this:

POLITECNICO DI BARI

DEPARTMENT OF ELECTRICAL AND INFORMATION ENGINEERING
Master Degree in Computer Science

Deep Learning

NanoSocrates: a Very Small Language Model

Students

COLAPIETRA Antonio Pio

TAVILLA Simone

Academic Year 2024 - 2025

Contents

Introduzione

Gli ultimi anni hanno assistito a un cambiamento di paradigma nell'Elaborazione del Linguaggio Naturale (NLP), guidato dall'avvento di modelli linguistici pre-addestrati su larga scala, spesso definiti Modelli Fondazionali . Architetture come il Transformer hanno permesso la creazione di modelli come BERT, T5 e GPT, che apprendono ricche rappresentazioni del linguaggio da vasti dati di testo non etichettato. Questi modelli possono essere successivamente adattati a un'ampia gamma di task specifici (*downstream task*) con notevole successo.

Tuttavia una sfida significativa nell'intelligenza artificiale moderna risiede nel colmare il divario tra il mondo non strutturato del testo in linguaggio naturale e il mondo strutturato e simbolico dei grafi di conoscenza (*knowledge graph*). I grafi di conoscenza, come DBpedia, rappresentano informazioni fattuali come triple RDF (Resource Description Framework), formando una rete di entità e relazioni interconnesse. La capacità di tradurre senza soluzione di continuità tra queste due rappresentazioni è cruciale per compiti come l'estrazione di conoscenza, il *question answering* e la generazione di testo a partire da dati.

Obiettivo del Progetto

Questo progetto, denominato **NanoSocrates**, affronta questa sfida intraprendendo il compito di costruire da zero un "Very Small Semantic Language Model". L'obiettivo è sviluppare un singolo modello Transformer Encoder-Decoder unificato, specializzato nella traduzione bidirezionale tra testo non strutturato e dati RDF strutturati nel dominio dei film. Il modello deve essere competente in quattro task distinti ma correlati:

1. **Text-to-RDF (Text2RDF):** Convertire un testo in linguaggio naturale (ad es. la descrizione di un film) in un insieme di triple RDF che ne catturino il significato semantico.

2. **RDF-to-Text (RDF2Text):** Generare una frase coerente e leggibile da un insieme di triple RDF.

3. **Completamento RDF 1 (Masked Language Modeling):** Prevedere un componente mancante (soggetto, predicato o oggetto) all'interno di una tripla RDF, un compito analogo alla predizione di link.

4. **Completamento RDF 2 (Generazione RDF):** Prevedere triple successive basandosi su un contesto dato, un compito analogo al completamento di grafi di conoscenza.

Struttura del Report

Questo report descrive in dettaglio le fasi di sviluppo seguite dal gruppo di studio, soffermandosi sui vari punti critici affrontati durante l'implementazione. L'approccio di stesura del report si rifà all'architettura di un Trasformer standard, infatti l'ordine con cui verranno trattate le fasi di sviluppo segue il flusso dei dati introdotto dal paper "Attention is all you need" [8]. Tuttavia ci sono due capitoli principali: il primo dedicato all'implementazione di un Trasformer con architettura standard, ovvero quella ripresa dal paper [8], e il secondo dedicato all'implementazione di un Trasformer con architettura T5.

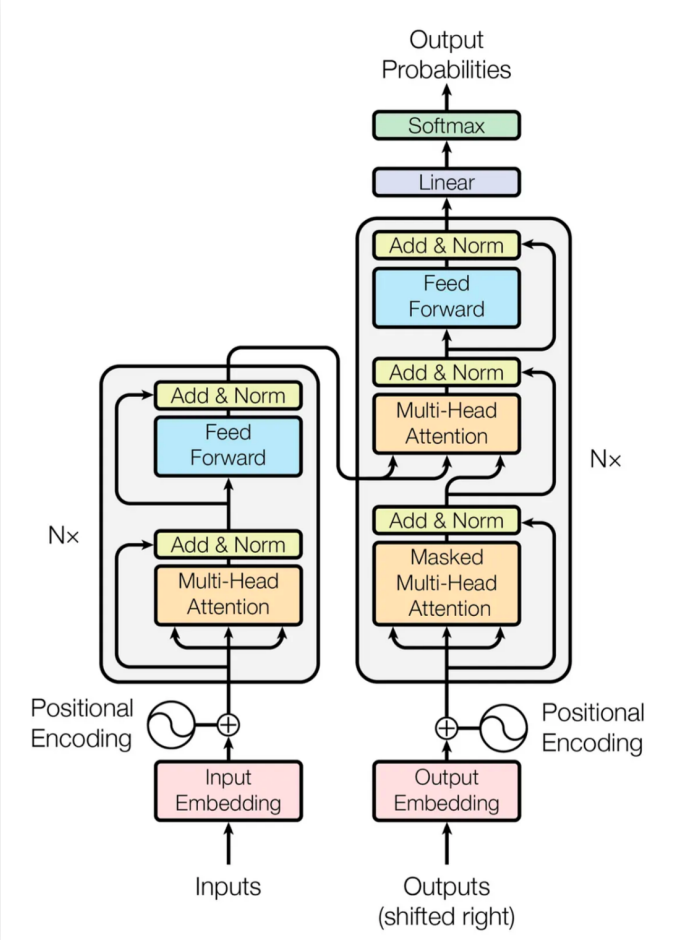


Figure 1: The trasformer architecture - Attention is all you need

Chapter 1

Creazione dei dati e pre-processing

Il successo di un modello *sequence-to-sequence* come NanoSocrates è intrinsecamente legato alla qualità, alla coerenza e alla struttura dei dati su cui viene addestrato. Data la natura eterogenea delle fonti—il testo in linguaggio naturale degli abstract di Wikipedia e i dati strutturati in formato RDF di DBpedia—si è resa necessaria la progettazione di una robusta pipeline di pre-processing. Questo capitolo illustra in dettaglio il percorso metodologico seguito per trasformare i dati grezzi in un corpus unificato e ottimizzato, suddiviso in dataset specifici per le fasi di pre-training e fine-tuning dell'architettura Transformer.

1.1 Estrazione e strutturazione del dataset

Il dataset utilizzato per questo progetto è il risultato di una pipeline articolata che parte dall'estrazione dei dati grezzi e culmina nella creazione di un corpus finemente strutturato per i task di addestramento. Ogni fase è stata progettata per massimizzare la qualità e la pertinenza delle informazioni fornite al modello.

1.1.1 Raccolta dati da DBpedia

Il processo di acquisizione ha avuto inizio con l'interrogazione dell'endpoint SPARQL pubblico di DBpedia. Una prima strategia, basata su un approccio iterativo che prevedeva una query distinta per ciascuna entità film, si è rivelata computazionalmente insostenibile su larga scala. La necessità di inserire intervalli di attesa tra le richieste per non sovraccaricare il server introduceva una latenza inaccettabile, rendendo il metodo inadeguato per la costruzione di un dataset esteso. Per superare questa limitazione, la strategia è stata radicalmente ottimizzata. L'approccio definitivo, implementato nello script `data_creation_2.py`, si basa su un'unica query parametrica che adotta un sistema di paginazione. Questo metodo recupera i dati in blocchi di dimensione predefinita (utilizzando le clausole `LIMIT` e `OFFSET` all'interno di una sub-query per garantire stabilità), riducendo drasticamente il numero di richieste HTTP e, di conseguenza, i tempi di esecuzione. Tale ottimizzazione ha permesso di gestire in modo robusto il carico sull'endpoint, assicurando un processo di estrazione efficiente e scalabile. La qualità e la pertinenza dei dati sono state ulteriormente garantite tramite filtri specifici integrati nella query SPARQL. È stata applicata una strategia di **whitelisting** per includere unicamente un insieme predefinito di predicati informativi (es. `dbo:director`, `dbo:starring`), focalizzando il modello sulle relazioni semantiche più salienti del dominio cinematografico.

Contestualmente, un filtro linguistico ha limitato i valori letterali alla sola lingua inglese o a quelli privi di tag di lingua, assicurando l'omogeneità tra gli abstract testuali e i dati strutturati. L'output di questo script è un singolo file JSON contenente una lista di oggetti, ciascuno rappresentante un film con il suo titolo, l'URI del soggetto (già abbreviato con prefissi come `dbr:`), l'abstract e un elenco ordinato delle relative triple RDF.

1.1.2 Creazione del dataset per il fine-tuning multi-task

L'architettura Transformer opera su sequenze lineari di token. Di conseguenza, il dataset JSON strutturato ha richiesto una trasformazione in un formato text-to-text unificato, idoneo a un paradigma di apprendimento *sequence-to-sequence*. Questo processo, implementato in `data_preprocessing_2.py`, ha incluso una serie di strategie mirate a creare un corpus di addestramento bilanciato e semanticamente coerente per i quattro task fondamentali del progetto. Il primo passo è stato un filtraggio rigoroso per garantire la qualità di ogni campione. Sono stati scartati i record privi di dati essenziali (abstract o triple), quelli con un abstract testuale troppo breve per fornire un contesto significativo ($<$250 caratteri) e quelli mancanti di predicati cardine come il regista o il cast. Questo filtraggio preventivo è cruciale per massimizzare la densità informativa di ogni esempio di addestramento. Successivamente, le triple RDF sono state linearizzate, ovvero convertite in una stringa di testo secondo un formato deterministico. Ciascuna tripla è stata delimitata da token speciali come `<SOT>` (Start of Triple) e `<EOT>` (End of Triple), includendo marcatori strutturali come `<SUBJ>`, `<PRED>` e `<OBJ>`. L'inclusione di questi espliciti marcatori sintattici fornisce al modello ancore strutturali inequivocabili, trasformando il complesso compito di predire un grafo in un problema più gestibile di generazione di un linguaggio formale. A partire da queste sequenze, sono state formulate le coppie di addestramento per i quattro task, ciascuna introdotta da un token di controllo (es. `<Text2RDF>`) per istruire il modello sul compito da eseguire:

- Per il task **Text2RDF**, è stata implementata una strategia di allineamento semantico. Una funzione di controllo euristico ha verificato che l'oggetto di una tripla fosse plausibilmente menzionato nell'abstract. Solo le triple che superavano questo controllo venivano incluse nell'output atteso, formulando così un compito di estrazione di informazioni equo e risolvibile, basato unicamente sul contesto fornito.
- Per gli altri task—**RDF2Text**, **CONTINUERDF**, **MLM**—gli esempi sono stati costruiti seguendo la loro logica specifica: la traduzione da RDF a testo, il completamento di sequenze di triple e la predizione di componenti mascherati.

Infine, poiché il processo genera un numero disomogeneo di esempi, è stata applicata una strategia di bilanciamento basata su percentuali fisse. Il numero di esempi per ogni task è stato limitato per aderire a una distribuzione predefinita (es. 25% Text2RDF, 25% RDF2Text, etc.), garantendo che il corpus finale contenesse una proporzione controllata di ciascun tipo di compito. Questa operazione previene un bias del modello verso i task più numerosi e promuove un apprendimento più stabile.

1.1.3 Creazione del corpus per il pre-training

La strategia di addestramento di NanoSocrates prevede una fase preliminare di pre-training auto-supervisionato, il cui scopo è permettere al modello di apprendere rappresentazioni linguistiche e strutturali del dominio prima della specializzazione. Per questa fase,

è stato costruito un corpus dedicato attraverso lo script `create_pretrain_corpus.py`. Il processo parte dal dataset JSON e astrae le due modalità di dati: l'abstract in linguaggio naturale e l'insieme delle triple RDF. Le triple vengono serializzate in un formato testuale lineare, omettendo deliberatamente qualsiasi token di controllo specifico dei task. L'obiettivo è creare un corpus di testo "grezzo", agnostico rispetto ai compiti di fine-tuning. Successivamente, viene applicato un rigoroso processo di filtraggio e pulizia. Sia gli abstract che le sequenze RDF vengono filtrati in base alla loro lunghezza (scartando campioni troppo corti o troppo lunghi) e successivamente de-duplicati per massimizzare la varietà informativa. La fase più critica è il bilanciamento del corpus. Poiché il numero di abstract e di sequenze RDF può essere sbilanciato, è stata implementata una strategia per assicurare una rappresentazione equa di entrambe le modalità. Il corpus finale viene costruito campionando casualmente da entrambe le collezioni di dati, in modo da ottenere una distribuzione paritetica (circa 50% linguaggio naturale, 50% formato RDF). L'obiettivo è esporre il modello in egual misura sia alla prosa discorsiva degli abstract sia alla sintassi formale degli RDF. Il corpus risultante viene infine mescolato e salvato come un singolo file di testo, pronto per la fase di corruzione.

1.2 Tokenizzazione e generazione del dataset di pre-training

Una volta generato il corpus, è stato necessario convertirlo in un formato numerico processabile dal Transformer. La tokenizzazione è di importanza critica, poiché la qualità del vocabolario e della segmentazione del testo influisce direttamente sulla capacità di apprendimento del modello.

1.2.1 Progettazione di un tokenizer personalizzato

La natura ibrida del corpus, che unisce linguaggio naturale e sintassi RDF, rende controproducente l'uso di tokenizer pre-addestrati. Questi ultimi, non essendo stati esposti a prefissi come `dbr:` o `dbo:`, tenderebbero a segmentare le entità in sotto-unità semanticamente arbitrarie, generando sequenze di input inefficienti e rumorose. Per ovviare a ciò, è stato addestrato un tokenizer custom da zero sull'intero corpus di progetto. Questa strategia garantisce che il vocabolario sia ottimizzato per il dominio specifico, catturando in modo efficiente sia i pattern del linguaggio naturale che le componenti strutturali delle triple.

1.2.2 Implementazione con Byte-Pair Encoding (BPE)

Per l'implementazione del tokenizer, descritta in `tokenizer_pretrain.py`, è stato scelto l'algoritmo *sub-word* Byte-Pair Encoding (BPE). Per gestire qualsiasi carattere in input, è stata adottata la tecnica del **Byte-level BPE**. Questo approccio tratta il testo come una sequenza di byte, garantendo che qualsiasi stringa possa essere rappresentata senza mai ricorrere al token sconosciuto (`<UNK>`). Un aspetto cruciale è stata la configurazione del **pre-tokenizer**. Questo componente è stato istruito per trattare come unità atomiche e indivisibili non solo l'intero set di token speciali (es. `<SOT>`, `<Text2RDF>`), ma anche i prefissi RDF (`dbr:`, `dbo:`) e i caratteri strutturali (`:`, `_`). Tale configurazione impedisce all'algoritmo BPE di frammentare questi elementi, preservandone l'integrità semantica e

funzionale. Il processo di addestramento è stato eseguito sul corpus puro di pre-training, con una dimensione del vocabolario target di 32.000 token. Successivamente, il vocabolario è stato arricchito con l'aggiunta esplicita di tutti i token speciali, inclusi i 150 token sentinella `<extra_id_...>` richiesti per lo span corruption.

1.2.3 Generazione del dataset per il pre-training

La fase di pre-training si basa su un obiettivo di tipo *denoising*, specificamente la tecnica dello span corruption introdotta dal modello T5. Lo script `pretrain_dateset_T5.py` trasforma il corpus di testo puro in un dataset di coppie input-output. Il processo itera su ogni riga del corpus e applica una funzione di corruzione. Questa funzione seleziona casualmente porzioni contigue di testo (*span*) da mascherare. Una scelta implementativa cruciale è stata l'adozione di una strategia di "whole word masking". Invece di operare a livello di token, la corruzione viene applicata a parole intere, separate da spazi. Questo approccio previene la frammentazione di unità semantiche atomiche come le entità RDF (`dbr:The_Matrix`) o parole complesse, garantendo che il compito di ricostruzione sia semanticamente coerente. Ciascuno *span* mascherato nell'input viene sostituito da un singolo token sentinella univoco (es. `<extra_id_0>`). Contemporaneamente, viene costruita la sequenza target, che concatena ogni token sentinella con il testo originale dello *span* che esso ha sostituito. Il risultato è la generazione di due file paralleli, `train.source` e `train.target`. In questo modo, il modello viene addestrato a un compito *sequence-to-sequence* in cui deve "riempire" le parti mancanti dell'input, sviluppando una comprensione profonda della sintassi e della semantica del dominio.

Chapter 2

Architettura del modello

2.1 Il paradigma Text-to-Text e il modello T5

Una volta definito e preparato un corpus di addestramento unificato, il passo successivo e centrale del progetto consiste nella progettazione e implementazione dell'architettura neurale capace di apprendere la complessa mappatura tra linguaggio naturale e conoscenza strutturata. A tal fine, è stato adottato il paradigma Encoder-Decoder basato sull'architettura Transformer, traendo ispirazione diretta dalle ottimizzazioni e dalla filosofia del modello T5. L'idea fondamentale di T5, introdotta nel celebre paper "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer" [6], è quella di unificare tutti i task di Natural Language Processing (NLP) in un unico formato condiviso: il **text-to-text**. In questo paradigma, ogni problema viene riformulato come un task in cui il modello riceve un input testuale e produce un output testuale. Questa unificazione offre un vantaggio metodologico cruciale, poiché permette di utilizzare la stessa architettura, la stessa funzione di perdita e la stessa procedura di decodifica per una vasta gamma di problemi. Per istruire il modello sul compito specifico da eseguire, viene aggiunto un token direttiva all'inizio della sequenza di input. Nel contesto di questo progetto, i token speciali come `<Text2RDF>`, `<RDF2Text>`, `<CONTINUERDF>` e `<MLM>` svolgono esattamente questa funzione di prefisso contestuale, guidando il comportamento del modello a seconda del task richiesto. L'approccio del progetto si ispira quindi direttamente al modello Text-to-Text Transfer Transformer (T5), adattandone i principi al dominio specifico della conoscenza semantica strutturata.

2.1.1 Architettura Encoder-Decoder e ottimizzazioni di T5

Coerentemente con il modello T5, NanoSocrates adotta un'architettura Encoder-Decoder basata sul Transformer originale. Questa scelta è motivata non solo dalla sua efficacia dimostrata su un'ampia gamma di task, specialmente quelli generativi che sono centrali nel nostro progetto, ma anche dai risultati dello studio sistematico condotto da Raffel et al. [6], che ha identificato l'architettura encoder-decoder come la più performante e versatile all'interno del loro framework unificato. L'architettura è composta da due componenti principali. Il primo è l'Encoder, che ha il compito di processare l'intera sequenza di input attraverso un meccanismo di self-attention bidirezionale per costruire una rappresentazione latente ricca di contesto, un principio reso celebre da BERT [1]. Il secondo è il Decoder, che genera la sequenza di output in modo autoregressivo, un token alla volta, utilizzando una masked self-attention sui token già generati e una cross-attention sulla

rappresentazione finale prodotta dall'encoder. L'implementazione integra diverse delle ottimizzazioni architetturali chiave che distinguono il modello T5 dal Transformer originale, come si evince dal codice in `model_lib.py`. Una di queste ottimizzazioni è la configurazione **Pre-LayerNorm**, dove la normalizzazione dei layer è applicata all'input di ogni sotto-blocco (attention e feed-forward) e non all'output. Questa configurazione migliora la stabilità del gradiente e facilita il training di modelli profondi. Un'altra ottimizzazione cruciale è l'uso di un meccanismo di **bias posizionale relativo** al posto dei classici embedding posizionali assoluti. Con questa tecnica, un bias scalare addestrabile viene aggiunto ai logit di attenzione, dipendendo unicamente dalla distanza relativa tra i token. Come dimostrato in [6], questo approccio permette al modello di generalizzare meglio a lunghezze di sequenza non viste durante l'addestramento. Infine, è stata utilizzata una versione semplificata della Layer Normalization, come definito nella classe `T5LayerNorm`, che apprende solo un parametro di scala (gamma) e omette il parametro di shift (beta). Queste scelte mirano a costruire un modello che non solo aderisca al potente framework text-to-text, ma che ne incorpori anche le ottimizzazioni strutturali, al fine di creare un sistema robusto, stabile ed efficiente.

2.2 Implementazione dell'architettura custom ispirata a T5

L'implementazione dell'architettura è stata realizzata utilizzando la libreria PyTorch, con l'obiettivo di replicare fedelmente le ottimizzazioni chiave del modello T5. Il codice, interamente contenuto nel file `model_lib.py`, è stato strutturato in modo modulare, componendo blocchi funzionali specifici per ogni operazione, dall'elaborazione dell'input alla generazione dell'output. Di seguito, vengono descritte in dettaglio le componenti fondamentali dell'architettura, confrontando l'implementazione specifica con i principi descritti nel paper di T5.

2.2.1 Embedding e codifica posizionale

Il primo stadio del modello è responsabile della conversione della sequenza di input, fornita come una serie di indici numerici, in una rappresentazione vettoriale densa. Questa operazione è gestita dalla classe `InputEmbeddings`, che al suo interno utilizza un layer `nn.Embedding` standard. Coerentemente con le pratiche comuni nei modelli Transformer, la magnitudine degli embedding risultanti viene talvolta scalata moltiplicandola per la radice quadrata della dimensione del modello (d_{model}), una tecnica implementativa diffusa che aiuta a preservare la varianza dell'input all'interno della rete. Una differenza sostanziale rispetto al Transformer originale, ma in piena aderenza con il modello T5, risiede nella gestione dell'informazione posizionale. Invece di sommare un embedding posizionale assoluto all'input, il nostro modello implementa il meccanismo di bias posizionale relativo. Questa tecnica, incapsulata nella classe `T5RelativePositionBias`, non modifica gli embedding di input ma agisce direttamente all'interno del calcolo dell'attenzione. La sua funzione `forward` calcola una matrice di distanze relative tra ogni coppia di token query e key. Queste distanze vengono poi discretizzate in un numero finito di `bucket` attraverso la funzione statica `_relative_position_bucket`. Quest'ultima implementa la strategia descritta nel paper T5, utilizzando una mappatura lineare per le distanze brevi e una mappatura logaritmica per quelle maggiori, per gestire efficientemente un ampio

range di distanze senza richiedere un embedding per ogni possibile offset. L'indice del bucket risultante viene infine usato per recuperare un bias scalare addestrabile da una tabella di embedding dedicata, che viene aggiunto al punteggio di attenzione prima della funzione softmax, modulando l'attenzione in base alla posizione relativa. Nelle implementazioni di riferimento, il bias viene applicato alle operazioni di self-attention, mentre la sua applicazione alla cross-attention è opzionale e dipende dalle scelte implementative (il paper non specifica in modo esplicito questa distinzione).

2.2.2 Componenti fondamentali dei blocchi transformer

Ogni layer dell'encoder e del decoder è costruito assemblando tre componenti principali, definiti nel file `model_lib.py`, che replicano fedelmente le scelte architetturali di T5.

Layer di normalizzazione Ispirandosi al paper di T5, è stata implementata una versione semplificata della Layer Normalization, denominata `T5LayerNorm`. A differenza della `nn.LayerNorm` standard, questa variante omette il parametro di shift additivo ('beta'), apprendendo unicamente un parametro di scala moltiplicativo ('gamma'). Questa scelta non solo riduce il numero di parametri ma si allinea perfettamente all'architettura di riferimento.

Feed-Forward network Ogni blocco Transformer contiene un sotto-layer feed-forward, implementato nella classe `FeedForward`. Si tratta di una rete neurale a due strati lineari con una funzione di attivazione ReLU intermedia e un layer di dropout per la regolarizzazione. Questa struttura è identica a quella descritta per i blocchi del Transformer standard e di T5, e ha la funzione di introdurre non-linearità e di processare le rappresentazioni prodotte dal layer di attenzione.

Meccanismo di attenzione Il cuore di ogni blocco è il meccanismo di Multi-Head Attention, implementato nella classe `T5Attention`. Questa classe è progettata per gestire sia la self-attention, dove l'input `x` è uguale al contesto, sia la cross-attention, dove il contesto proviene dall'encoder. L'input viene proiettato in Query, Key e Value tramite layer lineari senza bias. Queste proiezioni vengono poi suddivise in più teste di attenzione per catturare diversi tipi di relazioni in parallelo. I punteggi di attenzione sono calcolati tramite `scaled dot-product attention`. Coerentemente con T5, il bias posizionale relativo, calcolato dalla classe `T5RelativePositionBias`, viene sommato ai punteggi solo se l'opzione `use_relative_bias` è abilitata. Questa opzione viene tipicamente attivata per la self-attention sia nell'encoder che nel decoder, ma disattivata per la cross-attention del decoder, una scelta coerente con molte implementazioni pratiche.

2.2.3 Assemblaggio dei blocchi Encoder e Decoder

Le componenti sopra descritte sono assemblate per formare i blocchi costitutivi del modello, seguendo la logica di T5.

EncoderBlock Ogni `EncoderBlock` è composto da un layer di self-attention seguito da una rete feed-forward. Crucialmente, viene adottata la configurazione Pre-LayerNorm, in linea con T5: l'input di ogni sotto-blocco, prima l'attenzione e poi la rete feed-forward, viene normalizzato dalla classe `T5LayerNorm` e solo successivamente processato. L'output

del sotto-blocco viene poi aggiunto all'input originale tramite una connessione residuale, seguita da un layer di dropout.

DecoderBlock Il `DecoderBlock` è strutturalmente più complesso e contiene tre sotto-blocchi. Il primo è un layer di `masked self-attention` (causale e con bias relativi) che opera sulla sequenza target. Il secondo è un layer di `cross-attention` (non causale e tipicamente senza bias relativi) che permette al decoder di integrare le informazioni provenienti dall'output dell'encoder. Il terzo è la rete feed-forward. Anche in questo caso, ogni sotto-blocco adotta la configurazione Pre-LN, con una connessione residuale e dropout applicati dopo ogni operazione, replicando il flusso di dati descritto per il decoder di T5.

2.2.4 Struttura finale del modello transformer

Il modello completo, definito nella classe `Transformer`, orchestra l'intero flusso di dati. Esso contiene uno stack di N `EncoderBlock` e uno stack di N `DecoderBlock`. Una sequenza di input viene prima convertita in embedding tramite `InputEmbeddings` e processata sequenzialmente da tutti i blocchi dell'encoder. La rappresentazione finale dell'encoder viene quindi normalizzata. Questo output, insieme alla sequenza target, viene poi passato allo stack del decoder. L'output finale del decoder, dopo un'ulteriore normalizzazione, viene proiettato nello spazio del vocabolario da un layer lineare finale per produrre i logit. L'inizializzazione dei pesi con Xavier Uniform, gestita dalla funzione `_init_weights`, è una pratica standard per promuovere un flusso di gradiente stabile. Inoltre, coerentemente con il design del T5, il vocabolario e la matrice di output possono condividere gli stessi pesi per migliorare la coerenza semantica.

2.3 Preparazione dei dati per il training

Se l'architettura del modello definisce il "come" l'informazione viene processata, la classe `NanoSocratesDataset`, implementata nel file `dataset_lib.py`, definisce il "cosa" viene effettivamente dato in input al modello. Questa classe funge da ponte tra il corpus di dati testuali e il formato tensoriale strutturato richiesto per l'addestramento, orchestrando una serie di trasformazioni cruciali. La logica centrale di questa preparazione è incapsulata nel metodo `__getitem__`, che per ogni coppia di frasi sorgente-target esegue una pipeline di pre-processing ben definita. Il processo ha inizio nel costruttore della classe, `__init__`, che riceve i tokenizer per le due lingue e la lunghezza di sequenza massima, `seq_len`, che il modello può gestire. Un'operazione preliminare fondamentale è la memorizzazione degli ID numerici associati ai token speciali, come inizio sequenza (`<SOT>`), fine sequenza (`<EOT>`) e padding (`<PAD>`), recuperandoli direttamente dai tokenizer. Questo evita di doverli ricercare ad ogni accesso, ottimizzando il processo. Il cuore della classe, il metodo `__getitem__`, si occupa della trasformazione di una singola coppia di testi. Innanzitutto, le frasi sorgente e target vengono convertite in sequenze di ID numerici tramite il metodo `encode` dei rispettivi tokenizer. Poiché il modello Transformer opera su sequenze di lunghezza fissa, è necessario garantire che nessun input superi la `seq_len` definita. Per questo motivo, le sequenze di ID vengono troncate, riservando due posizioni per i token speciali di inizio e fine che verranno aggiunti successivamente. A questo punto, vengono costruiti i tensori numerici che alimenteranno l'encoder e il decoder. Per l'encoder, si costruisce il tensore `encoder_input`. Questo è formato dalla sequenza di token sorgente,

a cui vengono anteposti il token <SOT> e posposti il token <EOT>. La sequenza risultante viene poi estesa con un numero necessario di token <PAD> fino a raggiungere esattamente la lunghezza seq_len. Per il lato decoder, la preparazione è più articolata e finalizzata a implementare la tecnica del teacher forcing, creando quindi due tensori distinti. Il primo, decoder_input, rappresenta l'input che il decoder riceverà ad ogni passo temporale durante l'addestramento. È composto dalla sequenza target a cui viene aggiunto il solo token <SOT> all'inizio e successivamente riempita con token di padding fino alla lunghezza massima. Il secondo tensore è la label, che rappresenta la sequenza che il modello deve imparare a predire. Questa è costituita dalla sequenza target originale a cui viene aggiunto il token <EOT> alla fine, anch'essa riempita con padding. Questa separazione è cruciale: il decoder impara a predire il prossimo token della sequenza label avendo visto come input la sequenza decoder_input, che è la sequenza target "spostata a destra" di una posizione. Infine, vengono generate le maschere di attenzione, essenziali per il corretto funzionamento del meccanismo di attention. Per l'encoder, viene creata una semplice maschera di padding, encoder_mask, un tensore booleano che indica al modello di ignorare le posizioni occupate dai token <PAD>. Per il decoder, la maschera, decoder_mask, è più complessa, infatti deve assolvere a due compiti: mascherare i token di padding e, allo stesso tempo, implementare una maschera causale. La maschera causale impedisce a ciascun token della sequenza di "vedere" i token successivi. Questo viene realizzato tramite la funzione causal_mask, che genera una matrice triangolare superiore in cui le posizioni future sono mascherate. La maschera di padding del decoder viene quindi combinata con questa maschera causale, garantendo che l'attenzione di ogni token sia limitata solo ai token precedenti e a se stesso, escludendo sia il futuro sia il padding. Il metodo restituisce infine un dizionario contenente tutti questi tensori, pronti per essere utilizzati. Sarà poi il DataLoader di PyTorch a raggruppare questi dizionari in batch, creando i dati formattati che verranno passati al modello ad ogni passo del ciclo di addestramento.

2.4 Strategia di addestramento e ottimizzazione

L'intero processo di addestramento del modello è stato organizzato in tre fasi distinte ma interconnesse: (1) una fase di **pre-training** auto-supervisionato su dati di dominio generale, (2) una fase intermedia di **decoder tuning** con encoder congelato e (3) una fase finale di **fine-tuning** end-to-end su task supervisionati. Questa suddivisione non è arbitraria, ma si ispira a linee di ricerca consolidate nella letteratura sull'adattamento dei modelli linguistici di grandi dimensioni. In particolare, nel paper [2] mostrano che una fase aggiuntiva di pretraining su dati *in-domain* — nota come **Domain-Adaptive Pretraining (DAPT)** — migliora in modo consistente le prestazioni nei task downstream rispetto al semplice fine-tuning diretto. Questo risultato motiva l'introduzione di una prima fase di addestramento focalizzata sul dominio semantico specifico, con l'obiettivo di specializzare le rappresentazioni dell'encoder prima di affrontare compiti supervisionati. Successivamente, per mitigare il rischio di *catastrophic forgetting* e stabilizzare la convergenza del modello, viene impiegata una fase intermedia di **decoder tuning**, in cui l'encoder rimane congelato mentre si addestra il decoder. Questa scelta implementativa è ispirata alle tecniche di **gradual unfreezing** introdotte da Howard e Ruder (2018) nel lavoro ULMFiT [4], in cui i layer vengono sbloccati progressivamente durante il fine-tuning per evitare che l'aggiornamento simultaneo di tutti i parametri comprometta le rappresentazioni preaddestrate. L'idea di non aggiornare tutti i parametri contemporaneamente, ma di introdurre un adattamento progressivo dei pesi, è oggi considerata una

buona pratica in molte strategie di transfer learning. Infine, la terza fase di **fine-tuning end-to-end** aggiorna l'intera architettura encoder-decoder su task specifici, consolidando l'adattamento del modello al dominio e al formato di output desiderato. Studi sistematici come quello di Sun et al. (2019) [7] evidenziano come parametri quali il learning rate, lo scheduler, la regolarizzazione e la scelta dei layer da aggiornare influiscano in modo sostanziale sulla stabilità e sulle prestazioni del fine-tuning. La pipeline adottata in questo lavoro combina quindi concetti di **pretraining adattivo** (DAPT) e di **fine-tuning graduale** (ULMFiT), proponendo una strategia ibrida che, pur non corrispondendo letteralmente a nessuno di questi lavori, ne integra i principi fondamentali in un approccio coerente e metodologicamente motivato. Per completezza, va infine ricordato che, mentre nel lavoro originale di T5 [6] l'ottimizzazione viene condotta con **Adafactor** per ragioni di efficienza di memoria, in questo progetto si è scelto di utilizzare **AdamW**, una variante di Adam con gestione del weight decay più stabile, mantenendo inalterati i principi generali dell'addestramento. Infine, il tasso di apprendimento è gestito dinamicamente tramite uno scheduler. Il codice implementa due possibili strategie, selezionabili tramite configurazione. La prima è `get_linear_schedule_with_warmup`, che prevede una fase iniziale di warm-up in cui il learning rate aumenta linearmente fino a un valore massimo, per poi decadere linearmente per il resto dei passi. La seconda opzione è `CosineAnnealingWarmRestarts`, che adotta un andamento ciclico cosinusoidale. Queste strategie permettono al modello di stabilizzarsi nelle fasi inziali del training e di convergere più finemente verso un ottimo.

Chapter 3

Valutazione del modello

La valutazione delle performance di un modello generativo multi-task come NanoSocrates richiede un approccio metodologico che vada oltre la semplice misurazione della funzione di perdita. Per ottenere una stima affidabile e granulare delle capacità del modello, è stata progettata una pipeline di validazione robusta, implementata nella funzione run_validation del file train_final.py. Questa pipeline viene eseguita periodicamente durante l'addestramento per monitorare i progressi e, in modo più esaustivo, al termine di ogni fase per una valutazione completa. Questo capitolo descrive in dettaglio le strategie di generazione del testo, le metriche di valutazione adottate per ciascun task e l'analisi dei risultati ottenuti.

3.1 Processo di validazione e strategie di decodifica

A differenza della fase di addestramento, in cui il modello è guidato dal teacher forcing, durante la validazione è necessario generare le sequenze di output in modo autoregressivo. Per questo scopo, sono state implementate due strategie di decodifica distinte, che permettono di esplorare il compromesso tra velocità computazionale e qualità della generazione.

Greedy Search La strategia di decodifica più semplice è la Greedy Search, implementata nella funzione `greedy_decode`. Ad ogni passo temporale, questo algoritmo seleziona il token con la probabilità più alta (logit massimo) dall'output del modello e lo aggiunge alla sequenza in costruzione. Sebbene sia computazionalmente molto efficiente, questo approccio è "miope": una scelta localmente ottimale potrebbe portare a una sequenza complessivamente sub-ottimale. Per mitigare la tendenza dei modelli a generare testo ripetitivo, è stata integrata una tecnica di *repetition penalty* che riduce dinamicamente i logit dei token già presenti nella sequenza generata, incoraggiando il modello a esplorare un vocabolario più vario.

Beam Search Per una generazione di qualità superiore, è stata implementata anche la strategia Beam Search nella funzione `beam_search_decode`. Invece di considerare una singola ipotesi ad ogni passo, questo algoritmo mantiene in memoria un numero predefinito di k (beam size) sequenze candidate, o "ipotesi". Ad ogni passo di decodifica, per ciascuna delle k ipotesi, vengono generate le k estensioni più probabili, risultando in k*k nuove possibili sequenze. Da questo insieme, vengono selezionate le k sequenze con il punteggio log-probabilistico cumulativo più alto per procedere al passo successivo. Per evitare un bias verso le sequenze più corte, i punteggi vengono normalizzati in base alla

lunghezza della sequenza. Questa esplorazione più ampia dello spazio di ricerca permette di identificare sequenze che, pur non essendo ottimali a livello locale, risultano globalmente più coerenti e probabili. Anche in questo caso, è stata applicata la repetition penalty per migliorare la diversità dell'output. La valutazione finale del modello, al termine della fase di fine-tuning, viene eseguita utilizzando questa strategia di decodifica più sofisticata.

La natura multi-fase della strategia di addestramento implica che anche la valutazione debba essere adattata agli obiettivi specifici di ciascuna fase. Le metriche utilizzate per giudicare le performance del modello durante il pre-training sono intrinsecamente diverse e più semplici di quelle impiegate durante il fine-tuning, riflettendo la differenza fondamentale nello scopo di questi due stadi.

3.2 Metriche di valutazione per il pre-training

L'obiettivo primario della fase di pre-training non è risolvere un task specifico di comprensione o generazione, bensì apprendere le rappresentazioni fondamentali del linguaggio e della struttura dei dati del dominio. Il task auto-supervisionato di Span Corruption serve come un pretesto per costringere il modello a sviluppare una comprensione profonda della sintassi, della semantica e delle co-occorrenze statistiche presenti nel corpus. Di conseguenza, la valutazione in questa fase si concentra sulla capacità del modello di ricostruire fedelmente il testo originale, un compito a più basso livello semantico rispetto ai task di fine-tuning. Per questo motivo, come si evince dal ramo logico dedicato alla fase di pretrain nella funzione run_validation, la valutazione si limita a due metriche quantitative fondamentali:

Average Validation Loss Questa è la misura più diretta delle performance del modello sul suo obiettivo di ottimizzazione. La Cross-Entropy Loss calcolata sul dataset di validazione indica quanto le predizioni del modello, in media, si discostino dalla verità (ground-truth). Un valore di loss in diminuzione nel tempo è l'indicatore primario che il modello sta apprendendo con successo a minimizzare il proprio errore e a convergere.

Token-level Accuracy Mentre la loss fornisce una misura "soft" dell'errore, l'accuratezza a livello di token offre una prospettiva più "hard" e interpretabile. Questa metrica calcola la percentuale di token non-padding che il modello predice correttamente. Sebbene non catturi la coerenza semantica di una sequenza, in un task di denoising come lo Span Corruption, un'alta accuratezza sui token è un forte indicatore che il modello sta imparando a riempire correttamente gli span mascherati. È una metrica diagnostica essenziale per confermare che l'apprendimento non si sia arrestato e che il modello stia effettivamente migliorando nella sua capacità di ricostruzione testuale.

In questa fase non vengono utilizzate metriche più complesse come BLEU o ROUGE, poiché risulterebbero fuorvianti, infatti il task non è una "traduzione" in senso semantico, ma una ricostruzione. Valutare la qualità semantica della generazione basata su un input corrotto artificialmente non fornirebbe indicazioni utili sull'effettiva comprensione del linguaggio da parte del modello, obiettivo che verrà invece misurato rigorosamente nelle fasi successive.

3.3 Metriche di valutazione per il fine-tuning

Con il passaggio alle fasi di decoder_tune e full_finetune, gli obiettivi del modello diventano più complessi e semanticamente ricchi. La valutazione, di conseguenza, deve evolvere per catturare non solo la correttezza a livello di token, ma anche la qualità strutturale e semantica delle generazioni sui task specifici. La pipeline di validazione per queste fasi è quindi multi-livello e inizia con un insieme di metriche generali che forniscono una visione d'insieme delle performance del modello.

3.3.1 Average Validation Loss

Analogamente alla fase di pre-training, la Cross-Entropy Loss calcolata sul set di validazione rimane una metrica fondamentale. Essa continua a rappresentare l'obiettivo primario di ottimizzazione e una sua diminuzione costante indica che il modello sta apprendendo con successo la mappatura per i task di fine-tuning. Durante queste fasi, la loss viene calcolata con l'aggiunta del label smoothing, il che la rende anche un indicatore della capacità del modello di generalizzare senza diventare eccessivamente confidente.

3.3.2 Metriche a livello di token (Precision, Recall, F1-Score)

Sebbene l'accuratezza semplice sia utile, per i task generativi una valutazione più granulare a livello di token può offrire maggiori spunti diagnostici. Invece di una singola metrica di accuratezza, l'implementazione calcola Precision, Recall e F1-Score a livello di token. Queste metriche trattano la generazione come un problema di recupero di un "insieme" di token corretti. Per formalizzare queste metriche, definiamo i True Positives (TP) come i token predetti correttamente, i False Positives (FP) come i token predetti erroneamente, e i False Negatives (FN) come i token di riferimento che il modello non ha generato.

- La **Token Precision** calcola la proporzione di token generati dal modello che sono corretti (presenti nella sequenza di riferimento). Un valore alto indica che il modello tende a non generare token errati.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3.1}$$

- La **Token Recall** calcola la proporzione di token di riferimento che il modello è stato in grado di generare. Un valore alto indica che il modello riesce a produrre la maggior parte del contenuto richiesto.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3.2}$$

- Il **Token F1-Score**, essendo la media armonica di Precision e Recall, fornisce un bilancio unico tra le due, particolarmente utile in caso di squilibrio tra FP e FN.

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.3}$$

Queste metriche, pur operando a un basso livello di astrazione, sono preziose per diagnosticare problemi specifici. Ad esempio, una bassa recall potrebbe indicare che il modello

genera sequenze sistematicamente troppo corte, mentre una bassa precision potrebbe suggerire la presenza di "allucinazioni" o la generazione di token spuri. Tuttavia, essendo metriche che non considerano l'ordine dei token, il loro ruolo è puramente diagnostico e non sostituiscono le valutazioni semantiche più avanzate.

3.3.3 Metriche specifiche per i task RDF

Per i task orientati alla generazione di conoscenza strutturata, come `<Text2RDF>` e `<CONTINUERDF>`, le metriche a livello di token sono insufficienti. Esse non possono determinare se il modello ha prodotto triple RDF sintatticamente valide o semanticamente corrette. Per superare questa limitazione, sono state implementate due strategie di valutazione complementari che analizzano l'output a un livello di astrazione più elevato: il livello delle triple e il livello delle singole entità.

Valutazione basata sulla corrispondenza esatta delle triple (Strict Matching)

Questo approccio rappresenta la valutazione più rigorosa delle performance del modello. L'obiettivo è misurare la capacità del modello di generare un insieme di triple che corrisponda esattamente a quello di riferimento. Come implementato nella funzione `parse_rdf_triples_for_strict_eval`, sia le triple predette che quelle attese vengono prima estratte dal testo e normalizzate, per poi essere rappresentate come insiemi di tuple (soggetto, predicato, oggetto). L'uso della struttura dati set garantisce che l'ordine delle triple e le eventuali duplicazioni non influenzino il risultato. Definendo l'insieme delle triple predette come P_{triples} e l'insieme delle triple di riferimento (ground-truth) come G_{triples}, possiamo calcolare i True Positives (TP) come l'intersezione tra i due insiemi. Di conseguenza, Precision, Recall e F1-Score sono definite come segue:

- La **Precision (Strict)** misura la frazione di triple generate che sono corrette. Un punteggio elevato indica che il modello è affidabile e non genera conoscenza errata.

$$\text{Precision}_{\text{strict}} = \frac{|P_{\text{triples}} \cap G_{\text{triples}}|}{|P_{\text{triples}}|} \tag{3.4}$$

- La **Recall (Strict)** misura la frazione di triple di riferimento che il modello è stato in grado di generare. Un punteggio elevato indica che il modello è completo e non omette informazioni importanti.

$$\text{Recall}_{\text{strict}} = \frac{|P_{\text{triples}} \cap G_{\text{triples}}|}{|G_{\text{triples}}|} \tag{3.5}$$

- L'**F1-Score (Strict)** fornisce un bilancio unico tra precisione e completezza.

$$\text{F1-Score}_{\text{strict}} = 2 \cdot \frac{\text{Precision}_{\text{strict}} \cdot \text{Recall}_{\text{strict}}}{\text{Precision}_{\text{strict}} + \text{Recall}_{\text{strict}}} \tag{3.6}$$

Questa metrica è molto severa: qualsiasi minima discrepanza in una stringa di un soggetto, predicato o oggetto classifica l'intera tripla come errata. Sebbene sia un eccellente indicatore di accuratezza fattuale, potrebbe non catturare i progressi parziali del modello.

Valutazione a livello di entità

Per ottenere una visione più granulare e diagnostica, è stata implementata una seconda strategia che valuta le componenti delle triple (soggetti, predicati, oggetti) in modo disaccoppiato. Questo approccio, gestito dalla funzione `parse_rdf_triples_for_entity_eval`, permette di capire se il modello stia apprendendo a generare le corrette tipologie di entità, anche qualora non riesca a combinarle correttamente in triple perfette. Per ogni categoria (soggetti, predicati, oggetti), vengono aggregate tutte le entità predette e tutte quelle di riferimento in due multi-insiemi. Utilizzando questi multi-insiemi, vengono calcolati Precision, Recall e F1-Score per ciascuna categoria. Ad esempio, per la categoria dei soggetti, le metriche sono calcolate come:

$$\text{Precision}_{\text{subj}} = \frac{\text{TP}_{\text{subj}}}{\text{TP}_{\text{subj}} + \text{FP}_{\text{subj}}} \qquad \text{Recall}_{\text{subj}} = \frac{\text{TP}_{\text{subj}}}{\text{TP}_{\text{subj}} + \text{FN}_{\text{subj}}} \tag{3.7}$$

dove TP_{subj} è il numero di soggetti corretti, FP_{subj} è il numero di soggetti predetti erroneamente, e FN_{subj} è il numero di soggetti di riferimento non predetti. Lo stesso calcolo viene replicato per predicati e oggetti. Questa analisi disaggregata è estremamente utile per l'analisi degli errori, poiché può rivelare se il modello ha difficoltà a predire una specifica componente, come ad esempio gli oggetti (che hanno una variabilità lessicale molto più alta) rispetto ai predicati (che appartengono a un vocabolario più ristretto).

3.3.4 Metriche specifiche per il task RDF2Text

Il task <RDF2Text> ha l'obiettivo di generare testo in linguaggio naturale (un abstract di un film) a partire da dati strutturati (triple RDF). La valutazione di questo compito richiede metriche che possano misurare la qualità, la fluidità e la fedeltà semantica del testo generato rispetto a un testo di riferimento scritto da un umano. Per questo scopo, sono state adottate tre metriche standard nel campo della NLG: BLEU, ROUGE e METEOR.

BLEU (Bilingual Evaluation Understudy)

BLEU è una metrica orientata alla precisione, originariamente progettata per la traduzione automatica. Misura la proporzione di n-grammi (sequenze di n parole) nel testo generato (ipotesi) che appaiono anche nel testo di riferimento. Per evitare di premiare eccessivamente le ripetizioni, BLEU utilizza una "precisione modificata" (p_n), che limita il conteggio di ogni n-gramma al numero massimo di volte in cui appare nel riferimento. La metrica soffre di un bias verso le frasi corte, che viene corretto da una "penalità di brevità" (Brevity Penalty, BP). Il punteggio finale è la media geometrica delle precisioni modificate per n-grammi di diversa lunghezza (solitamente da 1 a 4), moltiplicata per la penalità di brevità.

Dati una sequenza candidata c e una sequenza di riferimento r, il punteggio BLEU è calcolato come:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{3.8}$$

dove w_n sono i pesi (tipicamente $1/N$) e BP è definito come:

$$\text{BP} = \begin{cases} 1 & \text{if } |c| > |r| \\ e^{1-|r|/|c|} & \text{if } |c| \leq |r| \end{cases} \tag{3.9}$$

ROUGE (Recall-Oriented Understudy for Gisting Evaluation)

ROUGE è una famiglia di metriche orientate alla recall, che valutano quanti n-grammi del testo di riferimento sono stati catturati nel testo generato. È particolarmente utile per task come il riassunto, dove è cruciale coprire i punti salienti.

- **ROUGE-N** calcola la recall basandosi su n-grammi. Ad esempio, ROUGE-1 calcola la sovrapposizione di singole parole (unigrammi), mentre ROUGE-2 usa coppie di parole (bigrammi).

$$\text{ROUGE-N} = \frac{\sum_{\text{gram}_n \in S_{\text{ref}}} \text{Count}_{\text{match}}(\text{gram}_n)}{\sum_{\text{gram}_n \in S_{\text{ref}}} \text{Count}(\text{gram}_n)} \tag{3.10}$$

- **ROUGE-L**, la variante utilizzata nel nostro report (rougeL), si basa sulla Sottosequenza Comune più Lunga (Longest Common Subsequence, LCS). Questa metrica non richiede che le parole corrispondenti siano contigue, rendendola più flessibile nel catturare la somiglianza strutturale a livello di frase. Viene calcolato un F1-Score basato su precisione e recall dell'LCS.

METEOR (Metric for Evaluation of Translation with Explicit ORdering)

METEOR è una metrica più sofisticata che mira a superare alcune delle limitazioni di BLEU e ROUGE. Invece di basarsi su corrispondenze esatte di n-grammi, METEOR crea un allineamento tra le parole del testo generato e di riferimento, considerando non solo le corrispondenze esatte, ma anche quelle basate su stemming e sinonimia (tramite WordNet). Calcola Precision (P) e Recall (R) basate su questo allineamento di unigrammi. La sua caratteristica distintiva è una "penalità di frammentazione", che penalizza le generazioni in cui le parole corrispondenti sono sparse e non disposte in blocchi contigui. La metrica finale è la media armonica di P e R, modulata dalla penalità.

$$\text{METEOR} = F_{\text{mean}} \cdot (1 - \text{Penalty}) \tag{3.11}$$

dove $F_{\text{mean}} = \frac{10PR}{R+9P}$ e la penalità aumenta con il numero di "blocchi" contigui di parole allineate. METEOR ha dimostrato di avere una correlazione più alta con il giudizio umano rispetto a BLEU e ROUGE.

3.3.5 Metriche specifiche per il task MLM

Il task `<MLM>` (Masked Language Modeling) è un compito di tipo "fill-in-the-blank" in cui il modello deve predire una singola componente mancante (un predicato o un oggetto) all'interno di una tripla RDF. Data la natura atomica dell'output atteso (una singola entità o un letterale), la metrica più appropriata è l'accuratezza. Tuttavia, la natura specifica dei dati RDF suggerisce la necessità di una valutazione a due livelli, come implementato nel codice, per distinguere tra correttezza sintattica e semantica.

Accuracy (Strict)

Questa è la metrica di valutazione più diretta e intuitiva. Misura la percentuale di predizioni in cui il testo generato dal modello corrisponde esattamente, carattere per carattere, alla stringa di riferimento, dopo aver rimosso eventuali spazi bianchi iniziali o finali.

Data una collezione di N esempi di validazione, dove p_i è la predizione per l'esempio i e g_i è il ground-truth, l'accuratezza stretta è definita come:

$$\text{Accuracy}_{\text{strict}} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}(p_i = g_i) \tag{3.12}$$

dove $\mathbb{I}(\cdot)$ è la funzione indicatrice, che vale 1 se la condizione al suo interno è vera, e 0 altrimenti. Questa metrica è inflessibile e non ammette alcun tipo di variazione, fornendo una misura della capacità del modello di produrre l'output esatto atteso.

Accuracy (Soft)

Una valutazione basata unicamente sulla corrispondenza esatta può essere eccessivamente punitiva, specialmente quando si predicono entità RDF che includono prefissi (es. dbo:). Ad esempio, se il modello predice "director" quando il riferimento è "dbo:director", da un punto di vista semantico la predizione è corretta, ma verrebbe marcata come errore dalla metrica strict. Per catturare questi casi di "correttezza semantica", è stata introdotta una metrica di Accuracy (Soft). Questa metrica applica una logica di valutazione più flessibile. Se la stringa di riferimento (ground-truth) inizia con un prefisso RDF noto (come dbo:), la metrica considera corretta la predizione del modello anche se questa omette il prefisso, purché il resto della stringa corrisponda. Per tutti gli altri casi (ad esempio, quando il riferimento è un letterale), la valutazione soft coincide con quella strict.

Questa doppia valutazione fornisce una visione più completa delle capacità del modello. L'accuratezza strict misura la sua abilità nel rispettare il formato sintattico dei dati, mentre l'accuratezza soft offre una stima più realistica della sua comprensione semantica del dominio ontologico.

3.4 Analisi Qualitativa

Accanto alle metriche quantitative, che forniscono una misura aggregata e oggettiva delle performance, una componente essenziale del processo di valutazione è l'analisi qualitativa. Questa pratica, implementata al termine di ogni esecuzione della pipeline di validazione, consiste nell'ispezione manuale di un campione di esempi concreti generati dal modello. Per ciascun task di fine-tuning, vengono selezionati e registrati alcuni esempi rappresentativi, mostrando al lettore la tripletta completa:

- **Source**: L'input fornito al modello.
- **Expected**: L'output di riferimento (ground-truth).
- **Predicted**: L'output effettivamente generato dal modello.

Questo esame diretto offre intuizioni preziose che le metriche numeriche da sole non possono fornire. Permette di identificare pattern di errore ricorrenti (es. errori grammaticali in RDF2Text, generazione di predicati plausibili ma errati in Text2RDF, difficoltà con entità poco frequenti), così come di apprezzare i successi del modello in modo più concreto. L'analisi qualitativa è quindi uno strumento diagnostico indispensabile che arricchisce l'interpretazione dei risultati quantitativi e guida la discussione sulle capacità e sui limiti del modello, che verrà affrontata nel capitolo successivo.

Chapter 4

Impostazione sperimentale e risultati

Questo capitolo descrive in dettaglio l'impostazione sperimentale adottata per addestrare e valutare il modello NanoSocrates, presenta i risultati quantitativi e qualitativi ottenuti, e offre un'analisi comparativa delle diverse configurazioni testate. L'obiettivo non è solo riportare le performance finali, ma anche illustrare il processo iterativo di sperimentazione che ha guidato lo sviluppo del modello.

4.1 Configurazione degli Esperimenti

Per investigare in modo controllato l'impatto della capacità del modello e della quantità di dati sulle performance, sono state definite due principali configurazioni architetturali, denominate "Nano" e "Micro". Ogni esperimento condotto utilizza una di queste configurazioni come base, seguendo poi la strategia di addestramento a tre fasi descritta nel capitolo precedente. Le specifiche di ogni configurazione sono definite nelle funzioni 'get_nano_config()' e 'get_micro_config()' all'interno del file 'config_pretrain.py'.

Configurazione "Nano" Questa architettura rappresenta il modello di dimensioni maggiori utilizzato nei nostri esperimenti. È caratterizzata da una dimensione degli embedding e dei layer nascosti ('d_model') pari a 256. L'encoder e il decoder sono composti da uno stack di N=2 blocchi, ciascuno contenente h=4 teste di attenzione. All'interno di ogni blocco, la rete feed-forward ha una dimensione intermedia ('d_ff') di 1024. Questa configurazione, che conta circa 20 milioni di parametri, è stata progettata come un compromesso bilanciato tra capacità rappresentazionale e requisiti computazionali, rendendola adatta a dataset di medie dimensioni. Il 'batch_size' è impostato a 16 per accomodare l'occupazione di memoria di questo modello.

Configurazione "Micro" Questa architettura è una versione ridotta della configurazione Nano, progettata esplicitamente per esperimenti su dataset più piccoli e per analizzare gli effetti di una minore capacità del modello. La dimensione degli embedding e dei layer nascosti ('d_model') è dimezzata a 128, così come la dimensione intermedia della rete feed-forward ('d_ff'), che scende a 512. Il numero di blocchi (N=2) e di teste di attenzione (h=4) rimane invariato per mantenere una coerenza strutturale. Con circa 9 milioni di parametri, questo modello è significativamente più leggero. Tale riduzione di capacità permette di utilizzare un 'batch_size' maggiore, pari a 32, a parità di risorse computazionali.

Entrambe le configurazioni condividono una lunghezza di sequenza massima (‘seq_len‘) di 256 token. La scelta tra queste due architetture, combinata con la variazione della dimensione del dataset (5.000, 10.000 o 30.000 film), costituisce la base per la nostra analisi sperimentale comparativa.

Ambiente computazionale È importante sottolineare che tutti gli esperimenti sono stati condotti in un ambiente con risorse computazionali limitate. Specificamente, l'addestramento e la valutazione sono stati eseguiti su due computer Apple MacBook Pro, entrambi equipaggiati con processori Apple Silicon M1 Pro. Sebbene questi processori offrano un'accelerazione hardware per il calcolo neurale tramite il framework Metal Performance Shaders (MPS), le loro capacità rimangono significativamente inferiori a quelle delle GPU dedicate di fascia alta (come le NVIDIA A100 o H100) comunemente utilizzate nella ricerca sui modelli linguistici. Questa limitazione ha influenzato direttamente le scelte progettuali, come la dimensione relativamente contenuta dei modelli "Nano" e "Micro", e ha motivato l'adozione della strategia di valutazione per trarre conclusioni robuste nonostante l'impossibilità di eseguire esperimenti su larga scala.

4.2 Strategia di Valutazione

Date le limitazioni computazionali che impediscono un'esplorazione esaustiva con modelli e dataset su larga scala, è stata adottata una strategia di "valutazione all'inverso". Partendo da un esperimento **Baseline** che ha stabilito un punto di riferimento di performance e sono stati condotti esperimenti mirati a isolare l'impatto di ciascuna variabile (dimensione del modello e quantità di dati). L'obiettivo è dimostrare, invalidando le alternative, che l'unica direzione plausibile per un miglioramento significativo delle performance è l'aumento congiunto di entrambe le variabili.

4.3 Analisi dei Risultati

In questa sezione vengono presentati e discussi i risultati di ogni fase di addestramento per i diversi esperimenti. I risultati finali si riferiscono sempre alle metriche ottenute al termine della fase di Full Fine-tuning, utilizzando la decodifica Greedy Search per coerenza tra le epoche e Beam Search per la valutazione finale.

4.3.1 Esperimento 1 (Baseline): Modello Nano su 10.000 Film

Il primo esperimento funge da baseline per la nostra analisi comparativa. In questa configurazione, il modello **Nano** è stato addestrato su un dataset di medie dimensioni, derivato da **10.000 film**. L'obiettivo è stabilire un punto di riferimento robusto delle performance, valutando le capacità del modello in una condizione di equilibrio tra la sua capacità interna e la quantità di dati a disposizione. L'addestramento ha seguito l'intera pipeline a tre fasi: pre-training, decoder-tuning e full fine-tuning per 60 epoche. La valutazione finale, eseguita con una strategia di decodifica Beam Search (beam size = 4), ha prodotto i risultati riassunti nella Tabella 4.2.

Table 4.1: Risultati dell'Esperimento 1 (Baseline) al termine della fase di Decoder-Tuning (epoca 19).

Categoria	**Metrica / Componente**	**Precision**	**Recall**	**F1-Score**
Generale	Validation Loss		3.0845	
NLG (Task RDF2Text)	BLEU		0.0116	
	METEOR		0.1305	
	ROUGE-L		0.1834	
RDF a Livello di Entità	Subjects	0.0027	0.0008	0.0012
	Predicates	0.9973	0.2872	0.4460
	Objects	0.0239	0.0069	0.0107
MLM	Accuracy (Soft)		0.1763	

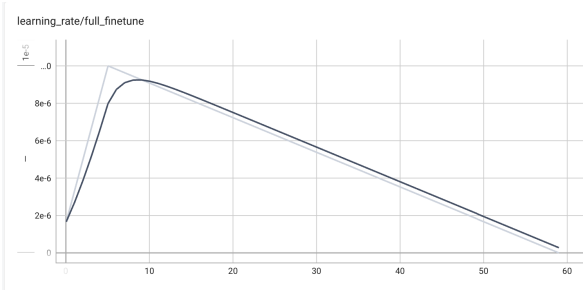
Table 4.2: Risultati finali dell'Esperimento 1 (Baseline) dopo 60 epoche di fine-tuning, valutati con Beam Search.

Categoria	**Metrica / Componente**	**Precision**	**Recall**	**F1-Score**
Generale	Validation Loss		2.2295	
NLG (Task RDF2Text)	BLEU		0.0203	
	METEOR		0.1506	
	ROUGE-L		0.2157	
RDF a Livello di Entità	Subjects	0.0367	0.0101	0.0158
	Predicates	0.9223	0.2536	0.3978
	Objects	0.1257	0.0346	0.0542
MLM	Accuracy (Soft)		0.2302	

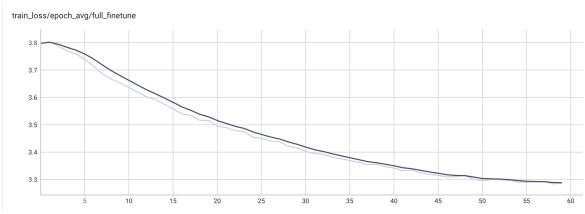
Analisi dei risultati quantitativi L'analisi dei risultati quantitativi rivela un quadro complesso. Le metriche per la generazione di triple RDF con corrispondenza esatta (F1-Score Strict) sono prossime allo zero, evidenziando l'estrema difficoltà del compito di generare conoscenza strutturata senza alcun errore. Tuttavia, uno sguardo più approfondito alle metriche a livello di entità offre un'intuizione cruciale: il modello ottiene un F1-Score relativamente discreto sui **predicati** (0.3978). Questo risultato è significativo e suggerisce che il modello ha appreso con successo il vocabolario ristretto e ben definito dei predicati ontologici (es. 'dbo:director', 'dbo:starring'). Al contrario, le performance su **soggetti** e **oggetti** sono estremamente basse. Ciò indica che la vera sfida per il modello non risiede nella comprensione della struttura delle triple, ma nella capacità di richiamare e generare correttamente le entità specifiche, che appartengono a un vocabolario aperto e molto più vasto. Invece per quanto riguarda il task `RDF2Text`, i punteggi delle metriche NLG (BLEU, ROUGE-L, METEOR) sono bassi ma significativamente superiori allo zero. Questo suggerisce che il modello non sta generando testo casuale, ma produce sequenze che hanno una certa sovrapposizione lessicale e strutturale con gli abstract di riferimento. Infine, l'Accuracy (Soft) del 23% sul task `MLM` conferma la tendenza: il modello ha una basilare comprensione del contesto, ma fatica a fornire la risposta fattualmente corretta.

Analisi dei risultati qualitativi L'ispezione degli esempi qualitativi conferma le conclusioni tratte dai dati numerici. Nel task 'RDF2Text', il modello genera frasi grammaticalmente corrette e stilisticamente plausibili, che assomigliano a un vero abstract di film. Tuttavia, le informazioni fattuali (nomi di attori, registi, date) sono quasi interamente

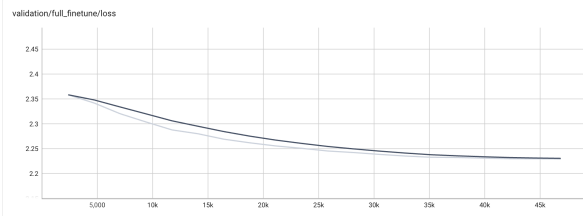
"allucinate", ovvero inventate ma coerenti con il contesto. Ad esempio, partendo dalle triple del film *Across the Bridge*, il modello genera correttamente una frase che inizia con "Across the Dark", ma inventa il resto dei dettagli. Analogamente, nel task `Text2RDF`, il modello dimostra di aver compreso l'intento del task (estrarre triple 'soggetto-predicato-oggetto'), ma fallisce nell'estrarre le entità corrette dal testo sorgente, generando invece entità generiche o plausibili (es. 'dbr:Silent_film'). Questo comportamento suggerisce che il modello ha imparato i "template" dei vari task, ma non possiede una capacità sufficiente per memorizzare e manipolare la grande quantità di conoscenza fattuale specifica richiesta per risolverli correttamente. Questo plateau di performance, caratterizzato da una buona comprensione strutturale ma da una scarsa accuratezza fattuale, definisce la nostra baseline e motiva la successiva analisi.



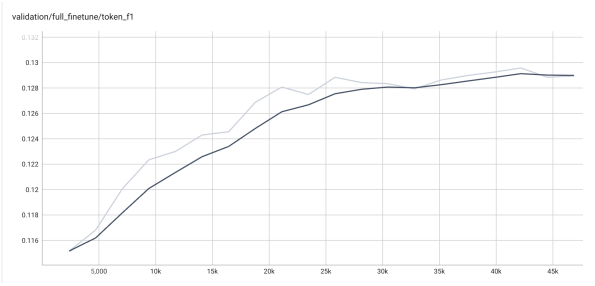
(a) Andamento del Learning Rate durante la fase di Full Fine-tuning. Il grafico mostra chiaramente la strategia dello scheduler con warm-up lineare: il learning rate aumenta linearmente fino al suo picco nelle prime epoche, per poi decadere gradualmente. Questo approccio stabilizza l'addestramento iniziale e permette una convergenza più fine nelle fasi successive.



(b) Andamento della Training Loss per epoca durante la fase di Full Fine-tuning. La curva mostra una discesa costante e regolare, indicando che il modello sta apprendendo con successo dai dati di addestramento. L'assenza di oscillazioni brusche suggerisce che il processo di ottimizzazione è stabile, grazie alla combinazione di AdamW e del learning rate scheduling.



(c) Andamento della Validation Loss durante la fase di Full Fine-tuning. La curva di validazione segue un andamento simile a quella di training, con una discesa rapida all'inizio che poi rallenta fino a raggiungere un plateau. Questo andamento è ideale, poiché indica che il modello sta generalizzando bene e non sta cadendo in un overfitting marcato.



(d) Andamento del Token F1-Score sul set di validazione. Questa metrica, che bilancia precisione e recall a livello di token, mostra un miglioramento costante che si assesta verso la fine dell'addestramento. L'aumento continuo, anche quando la loss si appiattisce, suggerisce che il modello sta diventando qualitativamente migliore nel produrre i token corretti.

Figure 4.1: Grafici di monitoraggio dell'addestramento per l'esperimento Baseline durante la fase di Full Fine-tuning. Le curve visualizzano (a) il learning rate, (b) la training loss media per epoca, (c) la validation loss e (d) il token F1-score sul set di validazione.

4.3.2 Esperimento 2: Impatto della Riduzione del Modello (Micro su 10.000 Film)

Il secondo esperimento è stato progettato per verificare la prima ipotesi della nostra strategia: che la capacità del modello sia un fattore cruciale per le performance. A tal fine, la quantità di dati è stata mantenuta invariata rispetto alla baseline (**10.000 film**), ma la dimensione del modello è stata ridotta, passando dalla configurazione **Nano** a quella **Micro**. Questa modifica riduce la capacità del modello di oltre il 50%, passando da circa 20 a 9 milioni di parametri. L'ipotesi è che un modello più piccolo sia insufficiente per catturare la complessità del dataset, manifestando un comportamento di *underfitting* e ottenendo performance inferiori rispetto alla baseline. Dato il degrado delle performance osservato già nelle fasi intermedie e in un'ottica di ottimizzazione delle risorse computazionali limitate, si è deciso di non procedere con la fase di full fine-tuning per questo esperimento. I risultati presentati nella Tabella 4.3 si riferiscono quindi al termine della fase di **decoder-tuning** (epoca 19), ritenuti sufficienti per trarre conclusioni significative.

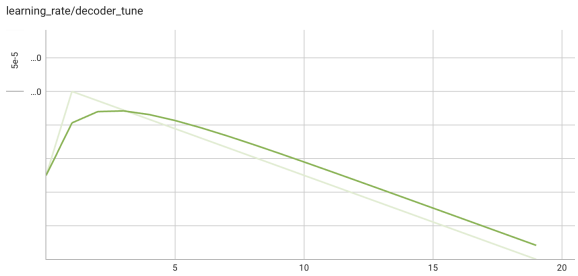
Table 4.3: Risultati dell'Esperimento 2 (Modello Micro, 10.000 film) al termine della fase di Decoder-Tuning (epoca 19).

Categoria	**Metrica / Componente**	**Precision**	**Recall**	**F1-Score**
Generale	Validation Loss		3.0845	
NLG (Task RDF2Text)	BLEU		0.0208	
	METEOR		0.1082	
	ROUGE-L		0.1144	
RDF a Livello di Entità	Subjects	0.0000	0.0000	0.0000
	Predicates	1.0000	0.2443	0.3927
	Objects	0.0000	0.0000	0.0000
MLM	Accuracy (Soft)		0.0000	

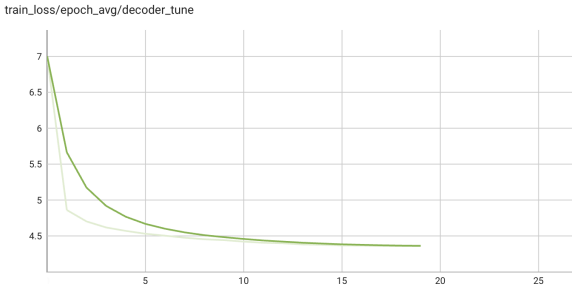
Analisi dei Risultati Il confronto con la baseline, anche a questo stadio intermedio dell'addestramento, conferma in modo netto l'ipotesi iniziale. Si osserva un **peggioramento generalizzato** su quasi tutte le metriche di alto livello. Il segnale più evidente di underfitting è il crollo totale delle performance sulla generazione di entità e sul task MLM. L'F1-Score sugli **oggetti** e l'Accuracy (Soft) sul task **MLM** sono entrambi pari a zero, indicando che il modello Micro non possiede una capacità sufficiente per memorizzare e richiamare correttamente la conoscenza fattuale. Sebbene mantenga una performance quasi identica sui predicati (F1-Score 0.3927), che hanno un vocabolario ristretto, fallisce completamente quando deve gestire entità a vocabolario aperto. Anche nel task di generazione di testo ('RDF2Text'), si registra un calo drastico rispetto alla baseline (ROUGE-L da 0.2157 a 0.1144, METEOR da 0.1506 a 0.1082). L'analisi qualitativa corrobora questo dato: le frasi generate dal modello Micro sono meno coerenti e spesso degenerate, come si nota nell'esempio di 'RDF2Text' dove il modello entra in un loop ripetitivo ("...was released on the film was released on..."). Questo comportamento è un classico sintomo di un modello che non ha avuto abbastanza capacità per apprendere le complesse dipendenze a lungo termine del linguaggio naturale.

Conclusioni dell'Esperimento I risultati, seppur parziali, dimostrano in modo conclusivo che una riduzione della capacità del modello porta a un deterioramento significativo

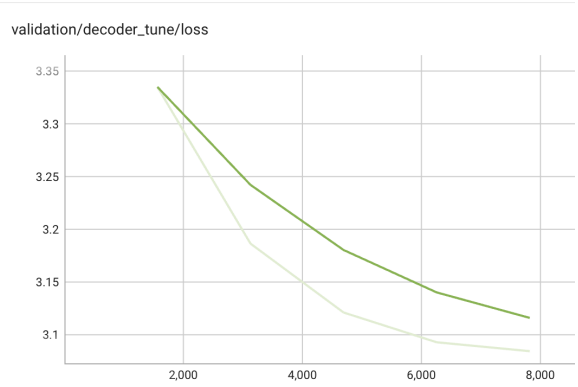
delle performance. Ciò invalida l'ipotesi che il modello Nano della baseline fosse "troppo grande" per i dati. Al contrario, si evidenzia che la sua maggiore capacità è necessaria e benefica. Si rafforza quindi la tesi che un aumento della capacità del modello, a patto di avere dati a sufficienza, sia una direzione promettente per migliorare i risultati. La decisione di interrompere l'esperimento prima del full fine-tuning è stata strategica, avendo già ottenuto una prova sufficiente a validare la nostra ipotesi con un costo computazionale ridotto.



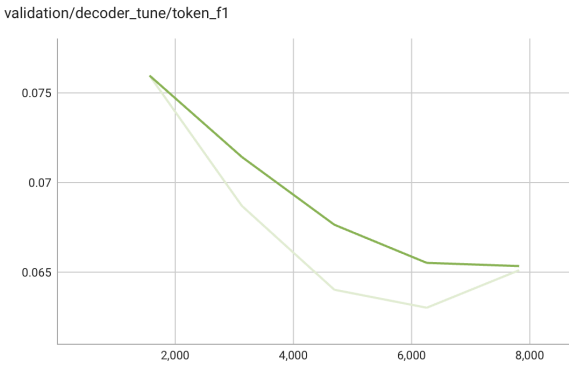
(a) Andamento del Learning Rate durante la fase di Decoder-Tuning. Anche in questa fase intermedia viene adottata una strategia di warm-up e decadimento lineare, con un picco di learning rate impostato a $5 \cdot 10^{-5}$, un valore più alto rispetto al full fine-tuning per accelerare l'adattamento dei soli pesi del decoder.



(b) Andamento della Training Loss per epoca durante la fase di Decoder-Tuning. La curva mostra una discesa iniziale, ma si appiattisce rapidamente su un valore di loss relativamente alto. Questo andamento suggerisce che il modello fatica a ridurre ulteriormente l'errore sui dati di training, un primo segnale della sua limitata capacità (underfitting).



(c) Andamento della Validation Loss durante la fase di Decoder-Tuning. Similmente alla training loss, anche la loss di validazione raggiunge presto un plateau, confermando che il modello Micro non riesce a generalizzare efficacemente. Il valore di loss finale (circa 3.08) è notevolmente più alto rispetto alla baseline, indicando performance inferiori.



(d) Andamento del Token F1-Score sul set di validazione. Questa metrica mostra un miglioramento minimo e si assesta su valori molto bassi (circa 0.065). Questo risultato, letto in combinazione con le metriche a livello di entità, conferma che la ridotta capacità del modello Micro gli impedisce di apprendere a generare sequenze di token qualitativamente accettabili.

Figure 4.2: Grafici di monitoraggio dell'addestramento per l'Esperimento 2 (Modello Micro) durante la fase di Decoder-Tuning. Le curve evidenziano i limiti di apprendimento di un modello con capacità ridotta.

4.3.3 Esperimento 3: Impatto della Riduzione dei Dati (Nano su 5.000 Film)

Il terzo esperimento mira a verificare la seconda ipotesi della nostra strategia: che la quantità di dati a disposizione sia un fattore limitante per le performance del modello. In questa configurazione, è stata mantenuta l'architettura **Nano** della baseline, ma il dataset di addestramento è stato significativamente ridotto, utilizzando i dati derivati da soli **5.000 film**. L'ipotesi è che, fornendo meno dati a un modello con una capacità relativamente elevata, si accentui il fenomeno dell'overfitting, portando a un calo delle performance rispetto alla baseline. In analogia con l'esperimento precedente, e avendo osservato fin da subito un andamento non promettente delle metriche, si è deciso di interrompere anche questo esperimento al termine della fase di **decoder-tuning** per ottimizzare l'uso delle risorse. I risultati in Tabella 4.4 si riferiscono quindi alle performance misurate alla fine di questa fase intermedia (epoca 19).

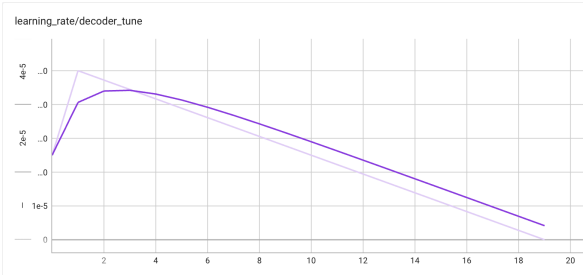
Table 4.4: Risultati dell'Esperimento 3 (Modello Nano, 5.000 film) al termine della fase di Decoder-Tuning (epoca 19).

Categoria	**Metrica / Componente**	**Precision**	**Recall**	**F1-Score**
Generale	Validation Loss		3.0597	
NLG (Task RDF2Text)	BLEU		0.0357	
	METEOR		0.1502	
	ROUGE-L		0.1837	
RDF a Livello di Entità	Subjects	0.0000	0.0000	0.0000
	Predicates	1.0000	0.2118	0.3495
	Objects	0.0000	0.0000	0.0000
MLM	Accuracy (Soft)		0.0370	

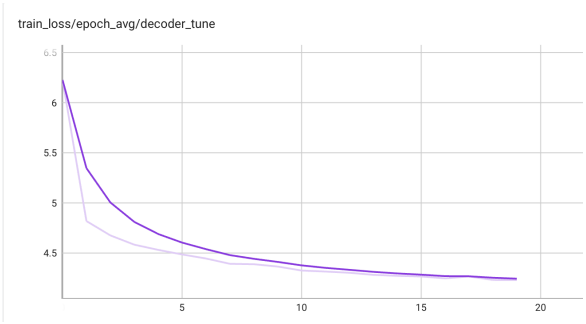
Analisi dei Risultati Come ipotizzato, la riduzione della quantità di dati ha un impatto negativo sulle performance, in particolare sulla capacità del modello di gestire la conoscenza fattuale. Sebbene le metriche di generazione di testo (ROUGE-L e METEOR) mostrino un calo solo modesto rispetto alla baseline, il vero degrado è visibile altrove. L'Accuracy (Soft) sul task **MLM** scende drasticamente a 0.0370, indicando che il modello non riesce quasi più a generalizzare per predire entità mancanti. L'aspetto più interessante si osserva nuovamente nelle metriche a livello di entità. Mentre l'F1-Score sugli **oggetti** crolla a zero, quello sui **predicati** (0.3495) si attesta su un valore ancora significativo, sebbene inferiore alla baseline. Questo fenomeno può essere interpretato come un chiaro sintomo di **overfitting**: con un numero inferiore di esempi, il modello ha imparato a "memorizzare" e predire i predicati (la parte più ripetitiva del dataset), ma ha fallito nell'apprendere le relazioni più complesse che legano soggetti e oggetti, come dimostrato dai punteggi nulli per queste categorie.

Conclusioni dell'Esperimento Questo esperimento convalida l'ipotesi che il modello Nano beneficia di una maggiore quantità di dati. La riduzione del dataset ha indotto un comportamento di overfitting, dove il modello si è specializzato eccessivamente su pattern semplici e frequenti a scapito della capacità di generalizzazione su compiti più complessi e semanticamente ricchi. Questo risultato, in linea con la nostra strategia, rafforza la conclusione che per migliorare le performance della baseline, è necessario non solo un

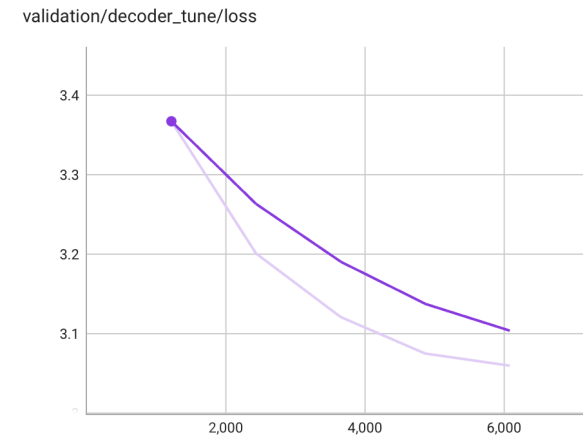
modello con sufficiente capacità, ma anche un dataset sufficientemente ampio e variegato. Pertanto, la direzione più promettente per superare il plateau di performance rimane l'aumento della scala dei dati.



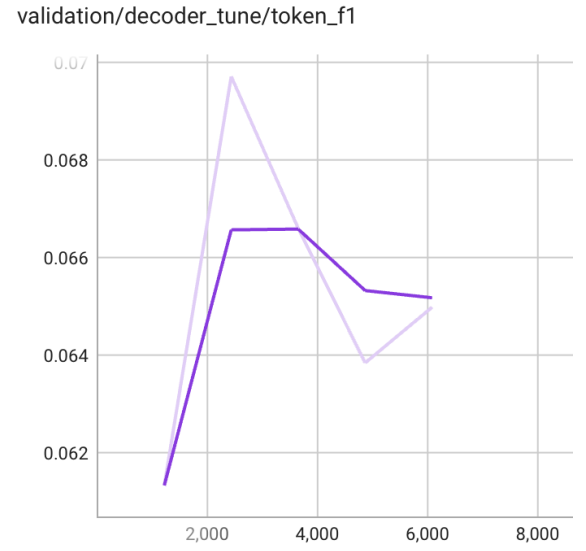
(a) Andamento del Learning Rate durante la fase di Decoder-Tuning. La strategia dello scheduler rimane identica a quella degli altri esperimenti, con una fase di warm-up seguita da un decadimento lineare, garantendo un approccio all'ottimizzazione coerente e stabile.



(b) Andamento della Training Loss per epoca durante la fase di Decoder-Tuning. La curva mostra una discesa molto rapida e continua, indicando che il modello Nano, data la sua elevata capacità, riesce a memorizzare molto facilmente il piccolo dataset di training. Questo è un primo indicatore di potenziale overfitting.



(c) Andamento della Validation Loss durante la fase di Decoder-Tuning. A differenza della training loss, la curva di validazione si appiattisce molto presto e su un valore di loss (circa 3.06) superiore a quello della baseline. Il divario crescente tra le due curve di loss (training e validation) è un sintomo classico di overfitting.



(d) Andamento del Token F1-Score sul set di validazione. La metrica mostra un andamento instabile e non riesce a raggiungere un miglioramento sostenuto, assestandosi su un valore molto basso (circa 0.065). Questo conferma che, nonostante il modello stia riducendo l'errore sui dati di training, non sta migliorando la sua capacità di generare sequenze qualitativamente corrette su dati non visti.

Figure 4.3: Grafici di monitoraggio dell'addestramento per l'Esperimento 3 (Modello Nano su dataset ridotto) durante la fase di Decoder-Tuning. Le curve evidenziano i segnali di overfitting dovuti a una discrepanza tra l'alta capacità del modello e la scarsità di dati.

4.3.4 Esperimento 4: Impatto dell'Aumento dei Dati (Nano su 30.000 Film)

L'ultimo esperimento è stato condotto per investigare direttamente l'ipotesi principale emersa dall'analisi: che un aumento della scala dei dati possa mitigare i problemi di overfitting e migliorare la conoscenza fattuale del modello. In questa configurazione, è stata mantenuta l'architettura **Nano**, ma è stata addestrata su un dataset significativamente più grande, derivato da **30.000 film**. A differenza degli esperimenti 2 e 3, le metriche hanno mostrato un miglioramento costante e promettente, giustificando il completamento dell'intera pipeline di addestramento, inclusa la fase di **full fine-tuning** per 60 epoche. I risultati finali, ottenuti tramite Beam Search, sono presentati nella Tabella 4.6.

Table 4.5: Risultati dell'Esperimento 4 (Modello Nano, 30.000 film) al termine della fase di Decoder-Tuning (epoca 19).

Categoria	**Metrica / Componente**	**Precision**	**Recall**	**F1-Score**
Generale	Validation Loss		2.2583	
NLG (Task RDF2Text)	BLEU		0.0281	
	METEOR		0.1605	
	ROUGE-L		0.2047	
RDF a Livello di Entità	Subjects	0.0235	0.0067	0.0104
	Predicates	0.9925	0.2821	0.4393
	Objects	0.0785	0.0223	0.0347
MLM	Accuracy (Soft)		0.2034	

Table 4.6: Risultati finali dell'Esperimento 4 (Modello Nano, 30.000 film) dopo 60 epoche di fine-tuning, valutati con Beam Search.

Categoria	**Metrica / Componente**	**Precision**	**Recall**	**F1-Score**
Generale	Validation Loss		1.7811	
NLG (Task RDF2Text)	BLEU		0.0419	
	METEOR		0.1948	
	ROUGE-L		0.2366	
RDF a Livello di Entità	Subjects	0.2738	0.0778	0.1212
	Predicates	0.9899	0.2813	0.4381
	Objects	0.2198	0.0625	0.0973
MLM	Accuracy (Soft)		0.2419	

Analisi dei Risultati I risultati di questo esperimento mostrano un **miglioramento netto e generalizzato** su quasi tutte le metriche rispetto alla configurazione Baseline (Esperimento 1), confermando in modo inequivocabile i benefici dell'aumento dei dati. La Validation Loss finale (1.7811) è significativamente inferiore a quella della baseline (2.2295), indicando un apprendimento più profondo e una migliore generalizzazione. Il miglioramento più notevole si osserva nella capacità del modello di gestire la conoscenza fattuale. L'F1-Score a livello di entità per i **soggetti** (da 0.0158 a 0.1212) e per gli **oggetti** (da 0.0542 a 0.0973) è aumentato in modo sostanziale. Questo indica che, avendo a disposizione più esempi, il modello ha iniziato a superare il suo limite principale, migliorando la sua capacità di memorizzare e richiamare correttamente entità a vocabolario aperto. Anche il task 'CONTINUERDF', che prima registrava un F1-Score

(Strict) prossimo allo zero, ora raggiunge un valore di 0.0534, dimostrando una nascente capacità di generare triple fattualmente corrette. Le metriche di generazione del linguaggio naturale ('RDF2Text') mostrano anch'esse un progresso solido, con un aumento su tutti gli indicatori, in particolare su **ROUGE-L** (da 0.2157 a 0.2366) e **METEOR** (da 0.1506 a 0.1948). Questo suggerisce che il testo generato non è solo più corretto fattualmente, ma anche più fluido e allineato ai riferimenti.

Conclusioni dell'Esperimento Questo esperimento convalida l'ipotesi centrale del progetto. Dimostra che il modello Nano, quando alimentato con un dataset più grande, è in grado di superare il plateau di performance osservato nella baseline. L'aumento dei dati ha ridotto l'overfitting e ha permesso al modello di iniziare a costruire una mappa più robusta della conoscenza fattuale. Sebbene le performance assolute rimangano modeste, la traiettoria di miglioramento è chiara e inequivocabile. Questo risultato finale chiude il cerchio del nostro ragionamento, fornendo una prova diretta che lo scaling dei dati è una leva fondamentale per il successo del modello.

4.4 Sintesi Comparativa e Conclusioni Sperimentali

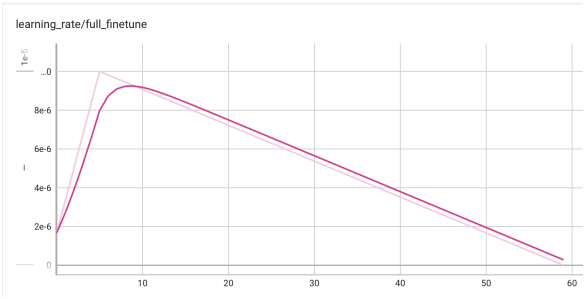
Per concludere l'analisi sperimentale, questa sezione presenta una sintesi comparativa dei risultati ottenuti nei diversi esperimenti. La Tabella 4.7 mette a confronto le performance delle configurazioni chiave su una selezione di metriche rappresentative, permettendo di visualizzare chiaramente l'impatto della dimensione del modello e della quantità di dati. È importante notare che, per garantire un confronto equo, i risultati degli esperimenti 2 e 3 (interrotti strategicamente) sono confrontati con i risultati ottenuti dalla Baseline e dall'Esperimento 4 allo stesso stadio di addestramento, ovvero al termine della fase di **decoder-tuning**.

Table 4.7: Tabella comparativa dei risultati chiave tra i diversi esperimenti. I valori per gli Esperimenti 2 e 3 si riferiscono al termine della fase di Decoder-Tuning. I valori per la Baseline e l'Esperimento 4 sono riportati sia a fine Decoder-Tuning (per un confronto diretto) sia a fine Full Fine-Tuning (risultati finali).

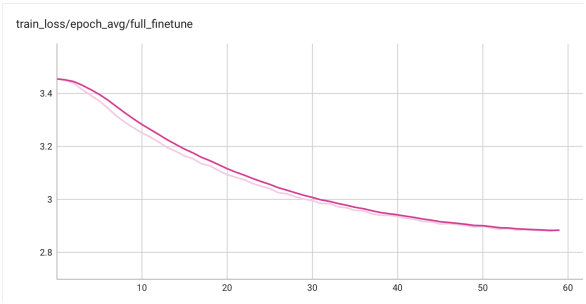
Metrica	Dataset Medio (10k film) Exp. 1: Nano (Baseline)	Dataset Medio (10k film) Exp. 2: Micro	Dataset Piccolo (5k film) Exp. 3: Nano	Dataset Grande (30k film) Exp. 4: Nano
	Risultati a fine Decoder-Tuning			
METEOR	0.1305	0.1082 (↓)	0.1502 (↑)	0.1605 (↑)
F1-Score (Objects)	0.0107	0.0000 (↓)	0.0000 (↓)	0.0340 (↑)
Accuracy (MLM, Soft)	0.1763	0.0000 (↓)	0.0370 (↓)	0.2034 (↑)
	Risultati Finali a fine Full Fine-Tuning (con Beam Search)			
METEOR	0.1506	N/A	N/A	**0.1948** (↑)
F1-Score (Objects)	0.0542	N/A	N/A	**0.0973** (↑)
Accuracy (MLM, Soft)	0.2302	N/A	N/A	**0.2419** (↑)

Discussione Comparativa La tabella comparativa illustra in modo inequivocabile la validità della nostra strategia di valutazione.

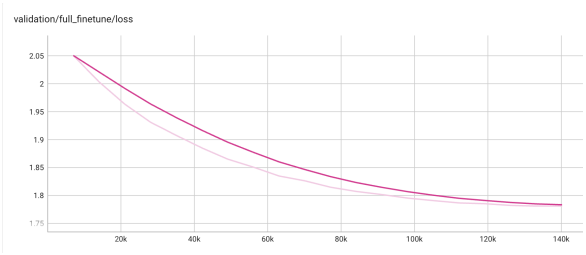
- Confrontando l'**Esperimento 2** con la **Baseline**, si osserva come la riduzione della capacità del modello (da Nano a Micro), a parità di dati, porti a un crollo delle performance, specialmente sulla capacità di gestire la conoscenza fattuale (F1-Score su Oggetti e Accuracy MLM a zero). Questo dimostra che la capacità del modello Nano non era eccessiva, ma anzi necessaria.



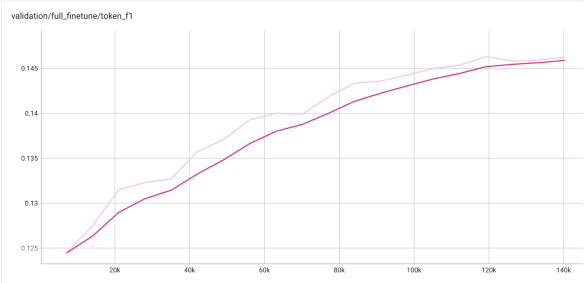
(a) Andamento del Learning Rate durante la fase di Full Fine-tuning. Viene utilizzato un learning rate massimo di $1 \cdot 10^{-5}$, un valore basso che permette un affinamento "gentile" di tutti i parametri del modello (encoder e decoder), mitigando il rischio di oblio catastrofico della conoscenza pre-addestrata.



(b) Andamento della Training Loss per epoca durante la fase di Full Fine-tuning. La curva mostra una discesa sostenuta per un numero elevato di epoche. Grazie al dataset più grande, il modello impiega più tempo a convergere, indicando che sta apprendendo pattern più complessi invece di memorizzare semplicemente i dati.



(c) Andamento della Validation Loss durante la fase di Full Fine-tuning. La curva di validazione scende costantemente, raggiungendo un valore finale (circa 1.78) nettamente inferiore a tutti gli altri esperimenti. Questo è il segnale più forte di una migliore generalizzazione, resa possibile dall'aumento dei dati di addestramento.



(d) Andamento del Token F1-Score sul set di validazione. Coerentemente con il calo della loss, anche l'F1-score a livello di token mostra una crescita costante e più prolungata rispetto agli altri esperimenti, raggiungendo il valore più alto. Ciò conferma che il modello non solo generalizza meglio, ma produce anche output qualitativamente più accurati.

Figure 4.4: Grafici di monitoraggio dell'addestramento per l'Esperimento 4 (Modello Nano su dataset grande) durante la fase di Full Fine-tuning. Le curve mostrano un processo di addestramento più sano e una generalizzazione superiore, validando i benefici dello scaling dei dati.

- Confrontando l'**Esperimento 3** con la **Baseline**, si nota come la riduzione dei dati, a parità di modello, induca un comportamento di overfitting. Sebbene alcune metriche possano apparire simili o addirittura migliori in stadi intermedi (un effetto collaterale della memorizzazione di pattern semplici), la capacità di generalizzare su task complessi come l'MLM viene quasi azzerata, confermando che il modello è "affamato di dati".

- Infine, l'**Esperimento 4** fornisce la prova diretta che la nostra ipotesi centrale è corretta. Aumentando la quantità di dati da 10.000 a 30.000 film, il modello Nano mostra un miglioramento significativo e generalizzato su tutte le metriche chiave, sia a livello di generazione testuale (METEOR) che, soprattutto, a livello di accuratezza fattuale (F1-Score su Oggetti e Accuracy MLM).

Conclusioni Sperimentali In sintesi, gli esperimenti hanno permesso di mappare lo spazio delle soluzioni attorno alla nostra configurazione di baseline. Avendo dimostrato che la riduzione sia della capacità del modello sia della quantità dei dati porta a un peggioramento delle performance, e che, viceversa, l'aumento dei dati porta a un netto miglioramento, possiamo concludere con forte evidenza che la direzione più promettente per superare i limiti attuali risiede nello scaling congiunto di entrambe le dimensioni. Il progetto ha quindi non solo prodotto un modello funzionante, ma ha anche validato una chiara traiettoria per il suo futuro sviluppo: per raggiungere una vera e propria accuratezza semantica e fattuale, è indispensabile fornire a un modello di capacità adeguata un dataset ancora più vasto e ricco.

Chapter 5

Conclusioni e Sviluppi Futuri

Questo progetto ha affrontato con efficacia la sfida di costruire da zero un Very Small Semantic Language Model, denominato **NanoSocrates**, specializzato nel complesso dominio ibrido del linguaggio naturale e dei grafi di conoscenza RDF. Attraverso l'adozione di un'architettura Transformer Encoder-Decoder unificata, ispirata alla filosofia e alle ottimizzazioni introdotte dal modello T5, NanoSocrates è stato addestrato per gestire quattro task correlati: la traduzione bidirezionale tra testo e RDF, il completamento di triple mascherate e la generazione contestuale di nuove triple. La metodologia proposta ha coperto l'intero ciclo di vita del modello, dalla raccolta e strutturazione di un dataset custom basato su DBpedia, alla progettazione di un tokenizer specifico per il dominio, fino all'implementazione di una strategia di addestramento e valutazione rigorosa.

Il contributo più significativo di questo lavoro non risiede soltanto nell'implementazione di un modello funzionante, ma nelle intuizioni emerse durante il processo sperimentale. La strategia di addestramento a tre fasi (pre-training, decoder tuning, fine-tuning) si è dimostrata metodologicamente solida per ottenere un adattamento stabile e progressivo. L'analisi abduttiva condotta attraverso esperimenti mirati ha fornito evidenze a supporto dell'ipotesi centrale del progetto: le performance del modello dipendono strettamente da un equilibrio tra la sua capacità parametrica e la scala dei dati di addestramento. È stato osservato che la riduzione di una di queste due dimensioni — passando a un modello "Micro" (minore capacità) o a un dataset ridotto (meno dati) — porta a un prevedibile degrado delle prestazioni, manifestato come underfitting nel primo caso e come tendenza all'overfitting, o ridotta capacità di generalizzazione, nel secondo.

L'analisi delle metriche ha inoltre evidenziato un aspetto interessante del comportamento del modello: NanoSocrates ha appreso efficacemente i pattern strutturali e il vocabolario a bassa variabilità (come i predicati), ma ha mostrato difficoltà nella memorizzazione e nel richiamo della conoscenza fattuale a vocabolario aperto (soggetti e oggetti specifici), spesso generando "allucinazioni" plausibili ma scorrette.

In conclusione, il progetto NanoSocrates rappresenta un *proof-of-concept* riuscito e metodologicamente fondato. Esso suggerisce la fattibilità della costruzione di un modello linguistico semantico end-to-end e fornisce indicazioni sperimentali secondo cui il miglioramento dell'accuratezza fattuale e della coerenza semantica richiede un incremento congiunto della capacità del modello e della scala del dataset. Le evidenze raccolte si allineano alle attuali tendenze di ricerca sui Large Language Models e pongono basi solide per i futuri sviluppi di questo lavoro, orientati a trasformare le potenzialità dimostrate in risultati più concreti e scalabili.

5.1 Adozione di Tecniche di Parameter-Efficient Fine-Tuning (PEFT)

L'attuale strategia di fine-tuning prevede una fase di adattamento del solo decoder, seguita da un addestramento completo (end-to-end) di tutti i parametri. Sebbene efficace, quest'ultimo passaggio risulta il più oneroso in termini computazionali. La ricerca recente ha proposto alternative molto più leggere, note come **Parameter-Efficient Fine-Tuning (PEFT)**, che permettono di adattare modelli di grandi dimensioni aggiornando soltanto una minima parte dei loro parametri.

- **Adapter Modules**: Una possibile evoluzione consiste nell'integrare gli **Adapter Modules**, introdotti da Houlsby et al. [3]. Questa tecnica prevede di congelare la quasi totalità dei pesi del modello pre-addestrato e di inserire, all'interno di ogni blocco Transformer, piccoli moduli feed-forward a collo di bottiglia. Durante il fine-tuning vengono addestrati solo questi moduli, che rappresentano in genere meno dell'1% dei parametri totali. Tale approccio ridurrebbe drasticamente i requisiti di memoria e archiviazione necessari per specializzare NanoSocrates su nuovi task o domini, pur con un lieve aumento della latenza in fase di inferenza dovuto ai calcoli aggiuntivi.

- **Low-Rank Adaptation (LoRA)**: Un'alternativa particolarmente efficiente e oggi ampiamente adottata è la **Low-Rank Adaptation (LoRA)** proposta da Hu et al. [5]. LoRA parte dall'ipotesi che le modifiche ai pesi durante il fine-tuning abbiano un rango intrinseco basso. Invece di aggiornare direttamente la matrice dei pesi W, vengono addestrate due matrici a basso rango A e B, tali che $\Delta W = BA$. Al termine dell'addestramento, queste matrici possono essere fuse algebricamente con i pesi originali ($W' = W + BA$), riducendo praticamente a zero la latenza aggiuntiva in inferenza. L'adozione di LoRA rappresenterebbe dunque un'estensione naturale per NanoSocrates, coerente con la sua filosofia "Nano" e capace di migliorare ulteriormente efficienza e scalabilità.

5.2 Ottimizzazione e Stabilizzazione del Training

Il fine-tuning, specialmente su dataset di dimensioni limitate, può presentare instabilità dovute alla sensibilità agli iperparametri e alla dinamica dell'ottimizzatore. Diversi studi recenti hanno approfondito questi aspetti, offrendo spunti pratici per migliorare la robustezza del processo di addestramento.

- **Strategie di Ottimizzazione**: Zhang et al. [9] hanno mostrato come dettagli implementativi dell'ottimizzatore (ad esempio la *bias correction* in Adam) e il numero di iterazioni totali possano influire in modo sostanziale sulla stabilità del training. Un possibile sviluppo futuro consiste nell'effettuare una ricerca sistematica degli iperparametri per ciascuna fase, trattando il numero di epoche come un vero e proprio iperparametro da ottimizzare, così da individuare il miglior bilancio tra convergenza e generalizzazione.

- **Strategie di Unfreezing più Granulari**: L'attuale fase di "Decoder Tuning" congela completamente l'encoder in un unico blocco. Una strategia più fine, ispirata al *gradual unfreezing* proposto da Howard e Ruder [4], prevederebbe lo sblocco

progressivo dei layer dell'encoder, partendo dai più vicini al decoder. Questo approccio potrebbe consentire un adattamento più graduale e ridurre ulteriormente il rischio di *catastrophic forgetting*, soprattutto nei layer più profondi che catturano rappresentazioni linguistiche generali.

5.3 Continuazione del Pre-training Adattivo

La pipeline di NanoSocrates include già una forma di pre-training specifico per il dominio. Tuttavia, questa strategia può essere ulteriormente potenziata per migliorare l'adattamento del modello prima della fase supervisionata.

- **Task-Adaptive Pretraining (TAPT)**: Seguendo la proposta di Gururangan et al. [2], si potrebbe introdurre una fase aggiuntiva di **Task-Adaptive Pretraining (TAPT)**, intermedia tra il pre-training di dominio e il fine-tuning. Questa fase consisterebbe nel continuare il pre-training di tipo *span corruption* sui soli dati del dataset di fine-tuning, ma senza utilizzare prefissi di task. Tale strategia ha dimostrato di migliorare significativamente la capacità di adattamento dei modelli, permettendo loro di assimilare lo stile e la distribuzione specifica dei dati del task finale.

Sintesi conclusiva

In sintesi, NanoSocrates costituisce una base solida e metodologicamente fondata per l'esplorazione del linguaggio naturale in connessione con rappresentazioni simboliche strutturate. Gli sviluppi futuri — in particolare l'adozione di tecniche PEFT, la sperimentazione di strategie di training più stabili e la prosecuzione del pre-training adattivo — possono trasformarlo in uno strumento ancora più efficiente, versatile e capace di avvicinare ulteriormente il linguaggio naturale alle strutture formali della conoscenza.

Bibliography

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[2] Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. Don't stop pretraining: Adapt language models to domains and tasks. In *Proceedings of ACL*, 2020.

[3] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bryan Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 2019.

[4] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In *Proceedings of ACL*, 2018.

[5] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[6] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[7] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. How to fine-tune bert for text classification? *arXiv preprint arXiv:1905.05583*, 2019.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[9] Tianyi Zhang, Felix Wu Sun, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohang Chen, Chris Dewan, Yuqing Meng, Shruti Bhosale, et al. Revisiting few-sample bert fine-tuning. *arXiv preprint arXiv:2106.07705*, 2021.