), "model.pth")
```

Qui, `state_dict` del modello è un oggetto dizionario Python che mappa ogni layer del modello ai suoi parametri addestrabili (pesi e bias); `"model.pth"` è un nome arbitrario per il file del modello che verrà salvato su disco. Possiamo dargli qualsiasi nome ed estensione, anche se `.pth` e `.pt` sono le estensioni più comuni.

Una volta salvato il modello, possiamo ripristinarlo dal disco:

```
model = NeuralNetwork(2, 2)
model.load_state_dict(torch.load("model.pth"))
```

La funzione `torch.load("model.pth")` legge il file `"model.pth"` e ricostruisce l'oggetto dizionario Python contenente i parametri del modello, mentre la funzione `model.load_state_dict()` applica questi parametri al modello, ripristinando di fatto lo stato di apprendimento di quando lo abbiamo salvato.

La riga `model = NeuralNetwork(2, 2)` non è strettamente necessaria, se si esegue questo codice nella stessa sessione in cui è stato salvato un modello. Tuttavia, l'ho inclusa per illustrare che abbiamo bisogno di avere un'istanza del modello in memoria per applicare i parametri salvati. Qui, l'architettura `NeuralNetwork(2, 2)` deve corrispondere esattamente al modello salvato.

Ottimizzazione delle prestazioni di addestramento con le GPU

Ora, vediamo come utilizzare le GPU, che accelerano l'addestramento delle reti neurali profonde rispetto alle normali CPU. Innanzitutto, esamineremo i concetti di base dell'uso delle GPU in PyTorch. Quindi, addestreremo un modello su una singola GPU. Infine, esamineremo l'addestramento distribuito su più GPU.

PyTorch su dispositivi GPU

Modificare il ciclo di addestramento per eseguirlo su una GPU è relativamente semplice, e richiede interventi solo su tre righe di codice. Prima di apportare le modifiche, è fondamentale comprendere il concetto alla base dei calcoli su GPU in PyTorch. In PyTorch, un dispositivo è il luogo in cui si verificano i calcoli e risiedono i dati. La CPU e la GPU sono esempi di dispositivi. Un tensore PyTorch risiede in un dispositivo e le sue operazioni vengono eseguite su quello stesso dispositivo.

Vediamo come funziona la cosa. Supponendo che abbiate installato una versione di PyTorch compatibile con l'impiego di GPU (come descritto in precedenza in questa stessa appendice), possiamo verificare che il runtime supporti effettivamente il calcolo su GPU tramite la seguente riga di codice:

```
print(torch.cuda.is_available())
```

Il risultato è il seguente:

```
True
```

Ora, supponiamo di avere due tensori da sommare; questo calcolo verrà eseguito sulla CPU, per default:

```
tensor_1 = torch.tensor([1., 2., 3.])
tensor_2 = torch.tensor([4., 5., 6.])
print(tensor_1 + tensor_2)
```

Questo produrrà il seguente output:

```
tensor([5., 7., 9.])
```

Ora possiamo usare il metodo `.to()`, lo stesso che usiamo per cambiare il tipo di dati di un tensore (vedi il Capitolo 2) per trasferire questi tensori su una GPU ed eseguire lì l'addizione:

```
tensor_1 = tensor_1.to("cuda")
tensor_2 = tensor_2.to("cuda")
print(tensor_1 + tensor_2)
```

L'output è il seguente:

```
tensor([5., 7., 9.], device='cuda:0')
```

Il tensore risultante ora include le informazioni sul dispositivo, `device='cuda:0'`, il che significa che i tensori risiedono sulla prima GPU. Se la vostra macchina è dotata di più GPU, potete specificare a quale GPU desiderate trasferire i tensori. Potete farlo indicando l'ID del dispositivo nel comando. Per esempio, potete usare `.to("cuda:0")`, `.to("cuda:1")` e così via.

Tuttavia, tutti i tensori devono essere sullo stesso dispositivo, altrimenti, il calcolo fallirà, come in questo caso in cui un tensore risiede sulla CPU e l'altro sulla GPU:

```
tensor_1 = tensor_1.to("cpu")
print(tensor_1 + tensor_2)
```

I risultati sono i seguenti:

```
RuntimeError Traceback (most recent call last)
<ipython-input-7-4ff3c4d20fc3> in <cell line: 2>()
1 tensor_1 = tensor_1.to("cpu")
----> 2 print(tensor_1 + tensor_2)
RuntimeError: Expected all tensors to be on the same device, but found at least
two devices, cuda:0 and cpu!
```

In sintesi, dobbiamo solo trasferire i tensori sullo stesso dispositivo GPU, e PyTorch si occuperà del resto.

Addestramento su una sola GPU

Ora che abbiamo familiarità col trasferimento dei tensori alla GPU, possiamo modificare il ciclo di addestramento per eseguirlo su una GPU. Questo passo richiede interventi solo su tre righe di codice, come mostra il Listato A.11.

Listato A.11 Un ciclo di addestramento su una GPU.

```
torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)

device = torch.device("cuda") (1)
model = model.to(device) (2)

optimizer = torch.optim.SGD(model.parameters(), lr=0.5)

num_epochs = 3
```

```
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, labels) in enumerate(train_loader):
        features, labels = features.to(device), labels.to(device) (3)
        logits = model(features)
        loss = F.cross_entropy(logits, labels) # Loss function

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        ### LOGGING
        print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
              f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
              f" | Train/Val Loss: {loss:.2f}")

    model.eval()
    # Collocare qui il codice opzionale di valutazione del modello
```

(1) Definisce una variabile `device` che di default è una GPU.

(2) Trasferisce il modello sulla GPU.

(3) Trasferisce i dati sulla GPU.

L’esecuzione del codice precedente produrrà quanto segue, simile ai risultati ottenuti sulla CPU:

```
Epoch: 001/003 | Batch 000/002 | Train/Val Loss 0.75
Epoch: 001/003 | Batch 001/002 | Train/Val Loss 0.65
Epoch: 002/003 | Batch 000/002 | Train/Val Loss 0.44
Epoch: 002/003 | Batch 001/002 | Train/Val Loss 0.13
Epoch: 003/003 | Batch 000/002 | Train/Val Loss 0.03
Epoch: 003/003 | Batch 001/002 | Train/Val Loss 0.00
```

Possiamo usare `.to("cuda")` invece di `device = torch.device("cuda")`. Trasferire un tensore a `"cuda"` invece che a `torch.device("cuda")` funziona altrettanto bene ed è più breve. Possiamo anche modificare l’istruzione per rendere lo stesso codice eseguibile su una CPU se non è disponibile una GPU. Questa è considerata una best practice quando si condivide il codice PyTorch:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Nel caso del ciclo di addestramento ora modificato, probabilmente non noteremo un’accelerazione dovuta al trasferimento della memoria dalla CPU alla GPU. Tuttavia, possiamo aspettarci un’accelerazione significativa nell’addestramento di reti neurali profonde, in particolare dei modelli LLM.

PyTorch su macOS

Se avete un Apple Mac con chip Apple Silicon (come i modelli M1, M2, M3 o più recenti) invece di un PC con una GPU Nvidia, potete cambiare…

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

… in …

```
device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")
```

Esercizio A.4

Confrontate il tempo di esecuzione della moltiplicazione di matrici su una CPU e su una GPU. A quali dimensioni (delle matrici) iniziate a vedere che la GPU è più veloce di una CPU? Suggerimento: in Jupyter usate il comando `%timeit` per confrontare i tempi di esecuzione. Per esempio, date le matrici `a` e `b`, eseguite il comando `%timeit a @ b` in una nuova cella del notebook.

Addestramento su più GPU

L'addestramento distribuito prevede la suddivisione dell'addestramento del modello su più GPU e macchine. Perché ne abbiamo bisogno? Anche potendo addestrare un modello su una singola GPU e macchina, il processo potrebbe richiedere molto tempo. Il tempo di addestramento può essere ridotto in modo significativo distribuendo il processo di addestramento su più macchine, ciascuna con, potenzialmente, più GPU. Ciò è particolarmente interessante nelle fasi sperimentali dello sviluppo, in cui potrebbero essere necessarie numerose iterazioni di addestramento per perfezionare i parametri e l'architettura del modello.

NOTA

Per questo libro, non è richiesto l'accesso o l'uso di più GPU. Questo paragrafo è dedicato a coloro che sono interessati al funzionamento del calcolo multi-GPU in PyTorch.

Cominciamo col caso più elementare di addestramento distribuito: la strategia `DistributedDataParallel` (DDP) di PyTorch consente di attuare

il parallelismo suddividendo i dati di input tra i dispositivi disponibili ed elaborando simultaneamente questi sottoinsiemi di dati.

Come funziona? PyTorch avvia un processo a sé su ogni GPU, e ogni processo riceve e conserva una copia del modello; queste copie saranno sincronizzate durante l'addestramento. Per illustrare l'operazione, supponiamo di avere due GPU che vogliamo usare per addestrare una rete neurale (Figura A.12).

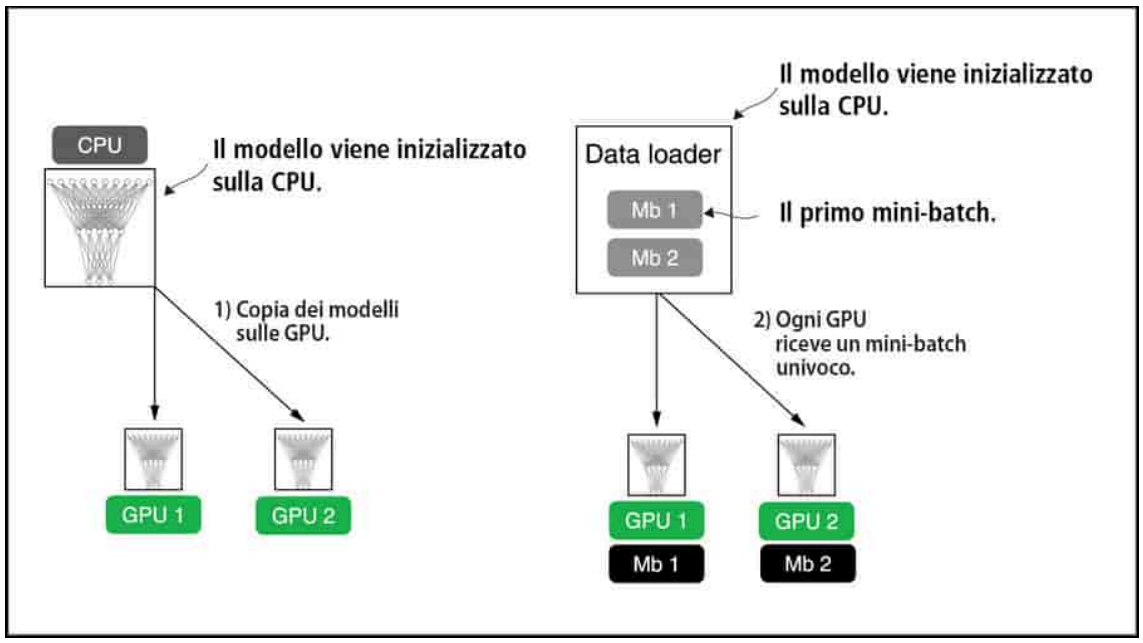


Figura A.12 Il trasferimento del modello e dei dati in DDP comporta due passi chiave. Innanzitutto, creiamo una copia del modello su ciascuna GPU. Quindi dividiamo i dati di input in mini-batch univoci che passiamo a ciascuna copia del modello.

Ognuna delle due GPU riceverà una copia del modello. Quindi, in ogni iterazione di addestramento, ogni modello riceverà un mini-batch dal data loader. Possiamo usare un `DistributedSampler` per garantire che ogni GPU riceva un mini-batch differente, senza sovrapposizioni.

Poiché ogni copia del modello vedrà un campione differente dei dati di addestramento, le copie del modello restituiranno `logits` differenti come output e calcoleranno gradienti differenti durante il

passo backward. Di questi gradienti viene calcolata la media e vengono sincronizzati durante l'addestramento, per aggiornare i modelli. In questo modo, ci assicuriamo che i modelli non divergano, come mostra la Figura A.13.

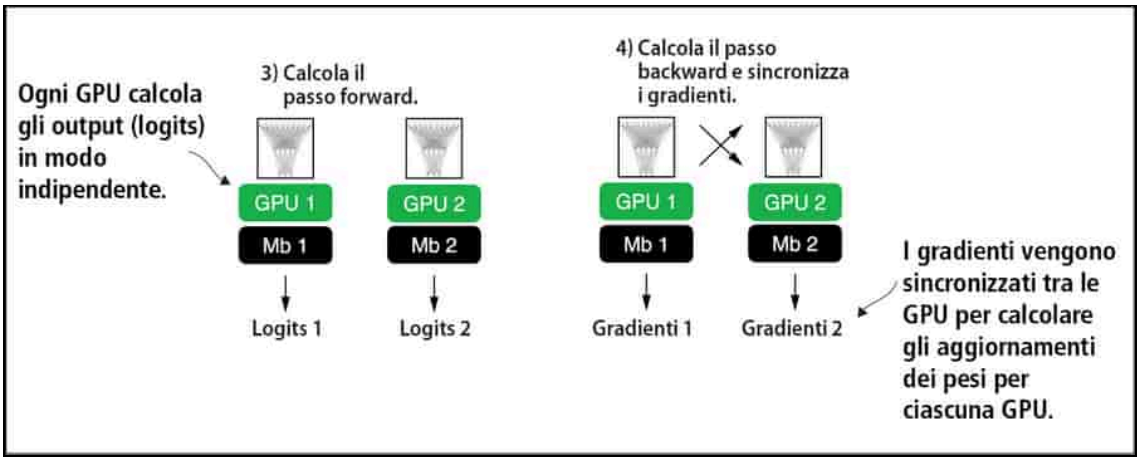


Figura A.13 I passi in avanti e all'indietro, in DDP, vengono eseguiti indipendentemente su ogni GPU, col suo sottoinsieme di dati. Una volta completati i passi in avanti e all'indietro, i gradienti di ogni copia del modello (di ogni GPU) vengono sincronizzati. Ciò garantisce che ogni copia del modello abbia i pesi aggiornati allo stesso modo.

Il vantaggio di DDP è la maggiore velocità nell'elaborazione del dataset rispetto a una singola GPU. A parte un piccolo overhead di comunicazione tra i dispositivi, DDP può teoricamente elaborare un'epoca di addestramento nella metà del tempo, impiegando due GPU invece di una sola. L'efficienza temporale aumenta col numero delle GPU, consentendoci di elaborare un'epoca otto volte più velocemente, se abbiamo otto GPU (e così via).

NOTA

DDP non funziona correttamente in ambienti Python interattivi, come i notebook Jupyter, che non gestiscono il multiprocessing nello stesso modo in cui lo fa uno script Python. Pertanto, il seguente codice deve essere eseguito come uno script, non all'interno di un'interfaccia come Jupyter. DDP deve generare più processi, e ogni processo deve avere la propria istanza di un interprete Python.

Vediamo ora come funzionano le cose in pratica. Per brevità, mi concentro sulle parti principali del codice che devono essere adattate per l'addestramento con DDP. Tuttavia, coloro che desiderano eseguire il codice sulla propria macchina multi-GPU o su un'istanza cloud a scelta dovrebbero utilizzare lo script autonomo fornito nel repository GitHub di questo libro, all'indirizzo `https://github.com/rasbt/LLMs-from-scratch`.

Per prima cosa importiamo alcuni sub-moduli, classi e funzioni per l'addestramento distribuito di PyTorch, come mostrato nel Listato A.12.

Listato A.12 Utility PyTorch per l'addestramento distribuito.

```
import torch.multiprocessing as mp
from torch.utils.data.distributed import DistributedSampler
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.distributed import init_process_group, destroy_process_group
```

Prima di approfondire le modifiche apportate per rendere l'addestramento compatibile con DDP, esaminiamo brevemente l'utilizzo di queste nuove utility, di cui abbiamo bisogno insieme alla classe `DistributedDataParallel`.

Il sub-modulo `multiprocessing` di PyTorch contiene funzioni come `multiprocessing.spawn()`, che utilizzeremo per generare più processi e per applicare una funzione a più input in parallelo. La utilizzeremo per generare un processo di addestramento per le GPU. Se generiamo più processi di addestramento, avremo bisogno di un modo per suddividere il dataset tra questi processi. Per questo, utilizzeremo `DistributedSampler`.

`init_process_group()` e `destroy_process_group()` vengono utilizzate per inizializzare e uscire dalle unità di addestramento distribuite. La funzione `init_process_group()` deve essere richiamata all'inizio dello script di addestramento per inizializzare un gruppo di processi e

`destroy_process_group()` deve essere richiamata alla fine dello script di addestramento per distruggere un dato gruppo di processi e rilasciarne le risorse. Il codice riportato nel Listato A.13 mostra come questi nuovi componenti vengono utilizzati per implementare l'addestramento DDP per il modello `NeuralNetwork` che abbiamo implementato in precedenza.

Listato A.13 Addestramento del modello con la strategia DistributedDataParallel.

```
def ddp_setup(rank, world_size):
    os.environ["MASTER_ADDR"] = "localhost" (1)
    os.environ["MASTER_PORT"] = "12345" (2)
    init_process_group(backend="nccl", (3)
                       rank=rank, (4)
                       world_size=world_size (5)
    )
    torch.cuda.set_device(rank) (6)

def prepare_dataset():
    # insert dataset preparation code
    train_loader = DataLoader(dataset=train_ds,
                              batch_size=2,
                              shuffle=False, (7)
                              pin_memory=True, (8)
                              drop_last=True,
                              sampler=DistributedSampler(train_ds) (9)
    )
    return train_loader, test_loader

def main(rank, world_size, num_epochs): (10)
    ddp_setup(rank, world_size)
    train_loader, test_loader = prepare_dataset()
    model = NeuralNetwork(num_inputs=2, num_outputs=2)
    model.to(rank)
    optimizer = torch.optim.SGD(model.parameters(), lr=0.5)
    model = DDP(model, device_ids=[rank])
    for epoch in range(num_epochs):
        for features, labels in train_loader:
            features, labels = features.to(rank), labels.to(rank) (11)
            # collocare qui il codice di predizione e retropropagazione del
modello
            print(f"[GPU{rank}] Epoch: {epoch+1:03d}/{num_epochs:03d}"
                  f" | Batchsize {labels.shape[0]:03d}"
                  f" | Train/Val Loss: {loss:.2f}")
    model.eval()
    train_acc = compute_accuracy(model, train_loader, device=rank)
    print(f"[GPU{rank}] Training accuracy", train_acc)
    test_acc = compute_accuracy(model, test_loader, device=rank)
    print(f"[GPU{rank}] Test accuracy", test_acc)
    destroy_process_group()  (12)

if __name__ == "__main__":
    print("Number of GPUs available:", torch.cuda.device_count())
```

```
    torch.manual_seed(123)
    num_epochs = 3
    world_size = torch.cuda.device_count()
    mp.spawn(main, args=(world_size, num_epochs), nprocs=world_size) (13)
```

(1) Indirizzo del nodo principale.

(2) Qualsiasi porta libera sulla macchina.

(3) `nccl` sta per NVIDIA Collective Communication Library.

(4) `rank` si riferisce all'indice della GPU che vogliamo utilizzare.

(5) `world_size` è il numero di GPU da utilizzare.

(6) Imposta il dispositivo GPU corrente su cui verranno allocati i tensori e su cui verranno eseguite le operazioni.

(7) Ora è il Distributed-Sampler a occuparsi del mescolamento.

(8) Abilita un trasferimento di memoria più rapido durante l'addestramento su GPU.

(9) Suddivide il dataset in sottoinsiemi distinti e non sovrapposti per ciascun processo (GPU).

(10) La funzione `main` che esegue l'addestramento del modello.

(11) `rank` è l'ID della GPU.

(12) Ripulisce l'allocazione delle risorse.

(13) Avvia la funzione `main` utilizzando più processi, dove `nprocs=world_size` significa "un processo per GPU".

Prima di eseguire questo codice, riassumiamo come funziona, tenendo conto delle annotazioni. Abbiamo una clausola `__name__ == "__main__"` contenente le istruzioni da impiegare quando eseguiamo il codice come script Python anziché importarlo come modulo. Questo codice visualizza il numero di GPU disponibili usando `torch.cuda.device_count()`, imposta un *seed* casuale per la riproducibilità e quindi genera nuovi processi usando la funzione PyTorch `multiprocessesing.spawn()`. Qui, la funzione `spawn()` avvia un processo per ogni GPU, data l'impostazione `nproces = world_size`, dove le dimensioni

del “mondo” sono pari al numero di GPU disponibili. Questa funzione `spawn()` avvia il codice nella funzione `main()`, che definiamo nello stesso script con alcuni argomenti aggiuntivi forniti tramite `args`. Notate che la funzione `main()` ha un argomento `rank` che non includiamo nella chiamata a `mp.spawn()`. Questo perché `rank`, che si riferisce all’ID del processo che utilizziamo come ID della GPU, viene già passato automaticamente.

La funzione `main()` imposta l’ambiente distribuito tramite `ddp_setup()` (un’altra funzione che abbiamo definito), carica i set di addestramento e test, imposta il modello ed esegue l’addestramento. Rispetto all’addestramento a GPU singola, ora trasferiamo il modello e i dati al dispositivo di destinazione tramite `.to(rank)`, che utilizziamo per fare riferimento all’ID del dispositivo GPU. Inoltre, includiamo il modello in un `DDP`, che consente la sincronizzazione dei gradienti tra le diverse GPU durante l’addestramento. Dopo che l’addestramento è terminato e abbiamo valutato i modelli, utilizziamo `destroy_process_group()` per uscire in modo pulito dall’addestramento distribuito e per liberare le risorse impegnate.

In precedenza, ho detto che ogni GPU riceverà un sub-campione differente dei dati di addestramento. Per assicurarci questo, impostiamo `sampler=DistributedSampler(train_ds)` nel loader di addestramento.

L’ultima funzione è `ddp_setup()`. Imposta l’indirizzo e la porta del nodo principale per consentire le comunicazioni tra i diversi processi, inizializza il gruppo di processi col backend NCCL (progettato per le comunicazioni fra GPU) e imposta il `rank` (identificatore di processo) e le dimensioni del “mondo” (numero totale di processi). Infine, specifica il dispositivo GPU corrispondente al rank del processo di addestramento.

Selezione delle GPU disponibili su una macchina multi-GPU

Se desiderate limitare il numero di GPU utilizzate per l'addestramento su una macchina multi-GPU, il modo più semplice consiste nell'utilizzare la variabile d'ambiente `CUDA_VISIBLE_DEVICES`. Supponiamo che la vostra macchina abbia più GPU e che vogliate utilizzare una sola GPU, per esempio la GPU con indice 0. Invece di `python nome_script.py`, potete eseguire la seguente riga di codice dal terminale:

```
CUDA_VISIBLE_DEVICES=0 python nome_script.py
```

Oppure, se la vostra macchina ha quattro GPU e volete usare solo la prima e la terza GPU, potete usare:

```
CUDA_VISIBLE_DEVICES=0,2 python nome_script.py
```

Questo uso di `CUDA_VISIBLE_DEVICES` è un modo semplice ed efficace per gestire l'allocazione delle GPU senza modificare gli script PyTorch.

Eseguiamo ora questo codice e vediamo come funziona avviando il codice come script dal terminale:

```
python ch02-DDP-script.py
```

Notate che dovrebbe funzionare su macchine sia con una sola GPU sia con più GPU. Se eseguiamo questo codice su una singola GPU, dovremmo ottenere il seguente output:

```
PyTorch version: 2.2.1+cu117
CUDA available: True
Number of GPUs available: 1
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.62
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.32
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.11
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.07
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.02
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.03
[GPU0] Training accuracy 1.0
[GPU0] Test accuracy 1.0
```

Il codice in output è simile a quello ottenuto utilizzando una singola GPU, il che rappresenta un buon controllo di integrità.

Ora, se eseguiamo lo stesso comando e codice su una macchina con due GPU, dovremmo ottenere quanto segue:

```
PyTorch version: 2.2.1+cu117
CUDA available: True
Number of GPUs available: 2
[GPU1] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.60
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.59
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.16
[GPU1] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.17
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.05
[GPU1] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.05
[GPU1] Training accuracy 1.0
[GPU0] Training accuracy 1.0
[GPU1] Test accuracy 1.0
[GPU0] Test accuracy 1.0
```

Come previsto, possiamo vedere che alcuni batch vengono elaborati sulla prima GPU (`GPU0`) e altri sulla seconda (`GPU1`). Tuttavia, vediamo righe di output duplicate quando visualizziamo le accuratezze di addestramento e test. Ogni processo (in altre parole, ogni GPU) stampa l'accuratezza del test in modo indipendente. Poiché DDP replica il modello su ogni GPU e ogni processo viene eseguito in modo indipendente, se avete un'istruzione di stampa all'interno del vostro ciclo di test, ogni processo la eseguirà, con questa ripetizione delle righe di output ripetute. Se la cosa vi dà fastidio, potete risolverla usando il rango di ogni processo per controllare le vostre istruzioni `print`:

```
if rank == 0: (1)
    print("Test accuracy: ", accuracy)
```

(1) Mostra solo il primo processo.

Ecco, in sintesi, come funziona l'addestramento distribuito tramite DDP. Per maggiori dettagli, vi consiglio di consultare la documentazione API ufficiale su `https://mng.bz/9dPr`.

Alternative a PyTorch per l'addestramento multi-GPU

Se preferite un modo più diretto per usare più GPU in PyTorch, potete prendere in considerazione anche altre API, come la libreria open source Fabric. Ne ho parlato in *Accelerating PyTorch Model Training: Using Mixed-Precision and Fully Sharded Data Parallelism* (`https://mng.bz/jXle`).

Riepilogo

- PyTorch è una libreria open source con tre componenti principali: una libreria di tensori, funzioni di differenziazione automatica e utility di deep learning.
- La libreria di tensori di PyTorch è simile alle librerie per array, come NumPy.
- Nel contesto di PyTorch, i tensori sono strutture di dati simili ad array che rappresentano valori scalari, vettori, matrici e array di dimensioni superiori.
- I tensori PyTorch possono essere eseguiti sulla CPU, ma uno dei principali vantaggi del formato per tensori di PyTorch è il supporto delle GPU, per accelerare i calcoli.
- Le funzionalità di differenziazione automatica (*autograd*) di PyTorch consentono di addestrare comodamente le reti neurali utilizzando la retropropagazione senza dover derivare manualmente i gradienti.
- Le utility di deep learning di PyTorch forniscono gli elementi costitutivi per la creazione di reti neurali profonde.
- PyTorch include le classi `Dataset` e `DataLoader` per impostare efficienti pipeline di caricamento dei dati.
- È più semplice addestrare i modelli su una CPU o una singola GPU.
- Utilizzare `DistributedDataParallel` è il modo più semplice in PyTorch per accelerare l'addestramento, se sono disponibili più GPU.

Appendice B

Riferimenti e approfondimenti

Capitolo 1

I modelli LLM personalizzati sono in grado di superare i modelli LLM generici, come ha dimostrato un team di Bloomberg tramite una versione di GPT pre-addestrata su dati finanziari. Il modello LLM personalizzato ha superato ChatGPT nelle attività finanziarie, mantenendo al contempo buone prestazioni nei benchmark LLM generici.

- *BloombergGPT: A Large Language Model for Finance*, (2023) di Wu et al., https://arxiv.org/abs/2303.17564

Anche i modelli LLM esistenti possono essere adattati e perfezionati per ottenere risultati migliori rispetto ai modelli LLM generici, come hanno dimostrato i team di Google Research e Google DeepMind in un contesto medico.

- *Towards Expert-Level Medical Question Answering with Large Language Models*, (2023) di Singhal et al., `https://arxiv.org/abs/2305.09617`.

Il seguente documento presenta l'architettura originale a transformer.

- *Attention Is All You Need*, (2017) di Vaswani et al., https://arxiv.org/abs/1706.03762.

Sul transformer originale in stile encoder, denominato BERT, vedere.

- *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, (2018) di Devlin et al., https://arxiv.org/abs/1810.04805.

Il documento che descrive il modello GPT-3 in stile decoder, che ha ispirato i modelli LLM attuali e utilizzato come template per l'implementazione di un modello LLM da zero in questo libro, è il seguente.

- *Language Models are Few-Shot Learners*, (2020) di Brown et al., https://arxiv.org/abs/2005.14165.

Di seguito viene illustrato il transformer originale per la classificazione delle immagini, il che dimostra che le architetture transformer non sono limitate ai soli input testuali.

- *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*, (2020) di Dosovitskiy et al., https://arxiv.org/abs/2010.11929.

Le seguenti architetture LLM sperimentali (meno diffuse) servono da esempio per dimostrare che non tutti i modelli LLM devono essere basati sull'architettura a transformer.

- *RWKV: Reinventing RNNs for the Transformer Era*, (2023) di Peng et al., https://arxiv.org/abs/2305.13048.
- *Hyena Hierarchy: Towards Larger Convolutional Language Models*, (2023) di Poli et al., https://arxiv.org/abs/2302.10866.
- *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*, (2023) di Gu e Dao, https://arxiv.org/abs/2312.00752.

Il modello di Meta AI è una nota implementazione di un modello di tipo GPT disponibile gratuitamente, a differenza di GPT-3 e ChatGPT.

- *Llama 2: Open Foundation and Fine-Tuned Chat Models*, (2023) di Touvron et al., https://arxiv.org/abs/2307.092881

Per i lettori interessati a ulteriori dett.agli sul dataset impiegato nel Capitolo 1, questo documento descrive il dataset *The Pile*, curato da Eleuther AI.

- *The Pile: An 800GB Dataset of Diverse Text for Language Modeling*, (2020) di Gao et al., https://arxiv.org/abs/2101.00027.

Il seguente documento fornisce informazioni su InstructGPT per il fine-tuning di GPT-3, menzionato nel Capitolo 1 ma discusso più in dettaglio nel Capitolo 7.

- *Training Language Models to Follow Instructions with Human Feedback*, (2022) di Ouyang et al., https://arxiv.org/abs/2203.02155.

Capitolo 2

I lettori interessati alla discussione e al confronto degli spazi di embedding con gli spazi latenti e al concetto generale di rappresentazione vettoriale possono trovare informazioni nel primo capitolo del mio libro.

- *Machine Learning Q and AI*, (2023) di Sebastian Raschka, https://leanpub.com/machine-learning-q-and-ai.

Il seguente documento fornisce informazioni approfondite sull'uso della codifica a coppie di byte come metodo di tokenizzazione.

- *Neural Machine Translation of Rare Words with Subword Units*, (2015) di Sennrich et al., https://arxiv.org/abs/1508.07909.

Il codice per il tokenizer di codifica a coppie di byte utilizzato per addestrare GPT-2 è stato reso open source da OpenAI.

- https://github.com/openai/gpt-2/blob/master/src/encoder.py.

OpenAI fornisce un'interfaccia utente web interattiva per illustrare il funzionamento del tokenizer a coppie di byte nei modelli GPT.

- https://platform.openai.com/tokenizer.

Per i lettori interessati a programmare e addestrare un tokenizer BPE partendo da zero, il repository GitHub `minbpe` di Andrej Karpathy offre un'implementazione compatta e leggibile.

- *A Minimal Implementation of a BPE Tokenizer*, https://github.com/karpathy/minbpe.

I lettori interessati a studiare altri schemi di tokenizzazione alternativi, utilizzati da altri modelli LLM possono trovare maggiori informazioni negli articoli su SentencePiece e WordPiece.

- *SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing*, (2018) di Kudo e Richardson, https://aclanthology.org/D18-2012/.
- *Fast WordPiece Tokenization*, (2020) di Song et al., https://arxiv.org/abs/2012.15524.

Capitolo 3

I lettori interessati a saperne di più sull'attenzione di Bahdanau per le reti neurali ricorrenti e la traduzione linguistica possono trovare

approfondimenti dettagliati nel seguente articolo.

- *Neural Machine Translation by Jointly Learning to Align and Translate*, (2014) di Bahdanau, Cho, and Bengio, https://arxiv.org/abs/1409.0473.

Il concetto di auto-attenzione come attenzione a prodotto scalare è stato introdotto nel documento originale sul transformer.

- *Attention Is All You Need*, (2017) di Vaswani et al., https://arxiv.org/abs/1706.03762.

FlashAttention è un'implementazione altamente efficiente di un meccanismo di auto-attenzione, che accelera il processo di calcolo ottimizzando le modalità di accesso alla memoria. FlashAttention è matematicamente uguale al meccanismo di auto-attenzione standard, ma ottimizza il processo di calcolo.

- *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*, (2022) di Dao et al., https://arxiv.org/abs/2205.14135.
- *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning*, (2023) di Dao, https://arxiv.org/abs/2307.08691.

PyTorch implementa una funzione di auto-attenzione e l'attenzione causale che supporta FlashAttention. Questa funzione è ancora in beta e soggetta a modifiche.

- Documentazione di scaled_dot_product_attention: https://mng.bz/NRJd.

PyTorch implementa anche una classe `MultiHeadAttention` basata sulla funzione `scaled_dot_product()`.

- Documentazione di MultiHeadAttention documentation: https://mng.bz/DdJV.

Il dropout è una tecnica di regolarizzazione utilizzata nelle reti neurali per prevenire l'overfitting eliminando casualmente determinate unità (insieme alle loro connessioni) dalla rete neurale durante l'addestramento.

- *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, (2014) di Srivastava et al., https://jmlr.org/papers/v15/srivastava14a.html.

Sebbene l'utilizzo dell'attenzione multi-head basata sull'attenzione a prodotto scalare rimanga la variante più comune di auto-attenzione, si è scoperto che è possibile ottenere buone prestazioni anche senza la matrice dei pesi e il layer di proiezione.

- *Simplifying Transformer Blocks*, (2023) di He and Hofmann, https://arxiv.org/abs/2311.01906.

Capitolo 4

Il seguente articolo introduce una tecnica che stabilizza le dinamiche dello stato nascosto delle reti neurali normalizzando gli input sommati ai neuroni all'interno di un layer nascosto, riducendo significativamente i tempi di addestramento rispetto ai metodi pubblicati in precedenza.

- *Layer Normalization*, (2016) di Ba, Kiros, and Hinton, https://arxiv.org/abs/1607.06450.

Post-LayerNorm, utilizzato nel modello a transformer originale, applica la normalizzazione dei layer dopo le reti di auto-attenzione e feed forward. Al contrario, Pre-LayerNorm, adottato in modelli LLM come GPT-2 e più recenti, applica la normalizzazione dei layer prima di questi componenti, il che può portare a dinamiche di

addestramento più stabili e ha dimostrato di migliorare le prestazioni in alcuni casi, come discusso nei seguenti documenti.

- *On Layer Normalization in the Transformer Architecture*, (2020) di Xiong et al., https://arxiv.org/abs/2002.04745.
- *ResiDual: Transformer with Dual Residual Connections*, (2023) di Tie et al., https://arxiv.org/abs/2304.14802.

Una variante popolare di LayerNorm, utilizzata nei modelli LLM odierni è RMSNorm che vanta una notevole efficienza di elaborazione. Questa variante semplifica il processo di normalizzazione, normalizzando gli input utilizzando solo il valore quadratico medio degli input, senza sottrarre la media prima del quadrato. Ciò significa che non centra i dati prima di calcolare la scala. RMSNorm è descritto più in dettaglio nel seguente testo.

- *Root Mean Square Layer Normalization*, (2019) di Zhang and Sennrich, https://arxiv.org/abs/1910.07467.

La funzione di attivazione GELU (*Gaussian Error Linear Unit*) combina, per modellare gli output dei livelli, le proprietà della classica funzione di attivazione ReLU e della funzione di distribuzione cumulativa della distribuzione normale, consentendo la regolarizzazione stocastica e le non linearità nei modelli di deep learning.

- *Gaussian Error Linear Units (GELUs)*, (2016) di Hendricks and Gimpel, https://arxiv.org/abs/1606.08415.

Il paper di GPT-2 ha introdotto una serie di modelli LLM basati su transformer con dimensioni variabili: 124 milioni, 355 milioni, 774 milioni e 1,5 miliardi di parametri.

- *Language Models Are Unsupervised Multitask Learners*, (2019) di Radford et al., https://mng.bz/lMgo.

GPT-3 di OpenAI utilizza fondamentalmente la stessa architettura di GPT-2, tranne per il fatto che la versione più grande (175 miliardi) è 100 volte più grande del modello GPT-2 più grande ed è stata addestrata su molti più dati. I lettori interessati possono fare riferimento al documento ufficiale di GPT-3 di OpenAI e alla panoramica tecnica di Lambda Labs, che calcola che addestrare GPT-3 su una singola GPU consumer RTX 8000 richiederebbe 665 anni.

- *Language Models are Few-Shot Learners*, (2023) di Brown et al., https://arxiv.org/abs/2005.14165.
- *OpenAI's GPT-3 Language Model: A Technical Overview*, https://lambdalabs.com/blog/demystifying-gpt-3.

NanoGPT è un repository di codice con un'implementazione minimalista ma efficiente di un modello GPT-2, simile al modello implementato in questo libro. Il codice di questo libro è diverso da quello di nanoGPT, ma questo repository ha ispirato la riorganizzazione di una grande implementazione della classe genitore Python in sub-moduli più piccoli.

- *NanoGPT, a Repository for Training Medium-Sized GPTs*, https://github.com/karpathy/nanoGPT.

Un post informativo che mostra come la maggior parte del calcolo nei modelli LLM venga svolto nei layer feed-forward anziché nei layer di attenzione quando le dimensioni del contesto sono inferiori a 32.000 token.

- *In the Long (Context) Run*, by Harm de Vries, https://www.harmdevries.com/post/context-length.

Capitolo 5

Per informazioni dettagliate sulla funzione loss e su come applicare una trasformazione logaritmica per semplificarne la gestione ai fini dell'ottimizzazione matematica, potete vedere il video della mia lezione.

- *L8.2 Logistic Regression Loss Function*, https://www.youtube.com/watch?v=GxJe0DZvydM.

La seguente lezione e l'esempio di codice spiegano come funzionano le funzioni a entropia incrociata di PyTorch.

- *L8.7.1 OneHot Encoding and Multi-category Cross Entropy*, https://www.youtube.com/watch?v=4n71-tZ94yk.
- *Understanding Onehot Encoding and Cross Entropy in PyTorch*, https://mng.bz/o05v.

I due documenti seguenti descrivono in dettaglio il dataset, gli iperparametri e i dettagli dell'architettura utilizzati per il pre-addestramento dei modelli LLM.

- *Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling*, (2023) di Biderman et al., https://arxiv.org/abs/2304.01373.
- *OLMo: Accelerating the Science of Language Models*, (2024) di Groeneveld et al., https://arxiv.org/abs/2402.00838.

Il seguente codice supplementare disponibile per questo libro contiene istruzioni per la preparazione di 60.000 libri di pubblico dominio del *Progetto Gutenberg* per l'addestramento del modello LLM.

- *Pretraining GPT on the Project Gutenberg Dataset*, https://mng.bz/Bdw2.

Il Capitolo 5 discute il pre-addestramento dei modelli LLM, e l'Appendice D tratta alcune funzioni di addestramento più avanzate, come *Linear Warmup* e *Cosine Annealing*. Il seguente documento rileva che tecniche simili possono essere applicate con successo per continuare il pre-addestramento di modelli LLM già pre-addestrati e offre suggerimenti e approfondimenti.

- *Simple and Scalable Strategies to Continually Pre-train Large Language Models*, (2024) di Ibrahim et al., https://arxiv.org/abs/2403.08763.

BloombergGPT è un esempio di modello LLM specifico per un dominio, creato tramite l'addestramento su corpora di testo sia generali sia specifici, in particolare nel campo della finanza.

- *BloombergGPT: A Large Language Model for Finance*, (2023) di Wu et al., https://arxiv.org/abs/2303.17564.

GaLore è un recente progetto di ricerca volto a rendere più efficiente il pre-addestramento di un modello LLM. La modifica richiesta si riduce semplicemente alla sostituzione dell'ottimizzatore PyTorch `AdamW` nella funzione di addestramento con l'ottimizzatore `GaLoreAdamW` fornito dal pacchetto Python `galore-torch`.

- *GaLore: Memory-Efficient LLM Training by Gradient Low-Rank Projection*, (2024) di Zhao et al., https://arxiv.org/abs/2403.03507.
- Repository di codice GaLore, https://github.com/jiaweizzhao/GaLore.

I seguenti documenti e risorse condividono dataset di pre-addestramento su larga scala, di pubblico dominio, per modelli LLM,

costituiti da centinaia di gigabyte o terabyte di dati testuali.

- *Dolma: An Open Corpus of Three Trillion Tokens for LLM Pretraining Research*, (2024) di Soldaini et al., https://arxiv.org/abs/2402.00159.
- *The Pile: An 800GB Dataset of Diverse Text for Language Modeling*, (2020) di Gao et al., https://arxiv.org/abs/2101.00027.
- *The RefinedWeb Dataset for Falcon LLM: Outperforming Curated Corpora with Web Data, and Web Data Only*, (2023) di Penedo et al., https://arxiv.org/abs/2306.01116.
- *RedPajama*, by Together AI, https://mng.bz/d6nw.
- Il dataset *FineWeb*, che include oltre 15.000 miliardi di token di dati web inglesi puliti e deduplicati provenienti da CommonCrawl, https://mng.bz/rVzy.

Il documento che ha introdotto il Top-k Sampling è il seguente.

- *Hierarchical Neural Story Generation*, (2018) di Fan et al., https://arxiv.org/abs/1805.04833.

Un'alternativa al Top-k Sampling è il campionamento top-p (non trattato nel Capitolo 5), che seleziona dal più piccolo insieme dei migliori token la cui probabilità cumulativa supera una soglia *p*, mentre il Top-k Sampling sceglie dai k migliori token in base alla probabilità.

- *Top-p Sampling*, https://en.wikipedia.org/wiki/Top-p_sampling.

Beam Search (non trattata nel Capitolo 5) è un algoritmo di decodifica alternativo che genera sequenze di output mantenendo a ogni passo solo le sequenze parziali con punteggio più elevato, per bilanciare l'efficienza e la qualità.

- *Diverse Beam Search: Decoding Diverse Solutions from Neural Sequence Models*, (2016) di Vijayakumar et al., https://arxiv.org/abs/1610.02424.

Capitolo 6

Ulteriori risorse che discutono i diversi tipi di fine-tuning sono i seguenti.

- *Using and Finetuning Pretrained Transformers*, https://mng.bz/VxJG.
- *Finetuning Large Language Models*, https://mng.bz/x28X.

Ulteriori esperimenti, tra cui un confronto tra il fine-tuning del primo e dell'ultimo token di output, possono essere trovati nel materiale supplementare su GitHub.

- *Additional spam classification experiments*, https://mng.bz/AdJx.

Per un'attività di classificazione binaria, come la classificazione dello spam, è tecnicamente possibile utilizzare un solo nodo di output anziché due, come spiegato nel seguente articolo.

- *Losses Learned – Optimizing Negative Log-Likelihood and Cross-Entropy in PyTorch*, https://mng.bz/ZEJA.

Ulteriori esperimenti sul fine-tuning di diversi layer di un modello LLM sono disponibili nel seguente articolo, che mostra come il fine-tuning dell'ultimo blocco transformer, oltre al layer di output, migliori sostanzialmente le predizioni.

- *Finetuning Large Language Models*, https://mng.bz/RZJv.

I lettori possono trovare risorse e informazioni aggiuntive su come gestire i dataset sbilanciati nella documentazione di imbalanced-learn.

- *Imbalanced-Learn User Guide*, https://mng.bz/2KNa.

Per i lettori interessati a classificare le e-mail di spam anziché gli sms di spam, la seguente risorsa fornisce un ampio dataset sulla classificazione delle e-mail in un comodo formato CSV simile al formato utilizzato nel Capitolo 6.

- *Email Spam Classification Dataset*, https://mng.bz/1GEq.

GPT-2 è un modello basato sul modulo decoder dell'architettura a transformer, e il suo scopo principale è quello di generare nuovo testo. In alternativa, anche modelli basati su encoder come BERT e RoBERTa possono essere efficaci per le attività di classificazione.

- *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, (2018) di Devlin et al., https://arxiv.org/abs/1810.04805.
- *RoBERTa: A Robustly Optimized BERT Pretraining Approach*, (2019) di Liu et al., https://arxiv.org/abs/1907.11692
- *Additional Experiments Classifying the Sentiment of 50k IMDB Movie Reviews*, https://mng.bz/PZJR

Articoli recenti dimostrano che le prestazioni di classificazione possono essere ulteriormente migliorate rimuovendo la maschera causale durante il fine-tuning per attività di classificazione, insieme ad altre modifiche.

- *Label Supervised LLaMA Finetuning*, (2023) di Li et al., https://arxiv.org/abs/2310.01208.

- *LLM2Vec: Large Language Models Are Secretly Powerful Text Encoders*, (2024) di BehnamGhader et al., https://arxiv.org/abs/2404.05961.

Capitolo 7

Il dataset Alpaca per il fine-tuning per rispondere a istruzioni contiene 52.000 coppie istruzione-risposta ed è uno dei primi e più popolari dataset di pubblico dominio per questo impiego.

- *Stanford Alpaca: An Instruction-Following Llama Model*, https://github.com/tatsu-lab/stanford_alpaca.

Altri dataset di pubblico dominio adatti per eseguire il fine-tuning per rispondere a istruzioni sono i seguenti.

- *LIMA*, https://huggingface.co/datasets/GAIR/lima. Per informazioni: *LIMA: Less Is More for Alignment*, Zhou et al., https://arxiv.org/abs/2305.11206.
- *UltraChat*, https://huggingface.co/datasets/openchat/ultrachat-sharegpt, un dataset di larga scala costituito da 805000 coppie istruzione-risposta; per informazioni: *Enhancing Chat Language Models by Scaling Highquality Instructional Conversations*, by Ding et al., https://arxiv.org/abs/2305.14233.
- *Alpaca GPT4*, https://mng.bz/Aa0p. Un dataset di tipo Alpaca con 52000 coppie istruzione-risposta generato con GPT-4 invece di GPT-3.5.

Phi-3 è un modello da 3.8 miliardi di parametri con una variante con istruzioni ottimizzate che si dice sia paragonabile a modelli proprietari molto più grandi, come GPT-3.5.

- *Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone*, (2024) di Abdin et al., https://arxiv.org/abs/2404.14219.

I ricercatori propongono un metodo sintetico di generazione dei dati a istruzioni che genera 300000 coppie istruzione-risposta di alta qualità da un modello Llama-3 con istruzione ottimizzata. Un pattern di base Llama-3 pre-addestrato e ottimizzato su questi esempi funziona in modo paragonabile al modello Llama-3 originale ottimizzato.

- *Magpie: Alignment Data Synthesis from Scratch by Prompting Aligned LLMs with Nothing*, (2024) di Xu et al., https://arxiv.org/abs/2406.08464.

La ricerca ha dimostrato che non mascherando le istruzioni e gli input nel fine-tuning per rispondere alle istruzioni migliora efficacemente le prestazioni in varie attività di elaborazione del linguaggio naturale e benchmark di generazione, in particolare se l'addestramento avviene su dataset con istruzioni lunghe e output brevi o quando si utilizza un numero limitato di esempi di addestramento.

- *Instruction Tuning with Loss Over Instructions*, (2024) di Shi, https://arxiv.org/abs/2405.14394.

Prometheus e PHUDGE sono LLM di pubblico dominio paragonabili a GPT-4 nella valutazione di risposte di lungo formato con criteri personalizzabili. Non li utilizziamo perché al momento della stesura, non sono supportati da Ollama, e quindi non possono essere eseguiti in modo efficiente su un laptop.

- *Prometheus: Inducing Finegrained Evaluation Capability in Language Models*, (2023) di Kim et al., https://arxiv.org/abs/2310.08491.
- *PHUDGE: Phi-3 as Scalable Judge*, (2024) di Deshwal and Chawla, https://arxiv.org/abs/2405.08029.
- *Prometheus 2: An Open Source Language Model Specialized in Evaluating Other Language Models*, (2024), by Kim et al., https://arxiv.org/abs/2405.01535.

I risultati nel seguente report supportano la visione secondo cui i modelli LLM acquisiscono conoscenze fattuali principalmente durante il pre-addestramento e il fine-tuning migliora principalmente la loro efficienza nell'uso di tali conoscenze. Inoltre, questo studio spiega come il fine-tuning dei modelli LLM con nuove informazioni fattuali influisce sulla loro capacità di usare conoscenze preesistenti, rivelando che i modelli apprendono nuovi fatti più lentamente e la loro introduzione durante il fine-tuning aumenta la tendenza del modello a generare informazioni errate.

- *Does Fine-Tuning LLMs on New Knowledge Encourage Hallucinations?*, (2024) di Gekhman, https://arxiv.org/abs/2405.05904.

Il fine-tuning delle preferenze è un passo facoltativo dopo il fine-tuning per rispondere a istruzioni, per allineare il modello LLM alle preferenze umane. I seguenti articoli dell'autore forniscono maggiori informazioni su questo processo.

- *LLM Training: RLHF and Its Alternatives*, https://mng.bz/ZVPm.
- Tips for LLM Pretraining and Evaluating Reward Models, *https://mng.bzRNXj*.

Appendice A

Sebbene l'Appendice A dovrebbe essere sufficiente di per sé, se state cercando introduzioni più complete al deep learning, vi consiglio i seguenti libri.

- *Machine learning with PyTorch and Scikit-Learn* (2022) di Sebastian Raschka, Hayden Liu e Vahid Mirjalili. ISBN 978-1801819312.
- *Deep Learning with PyTorch* (2021) di Eli Stevens, Luca Antiga e Thomas Viehmann. ISBN 978-1617295263.

Per un'introduzione più approfondita ai concetti dei tensori, i lettori possono trovare un mio tutorial video di 15 minuti.

- *Lecture 4.1: Tensors in Deep Learning*, `https://www.youtube.com/watch?v=JXfDlgrfOBY`.

Se volete saperne di più sulla valutazione del modello nel machine learning, vi consiglio questo mio articolo.

- *Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning*, (2018) di Sebastian Raschka, *https://arxiv.org/abs/1811.12808*.

Per i lettori interessati a un ripasso o a una breve introduzione al calcolo, ho scritto un capitolo sul calcolo che è disponibile gratuitamente sul mio sito web.

- *Introduction to Calculus*, by Sebastian Raschka, *https://mng.bz/WEyW*.

Perché PyTorch non richiama automaticamente in background `optimizer.zero_grad()`? In alcuni casi, potrebbe essere desiderabile accumulare i gradienti, e PyTorch ci lascia questa opzione. Se volete

saperne di più sull’accumulo dei gradienti, consultate il seguente articolo.

- *Finetuning Large Language Models on a Single GPU Using Gradient Accumulation*, by Sebastian Raschka, *https://mng.bz/8wPD*.

Questa appendice tratta DDP, un approccio molto utilizzato per l’addestramento di modelli di deep learning su più GPU. Per casi d’uso più avanzati, in cui un singolo modello non si adatta a una GPU, potete anche prendere in considerazione il metodo FSDP (*Fully Sharded Data Parallel*) di PyTorch, che esegue il parallelismo dei dati distribuiti e distribuisce i layer troppo grandi su più GPU. Per maggiori informazioni, consultate questa panoramica con ulteriori link alla documentazione API.

- *Introducing PyTorch Fully Sharded Data Parallel (FSDP) API*, *https://mng.bz/EZJR*.

Appendice C
Soluzioni degli esercizi

Gli esempi di codice completi con le risposte agli esercizi sono disponibili nel repository GitHub all'indirizzo https://github.com/rasbt/LLMs-from-scratch.

Capitolo 2

Esercizio 2.1

Potete ottenere i singoli token-ID richiedendo all'encoder una stringa alla volta:

```
print(tokenizer.encode("Ak"))
print(tokenizer.encode("w"))
# ...
```

Questo codice produce il seguente output:

```
[33901]
[86]
# ...
```

Poi potete utilizzare la seguente riga di codice per assemblare la stringa originale:

```
print(tokenizer.decode([33901, 86, 343, 86, 220, 959]))
```

Questo produce il seguente output:

```
'Akwirw ier'
```

Esercizio 2.2

Il codice per il data loader con `max_length=2` e `stride=2`:

```
dataloader = create_dataloader(
    raw_text, batch_size=4, max_length=2, stride=2
)
```

Questo produce batch del seguente formato:

```
tensor([[  40,  367],
        [2885, 1464],
        [1807, 3619],
        [ 402, 271]]
)
```

Il codice del secondo data loader con `max_length=8` e `stride=2`:

```
dataloader = create_dataloader(
    raw_text, batch_size=4, max_length=8, stride=2
)
```

Un esempio di batch prodotto:

```
tensor([[   40,   367,  2885,  1464,  1807,  3619,   402,   271],
        [ 2885,  1464,  1807,  3619,   402,   271, 10899,  2138],
        [ 1807,  3619,   402,   271, 10899,  2138,   257,  7026],
        [  402,   271, 10899,  2138,   257,  7026, 15632,   438]]
)
```

Capitolo 3

Esercizio 3.1

L’assegnazione corretta dei pesi è la seguente:

```
sa_v1.W_query = torch.nn.Parameter(sa_v2.W_query.weight.T)
sa_v1.W_key = torch.nn.Parameter(sa_v2.W_key.weight.T)
sa_v1.W_value = torch.nn.Parameter(sa_v2.W_value.weight.T)
```

Esercizio 3.2

Per ottenere una dimensione di output pari a 2, simile a quella che avevamo nell’attenzione single-head, dobbiamo modificare le dimensioni di proiezione `d_out` in `1`.

```
d_out = 1
mha = MultiHeadAttentionWrapper(d_in, d_out, block_size, 0.0, num_heads=2)
```

Esercizio 3.3

L’inizializzazione per il modello GPT-2 più piccolo è la seguente:

```
block_size = 1024
d_in, d_out = 768, 768
num_heads = 12
mha = MultiHeadAttention(d_in, d_out, block_size, 0.0, num_heads)
```

Capitolo 4

Esercizio 4.1

Possiamo calcolare il numero di parametri nei moduli feed forward e di attenzione come segue:

```
block = TransformerBlock(GPT_CONFIG_124M)

total_params = sum(p.numel() for p in block.ff.parameters())
print(f"Total number of parameters in feed forward module: {total_params:,}")

total_params = sum(p.numel() for p in block.att.parameters())
print(f"Total number of parameters in attention module: {total_params:,}")
```

Come potete vedere, il modulo feed forward contiene circa il doppio dei parametri del modulo di attenzione:

```
Total number of parameters in feed forward module: 4,722,432
Total number of parameters in attention module: 2,360,064
```

Esercizio 4.2

Per istanziare le altre dimensioni del modello GPT, possiamo modificare il dizionario di configurazione come segue (qui mostrato per GPT-2 XL):

```
GPT_CONFIG = GPT_CONFIG_124M.copy()
GPT_CONFIG["emb_dim"] = 1600
GPT_CONFIG["n_layers"] = 48
GPT_CONFIG["n_heads"] = 25
model = GPTModel(GPT_CONFIG)
```

Quindi, riutilizzando il codice già visto per calcolare il numero di parametri e i requisiti di RAM, troviamo:

```
gpt2-xl:
Total number of parameters: 1,637,792,000
Number of trainable parameters considering weight tying: 1,557,380,800
Total size of the model: 6247.68 MB
```

Esercizio 4.3

Ci sono tre punti distinti nel Capitolo 4 in cui abbiamo utilizzato i dropout layer: il layer di embedding, il layer di collegamento e il modulo di attenzione multi-head. Possiamo controllare i tassi di dropout per ciascuno dei layer codificandoli separatamente nel file di configurazione e quindi modificando di conseguenza l'implementazione del codice.

La configurazione modificata è la seguente:

```
GPT_CONFIG_124M = {
    "vocab_size": 50257,
    "context_length": 1024,
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate_attn": 0.1, (1)
    "drop_rate_shortcut": 0.1, (2)
    "drop_rate_emb": 0.1, (3)
    "qkv_bias": False
}
```

(1) Dropout per attenzione multi-head.

(2) Dropout per collegamento scorciatoia.

(3) Dropout per layer di embedding.

Le classi `TransformerBlock` e `GPTModel` modificate hanno il seguente aspetto:

```
class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate_attn"], (1)
            qkv_bias=cfg["qkv_bias"])
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_shortcut = nn.Dropout(cfg["drop_rate_shortcut"]) (2)
```

```
    def forward(self, x):
        shortcut = x
        x = self.norm1(x)
        x = self.att(x) (1)
        x = self.drop_shortcut(x)
        x = x + shortcut

        shortcut = x
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x)
        x = x + shortcut
        return x

class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate_emb"]) (3)
        self.trf_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])]
        )
        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(cfg["emb_dim"], cfg["vocab_size"], bias=False)

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)
        pos_embeds = self.pos_emb(torch.arange(seq_len, device=in_idx.device))
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits
```

(1) Dropout per attenzione multi-head.

(2) Dropout per collegamento scorciatoia.

(3) Dropout per layer di embedding.

Capitolo 5

Esercizio 5.1

Possiamo visualizzare il numero di volte in cui il token (o la parola) “pizza” viene campionato usando la funzione

`print_sampled_tokens()` che abbiamo definito in questo paragrafo. Iniziamo col codice che abbiamo definito in precedenza.

Il token “pizza” viene campionato 0× se la temperatura è 0 o 0.1, e viene campionato 32× se la temperatura è aumentata a 5. La probabilità stimata è 32 / 1000 × 100 percento = 3.2 percento.

La probabilità effettiva è del 4.3 percento ed è contenuta nel tensore delle probabilità softmax ridimensionato (`scaled_probas[2][6]`).

Esercizio 5.2

Top-k Sampling e Temperature Scaling sono impostazioni che devono essere regolate in base al modello LLM e al grado desiderato di diversità e casualità nell’output.

Quando si utilizzano valori top-k relativamente piccoli (per esempio, inferiori a 10) e quando la temperatura è impostata a meno di 1, l’output del modello diventa meno casuale e più deterministico. Questa impostazione è utile quando abbiamo bisogno che il testo generato sia più prevedibile, coerente e più vicino ai risultati più probabili in base ai dati di addestramento.

Le applicazioni per impostazioni di k e temperatura così basse includono la generazione di documenti o report formali in cui la chiarezza e l’accuratezza sono particolarmente importanti. Altri esempi di applicazioni includono le analisi tecniche o le attività di generazione di codice, in cui l’accuratezza è fondamentale. Inoltre, anche le risposte alle domande e i contenuti educativi richiedono risposte accurate, in cui è utile impiegare una temperatura inferiore a 1.

Al contrario, valori top-k più grandi (per esempio valori compresi tra 20 e 40) e valori di temperatura superiori a 1 sono utili quando si utilizzano i modelli LLM per il brainstorming o per generare contenuti creativi, come la narrativa.

Esercizio 5.3

Esistono diversi modi per forzare un comportamento deterministico con la funzione `generate()`:

1. impostando `top_k=None` e non applicando alcuno scaling alla temperatura;
2. impostando `top_k=1`.

Esercizio 5.4

In sostanza, dobbiamo caricare il modello e l'ottimizzatore che abbiamo salvato nel capitolo:

```
checkpoint = torch.load("model_and_optimizer.pth")
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-4, weight_decay=0.1)
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
```

Quindi, richiamare la funzione `train_simple_function()` con `num_epochs=1` per addestrare il modello per un'altra epoca.

Esercizio 5.5

Possiamo utilizzare il seguente codice per calcolare i valori loss del set di addestramento e di validazione del modello GPT:

```
train_loss = calc_loss_loader(train_loader, gpt, device)
val_loss = calc_loss_loader(val_loader, gpt, device)
```

I valori loss risultanti per il parametro da 124 milioni sono le seguenti:

```
Training loss: 3.754748503367106
Validation loss: 3.559617757797241
```

L'osservazione principale è che le performance dei set di addestramento e di validazione sono nella stessa fascia. Ciò può avere molteplici spiegazioni.

- *The Verdict* non faceva parte del dataset di pre-addestramento quando OpenAI ha addestrato GPT-2. Quindi, il modello non va esplicitamente in overfitting al set di addestramento e funziona altrettanto bene nelle porzioni del set di addestramento e validazione di *The Verdict*. Il valore loss del set di validazione è leggermente inferiore a quello del set di addestramento, il che è insolito nel deep learning. Tuttavia, è probabile che ciò sia dovuto a rumore casuale, poiché il dataset è relativamente piccolo. All'atto pratico, se non c'è un overfitting, ci si aspetta che le prestazioni dei set di addestramento e di validazione siano pressoché identiche.
- *The Verdict* faceva parte del set di addestramento di GPT-2. In questo caso, non possiamo dire se il modello sta andando in overfitting sui dati di addestramento, perché il set di validazione sarebbe stato utilizzato anche per l'addestramento. Per valutare il grado di overfitting, avremmo bisogno di un nuovo dataset generato dopo che OpenAI ha terminato l'addestramento di GPT-2, per assicurarci che non potesse aver fatto parte del pre-addestramento.

Esercizio 5.6

Nel capitolo, abbiamo sperimentato col modello GPT-2 più piccolo, che ha solo 124 milioni di parametri. Il motivo era mantenere compatti i requisiti di risorse. Tuttavia, potete facilmente sperimentare con modelli più grandi con piccole modifiche al codice. Per esempio, per caricare i pesi del modello da 1558 milioni anziché di quello da 124 milioni, le uniche due righe di codice che dobbiamo modificare sono le seguenti:

```
hparams, params = download_and_load_gpt2(model_size="124M", models_dir="gpt2")
model_name = "gpt2-small (124M)"
```

Il codice aggiornato è il seguente:

```
hparams, params = download_and_load_gpt2(model_size="1558M", models_dir="gpt2")
model_name = "gpt2-xl (1558M)"
```

Capitolo 6

Esercizio 6.1

Possiamo estendere gli input al numero massimo di token supportati dal modello impostando la lunghezza massima su `max_length = 1024` durante l'inizializzazione dei dataset:

```
train_dataset = SpamDataset(..., max_length=1024, ...)
val_dataset = SpamDataset(..., max_length=1024, ...)
test_dataset = SpamDataset(..., max_length=1024, ...)
```

Tuttavia, l'ulteriore padding determina un'accuratezza del test sostanzialmente peggiore, pari al 78.33 percento (rispetto al 95.67 percento del capitolo).

Esercizio 6.2

Invece di eseguire il fine-tuning solo del blocco transformer finale, possiamo eseguire il fine-tuning dell'intero modello rimuovendo le seguenti righe dal codice:

```
for param in model.parameters():
    param.requires_grad = False
```

Questa modifica determina un miglioramento dell'1 percento nell'accuratezza del test, che si attesta al 96.67 percento (rispetto al 95.67 percento del capitolo).

Esercizio 6.3

Invece di eseguire il fine-tuning dell'ultimo token di output, possiamo eseguire il fine-tuning il primo token di output modificando

`model(input_batch)[:, -1,:]` in `model(input_batch)[:, 0,:]` ovunque nel codice.

Come previsto, poiché il primo token contiene meno informazioni dell'ultimo, questa modifica determina un'accuratezza del test sostanzialmente peggiore, pari al 75.00 percento (rispetto al 95.67 percento del capitolo).

Capitolo 7

Esercizio 7.1

Il formato del prompt Phi-3, rappresentato nella Figura 7.4, ha il seguente aspetto per un dato input di esempio:

```
<user>
Identify the correct spelling of the following word: 'Occasion'

<assistant>
The correct spelling is 'Occasion'.
```

Per utilizzare questo modello, possiamo modificare la funzione `format_input()` come segue:

```
def format_input(entry):
    instruction_text = (f"<|user|>\n{entry['instruction']}")
    input_text = f"\n{entry['input']}" if entry["input"] else ""
    return instruction_text + input_text
```

Infine, dobbiamo anche aggiornare il modo in cui estraiamo la risposta generata quando raccogliamo le risposte del set di test:

```
for i, entry in tqdm(enumerate(test_data), total=len(test_data)):
    input_text = format_input(entry)
    tokenizer=tokenizer
    token_ids = generate(
        model=model,
        idx=text_to_token_ids(input_text, tokenizer).to(device),
        max_new_tokens=256,
        context_size=BASE_CONFIG["context_length"],
        eos_id=50256
    )
    generated_text = token_ids_to_text(token_ids, tokenizer)
    response_text = (
        generated_text[len(input_text):]
        .replace("<|assistant|>:", "") (1)
        .strip()
```

```
    )
    test_data[i]["model_response"] = response_text
```

(1) Nuovo: ###Response to <|assistant|>

Il fine-tuning del modello col template Phi-3 è più veloce di circa il 17 percento, poiché comporta input più brevi. Il punteggio è vicino a 50, che è più o meno lo stesso che abbiamo ottenuto in precedenza con i prompt in stile Alpaca.

Esercizio 7.2

Per mascherare le istruzioni come mostra la Figura 7.13, dobbiamo apportare piccole modifiche alla classe `InstructionDataset` e alla funzione `custom_collate_fn()`. Possiamo modificare la classe `InstructionDataset` per raccogliere le lunghezze delle istruzioni, che useremo nella funzione di collazione per individuare le posizioni delle istruzioni nei target quando codifichiamo la funzione di collazione, come segue:

```
class InstructionDataset(Dataset):
    def __init__(self, data, tokenizer):
        self.data = data
        self.instruction_lengths = [] (1)
        self.encoded_texts = []

        for entry in data:
            instruction_plus_input = format_input(entry)
            response_text = f"\n\n### Response:\n{entry['output']}"
            full_text = instruction_plus_input + response_text

            self.encoded_texts.append(tokenizer.encode(full_text))
            instruction_length = (len(tokenizer.encode(instruction_plus_input))
            self.instruction_lengths.append(instruction_length) (2)

    def __getitem__(self, index): (3)
        return self.instruction_lengths[index], self.encoded_texts[index]

    def __len__(self):
        return len(self.data)
```

(1) Lista separata per le lunghezze delle istruzioni.

(2) Raccoglie le lunghezze delle istruzioni.

(3) Restituisce separatamente sia le lunghezze delle istruzioni sia i testi.

Ora, aggiorniamo `custom_collate_fn()` dove ogni `batch` è ora una tupla contenente `(instruction_length, item)` anziché solo `item`, a causa delle modifiche nel dataset `InstructionDataset`. Inoltre, ora mascheriamo i token delle istruzioni corrispondenti nella lista degli ID target:

```
def custom_collate_fn(batch,
                      pad_token_id=50256,
                      ignore_index=-100,
                      allowed_max_length=None,
                      device="cpu"
):

    batch_max_length = max(len(item)+1 for instruction_length, item in batch)
    inputs_lst, targets_lst = [], [] (1)

    for instruction_length, item in batch: (1)
        new_item = item.copy()
        new_item += [pad_token_id]
        padded = (new_item + [pad_token_id] * (batch_max_length - len(new_item))
        inputs = torch.tensor(padded[:-1])
        targets = torch.tensor(padded[1:])
        mask = targets == pad_token_id
        indices = torch.nonzero(mask).squeeze()
        if indices.numel() > 1:
            targets[indices[1:]] = ignore_index

        targets[:instruction_length-1] = -100 (2)

        if allowed_max_length is not None:
            inputs = inputs[:allowed_max_length]
            targets = targets[:allowed_max_length]

        inputs_lst.append(inputs)
        targets_lst.append(targets)

    inputs_tensor = torch.stack(inputs_lst).to(device)
    targets_tensor = torch.stack(targets_lst).to(device)

    return inputs_tensor, targets_tensor
```

(1) Ora il batch è una tupla.

(2) Maschera tutti i token di input e delle istruzioni nei target.

Nel valutare un modello LLM dopo il fine-tuning con questo metodo di mascheramento delle istruzioni, le sue prestazioni risultano leggermente peggiori (circa 4 punti utilizzando il metodo Ollama Llama 3 del Capitolo 7). Ciò è coerente con le osservazioni riportate nel documento *Instruction Tuning With Loss Over Instructions* (https://arxiv.org/abs/2405.14394).

Esercizio 7.3

Per eseguire il fine-tuning del modello sul dataset originale Stanford Alpaca (https://github.com/tatsu-lab/stanford_alpaca), dobbiamo solo cambiare l'URL del file da…

```
url = "https://raw.githubusercontent.com/rasbt/LLMs-from-
scratch/main/ch07/01_main-chapter-code/instruction-data.json"
```

… a …

```
url = "https://raw.githubusercontent.com/tatsu-
lab/stanford_alpaca/main/alpaca_data.json"
```

Notate che il dataset contiene 52000 elementi (50 volte in più rispetto al Capitolo 7) e che tali elementi sono più lunghi di quelli con cui abbiamo lavorato nel Capitolo 7.

Pertanto, si consiglia vivamente di eseguire l'addestramento su una GPU.

Se si verificano errori di memoria, potete valutare di ridurre le dimensioni del batch da 8 a 4, 2 o 1. Oltre a ridurre le dimensioni del batch, potreste anche valutare di ridurre `allowed_max_length` da `1024` a `512` o `256`.

Esercizio 7.4

Per eseguire il fine-tuning del modello utilizzando LoRA, utilizzate le classi e le funzioni pertinenti dall'Appendice E:

```
from appendix_E import LoRALayer, LinearWithLoRA, replace_linear_with_lora
```

Ora, aggiungere le seguenti righe di codice sotto il codice di caricamento del modello (Listato 7.7):

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters before: {total_params:,}")

for param in model.parameters():
    param.requires_grad = False

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters after: {total_params:,}")
replace_linear_with_lora(model, rank=16, alpha=16)
```

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable LoRA parameters: {total_params:,}")
model.to(device)
```

Notate che, su una GPU Nvidia L4, il fine-tuning con LoRA impiega 1.30 minuti per essere eseguito su una L4. Sulla stessa GPU, il codice originale impiega 1.80 minuti. Quindi, LoRA è circa il 28 percento più veloce, in questo caso. Il punteggio, valutato col metodo Ollama Llama 3 del Capitolo 7, è circa 50, che è nella stessa fascia del modello originale.

Appendice A

Esercizio A.1

La rete ha due input e due output. Inoltre, ci sono due layer nascosti con 30 e 20 nodi, rispettivamente. A livello di programmazione, possiamo calcolare il numero di parametri come segue:

```
model = NeuralNetwork(2, 2)
num_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print("Total number of trainable model parameters:", num_params)
```

Questo produce il seguente output:

```
752
```

Possiamo anche calcolarlo manualmente.

- *Primo layer nascosto*: 2 input per 30 unità nascoste più 30 unità di bias.
- *Secondo layer nascosto*: 30 unità in arrivo per 20 nodi più 20 unità di bias.
- *Layer di output:* 20 nodi in arrivo moltiplicati per 2 nodi di output più 2 unità di bias.

Quindi, sommando tutti i parametri in ogni layer si ottiene $2 \times 30 + 30 + 30 \times 20 + 20 + 20 \times 2 + 2 = 752$.

Esercizio A.2

I risultati esatti in fase di esecuzione saranno specifici per l'hardware utilizzato per questo esperimento. Nei miei esperimenti, ho osservato accelerazioni significative anche per piccole moltiplicazioni di matrici come la seguente quando si utilizza un'istanza di Google Colab connessa a una GPU V100:

```
a = torch.rand(100, 200)
b = torch.rand(200, 300)
%timeit a@b
```

Sulla CPU, ciò ha comportato 63,8 µs ± 8,7 µs per ciclo.

Quando eseguito su una GPU:

```
a, b = a.to("cuda"), b.to("cuda")
%timeit a @ b
```

Il risultato è stato 13,8 µs ± 425 ns per ciclo.

In questo caso, su una V100, il calcolo è stato circa quattro volte più veloce.

Esercizio A.3

La rete ha due input e due output. Inoltre, ci sono 2 layer nascosti con 30 e 20 nodi, rispettivamente. A livello di programmazione, possiamo calcolare il numero di parametri come segue:

```
model = NeuralNetwork(2, 2)
num_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print("Total number of trainable model parameters:", num_params)
```

Questo produce il seguente output:

```
752
```

Possiamo anche calcolarlo manualmente come segue.

- *Primo layer nascosto*: 2 input per 30 unità nascoste più 30 unità di bias.
- *Secondo layer nascosto*: 30 unità in arrivo per 20 nodi più 20 unità di bias.

- *Layer di output*: 20 nodi in arrivo per 2 nodi di output più 2 unità di bias.

Quindi, sommando tutti i parametri in ogni layer si ottiene $2\times30+30 + 30\times20+20 + 20\times2 +2 = 752$.

Esercizio A.4

I risultati esatti in fase di esecuzione saranno specifici per l'hardware utilizzato per questo esperimento. Nei miei esperimenti, ho osservato accelerazioni significative anche per piccole moltiplicazioni di matrici nell'utilizzo di un'istanza di Google Colab connessa a una GPU V100:

```
a = torch.rand(100, 200)
b = torch.rand(200, 300)
%timeit a@b
```

Sulla CPU questo ha portato a 63,8 µs ± 8,7 µs per ciclo.

Quando eseguito su una GPU:

```
a, b = a.to("cuda"), b.to("cuda")
%timeit a @ b
```

Il risultato è stato 13,8 µs ± 425 ns per ciclo.

In questo caso, su una V100, il calcolo è stato circa quattro volte più veloce.

Appendice D

Nuovi dettagli per il ciclo di addestramento

In questa appendice, miglioriamo la funzione di addestramento per i processi di pre-addestramento e fine-tuning trattati nei Capitoli da 5 a 7. In particolare, l'appendice tratta le tecniche *Learning Rate Warmup*, *Cosine Decay* e *Gradient Clipping*. Incorporeremo queste tecniche nella funzione di addestramento, e pre-addestreremo un modello LLM.

Per rendere il codice autosufficiente, reinizializziamo il modello che abbiamo addestrato nel Capitolo 5:

```
import torch
from chapter04 import GPTModel
GPT_CONFIG_124M = {"vocab_size": 50257, (1)
                   "context_length": 256, (2)
                   "emb_dim": 768, (3)
                   "n_heads": 12, (4)
                   "n_layers": 12, (5)
                   "drop_rate": 0.1, (6)
                   "qkv_bias": False (7)
}
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
model.eval()
```

(1) Dimensioni del vocabolario.

(2) Lunghezza del contesto ridotta (originalmente `1024`).

(3) Dimensioni embedding.

(4) Numero di punti di attenzione.

(5) Numero di layer.

(6) Tasso di dropout.

(7) Bias query-chiave-valore.

Dopo aver inizializzato il modello, dobbiamo inizializzare i data loader. Innanzitutto, carichiamo il racconto breve *The Verdict*:

```
import os
import urllib.request

file_path = "the-verdict.txt"

url = ("https://raw.githubusercontent.com/rasbt/LLMs-from-scratch/"
       "main/ch02/01_main-chapter-code/the-verdict.txt"
)

if not os.path.exists(file_path):
    with urllib.request.urlopen(url) as response:
        text_data = response.read().decode('utf-8')
    with open(file_path, "w", encoding="utf-8") as file:
        file.write(text_data)
else:
    with open(file_path, "r", encoding="utf-8") as file:
        text_data = file.read()
```

Ora, carichiamo i `text_data` nei data loader:

```
from previous_chapters import create_dataloader_v1

train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
torch.manual_seed(123)
train_loader = create_dataloader_v1(
    text_data[:split_idx],
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=True,
    shuffle=True,
    num_workers=0
)
val_loader = create_dataloader_v1(
    text_data[split_idx:],
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=False,
    shuffle=False,
    num_workers=0
)
```

Learning Rate Warmup

L’implementazione di questa tecnica può stabilizzare l’addestramento di modelli complessi come i modelli LLM. Questo processo comporta l’aumento graduale (il “riscaldamento”) del tasso di apprendimento da un valore iniziale molto basso (`initial_lr`) a un valore massimo specificato dall’utente (`peak_lr`). Iniziare l’addestramento con aggiornamenti più piccoli dei pesi riduce il rischio che il modello vada incontro ad aggiornamenti grandi e destabilizzanti durante la sua fase di addestramento.

Supponiamo di pianificare l’addestramento di un modello LLM per 15 epoche, partendo da un tasso di apprendimento iniziale di `0.0001` e aumentandolo fino a un tasso di apprendimento massimo di `0.01`:

```
n_epochs = 15
initial_lr = 0.0001
peak_lr = 0.01
warmup_steps = 20
```

Il numero di step di “riscaldamento” (`warmup_step`) è solitamente impostato tra lo `0.1` e il `20` percento del numero totale di step, che possiamo calcolare come segue:

```
total_steps = len(train_loader) * n_epochs
warmup_steps = int(0.2 * total_steps) (1)
print(warmup_steps)
```

(1) Warmup pari al 20 percento.

Si ottiene `27`, il che significa che abbiamo 20 fasi di warmup per aumentare il tasso di apprendimento iniziale da 0.0001 a 0.01 nelle prime 27 fasi di addestramento.

Ora, implementiamo un semplice ciclo di addestramento per illustrare questo processo di “riscaldamento”:

```
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.1)
lr_increment = (peak_lr - initial_lr) / warmup_steps (1)

global_step = -1
track_lrs = []
```

```
for epoch in range(n_epochs): (2)
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps: (3)
            lr = initial_lr + global_step * lr_increment
        else:
            lr = peak_lr

for param_group in optimizer.param_groups: (4)
    param_group["lr"] = lr
track_lrs.append(optimizer.param_groups[0]["lr"]) (5)
```

(1) Questo incremento è determinato da quanto aumentiamo il valore di `inital_lr` in ciascuna delle 20 fasi di warmup.

(2) Esegue un tipico ciclo di addestramento iterando sui batch nel loader di addestramento in ogni epoca.

(3) Aggiorna il tasso di apprendimento se ci troviamo ancora nella fase di warmup.

(4) Applica all'ottimizzatore il tasso di apprendimento calcolato.

(5) In un ciclo di addestramento completo, verrebbero calcolati il valore loss e gli aggiornamenti del modello, che qui vengono omessi per semplicità.

Dopo aver eseguito il codice precedente, notiamo come il tasso di apprendimento sia stato modificato dal ciclo di addestramento per verificare che il *Learning Rate Warmup* si comporti come previsto:

```
import matplotlib.pyplot as plt

plt.ylabel("Learning rate")
plt.xlabel("Step")
total_training_steps = len(train_loader) * n_epochs
plt.plot(range(total_training_steps), track_lrs);
plt.show()
```

Il grafico risultante mostra che il tasso di apprendimento inizia con un valore basso e aumenta per 20 passi fino a raggiungere il valore massimo dopo 20 passi (Figura D.1).

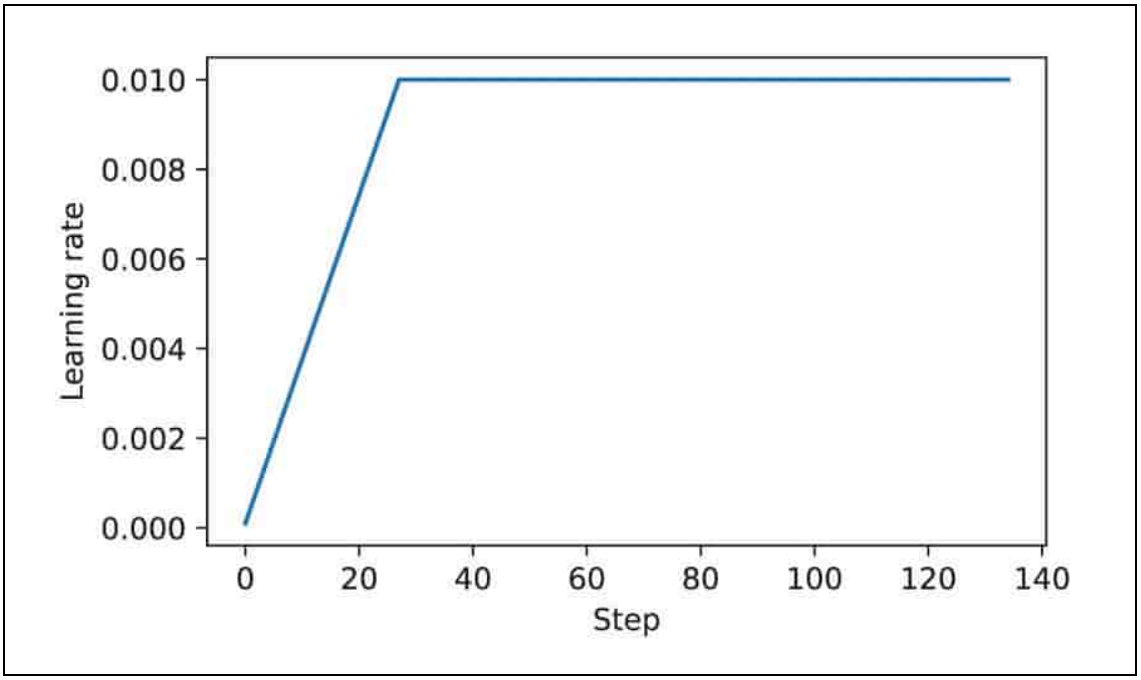


Figura D.1 La tecnica Learning Rate Warmup aumenta il tasso di apprendimento per i primi 20 passi dell'addestramento. Dopo 20 passi, il tasso di apprendimento raggiunge il picco di 0.01 e rimane costante per il resto dell'addestramento.

Ora, modificheremo ulteriormente il tasso di apprendimento in modo che diminuisca dopo aver raggiunto il livello massimo, il che contribuirà ulteriormente a migliorare l'addestramento del modello.

Cosine Decay

Un'altra tecnica ampiamente adottata per l'addestramento delle reti neurali profonde complesse e modelli LLM è *Cosine Decay*, che modula il tasso di apprendimento durante le epoche di addestramento, facendogli seguire una curva a coseno dopo la fase di warmup.

Nella sua variante più utilizzata, Cosine Decay riduce ("fa decadere") il tasso di apprendimento fino a quasi zero, imitando la curva di un ciclo di semi-coseno. La graduale diminuzione dell'apprendimento del Cosine Decay mira a rallentare il ritmo con

cui il modello aggiorna i suoi pesi. Ciò è particolarmente importante perché aiuta a minimizzare il rischio di superare i minimi di loss durante il processo di addestramento, il che è essenziale per garantire la stabilità dell'addestramento durante le fasi successive.

Possiamo modificare il modello del ciclo di addestramento aggiungendo la tecnica Cosine Decay:

```
import math

min_lr = 0.1 * initial_lr
track_lrs = []
lr_increment = (peak_lr - initial_lr) / warmup_steps
global_step = -1

for epoch in range(n_epochs):
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps: (1)
            lr = initial_lr + global_step * lr_increment
        else: (2)
            progress = ((global_step - warmup_steps) /
                        (total_training_steps - warmup_steps))
            lr = min_lr + (peak_lr - min_lr) * 0.5 * (
                1 + math.cos(math.pi * progress)
            )

            for param_group in optimizer.param_groups:
                param_group["lr"] = lr
            track_lrs.append(optimizer.param_groups[0]["lr"])
```

(1) Applica un warmup lineare.

(2) Utilizza Cosine Decay dopo il warmup.

Ancora una volta, per verificare che il tasso di apprendimento sia cambiato come previsto, tracciamone il grafico:

```
plt.ylabel("Learning rate")
plt.xlabel("Step")
plt.plot(range(total_training_steps), track_lrs)
plt.show()
```

Il grafico mostra che il tasso di apprendimento inizia con una fase di warmup lineare, che aumenta per 20 passi fino a raggiungere il valore massimo dopo 20 passi. Dopo i 20 passi di warmup lineare, entra in gioco Cosine Decay, che riduce gradualmente il tasso di apprendimento fino a raggiungere il minimo (Figura D.2).

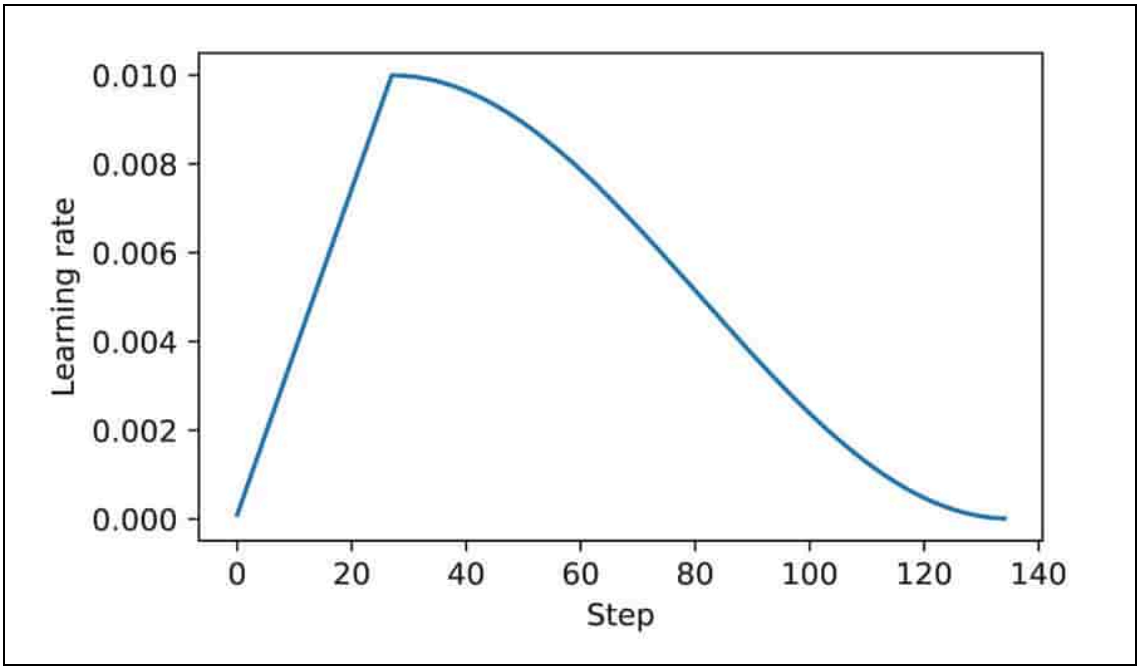


Figura D.2 I primi 20 passi del Learning Rate Warmup lineare sono seguiti da un Cosine Decay, che riduce il tasso di apprendimento in un ciclo di mezzo coseno fino a raggiungere il punto minimo alla fine dell'addestramento.

Gradient Clipping

Gradient Clipping è un'altra tecnica importante per migliorare la stabilità durante l'addestramento del modello LLM. Questo metodo comporta l'impostazione di una soglia al di sopra della quale i gradienti vengono riportati a una magnitudine massima predeterminata. Questo processo assicura che gli aggiornamenti dei parametri del modello durante la retropropagazione rimangano entro un intervallo accettabile.

Per esempio, applicando l'impostazione `max_norm=1.0` all'interno della funzione PyTorch `clip_grad_norm()`ci assicuriamo che la norma dei gradienti non superi 1.0. Qui, il termine "norma" indica la misura della lunghezza del vettore gradiente, o magnitudine, all'interno dello

spazio dei parametri del modello, riferendosi per la precisione alla norma L2, nota anche come *norma euclidea*.

In termini matematici, dato un vettore $v = [v_1, v_2, ..., v_n]$, la norma L2 è:

$$|v|_2 = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

Questo metodo di calcolo è applicato anche alle matrici. Per esempio, considerate la seguente matrice di gradienti:

$$\mathrm{G} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Se vogliamo limitare questi gradienti a un `max_norm` pari a `1`, calcoliamo prima la norma L2 di questi gradienti, che è:

$$|\mathrm{G}|_2 = \sqrt{1^2 + 2^2 + 2^2 + 4^2} = \sqrt{25} = 5$$

Dato che $|G|_2 = 5$ supera la nostra `max_norm`, `1`, riduciamo i gradienti per garantire che la norma sia esattamente uguale a `1`. Ciò si ottiene tramite un fattore di scala, calcolato come `max_norm` / $|G|_2 = 1/5$. Di conseguenza, la matrice G', aggiustata, diventa:

$$\mathrm{G}' = \frac{1}{5} \times G = \begin{bmatrix} \frac{1}{5} & \frac{2}{5} \\ \frac{2}{5} & \frac{4}{5} \end{bmatrix}$$

Per illustrare il processo di Gradient Clipping, inizializziamo un nuovo modello e calcoliamo il valore loss per un batch di addestramento, in modo simile alla procedura di un ciclo di addestramento standard:

```
from chapter05 import calc_loss_batch

torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
loss = calc_loss_batch(input_batch, target_batch, model, device)
loss.backward()
```

Dopo aver richiamato il metodo `.backward()`, PyTorch calcola i gradienti del valore loss e li memorizza in un attributo `.grad` per ciascun tensore (parametro) dei pesi del modello.

Per chiarire, possiamo definire la seguente funzione di servizio `find_highest_gradient()` per identificare il valore del gradiente più elevato eseguendo la scansione di tutti gli attributi `.grad` dei tensori dei pesi del modello dopo aver richiamato `.backward()`:

```
def find_highest_gradient(model):
    max_grad = None
    for param in model.parameters():
        if param.grad is not None:
            grad_values = param.grad.data.flatten()
            max_grad_param = grad_values.max()
            if max_grad is None or max_grad_param > max_grad:
                max_grad = max_grad_param
    return max_grad
print(find_highest_gradient(model))
```

Il gradiente più grande identificato dal codice precedente è il seguente:

```
tensor(0.0411)
```

Applichiamo ora il Gradient Clipping e vediamo come influisce sul valore del gradiente più grande:

```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
print(find_highest_gradient(model))
```

Il valore del gradiente più grande dopo l’applicazione del Gradient Clipping con norma massima pari a `1` è notevolmente più piccolo:

```
tensor(0.0185)
```

La funzione di addestramento modificata

Infine, miglioriamo la funzione di addestramento `train_model_simple()` (vedi il Capitolo 5) aggiungendo i tre concetti appena introdotti: Linear Warmup, Cosine Decay e Gradient Clipping. Insieme, questi metodi aiutano a stabilizzare l'addestramento del modello LLM.

Il codice, con annotate le modifiche rispetto al `train_model_simple()`, è il seguente:

```
from chapter05 import evaluate_model, generate_and_print_sample

def train_model(model, train_loader, val_loader, optimizer, device, n_epochs,
                eval_freq, eval_iter, start_context, tokenizer,warmup_steps,
                initial_lr=3e-05, min_lr=1e-6):

    train_losses, val_losses, track_tokens_seen, track_lrs = [], [], [], []
    tokens_seen, global_step = 0, -1

    peak_lr = optimizer.param_groups[0]["lr"] (1)
    total_training_steps = len(train_loader) * n_epochs (2)
    lr_increment = (peak_lr - initial_lr) / warmup_steps (3)

    for epoch in range(n_epochs):
        model.train()
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()
            global_step += 1

            if global_step < warmup_steps: (4)
                lr = initial_lr + global_step * lr_increment
            else:
                progress = ((global_step - warmup_steps) /
                            (total_training_steps - warmup_steps))
                lr = min_lr + (peak_lr - min_lr) * 0.5 * (1 + math.cos(math.pi *
progress))

            for param_group in optimizer.param_groups: (5)
                param_group["lr"] = lr
            track_lrs.append(lr)
            loss = calc_loss_batch(input_batch, target_batch, model, device)
            loss.backward()

            if global_step > warmup_steps: (6)
                torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

            optimizer.step() (7)
            tokens_seen += input_batch.numel()

            if global_step % eval_freq == 0:
                train_loss, val_loss = evaluate_model(model, train_loader,
```

```
val_loader,
                                                    device, eval_iter
                )
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                track_tokens_seen.append(tokens_seen)
                print(f"Ep {epoch+1} (Iter {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, "
                      f"Val loss {val_loss:.3f}"
                )

    generate_and_print_sample(model, tokenizer, device, start_context)

return train_losses, val_losses, track_tokens_seen, track_lrs
```

(1) Recupera il tasso di apprendimento iniziale dall'ottimizzatore, supponendo di utilizzarlo come tasso di apprendimento massimo.
(2) Calcola il numero totale di iterazioni nel processo di addestramento.
(3) Calcola l'incremento del tasso di apprendimento durante la fase di warmup.
(4) Regola il tasso di apprendimento in base alla fase corrente (Warmup o Cosine Decay).
(5) Applica all'ottimizzatore il tasso di apprendimento calcolato.
(6) Applica Gradient Clipping dopo la fase di warmup per evitare l'esplosione dei gradienti.
(7) Tutto quanto segue rimane invariato rispetto alla funzione `train_model_simple()` utilizzata nel Capitolo 5.

Dopo aver definito la funzione `train_model()`, possiamo utilizzarla in modo simile per addestrare il modello rispetto al metodo `train_model_simple()` che abbiamo utilizzato per il pre-addestramento:

```
import tiktoken

torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
peak_lr = 5e-4
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.1)
tokenizer = tiktoken.get_encoding("gpt2")

n_epochs = 15
train_losses, val_losses, tokens_seen, lrs = train_model(
    model, train_loader, val_loader, optimizer, device, n_epochs=n_epochs,
eval_freq=5,
```

```
    eval_iter=1, start_context="Every effort moves you", tokenizer=tokenizer,
    warmup_steps=warmup_steps, initial_lr=1e-5, min_lr=1e-5
)
```

L’addestramento richiederà circa 5 minuti su un MacBook Air o un laptop simile, e produrrà i seguenti output:

```
Ep 1 (Iter 000000): Train loss 10.934, Val loss 10.939
Ep 1 (Iter 000005): Train loss 9.151, Val loss 9.461
Every effort moves you,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
Ep 2 (Iter 000010): Train loss 7.949, Val loss 8.184
Ep 2 (Iter 000015): Train loss 6.362, Val loss 6.876
Every effort moves you,,,,,,,,,,,,,,,,,, the,,,,,,,,, the,,,,,,,,,, the,,,,,,,,
...
Ep 15 (Iter 000130): Train loss 0.041, Val loss 6.915
Every effort moves you?" "Yes--quite insensible to the irony. She wanted him
vindicated--and by me!" He laughed again, and threw back his head to look up at
the sketch of the donkey. "There were days when I
```

Come nel pre-addestramento, il modello inizia ad andare in overfitting dopo alcune epoche, poiché il dataset è molto piccolo e lo iteriamo più volte. Tuttavia, possiamo vedere che la funzione è efficace, poiché riduce al minimo il valore loss del set di addestramento.

Incoraggio i lettori ad addestrare il modello su un dataset di testi più ampio e a confrontare i risultati ottenuti con questa funzione di addestramento più sofisticata con i risultati che possono essere ottenuti con la funzione `train_model_simple()`.

Appendice E

Fine-tuning dei parametri con LoRA

LoRA (*Low-Rank Adaptation*) è una delle tecniche più utilizzate per il *fine-tuning dei parametri*. La seguente discussione si basa sull'esempio di fine-tuning della classificazione dello spam fornito nel Capitolo 6. Tuttavia, il fine-tuning LoRA è applicabile anche al fine-tuning delle istruzioni discussa nel Capitolo 7.

Introduzione a LoRA

LoRA è una tecnica che adatta un modello pre-addestrato per allinearlo meglio a un determinato dataset, spesso più piccolo, regolando solo un piccolo sottoinsieme dei parametri dei pesi del modello. La parte "Low-Rank" si riferisce al concetto matematico di limitare gli aggiustamenti del modello a un sottospazio dimensionale più piccolo dello spazio totale dei parametri dei pesi. Ciò cattura efficacemente le direzioni più influenti delle modifiche dei parametri dei pesi durante l'addestramento. Il metodo LoRA è utile, e molto utilizzato, perché consente un'efficiente fine-tuning di modelli di grandi dimensioni su dati specifici dell'attività, riducendo significativamente i costi e le risorse computazionali solitamente richiesti per il fine-tuning.

Supponiamo che a un determinato layer sia associata una grande matrice di pesi *W*. LoRA può essere applicato a tutti i layer lineari di un modello LLM. Tuttavia, a scopo illustrativo, ci concentreremo su un singolo layer.

Durante l'addestramento delle reti neurali profonde, nella fase di retropropagazione, apprendiamo una matrice ΔW, che contiene informazioni su quanto vogliamo aggiornare i parametri dei pesi originali per minimizzare la funzione loss durante l'addestramento. D'ora in poi, userò il termine "pesi" come abbreviazione di "parametri dei pesi del modello".

Nell'addestramento regolare e nel fine-tuning, l'aggiornamento dei pesi è definito come segue:

$$W_{\text{aggiornati}} = W + \Delta W$$

Il metodo LoRA, proposto da Hu et al. (https://arxiv.org/abs/2106.09685), offre un'alternativa più efficiente al calcolo degli aggiornamenti dei pesi ΔW apprendendone un'approssimazione:

$$\Delta W = AB$$

dove *A* e *B* sono due matrici molto più piccole di *W*, e *AB* rappresenta il prodotto delle matrici *A* e *B*.

Utilizzando LoRA, possiamo quindi riformulare l'aggiornamento dei pesi definito in precedenza:

$$W_{\text{aggiornati}} = W + AB$$

La Figura E.1 illustra le formule di aggiornamento dei pesi nel normale fine-tuning e con LoRA.

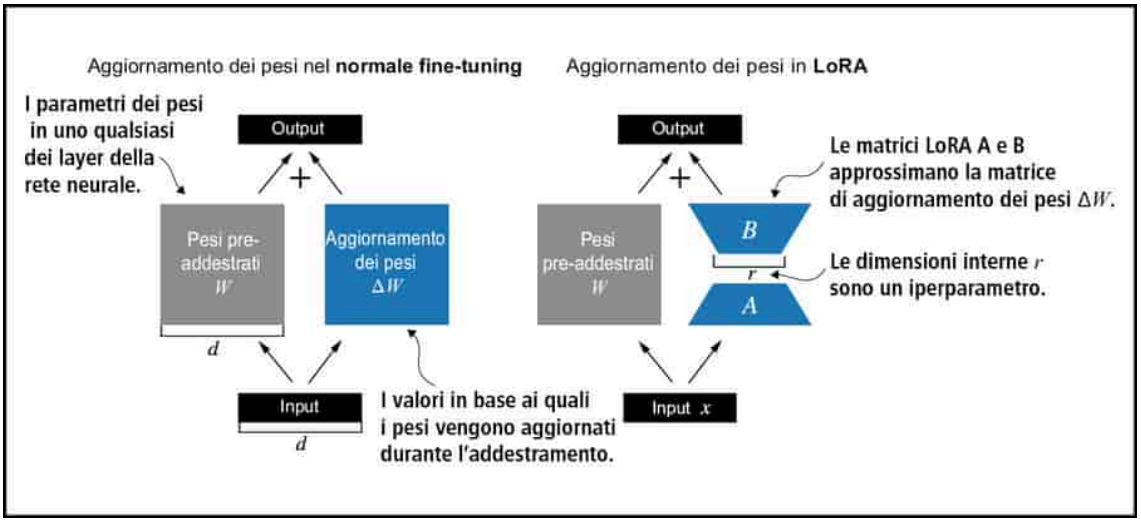


Figura E.1 Un confronto tra i due metodi di aggiornamento dei pesi: normale fine-tuning e LoRA. Il primo (a sinistra) comporta l'aggiornamento della matrice dei pesi pre-addestrata *W* direttamente con Δ*W*. LoRA (a destra) utilizza due matrici più piccole, *A* e *B*, per approssimare Δ*W*, dove il prodotto *AB* viene sommato a *W* e *r* denota le dimensioni interne, un iperparametro regolabile.

Potreste aver notato che le rappresentazioni visuali del fine-tuning e di LoRA nella Figura E.1 differiscono leggermente dalle formule presentate in precedenza. Questa variazione è attribuita alla legge distributiva della moltiplicazione delle matrici, che ci consente di separare i pesi originali e aggiornati anziché combinarli. Per esempio, nel caso del normale fine-tuning con dati di input *x*, possiamo esprimere il calcolo come:

$$x(W + \Delta W) = xW + x\Delta W$$

Allo stesso modo, possiamo scrivere quanto segue per LoRA:

$$x(W + AB) = xW + xAB$$

Oltre a ridurre il numero di pesi da aggiornare durante l'addestramento, la capacità di mantenere le matrici dei pesi LoRA separate dai pesi del modello originale rende LoRA ancora più utile. All'atto pratico, ciò consente ai pesi del modello pre-addestrato di

rimanere invariati, e le matrici LoRA vengono applicate dinamicamente dopo l'addestramento, quando si utilizza il modello.

Mantenere separati i pesi LoRA è molto utile perché consente di personalizzare il modello senza doverne salvare più versioni complete. Ciò riduce i requisiti di archiviazione e migliora la scalabilità, poiché solo le matrici LoRA, più piccole, devono essere regolate e salvate quando è necessario personalizzare i modelli LLM per ogni cliente o applicazione specifica.

Ora vediamo come LoRA può essere utilizzato per eseguire il fine-tuning di un modello LLM per la classificazione dello spam, in modo simile all'esempio di fine-tuning presentato nel Capitolo 6.

Preparazione del dataset

Prima di applicare LoRA all'esempio di classificazione dello spam, dobbiamo caricare il dataset e il modello pre-addestrato con cui lavoreremo. Il codice qui riportato ripete la preparazione dei dati del Capitolo 6. Invece di ripetere il codice, potreste aprire ed eseguire il notebook del Capitolo 6 e inserire lì il codice LoRA che presenterò alla fine di questa appendice.

Per prima cosa scarichiamo il dataset e salviamolo come file CSV.

Listato E.1 Download e preparazione del dataset.

```
from pathlib import Path
import pandas as pd
from ch06 import (download_and_unzip_spam_data,
                  create_balanced_dataset,
                  random_split
)

url = "https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip"
zip_path = "sms_spam_collection.zip"
extracted_path = "sms_spam_collection"
data_file_path = Path(extracted_path) / "SMSSpamCollection.tsv"

download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_path)

df = pd.read_csv(data_file_path, sep="\t", header=None, names=["Label", "Text"])
balanced_df = create_balanced_dataset(df)
```

```
balanced_df["Label"] = balanced_df["Label"].map({"ham": 0, "spam": 1})

train_df, validation_df, test_df = random_split(balanced_df, 0.7, 0.1)
train_df.to_csv("train.csv", index=None)

validation_df.to_csv("validation.csv", index=None)
test_df.to_csv("test.csv", index=None)
```

Ora, creiamo le istanze di `SpamDataset`.

Listato E.2 Istanziazione dei dataset PyTorch.

```
import torch
from torch.utils.data import Dataset
import tiktoken
from chapter06 import SpamDataset

tokenizer = tiktoken.get_encoding("gpt2")
train_dataset = SpamDataset("train.csv", max_length=None, tokenizer=tokenizer)
val_dataset = SpamDataset("validation.csv", max_length=train_dataset.max_length,
                          tokenizer=tokenizer
)
test_dataset = SpamDataset("test.csv", max_length=train_dataset.max_length,
                           tokenizer=tokenizer
)
```

Dopo aver creato gli oggetti del dataset PyTorch, creiamo i data loader.

Listato E.3 Creazione di data loader PyTorch.

```
from torch.utils.data import DataLoader

num_workers = 0
batch_size = 8

torch.manual_seed(123)

train_loader = DataLoader(dataset=train_dataset,
                          batch_size=batch_size,
                          shuffle=True,
                          num_workers=num_workers,
                          drop_last=True,
)

val_loader = DataLoader(dataset=val_dataset,
                        batch_size=batch_size,
                        num_workers=num_workers,
                        drop_last=False,
)

test_loader = DataLoader(dataset=test_dataset,
                         batch_size=batch_size,
                         num_workers=num_workers,
                         drop_last=False,
)
```

Come verifica, eseguiamo un'iterazione sui data loader e verifichiamo che i batch contengano otto esempi di addestramento ciascuno, dove ogni esempio di addestramento è costituito da 120 token:

```
print("Train loader:")
for input_batch, target_batch in train_loader:
    pass

print("Input batch dimensions:", input_batch.shape)
print("Label batch dimensions", target_batch.shape)
```

L'output è il seguente:

```
Train loader:
Input batch dimensions: torch.Size([8, 120])
Label batch dimensions torch.Size([8])
```

Infine, visualizziamo il numero totale di batch in ciascun dataset:

```
print(f"{len(train_loader)} training batches")
print(f"{len(val_loader)} validation batches")
print(f"{len(test_loader)} test batches")
```

In questo caso, abbiamo il seguente numero di batch per dataset:

```
130 training batches
19 validation batches
38 test batches
```

Inizializzazione del modello

Ripetiamo il codice del Capitolo 6 per caricare e preparare il modello GPT pre-addestrato. Iniziamo scaricando i pesi del modello e caricandoli nella classe `GPTModel`.

Listato E.4 Caricamento di un modello GPT pre-addestrato.

```
from gpt_download import download_and_load_gpt2
from chapter04 import GPTModel
from chapter05 import load_weights_into_gpt

CHOOSE_MODEL = "gpt2-small (124M)"
INPUT_PROMPT = "Every effort moves"

BASE_CONFIG = {"vocab_size": 50257, (1)
               "context_length": 1024, (2)
               "drop_rate": 0.0, (3)
               "qkv_bias": True (4)
}
```

```
model_configs = {"gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads":
12},
                 "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24,
"n_heads": 16},
                 "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads":
20},
                 "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads":
25},
}

BASE_CONFIG.update(model_configs[CHOOSE_MODEL])

model_size = CHOOSE_MODEL.split(" ")[-1].lstrip("(").rstrip(")")
settings, params = download_and_load_gpt2(model_size=model_size,
models_dir="gpt2")

model = GPTModel(BASE_CONFIG)
load_weights_into_gpt(model, params)
model.eval()
```

(1) Dimensioni vocabolario.

(2) Lunghezza contesto.

(3) Tasso dropout.

(4) Bias query-chiave-valore.

Per assicurarci che il modello sia stato caricato correttamente, verifichiamo che generi un testo coerente:

```
from chapter04 import generate_text_simple
from chapter05 import text_to_token_ids, token_ids_to_text

text_1 = "Every effort moves you"

token_ids = generate_text_simple(model=model,
                                 idx=text_to_token_ids(text_1, tokenizer),
                                 max_new_tokens=15,
                                 context_size=BASE_CONFIG["context_length"]
)

print(token_ids_to_text(token_ids, tokenizer))
```

Il seguente output mostra che il modello genera un testo coerente, il che indica che i pesi del modello sono caricati correttamente:

```
Every effort moves you forward.
The first step is to understand the importance of your work
```

Ora, prepariamo il modello per il fine-tuning per attività di classificazione, in modo simile a quanto fatto nel Capitolo 6, dove sostituiamo il layer di output:

```
torch.manual_seed(123)
num_classes = 2
model.out_head = torch.nn.Linear(in_features=768, out_features=num_classes)
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

Infine, calcoliamo l'accuratezza della classificazione iniziale del modello non ottimizzato (prevediamo che si aggiri intorno al 50 percento, il che significa che il modello non è ancora in grado di distinguere in modo affidabile tra messaggi spam e non spam):

```
torch.manual_seed(123)
train_accuracy = calc_accuracy_loader(train_loader, model, device, num_batches=10)
val_accuracy = calc_accuracy_loader(val_loader, model, device, num_batches=10)
test_accuracy = calc_accuracy_loader(test_loader, model, device, num_batches=10)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

Le accuratezze iniziali della previsione sono le seguenti:

```
Training accuracy: 46.25%
Validation accuracy: 45.00%
Test accuracy: 48.75%
```

Fine-tuning dei parametri con LoRA

Ora, modifichiamo e perfezioniamo il modello LLM usando LoRA. Iniziamo inizializzando un `LoRALayer` che crea le matrici *A* e *B*, insieme al fattore di scala `alfa` e all'impostazione `rank` (*r*). Questo layer può accettare un input e calcolare l'output corrispondente, come mostra la Figura E.2.

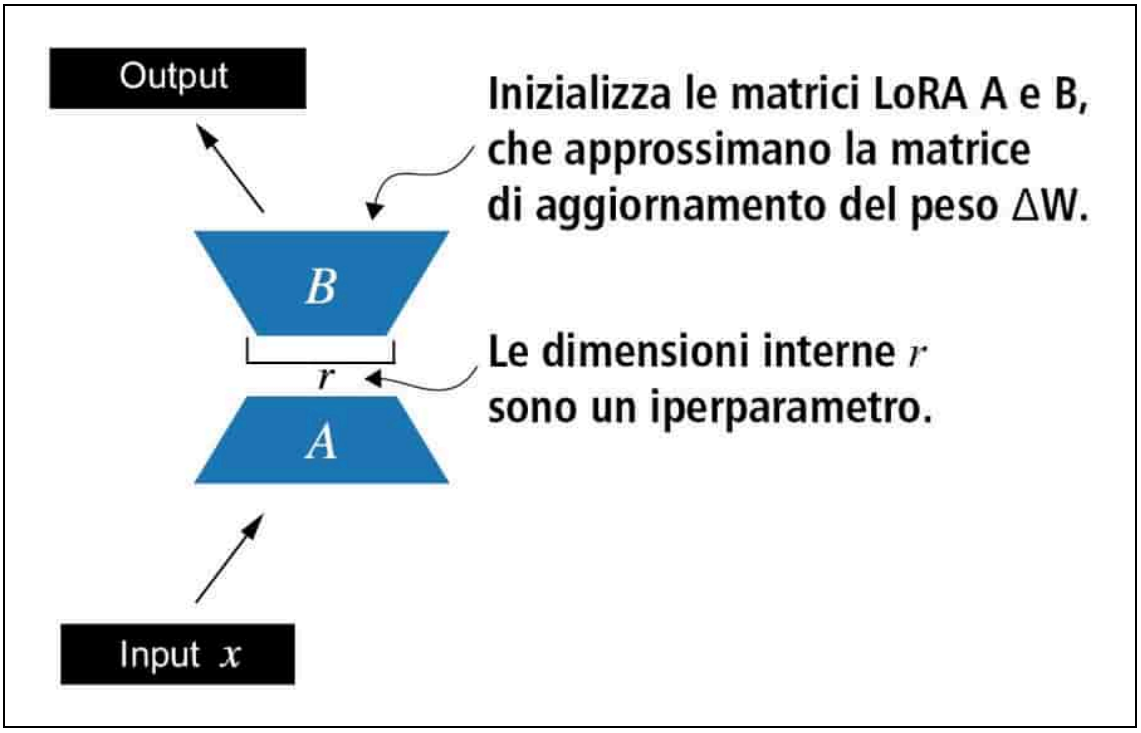


Figura E.2 Le matrici LoRA *A* e *B* vengono applicate agli input del layer e sono coinvolte nel calcolo degli output del modello. Le dimensioni interne *r* di queste matrici regolano il numero di parametri addestrabili variando le dimensioni di *A* e *B*.

Nel codice, questo layer LoRA può essere implementato come segue.

Listato E.5 Implementazione di un layer LoRA.

```
import math

class LoRALayer(torch.nn.Module):
    def __init__(self, in_dim, out_dim, rank, alpha):
        super().__init__()
        self.A = torch.nn.Parameter(torch.empty(in_dim, rank))
        torch.nn.init.kaiming_uniform_(self.A, a=math.sqrt(5)) (1)
        self.B = torch.nn.Parameter(torch.zeros(rank, out_dim))
        self.alpha = alpha

    def forward(self, x):
        x = self.alpha * (x @ self.A @ self.B)
        return x
```

(1) Stessa inizializzazione utilizzata per i layer lineari in PyTorch.

Il rango (`rank`) governa le dimensioni interne delle matrici *A* e *B*. In sostanza, questa impostazione determina il numero di parametri aggiuntivi introdotti da LoRA, creando un equilibrio tra l'adattabilità del modello e la sua efficienza tramite il numero di parametri utilizzati.

L'altra impostazione importante, `alpha`, opera da fattore di scala per l'output fornito dall'adattamento Low Rank. Determina principalmente il grado in cui l'output fornito dal layer adattato può influenzare l'output del layer originale. Questo può essere visto come un modo per regolare l'effetto dell'adattamento di Low Rank sull'output del layer. La classe `LoRALayer` che abbiamo implementato finora ci consente di trasformare gli input di un layer.

In LoRA, l'obiettivo tipico è sostituire i layer lineari esistenti, consentendo di aggiornare direttamente i pesi preesistenti pre-addestrati, come mostra la Figura E.3.

Per integrare i pesi originali del layer `Linear`, creiamo un layer `LinearWithLoRA` che utilizza il `LoRALayer` precedentemente implementato ed è progettato per sostituire i layer `Linear` esistenti all'interno di una rete neurale, come i moduli di auto-attenzione o i moduli feed-forward del `GPTModel`.

Listato E.6 Sostituzione di un layer LinearWithLoRA ai livelli Linear.

```
class LinearWithLoRA(torch.nn.Module):
    def __init__(self, linear, rank, alpha):
        super().__init__()
        self.linear = linear
        self.lora = LoRALayer(linear.in_features, linear.out_features, rank,
alpha)

    def forward(self, x):
        return self.linear(x) + self.lora(x)
```

Questo codice combina un layer `Linear` standard col layer `LoRALayer`. Il metodo `forward()` calcola l'output sommando i risultati del layer

lineare originale e del layer LoRA.

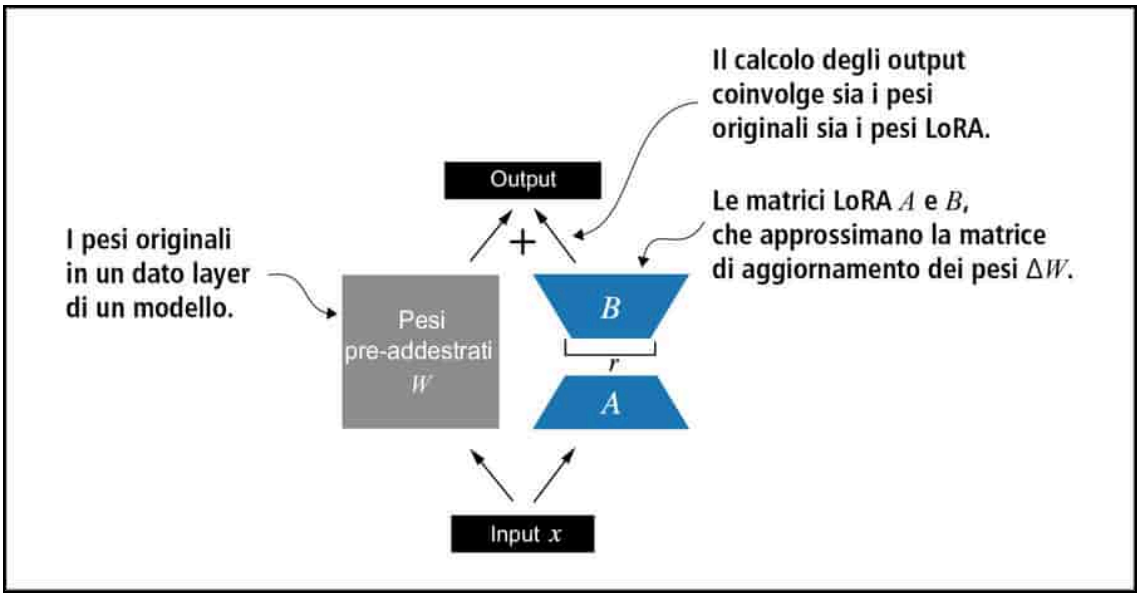


Figura E.3 L'integrazione di LoRA in un layer del modello. I pesi pre-addestrati originali (*W*) di un layer vengono combinati con gli output delle matrici LoRA (*A* e *B*), che approssimano la matrice di aggiornamento dei pesi (Δ*W*). L'output finale viene calcolato aggiungendo all'output originale l'output del layer adattato (utilizzando i pesi LoRA).

Poiché la matrice dei pesi *B* (`self.B` in `LoRALayer`) è inizializzata con valori `0`, il prodotto delle matrici *A* e *B* produce una matrice zero. Ciò garantisce che la moltiplicazione non alteri i pesi originali, poiché l'aggiunta di zero non crea modifiche.

Per applicare LoRA al `GPTModel` definito in precedenza, introduciamo una funzione `replace_linear_with_lora()`. Questa funzione scambierà tutti i layer `Linear` esistenti nel modello con i layer `LinearWithLoRA` appena creati:

```
def replace_linear_with_lora(model, rank, alpha):
    for name, module in model.named_children():
        if isinstance(module, torch.nn.Linear): (1)
            setattr(model, name, LinearWithLoRA(module, rank, alpha))
        else: (2)
            replace_linear_with_lora(module, rank, alpha)
```

(1) Sostituisce il livello `Linear` con `LinearWithLoRA`.

(2) Applica ricorsivamente la stessa funzione ai moduli figli.

Abbiamo implementato tutto il codice necessario per sostituire i layer `Linear` nel `GPTModel` con i layer `LinearWithLoRA` per un fine-tuning efficiente dei parametri. Ora, applicheremo l'aggiornamento `LinearWithLoRA` a tutti i layer `Linear` nei moduli di attenzione multi-head, feed-forward e nel layer di output del `GPTModel`, come mostra la Figura E.4.

Prima di applicare gli aggiornamenti del layer `LinearWithLoRA`, congeliamo i parametri del modello originale:

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters before: {total_params:,}")

for param in model.parameters():
    param.requires_grad = False

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters after: {total_params:,}")
```

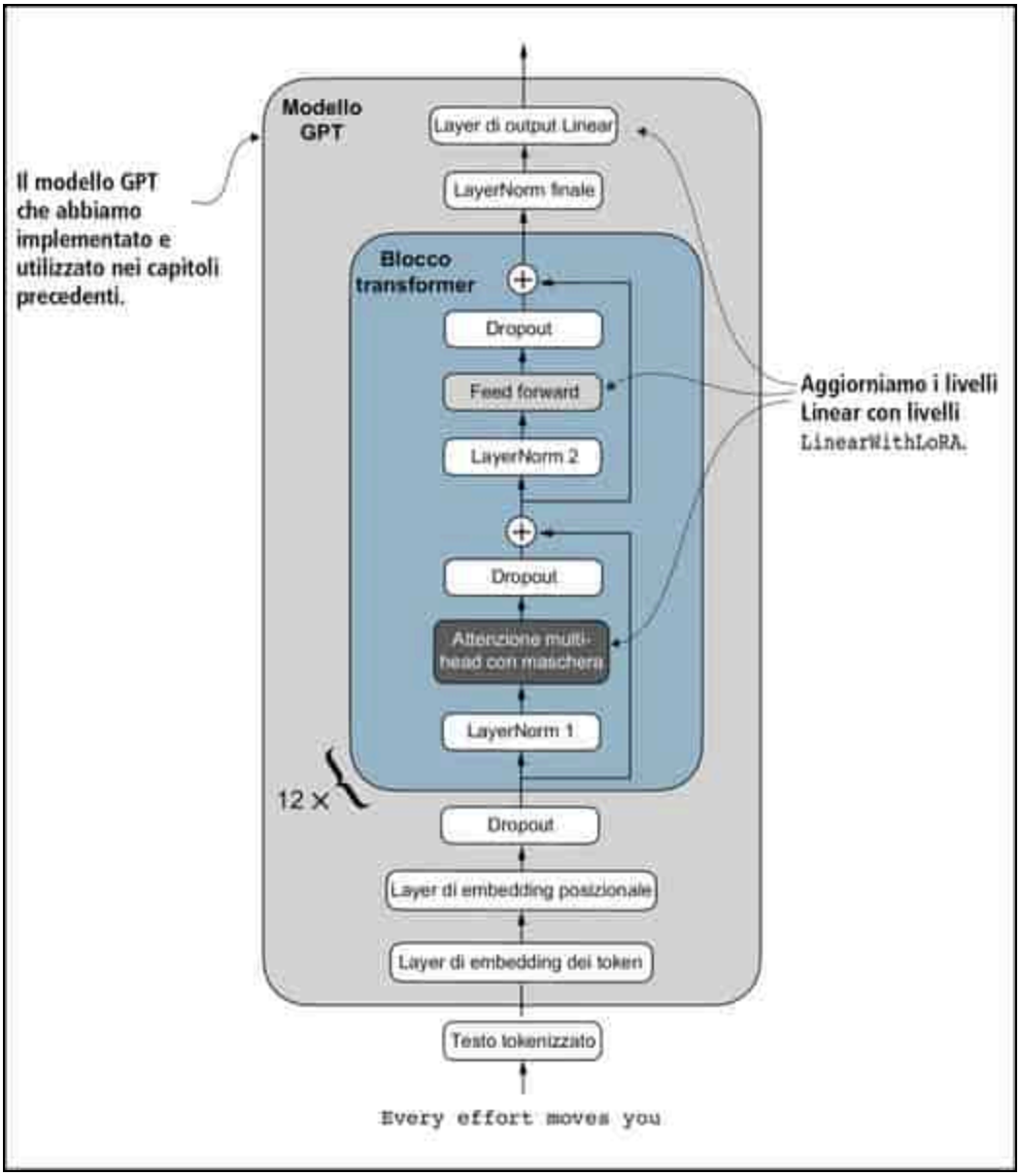


Figura E.4 L'architettura del modello GPT. Sono evidenziate le parti del modello in cui i layer Linear vengono aggiornati a layer LinearWithLoRA per un fine-tuning efficiente dei parametri.

Ora possiamo vedere che nessuno dei 124 milioni di parametri del modello è addestrabile:

```
Total trainable parameters before: 124,441,346
Total trainable parameters after: 0
```

Ora, utilizziamo `replace_linear_with_lora()` per sostituire i layer `Linear`:

```
replace_linear_with_lora(model, rank=16, alpha=16)
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable LoRA parameters: {total_params:,}")
```

Dopo aver aggiunto i layer LoRA, il numero di parametri addestrabili è il seguente:

```
Total trainable LoRA parameters: 2,666,528
```

Come potete vedere, abbiamo ridotto il numero di parametri addestrabili di quasi 50 volte utilizzando LoRA. Un `rank` e un `alpha` pari a 16 sono buone scelte standard, ma è pratica comune aumentare il parametro `rank`, che a sua volta aumenta il numero di parametri addestrabili. Di solito, l'`alpha` viene scelto in modo che sia la metà, il doppio o uguale a `rank`.

Verifichiamo che i layer siano stati modificati come previsto mostrando l'architettura del modello:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
print(model)
```

L'output è il seguente:

```
GPTModel(
  (tok_emb): Embedding(50257, 768)
  (pos_emb): Embedding(1024, 768)
  (drop_emb): Dropout(p=0.0, inplace=False)
  (trf_blocks): Sequential(
  ...
    (11): TransformerBlock(
      (att): MultiHeadAttention(
        (W_query): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (W_key): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (W_value): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (out_proj): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (dropout): Dropout(p=0.0, inplace=False)
```

```
      )
      (ff): FeedForward(
        (layers): Sequential(
          (0): LinearWithLoRA(
            (linear): Linear(in_features=768, out_features=3072, bias=True)
            (lora): LoRALayer()
          )
          (1): GELU()
          (2): LinearWithLoRA(
            (linear): Linear(in_features=3072, out_features=768, bias=True)
            (lora): LoRALayer()
          )
        )
      )
      (norm1): LayerNorm()
      (norm2): LayerNorm()
      (drop_resid): Dropout(p=0.0, inplace=False)
    )
  )
  (final_norm): LayerNorm()
  (out_head): LinearWithLoRA(
    (linear): Linear(in_features=768, out_features=2, bias=True)
    (lora): LoRALayer()
  )
)
```

Il modello ora include i nuovi livelli `LinearWithLoRA`, che a loro volta sono costituiti dai livelli `Linear` originali, impostati come non addestrabili, e dai nuovi layer LoRA, dei quali eseguiremo il fine-tuning.

Prima di iniziare a eseguire il fine-tuning del modello, calcoliamo l'accuratezza della classificazione iniziale:

```
torch.manual_seed(123)

train_accuracy = calc_accuracy_loader(train_loader, model, device, num_batches=10)
val_accuracy = calc_accuracy_loader(val_loader, model, device, num_batches=10)
test_accuracy = calc_accuracy_loader(test_loader, model, device, num_batches=10)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

I valori di accuratezza risultanti sono i seguenti:

```
Training accuracy: 46.25%
Validation accuracy: 45.00%
Test accuracy: 48.75%
```

Questi valori di accuratezza sono identici ai valori del Capitolo 6. Questo risultato si verifica perché abbiamo inizializzato la matrice LoRA *B* con valori `0`. Di conseguenza, il prodotto delle matrici *AB*

produce una matrice zero. Ciò garantisce che la moltiplicazione non alteri i pesi originali, poiché l'aggiunta di zero non li modifica.

Passiamo ora alla parte più interessante: il fine-tuning del modello tramite la funzione di addestramento del Capitolo 6. L'addestramento dura circa 15 minuti su un laptop MacBook Air M3 e meno di mezzo minuto su una GPU V100 o A100.

Listato E.7 Ottimizzazione di un modello con layer LoRA.

```
import time
from chapter06 import train_classifier_simple

start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5, weight_decay=0.1)

num_epochs = 5
train_losses, val_losses, train_accs, val_accs, examples_seen =
train_classifier_simple(
    model, train_loader, val_loader, optimizer, device, num_epochs=num_epochs,
    eval_freq=50, eval_iter=5, tokenizer=tokenizer
)

end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Training completed in {execution_time_minutes:.2f} minutes.")
```

L'output che vediamo durante l'addestramento è il seguente:

```
Ep 1 (Step 000000): Train loss 3.820, Val loss 3.462
Ep 1 (Step 000050): Train loss 0.396, Val loss 0.364
Ep 1 (Step 000100): Train loss 0.111, Val loss 0.229
Training accuracy: 97.50% | Validation accuracy: 95.00%
Ep 2 (Step 000150): Train loss 0.135, Val loss 0.073
Ep 2 (Step 000200): Train loss 0.008, Val loss 0.052
Ep 2 (Step 000250): Train loss 0.021, Val loss 0.179
Training accuracy: 97.50% | Validation accuracy: 97.50%
Ep 3 (Step 000300): Train loss 0.096, Val loss 0.080
Ep 3 (Step 000350): Train loss 0.010, Val loss 0.116
Training accuracy: 97.50% | Validation accuracy: 95.00%
Ep 4 (Step 000400): Train loss 0.003, Val loss 0.151
Ep 4 (Step 000450): Train loss 0.008, Val loss 0.077
Ep 4 (Step 000500): Train loss 0.001, Val loss 0.147
Training accuracy: 100.00% | Validation accuracy: 97.50%
Ep 5 (Step 000550): Train loss 0.007, Val loss 0.094
Ep 5 (Step 000600): Train loss 0.000, Val loss 0.056
Training accuracy: 100.00% | Validation accuracy: 97.50%

Training completed in 12.10 minutes.
```

L'addestramento del modello con LoRA ha richiesto più tempo rispetto a quello senza LoRA (vedi il Capitolo 6) perché i layer LoRA

introducono un calcolo aggiuntivo durante il passo forward. Tuttavia, per modelli più grandi, in cui la retropropagazione diventa più costosa, i modelli in genere si addestrano più velocemente con LoRA che senza.

Come potete vedere, il modello ha ricevuto un addestramento perfetto e un'accuratezza di validazione molto elevata. Visualizziamo anche le curve di loss, per vedere meglio se l'addestramento converge:

```
from chapter06 import plot_values

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_losses))

plot_values(epochs_tensor, examples_seen_tensor,train_losses, val_losses,
label="loss")
```

La Figura E.5 riporta i risultati.

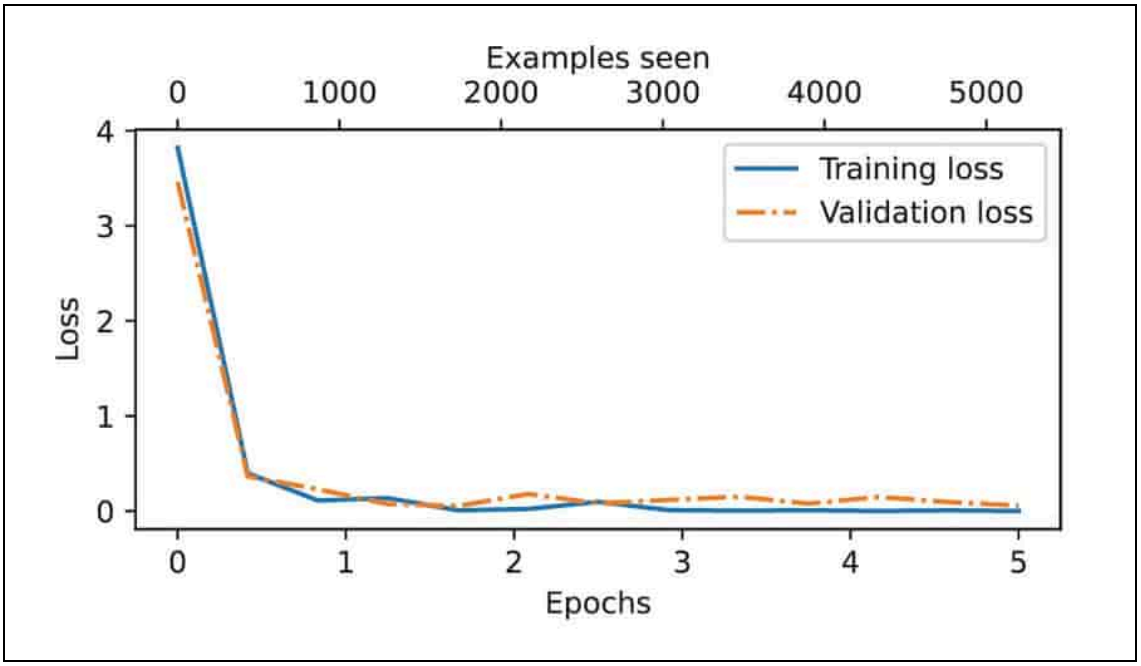


Figura E.5 Le curve di loss di addestramento e validazione su sei epoche per un modello di machine learning. Inizialmente, il valore loss sia di addestramento sia di validazione diminuiscono bruscamente e poi si stabilizzano, indicando che il modello sta

convergendo, il che significa che non ci si aspetta che migliori particolarmente con un ulteriore addestramento.

Oltre a valutare il modello in base alle curve di loss, calcoliamo anche le accuratezze sugli interi set di addestramento, validazione e test (durante l'addestramento, abbiamo approssimato le accuratezze del set di addestramento e validazione da cinque batch tramite l'impostazione `eval_iter=5`):

```
train_accuracy = calc_accuracy_loader(train_loader, model, device)
val_accuracy = calc_accuracy_loader(val_loader, model, device)
test_accuracy = calc_accuracy_loader(test_loader, model, device)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

I valori di accuratezza risultanti sono i seguenti:

```
Training accuracy: 100.00%
Validation accuracy: 96.64%
Test accuracy: 98.00%
```

Questi risultati mostrano che il modello funziona bene nei set di addestramento, validazione e test. Con un'accuratezza di addestramento del 100 percento, il modello ha appreso perfettamente i dati di addestramento. Tuttavia, le accuratezze di validazione e test leggermente inferiori (rispettivamente 96.64 e 97.33 percento) suggeriscono un piccolo grado di overfitting, poiché il modello non si generalizza altrettanto bene su dati non visti rispetto al set di addestramento. Nel complesso, i risultati sono davvero impressionanti, considerando che abbiamo eseguito il fine-tuning di un numero relativamente piccolo di pesi del modello (2.7 milioni di pesi LoRA invece dei 124 milioni di pesi del modello originale).

Glossario

A/B testing Metodo di confronto tra due versioni (A e B) di un elemento (per esempio una pagina web, un annuncio, un'app) per determinare quale si comporta meglio in base a una determinata metrica (per esempio clic, conversioni). Si usa per ottimizzare le decisioni basandosi su dati reali.

Accuratezza (*accuracy*) Metrica che misura la percentuale di previsioni corrette rispetto al totale delle previsioni in un modello di machine learning o statistica. Se un modello classifica correttamente 90 esempi su 100, la sua accuracy è del 90%. *Nota*: è utile in contesti bilanciati, ma in caso di classi sbilanciate può essere fuorviante.

AGI (*Artificial General Intelligence*) Tipo di intelligenza artificiale in grado di comprendere, apprendere e applicare la conoscenza in modo autonomo e versatile, operando come un essere umano, su qualsiasi compito cognitivo. A differenza dell'AI specializzata (*narrow AI*), ha capacità generali e adattive.

Airflow *Apache Airflow* è una piattaforma open-source per programmare, orchestrare e monitorare flussi di lavoro complessi, automatizzandone l'esecuzione e la gestione tramite codice. Si usa nel *data engineering*, permettendo di definire, pianificare e tracciare task in modo programmatico con Python.

Alberi a ricorsione Diagrammi che rappresentano visivamente le chiamate ricorsive di una funzione, mostrando come un problema può essere scomposto in sotto-problemi più piccoli, fino a raggiungere il

caso base. Sono utili per analizzare la complessità temporale degli algoritmi ricorsivi.

Alberi decisionali Modelli di machine learning che suddividono i dati in base a regole if-then, formando una struttura ad albero per classificare o prevedere i risultati. Sono intuitivi, facili da interpretare e utili per problemi sia di classificazione sia di regressione.

Algoritmi genetici Metodi di ottimizzazione ispirati all'evoluzione naturale, che utilizzano meccanismi come la selezione, il crossover ("incrocio") e la mutazione per generare soluzioni migliori per problemi complessi.

Allucinazione Fenomeno in cui un modello di intelligenza artificiale genera informazioni errate, distorte o inventate, presentandole come se fossero vere o plausibili. Accade spesso con modelli per la generazione di linguaggio naturale (come GPT) quando producono risposte basate non su dati di input o conoscenze reali, ma su pattern appresi in modo imperfetto durante l'addestramento. Per esempio, un chatbot può inventare fatti, anche storici, inesistenti o citare fonti inattendibili o inventate.

Analisi esplorativa dei dati (EDA - *Exploratory Data Analysis*) Approccio statistico per esaminare, riassumere e visualizzare i dati con lo scopo di identificarne modelli, relazioni, anomalie o tendenze, spesso prima di applicare i modelli formali. L'obiettivo è quello di comprendere la struttura dei dati e di generare ipotesi.

Analisi multivariata Insieme di tecniche statistiche che esaminano contemporaneamente più variabili per studiarne le relazioni, i modelli e le strutture sottostanti. Viene utilizzata per analizzare dati complessi.

ANN (*Artificial Neural Network*) Modello computazionale ispirato alle reti neurali del cervello, composto da layer (strati) di

nodi (i “neuroni”) interconnessi che elaborano i dati per apprendere pattern complessi.

Apache Spark Framework open-source per l’elaborazione distribuita di big data, ottimizzato per velocità e facilità d’uso. Supporta l’elaborazione batch, lo streaming, il machine learning e le query SQL, sfruttando la memoria RAM per offrire prestazioni superiori.

Attention mechanism Questo “meccanismo d’attenzione” è un componente dei modelli di deep learning (specialmente nelle reti neurali sequenziali, come i transformer) che permette di “pesare” l’importanza delle diverse parti dell’input (per esempio le parole di una frase) per migliorare l’elaborazione dei dati.

Autoencoder Rete neurale senza supervisione che impara a comprimere e ricostruire i dati, riducendoli in una rappresentazione più compatta (encoding) e poi decodificandoli per riavvicinarli all’input originale. Si usa per ridurre la dimensionalità e la rumorosità dei dati o per generare nuovi dati.

Backpropagation La “retropropagazione” è l’algoritmo di apprendimento per reti neurali che calcola il gradiente dell’errore rispetto ai pesi, propagando all’indietro gli errori per ottimizzare i parametri attraverso la discesa del gradiente.

Bagging (*Bootstrap Aggregating*) Tecnica di machine learning che migliora la stabilità e l’accuratezza di un modello addestrando più versioni dello stesso algoritmo su sottoinsiemi casuali differenti (e con ripetizione) dei dati e combinando le previsioni ottenute, tramite media o voto maggioritario.

Batch size Numero di campioni (il “lotto” degli esempi di addestramento) elaborati contemporaneamente da un modello di machine learning in una singola iterazione durante l’addestramento.

BERT (*Bidirectional Encoder Representations from Transformers*) Modello per il linguaggio basato su reti neurali a transformer, sviluppato da Google per l'elaborazione del linguaggio naturale (NLP - *Natural Language Processing*). A differenza dei modelli precedenti, BERT analizza il testo in modo *bidirezionale*, per estrarre meglio il contesto delle parole. È usato per compiti come la traduzione, le risposte a domande e la classificazione dei testi.

Bias cognitivo Errore algoritmico sistematico commesso dai modelli di machine learning che porta a risultati ingiusti o discriminatori, spesso a causa di dati di addestramento distorti o di presupposti errati. Per esempio, un sistema di assunzione automatizzato può favorire candidati di un certo genere o etnia a causa di pregiudizi insiti nei dati storici.

Bias-variance tradeoff Compromesso nella modellazione predittiva tra un bias (errore sistematico) dovuto a un modello troppo semplice e la varianza, una sensibilità eccessiva alle fluttuazioni del set di addestramento dovuta a un modello troppo complesso. L'obiettivo è quello di trovare un equilibrio carente (underfitting) o eccessivo (overfitting) adattamento ai dati di addestramento.

Big Data Enormi volumi di dati (strutturati o non strutturati) caratterizzati da "3 V: grande volume (quantità), grande velocità (di generazione/elaborazione) e grande varietà (di formati e/o fonti).

Boosting Tecnica di machine learning che combina più modelli deboli (per esempio alberi decisionali semplici) a formare un unico modello forte, addestrandoli sequenzialmente per correggere gli errori dei precedenti.

Chatbot Software progettato per simulare una conversazione con utenti umani, spesso tramite messaggi testuali o vocali, per fornire risposte automatizzate, assistenza o eseguire compiti specifici.

Clustering Tecnica di machine learning che raggruppa dati simili in insiemi (*cluster*) in base alle loro caratteristiche, senza conoscere a priori le categorie. Utile per la segmentazione dei clienti e il riconoscimento di pattern.

CNN (*Convolutional Neural Network*) Tipo di rete neurale artificiale specializzata nell'elaborazione di dati con struttura a griglia, come immagini. Utilizzano operazioni di convoluzione per estrarre automaticamente caratteristiche gerarchiche (come bordi, tramature, oggetti), riducendo al contempo i parametri grazie a meccanismi come il pooling. Sono ampiamente usate in computer vision per compiti come classificazione, rilevamento di oggetti e segmentazione.

Computer vision Campo dell'intelligenza artificiale che permette ai computer di interpretare e comprendere immagini e video, simulando la vista umana per compiti come il riconoscimento di oggetti, la classificazione e l'analisi visiva.

Confusion matrix Questa "matrice di confusione" è una tabella di quattro elementi che riassume le performance di un modello di classificazione, mostrando i veri positivi (correttamente previsti come positivi), i falsi positivi (erroneamente previsti come positivi), i veri negativi (correttamente previsti come negativi) e i falsi negativi (erroneamente previsti come negativi). Utile per calcolare metriche come l'accuratezza, la precisione, il recall e l'F1-score.

Correlazione vs causalità La *correlazione* si ha quando vi è una relazione statistica tra due variabili, ma senza implicare che una causi l'altra. La causalità, invece, implica che un cambiamento in una variabile (la causa) provochi direttamente un cambiamento in un'altra (l'effetto). Per esempio, se vi è una maggiore vendita di ghiaccioli, ci saranno anche più casi di scottature solari: vi è una *correlazione*, poiché entrambe le cose sono legate al caldo. Al

contrario l’esposizione al sole senza protezione (la causa) provoca (l’effetto): qui vi è una *causalità*. Questo ha un’importante conseguenza: che la correlazione non implica la causalità: solo perché due cose variano insieme, non significa che una influenzi l’altra.

Cross-validation La “validazione incrociata” (o “convalida incrociata”) è una tecnica statistica utilizzata per valutare e confrontare i modelli di machine learning, suddividendo i dati in più parti (fold) per allenare il modello su determinati sottoinsiemi e sottoporlo a test su altri, per stimarne le prestazioni in modo solido e affidabile.

Data cleaning Processo di identificazione e correzione degli errori, delle incoerenze e dei dati mancanti in un dataset, per migliorarne la qualità e l’affidabilità.

Data ethics Ramo dell’etica che si occupa del trattamento responsabile dei dati, a riguardo di aspetti come la privacy, l’equità, la trasparenza e la responsabilità nella raccolta, nell’analisi e nell’utilizzo delle informazioni, specialmente in contesti come l’intelligenza artificiale e i big data.

Data governance Insieme di processi, politiche, standard e ruoli che garantiscono un uso corretto, sicuro e conforme dei dati all’interno di un’azienda o organizzazione.

Data lake Sistema di archiviazione che raccoglie e conserva grandi volumi di dati grezzi, strutturati e non strutturati, in formato nativo, senza uno schema predefinito. Consente un’analisi flessibile e scalabile per diverse esigenze, come business intelligence, machine learning ed elaborazione di big data. Può trattarsi di file log, file CSV, immagini e dati di sensori.

Data pipeline Sistema automatizzato che trasferisce, elabora e trasforma dati da diverse fonti a una destinazione (come un database

o un’applicazione), garantendo flusso, qualità e utilità delle informazioni.

Data visualization Rappresentazione grafica di dati e informazioni, utilizzando elementi visivi come grafici, mappe e diagrammi per facilitarne la comprensione e l’analisi.

Data warehouse Sistema di archiviazione e gestione dei dati progettato per l’analisi e il reporting, che integra informazioni da più fonti in un formato strutturato e ottimizzato per consentire l’esecuzione di query complesse.

Data wrangling Processo di pulizia, trasformazione e organizzazione dei dati grezzi per renderli più adatti all’analisi.

Dataset sbilanciato Insieme di dati in cui le classi o categorie sono rappresentate in modo disomogeneo, con una o più classi predominanti (maggioranza) e altre molto meno presenti (minoranza). Per esempio, in un dataset per rilevare frodi, le transazioni fraudolente (1%) sono molto meno frequenti di quelle legittime (99%).

DBSCAN (*Density-Based Spatial Clustering of Applications with Noise*) Algoritmo di clustering basato sulla densità, che raggruppa in cluster i punti vicini, identificando anche i punti di “rumore” (*outlier*). Impiega due parametri principali: ε, *epsilon*, (distanza massima per considerare due punti come “vicini”) e MinPts (numero minimo di punti per formare un cluster).

Discesa del gradiente (gradient descent) Algoritmo iterativo di ottimizzazione utilizzato per “minimizzare” una funzione di costo, aggiornando i parametri del modello nella direzione opposta al gradiente della funzione rispetto a tali parametri. In breve, calcola il gradiente (derivata parziale) della funzione di costo, aggiorna i parametri muovendosi nella direzione opposta al gradiente con un

certo *learning rate* (passo di apprendimento) e ripete fino a giungere a una convergenza.

DNN (*Deep Neural Networks*) Rete neurale artificiale composta da più layer (strati) di neuroni interconnessi, progettata per apprendere rappresentazioni gerarchiche e complesse dai dati. È un'estensione delle reti neurali tradizionali, con più layer nascosti, utilizzata in applicazioni avanzate come il riconoscimento di immagini, l'elaborazione del linguaggio naturale e altro ancora.

Dropout Tecnica di regolarizzazione usata nelle reti neurali per prevenire l'overfitting. Durante l'addestramento, disattiva casualmente alcuni neuroni (con una probabilità p), costringendo la rete a imparare con sottoinsiemi diversi di unità, migliorando così la generalizzazione.

DVC (*Data Version Control*) Strumento open-source che permette di gestire le versioni di dati, modelli di machine learning e pipeline, integrandosi con Git per gestire in modo efficiente file di grandi dimensioni.

Embedding Rappresentazione numerica (vettoriale) di oggetti come parole, immagini o dati, che cattura le loro caratteristiche e relazioni in uno spazio multidimensionale. Per esempio, il word embedding rappresenta le parole come vettori, dove parole simili (come "cane" e "cagnolino") hanno rappresentazioni vettoriali vicine.

Ensemble metodi/modelli Tecniche di machine learning che combinano più modelli (*modelli base* o *weak learner*) per ottenere prestazioni complessive migliori, sfruttando la diversità o la media delle loro previsioni. Gli esempi più comuni includono il *bagging* (foreste casuali), il boosting (AdaBoost, XGBoost) e lo stacking (combinazione di modelli eterogenei). L'obiettivo è quello di ridurre l'overfitting, aumentare la robustezza e migliorare l'accuratezza.

ETL (*Extract, Transform, Load*) Processo di integrazione dei dati che prevede l'estrazione dei dati da diverse fonti (database, file, API, ecc.), la trasformazione dei dati (pulizia, normalizzazione, aggregazione) e il caricamento in un sistema di destinazione (data warehouse, database analitico). Lo scopo principale è quello di consolidare i dati per scopi di analisi e reporting.

Euristica Strategia cognitiva che semplifica i processi decisionali, utilizzando regole pratiche o "scorciatoie mentali" per arrivare rapidamente a soluzioni approssimate, spesso senza analizzare tutti i dati. È utile in contesti di incertezza o complessità, ma può portare a errori sistematici (i cosiddetti bias cognitivi).

F1-score Metrica che combina precisione e recall in un unico valore, calcolato come la media armonica delle due. La formula è:

$$F1 = 2 \times (\text{Precisione} \times \text{Recall}) / (\text{Precisione} + \text{Recall})$$

Dove:

$$\text{Precisione} = \text{Veri positivi} / (\text{Veri positivi} + \text{Falsi positivi})$$

$$\text{Recall} = \text{Veri positivi} / (\text{Veri positivi} + \text{Falsi negativi})$$

È utile quando le classi sono sbilanciate e si vuole avere un punteggio equilibrato tra accuratezza e copertura. Il valore massimo è 1 (perfetto), il minimo è 0.

Feature engineering Processo di selezione, trasformazione e creazione di caratteristiche (*feature*) dai dati grezzi per migliorare le prestazioni di un modello di machine learning.

Feature selection Processo di selezione di un sottoinsieme di caratteristiche (*feature*) rilevanti da un dataset, eliminando quelle ridondanti o irrilevanti, per migliorare le prestazioni del modello e ridurne la complessità computazionale.

Fine-tuning Processo di ottimizzazione in cui un modello pre-addestrato (per esempio, un modello di machine learning o di deep

learning) viene ulteriormente addestrato su un determinato dataset per migliorarne le prestazioni in un compito specifico.

Funzione di costo In inglese *loss function*, è una misura matematica che quantifica l'errore tra i valori previsti da un modello e i valori reali. L'obiettivo è quello di minimizzare questa funzione per migliorare le prestazioni del modello.

GAN (*Generative Adversarial Networks*) Modello di apprendimento automatico composto da due reti neurali in competizione: il *generatore* crea dati sintetici simili a quelli reali; il *discriminatore* cerca di distinguere i dati reali da quelli generati. Le due reti migliorano reciprocamente attraverso un addestramento "ad avversario", producendo risultati sempre più realistici. Usate per generare immagini, video, musica e così via.

GenAI L'intelligenza artificiale generativa si riferisce a sistemi di intelligenza artificiale in grado di generare nuovi contenuti, come testo, immagini, musica o codice, in modo autonomo e creativo. Esempi noti includono ChatGPT per il testo o DALL-E per le immagini.

Generalizzazione Processo mediante il quale un apprendimento, una regola o una conclusione tratta da casi specifici viene estesa a contesti più ampi o a una categoria generale. Nel *machine learning* è un modello che funziona bene con dati nuovi, non solo quelli di addestramento.

GPT (*Generative Pre-trained Transformer*) Modello di intelligenza artificiale basato su architetture a *transformer*, progettato per generare testo simile a quello umano. È *generativo* poiché crea nuovi contenuti (testo, codice e così via); è *pre-trained* perché apprende da grandi quantità di dati prima di essere raffinato; usa *transformer* per capire il contesto nelle sequenze (per esempio le frasi). ChatGPT è basato su modelli GPT.

Gradient descent Vedi Discesa del gradiente.

Gradient exploding Problema nell'addestramento delle reti neurali in cui i gradienti diventano estremamente grandi, causando un aggiornamento eccessivo dei pesi e una conseguente instabilità nel modello. Ciò può portare a divergenze o prestazioni scadenti.

Gradient vanishing Problema nell'addestramento delle reti neurali profonde in cui i gradienti diventano estremamente piccoli durante la backpropagation, impedendo l'aggiornamento efficace dei pesi nei layer iniziali e rallentando (o bloccando) l'apprendimento. È tipico nelle reti con funzioni di attivazione come *sigmoid* o *tanh* e in architetture molto profonde. Soluzioni comuni includono l'uso di *ReLU*, *residual connections* (ResNet) o normalizzazione dei layer (*BatchNorm*).

GRU (*Gated Recurrent Unit*) Tipo di rete neurale ricorrente (*RNN*) che utilizza meccanismi a *gate* per controllare il flusso delle informazioni, simile alla LSTM ma con una struttura semplificata (ha solo due gate: *reset* e *update*).

Hadoop Framework open-source per l'archiviazione e l'elaborazione distribuita di grandi volumi di dati su cluster di computer, basato sul modello *MapReduce* e sul file system *HDFS* (Hadoop Distributed File System). È progettato per essere scalabile orizzontalmente, per gestire *big data* in modo efficiente e per tollerare guasti hardware.

HFT (*Hugging Face Transformers*) Libreria open-source di Python che fornisce modelli di machine learning all'avanguardia, in particolare per l'elaborazione del linguaggio naturale, basati sull'architettura transformer. Include modelli preaddestrati come BERT, GPT e RoBERTa, facilitandone il download, l'addestramento e l'utilizzo.

IA (intelligenza artificiale) Branca della scienza dei computer che sviluppa sistemi in grado di svolgere compiti tipicamente umani, come l'apprendimento, il ragionamento e la decisione, attraverso algoritmi e dati.

Iperparametro Parametro di un modello di machine learning che viene impostato *prima* dell'addestramento e non viene appreso dai dati. Alcuni esempi: il tasso di apprendimento (*learning rate*), il numero di layer o neuroni in una rete neurale, il coefficiente di regolarizzazione (*lambda*). Gli iperparametri controllano il comportamento dell'algoritmo e influenzano le prestazioni del modello.

Julia Linguaggio di programmazione *open-source*, ad alte prestazioni, progettato per il calcolo scientifico e tecnico. Combina la velocità dei linguaggi compilati (come il C) con la facilità d'uso di linguaggi dinamici (come Python). Le sue caratteristiche principali sono la velocità (grazie alla compilazione JIT - *Just-in-Time)*, la sintassi intuitiva (simile a MATLAB/R/Python), il parallelismo nativo (con supporto per calcoli distribuiti e multithreading) e l'interoperabilità (potendo richiamare direttamente codice C, Python e di altri linguaggi). È ideale per analisi dati, machine learning, simulazioni numeriche e calcoli ad alta intensità computazionale.

Jupyter Notebook Applicazione web open-source che permette di creare e condividere documenti interattivi contenenti codice, testo, visualizzazioni e altri elementi, supportando diversi linguaggi di programmazione come Python, R e Julia. Ideale per l'analisi dati, il machine learning e la ricerca scientifica.

Keras API open source per lo sviluppo e l'addestramento di modelli di deep learning. Scritta in Python, è progettata per essere modulare, user-friendly e integrata con backend come TensorFlow, Theano o CNTK. In breve, è una libreria high-level per la creazione

di reti neurali, opera un’astrazione su framework più complessi (come TensorFlow) ed è ideale per scopi di prototipazione veloce e ricerca.

K-Means Algoritmo di *clustering* senza supervisione che partiziona un dataset in *K* gruppi (cluster) basandosi sulla similarità dei dati, minimizzando la varianza interna ai cluster. In breve, sceglie K centroidi iniziali (casuali o fissati); assegna ogni punto al cluster appartenente al centroide più vicino; ricalcola i centroidi come medie dei punti assegnati ai rispettivi cluster; ripete fino a raggiungere una convergenza (nessun cambiamento significativo).

KNN (*K-Nearest Neighbors*) Algoritmo di machine learning con supervisione utilizzato per scopi di classificazione e regressione. Funziona così: memorizza tutti i dati di addestramento; per una nuova osservazione, trova i *K esempi più vicini* nello spazio delle caratteristiche; per scopi di *classificazione*, assegna al nuovo punto la classe più frequente tra i K vicini; per scopi di *regressione*, calcola la media (o la mediana) dei valori vicini. È semplice ma efficace per dataset piccoli/non complessi, ma è sensibile alla scelta di *K* e al tipo di *distanza* usata e non impara un modello a priori; quindi, può essere lento in presenza di molti dati.

Learning rate Parametro chiave negli algoritmi di ottimizzazione (come la discesa del gradiente) che determina l’ampiezza dei passi compiuti durante l’aggiornamento dei pesi di un modello per minimizzare la funzione loss. Un valore alto offre una convergenza più veloce, ma rischia di saltare il vero minimo; un valore basso offre convergenza più lenta, ma più precisa. È un iperparametro critico per l’addestramento di reti neurali e modelli di machine learning.

LLaMA Famiglia di modelli di linguaggio avanzati sviluppati da Meta (Facebook). Si tratta di modelli open-source basati sull’architettura transformer, progettati per generare testo, rispondere

a domande e svolgere altre attività di elaborazione del linguaggio naturale (NLP).

LLM (*Large Language Model*) Modello di intelligenza artificiale addestrato su enormi quantità di dati testuali, in grado di generare, comprendere ed elaborare il linguaggio naturale in modo coerente e contestuale. Esempi famosi includono GPT-4, Gemini di Google e Llama di Meta.

Logica fuzzy Estensione della logica classica che permette di rappresentare e ragionare su concetti vaghi o incerti, utilizzando gradi di verità compresi tra 0 (falso) e 1 (vero), invece dei tradizionali valori binari (vero/falso). È utile in contesti nei quali le categorie non sono ben definite (“caldo”, “alto”, “vecchio”).

Loss function Misura che quantifica quanto le previsioni di un modello si discostano dai valori reali. L’obiettivo durante l’addestramento di un modello è minimizzare questa funzione per migliorare l’accuratezza delle previsioni.

LSTM (*Long Short-Term Memory*) tipo di rete neurale ricorrente (RNN) progettata per catturare dipendenze a lungo termine nei dati sequenziali, evitando il problema del gradient vanishing tipico delle RNN tradizionali. Sono un componente chiave nei modelli avanzati come i transformer e le reti encoder-decoder.

Machine learning Ramo dell’intelligenza artificiale che si basa sull’uso di algoritmi e modelli statistici per permettere ai computer di “apprendere” dai dati e migliorare le proprie prestazioni in un compito specifico, senza essere esplicitamente programmati.

Matrice di confusione Vedi *Confusion matrix*.

Metriche (Precision/Recall/Accuractezza/F1-score) Metriche per la valutazione di modelli di classificazione. Vedi le singole voci.

MLflow Piattaforma *open-source* per gestire il ciclo di vita del *machine learning*, incluse fasi come la sperimentazione, la

riproducibilità, il deployment e il monitoraggio dei modelli.

Modelli ensemble Vedi *Ensemble metodi/modelli*.

Modello predittivo Strumento matematico o statistico che utilizza dati esistenti per prevedere risultati o comportamenti futuri. Viene spesso impiegato nel machine learning, nell'analisi dati e nella business intelligence per anticipare tendenze, rischi o opportunità.

Naive Bayes Algoritmo di machine learning basato sul teorema di Bayes, che presume un'indipendenza tra le caratteristiche (da cui "naive", ingenuo). Utilizzato principalmente per problemi di classificazione, come il filtraggio dello spam o l'analisi del sentiment. È semplice e veloce e funziona bene con dataset di piccole dimensioni, ma a volte il presupposto di indipendenza tra le caratteristiche non è realistica.

NLP (*Natural Language Processing*) Campo dell'intelligenza artificiale di elaborazione del linguaggio naturale, che si occupa dell'interazione tra computer e linguaggio umano, permettendo alle macchine di comprendere, elaborare e generare testo o parlato.

NumPy Libreria Python per il calcolo numerico che fornisce strutture dati efficienti (come gli array multidimensionali) e funzioni per operazioni matematiche avanzate, ottimizzate per le prestazioni. È fondamentale per la data science, il machine learning e l'analisi scientifica in Python.

ONNX (*Open Neural Network Exchange*) Formato open standard per rappresentare modelli di machine learning, progettato per favorire l'interoperabilità tra framework (come PyTorch, TensorFlow). Consente di esportare, importare e utilizzare modelli in vari ambienti senza dipendenze specifiche.

Outlier Dato o osservazione che si discosta significativamente dagli altri dati del dataset, spesso a causa di errori, anomalie o variabilità intrinseca. Gli outlier possono influenzare negativamente

l’addestramento dei modelli, ma possono anche essere oggetto di interesse (per esempio nel rilevamento di frodi o guasti).

Overfitting È un “*sovradattamento*” che si verifica quando un modello di machine learning impara “troppo bene” i dati di addestramento, includendo anche il rumore e i dettagli irrilevanti, perdendo la capacità di generalizzare su nuovi dati.

Pandas Libreria open-source per Python specializzata nella manipolazione e nell’analisi di dati strutturati (tabelle), offrendo strutture efficienti come DataFrame e strumenti per la pulizia, la trasformazione e l’esplorazione dei dati. È ampiamente usata per preelaborare i dataset prima di applicare gli algoritmi di machine learning.

PCA (*Principal Component Analysis*) Tecnica di riduzione della dimensionalità che trasforma dati complessi in un insieme più piccolo di componenti principali, mantenendo la maggior parte della varianza originale. Si usa per semplificare i modelli, per visualizzare i dati e per migliorare l’efficienza computazionale.

Perceptron Algoritmo di apprendimento con supervisione per classificare i dati, rappresentando il modello più semplice di rete neurale artificiale. È composto da un input (i dati in ingresso), dei pesi (parametri regolabili), una funzione di attivazione e un output (il risultato della classificazione, spesso 0 o 1). Fu introdotto da Frank Rosenblatt nel 1957 e costituisce la base per reti neurali più complesse.

Pianificazione automatica Processo mediante il quale un sistema intelligente genera autonomamente una sequenza di azioni per raggiungere un determinato obiettivo, partendo da uno stato iniziale e considerando vincoli e condizioni ambientali. Si basa su algoritmi che esplorano i possibili percorsi di azione, ottimizzando l’efficienza, il tempo o le risorse.

Pipeline di dati Processo strutturato che raccoglie, elabora, trasforma e distribuisce i dati attraverso diverse fasi per addestrare, valutare o utilizzare modelli di intelligenza artificiale.

Precision Metrica che valuta l'accuratezza dei valori predetti come positivi da un modello. Si calcola come:

Precision = Veri Positivi / (Veri Positivi + Falsi Positivi) In breve, indica quanti degli elementi classificati come positivi sono effettivamente corretti. Per esempio, in un sistema di riconoscimento dello spam, una precisione alta significa che la maggior parte delle e-mail segnalate come spam sono effettivamente di spam. Spesso si usa insieme al recall (vedi) per valutare le prestazioni del modello in modo bilanciato.

Pre-training Fase iniziale in cui un modello (come un LLM, vedi) viene addestrato su un vasto dataset generico per apprendere pattern linguistici o caratteristiche generali, prima di essere affinato (*fine-tuning*) per compiti specifici. Per esempio, ChatGPT viene pre-addestrato su testi tratti da internet prima di essere ottimizzato per i dialoghi.

Prompt engineering Arte e scienza di progettare input (prompt) efficaci per guidare i modelli di intelligenza artificiale generativa (come ChatGPT) a produrre risposte accurate e pertinenti. Punta a ottimizzare la formulazione delle richieste per ottenere il miglior output possibile.

Python Linguaggio di programmazione molto popolare e versatile, ampiamente utilizzato per lo sviluppo di modelli di machine learning, deep learning e data science grazie alla sua semplicità, alla vasta gamma di librerie specializzate (come TensorFlow, PyTorch e scikit-learn) e alla sua community. È lo strumento preferito da molti ricercatori e sviluppatori per scopi di prototipazione rapida e implementazione di algoritmi di intelligenza artificiale.

PyTorch Framework open-source per il machine learning e l'intelligenza artificiale, basato su Python, che offre strumenti flessibili per la creazione e l'addestramento di reti neurali, con un'attenzione particolare al calcolo tensoriale e alla differenziazione automatica (*autograd*). È noto per la sua facilità d'uso e l'integrazione con l'ecosistema Python.

R Linguaggio open source specializzato in analisi statistica, machine learning e rappresentazione dei dati, molto usato nella ricerca e data science.

Random forest Questa "foresta casuale" è un algoritmo di machine learning che combina più alberi decisionali per migliorare l'accuratezza e ridurre l'overfitting. Funzionamento: crea molti alberi decisionali su sottoinsiemi casuali dei dati (bagging); ogni albero dà una previsione; il risultato finale è la media (regressione) o la maggioranza (classificazione) delle previsioni. I suoi vantaggi sono la robustezza e la capacità di gestire dati complessi e caratteristiche non lineari.

Recall Metrica di "sensibilità" che misura la capacità di un modello di identificare correttamente tutti i casi positivi rilevanti in un dataset. Si calcola come:

$$\text{Recall} = TP / (TP + FN)$$

Qui:

- TP (True Positives) sono i casi positivi correttamente identificati;
- FN (False Negatives) sono i casi positivi erroneamente classificati come negativi.

Un recall alto indica che il modello "cattura" la maggior parte degli elementi rilevanti (utile in contesti critici come le diagnosi mediche). Spesso è in tensione con la precision, che invece misura l'accuratezza delle predizioni positive.

Reinforcement learning Paradigma del machine learning in cui un agente impara a compiere azioni ottimali interagendo con un ambiente, ricevendo ricompense (o penalità) in base alle sue scelte. L'obiettivo è quello di massimizzare la ricompensa cumulativa nel tempo, attraverso un processo di apprendimento per prove ed errori.

RNN (*Recurrent Neural Network*) Una "rete neurale ricorrente" è un tipo di rete neurale progettata per elaborare sequenze di dati, dove le connessioni tra nodi formano cicli per mantenere una "memoria" delle informazioni precedenti. È utile per compiti di elaborazione del linguaggio naturale, riconoscimento vocale e previsioni temporali. Ha la capacità di gestire input/output di lunghezza variabile grazie alla sua struttura ricorrente.

ROC curve (*Receiver Operating Characteristic curve*) Grafico che mostra le prestazioni di un modello di classificazione nell'intelligenza artificiale, confrontando il tasso di veri positivi (TPR) e il tasso di falsi positivi (FPR) al variare della soglia di decisione.

Scikit-learn Libreria open source per il machine learning in Python, che offre strumenti semplici ed efficienti per l'analisi dei dati, la costruzione di modelli predittivi e l'implementazione di algoritmi di apprendimento con e senza supervisione. È basata su NumPy e SciPy, ed è ampiamente utilizzata per compiti di classificazione, regressione, clustering e pre-elaborazione dei dati.

Sistema esperto Programma che utilizza conoscenze specializzate per risolvere problemi o fornire consulenze in un dominio specifico, simulando il ragionamento di un esperto umano.

Spark Vedi Apache Spark.

Statistiche descrittive Insieme di tecniche per riassumere e visualizzare dati, utilizzate nell'intelligenza artificiale per analizzare pattern, distribuzioni e caratteristiche fondamentali dei dataset prima

dell’addestramento dei modelli. Include misure come media, deviazione standard, percentili e visualizzazioni (grafici, istogrammi).

Supervisione Tipo di apprendimento in cui un modello viene addestrato su un dataset etichettato, cioè con input correlati a output corretti già noti. L’obiettivo è quello di apprendere una funzione che mappi gli input agli output in modo generalizzabile a nuovi dati. Un esempio è la classificazione di immagini (foto etichettate come di “gatto” o di “cane”).

SVM (*Support Vector Machine*) Algoritmo di apprendimento con supervisione utilizzato per compiti di classificazione e regressione. Le SVM trovano il confine decisionale ottimale (iperpiano) che massimizza il margine tra diverse classi di dati. È particolarmente efficace in spazi ad alta dimensionalità e con dataset non troppo grandi.

T5 (*Text-To-Text Transfer Transformer*) Modello di linguaggio sviluppato da Google che tratta ogni compito di elaborazione del linguaggio naturale come un problema di conversione di “testo in testo”.

TensorFlow Framework open source sviluppato da Google per il machine learning e l’intelligenza artificiale, specializzato nella creazione, nell’addestramento e nell’impiego di reti neurali attraverso calcoli basati su tensori (strutture dati multidimensionali). Supporta CPU, GPU e TPU, ed è utilizzato per compiti come la visione artificiale, l’elaborazione del linguaggio naturale e i modelli generativi.

Tokenizzazione Processo che consiste nel suddividere un testo in unità più piccole chiamate token (parole, parti di parole o singoli caratteri), per facilitarne l’elaborazione da parte di modelli linguistici.

Transfer learning Tecnica di machine learning in cui un modello pre-addestrato su un compito viene riutilizzato e adattato per un compito diverso ma correlato, riducendo il tempo e i dati necessari per l'addestramento da zero. Per esempio, si può usare un modello addestrato sul riconoscimento di oggetti generici (come automobili o animali) per specializzarsi nella classificazione di modelli specifici di auto.

Transformer Architettura di rete neurale basata su meccanismi di attenzione (*self-attention*) che elabora dati in parallelo, catturando relazioni a lungo raggio in sequenze (come testo o immagini). Introdotto da Google nel 2017, è alla base di modelli come GPT e BERT.

Underfitting Si verifica quando un modello di machine learning è troppo semplice e non riesce a catturare adeguatamente le relazioni tra i dati, ottenendo prestazioni scarse sia sui dati di addestramento sia su quelli nuovi.

Word embedding Rappresentazioni numeriche dense e vettoriali delle parole, utilizzate nell'intelligenza artificiale per catturare il significato, le relazioni semantiche e il contesto linguistico. In pratica, si trasformano le parole in vettori di numeri reali, in modo che parole con significati simili abbiano rappresentazioni vettoriali vicine nello spazio multidimensionale.

XAI (*Explainable AI*) Insieme di tecniche e metodi che rendono comprensibile il funzionamento dei modelli di intelligenza artificiale, soprattutto quelli complessi come le reti neurali, permettendo agli utenti di capire come vengono prese le decisioni. L'obiettivo è quello di aumentare la trasparenza, la fiducia e l'auditability dei sistemi di intelligenza artificiale, specialmente in contesti critici come la medicina, la finanza o la giustizia.

Indice

Premessa

Ringraziamenti

Introduzione

- A chi si rivolge questo libro
- Struttura del libro: una tabella di marcia
- Il codice
- Altre risorse online
- L’autore

Capitolo 1 - I modelli LLM

- Che cosa è un modello LLM?
- Applicazioni dei modelli LLM
- Fasi di costruzione e utilizzo dei modelli LLM
- Introduzione all’architettura a transformer
- Utilizzo di grandi dataset
- L’architettura GPT
- Costruzione di un modello LLM
- Riepilogo

Capitolo 2 - Operare su dati testuali

- Embedding di parole
- Tokenizzazione del testo

Conversione dei token in token-ID
Aggiunta di token contestuali speciali
BPE: codifica a coppie di byte
Campionamento dei dati a finestra scorrevole
Creazione degli embedding dei token
Codifica della posizione delle parole
Riepilogo

Capitolo 3 - Programmazione dei meccanismi di attenzione

Il problema della modellazione di sequenze lunghe
Cattura delle dipendenze nei dati con i meccanismi di attenzione
Attenzione alle diverse parti dell'input con l'auto-attenzione
Implementazione dell'auto-attenzione con pesi addestrabili
Nascondere le parole future con l'attenzione causale
Estensione dell'attenzione single-head all'attenzione multi-head
Riepilogo

Capitolo 4 - Implementazione di un modello GPT da zero per generare testo

Programmazione di un'architettura LLM
Normalizzazione delle attivazioni con la normalizzazione dei layer
Implementazione di una rete feed forward con attivazioni GELU
Aggiunta delle connessioni scorciatoia
Collegamento dei layer di attenzione e lineari in un blocco transformer

Programmazione del modello GPT
Generazione del testo
Riepilogo

Capitolo 5 - Pre-addestramento su dati non etichettati

Valutazione dei modelli generativi per il testo
Addestramento di un modello LLM
Strategie di decodifica per controllare la casualità
Caricamento e salvataggio dei pesi del modello in PyTorch
Caricamento di pesi pre-addestrati da OpenAI
Riepilogo

Capitolo 6 - Fine-tuning per attività di classificazione

Diverse categorie di fine-tuning
Preparazione del dataset
Creazione dei data loader
Inizializzazione di un modello con pesi pre-addestrati
Aggiunta di una “testa” per la classificazione
Calcolo del valore loss e dell’accuratezza della classificazione
Fine-tuning del modello su dati supervisionati
Utilizzo del modello LLM come classificatore dello spam
Riepilogo

Capitolo 7 - Fine-tuning per rispondere a istruzioni

Introduzione al fine-tuning per rispondere a istruzioni
Preparazione di un dataset per il fine-tuning
Organizzazione dei dati in batch di addestramento
Creazione di data loader per un dataset delle istruzioni
Caricamento di un modello LLM pre-addestrato

Fine-tuning del modello LLM sui dati delle istruzioni
Estrazione e salvataggio delle risposte
Valutazione del modello LLM dopo il fine-tuning
Conclusioni
Riepilogo

Appendice A - Introduzione a PyTorch

Che cos’è PyTorch?
Che cosa sono i tensori
I modelli come grafi di calcoli
L’ABC della differenziazione automatica
Implementazione delle reti neurali multi-layer
Impostazione di data loader efficienti
Un tipico ciclo di addestramento
Salvataggio e caricamento dei modelli
Ottimizzazione delle prestazioni di addestramento con le GPU
Riepilogo

Appendice B - Riferimenti e approfondimenti

Capitolo 1
Capitolo 2
Capitolo 3
Capitolo 4
Capitolo 5
Capitolo 6
Capitolo 7
Appendice A

Appendice C - Soluzioni degli esercizi

Capitolo 2
Capitolo 3
Capitolo 4
Capitolo 5
Capitolo 6
Capitolo 7
Appendice A

Appendice D - Nuovi dettagli per il ciclo di addestramento

Learning Rate Warmup
Cosine Decay
Gradient Clipping
La funzione di addestramento modificata

Appendice E - Fine-tuning dei parametri con LoRA

Introduzione a LoRA
Preparazione del dataset
Inizializzazione del modello
Fine-tuning dei parametri con LoRA

Glossario