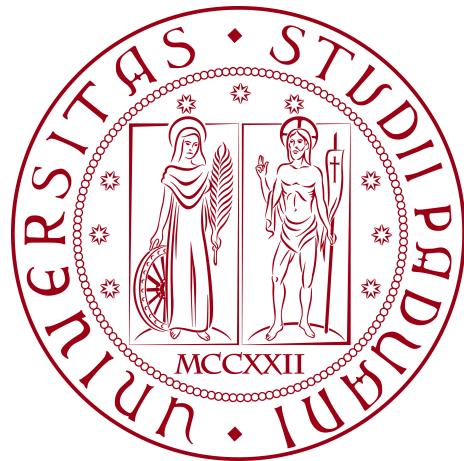


Università degli Studi di Padova
DIPARTIMENTO DI MATEMATICA “TULLIO LEVI-CIVITA”
CORSO DI LAUREA IN INFORMATICA



Test Automatici con Large Language Model

Tesi di Laurea Triennale

Relatore

Prof. Ballan Lamberto

Laureando

Dugo Alberto

Matricola 2042382

Ringraziamenti

Padova, Luglio 2024

Dugo Alberto

Abstract

Il presente documento illustra l'attività di stage svolta dal laureando Alberto Dugo presso l'azienda Zucchetti Spa.

Durante il periodo di stage, della durata di 320 ore, vi è stata l'opportunità di approfondire le conoscenze in ambito **machine learning**_G. In particolare era richiesto lo studio e l'implementazione di test automatici derivanti direttamente dal codice, sfruttando le abilità dei sistemi di intelligenza artificiale ed in particolare del **Large Language Model (LLM)**_G. Vi è stato inoltre la possibilità di fare **fine-tuning**_G dei modelli **LLM** attraverso il metodo **Low-Rank Adaptation (LoRA)**_G e quantizzare i modelli stessi in modo da renderli più efficienti.

Indice

Elenco delle figure

Elenco delle tavelle

Capitolo 1

Introduzione

1.1 L'azienda



Figura 1.1: logo di Zucchetti

Zucchetti S.p.a. è la prima software house in Italia per fatturato, opera nel settore dell'*Information Technology*, ed è stata fondata nel 1978 da Fabrizio Bernini a Lodi. L'azienda è specializzata nella realizzazione di software gestionali per pianificazione delle risorse d'impresa, soluzioni per il controllo degli accessi e sistemi di automazione industriale. Al giorno d'oggi Zucchetti oltre ad avere sedi in tutta Italia è presente anche in 50 paesi esteri, tra cui Cina, Germania, USA e Svizzera e conta più di 8.000 dipendenti e più di 1650 *partner*.

1.2 Il progetto

Lo *stage* si è svolto presso l'azienda Zucchetti, con sede a Padova. Il progetto di *stage* ha previsto la ricerca e lo sviluppo di test automatici derivanti direttamente dal codice e dalla documentazione, sfruttando le abilità dei sistemi di *intelligenza artificiale* ed in particolare dei **Large Language Models**. Lo *stage* è stato diviso in due macroperiodi di quattro settimane ciascuno. Nel primo periodo ho analizzato, attraverso lo studio di paper accademici e documentazioni, le tecniche di *testing* che sfruttano **Large Language Models**. Successivamente ho implementato un **prototipo** di generatore di test automatici in *Python* che usufruisce di un modello basato su *natural language processing*. Nella seconda parte del primo periodo ho decorato il codice attraverso commenti e ho generato *test*. I risultati di questi sono stati poi confrontati con quelli ottenuti dalla prima parte di periodo. Nel secondo periodo invece

ho eseguito **fine-tuning** dei modelli **LLMs** con il metodo **LoRA**, per poi confrontare i risultati ottenuti con quelli nel primo periodo. È stato inoltre richiesto lo studio di tecniche di *quantizzazione* per ridurre la dimensione dei modelli **LLMs** e la loro complessità computazionale.

1.3 Organizzazione del testo

Il secondo capitolo descrive i processi e le metodologie utilizzate durante lo *stage*.

In particolare si approfondiranno i processi di sviluppo *software* e gli strumenti utilizzati per fare ciò.

Il terzo capitolo si propone di delineare il dominio applicativo del progetto, mediante un'analisi dettagliata del tema accompagnata da esempi pratici di utilizzo. Inoltre, si provvederà a fornire una descrizione esaustiva del funzionamento di *Assured LLMS*. In questa sezione, sarà altresì redatta una lista esaustiva di rischi, requisiti e obiettivi del progetto. Successivamente, si procederà con la descrizione del prodotto sviluppato, che includerà *script* e *benchmarks*.

Il quarto capitolo descrive il processo di applicazione di **LoRA** e le possibili ottimizzazioni, andando ad approfondire le tecniche utilizzate durante il processo.

Il quinto capitolo concluderà il documento, presentando una valutazione retrospettiva personale dell'esperienza di *stage* e delle conoscenze acquisite.

In seguito si possono trovare le convenzioni tipografiche utilizzate per la stesura del documento:

- gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
- per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente nomenclatura: *parola_G*;
- i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

Capitolo 2

Processi e metodologie

2.1 Processo di sviluppo del prodotto

Il processo di sviluppo del prodotto è iniziato con la ricerca di articoli accademici inerenti agli argomenti sui quali si sarebbe dovuto basare il prodotto. Successivamente si è proceduto con la lettura e la comprensione degli stessi, integrando le conoscenze attraverso libri di *machine learning* e *deep learning*. Durante le ore lavorative vi è stata inoltre la possibilità di accrescere le mie conoscenze teoriche e pratiche grazie all'aiuto dei colleghi i quali, sin da subito, hanno mostrato interesse nell'argomento. Nell'ufficio di ricerca e sviluppo di Zucchetti, ove ho svolto lo stage, si lavora in un ambiente rilassato ma allo stesso tempo incentrato a portare valore al prodotto, questo mi ha spinto a lavorare in modo creativo e costantemente alla ricerca di accrescere le mie conoscenze. Oltre a ciò, vi è stata la possibilità di constatare le conoscenze apprese attraverso esposizioni durante le riunioni interne all'azienda. Questi *meeting* mi hanno quindi permesso di verificare la comprensione dell'argomento e di ricevere feedback da parte dei colleghi.

Durante la fase di sviluppo invece il lavoro è stato svolto in autonomia, con la possibilità di confrontarmi con i colleghi in caso di dubbi o problemi. Mi sono inoltre confrontato con gli altri stagisti che lavoravano su progetti simili, per fare *brainstorming* e per discutere delle soluzioni da noi adottate alla ricerca di idee e soluzioni a difficoltà comuni. Infine, il prodotto è stato testato autonomamente e successivamente con i colleghi per valutare la qualità del prodotto e ricevere feedback per eventuali miglioramenti.

2.2 Strumenti utilizzati

gli strumenti utilizzati per la realizzazione del progetto sono stati:

- **Git**: sistema di controllo di versione distribuito;
- **GitHub**: servizio di hosting per progetti software che utilizzano Git;

- **Google Colab Pro:** servizio di Google che permette di eseguire codice Python in cloud;
- **Hugging Face:** libreria Python che fornisce modelli di Machine Learning pre-addestrati;
- **LM Studio:** software che permette di scaricare ed utilizzare localmente alcuni dei modelli di Hugging Face;
- **PyCharm:** Integrated Development Environment (IDE)_G per lo sviluppo in Python;
- **Python:** linguaggio di programmazione ad alto livello, interpretato, interattivo, orientato agli oggetti, adatto per lo sviluppo di applicazioni legate al machine learning;

Capitolo 3

Test automatici generati da LLM

3.1 Analisi del dominio applicativo

Durante la fase di analisi del dominio applicativo si è proceduto a delineare il contesto in cui il progetto si colloca, analizzando il tema e fornendo esempi pratici di utilizzo. Inoltre, si è provveduto a fornire una descrizione esaustiva del funzionamento di Assured LLM-Based Software Engineering (LLMSE)_G, tipologia di Large Language Model che sta alla base del funzionamento del prodotto.

3.1.1 Analisi del tema

Il tema del progetto riguarda la realizzazione di *test* automatici per il codice sorgente, ponendosi come obiettivo la semplificazione del processo di *testing* affidato ai programmatore attraverso l'utilizzo di Large Language Models. In particolare, il progetto consiste in uno *script* con il quale vengono estratti i metodi e le classi da testare cercando le relazioni tra di essi, in seguito verrà spigato dettagliatamente come è stato eseguito questo passaggio. Lo *script* in questione quindi utilizza predizioni dette dal Large Language Model per generare *test* automatici. Quest'ultimi sono stati successivamente eseguiti e i risultati riportati in grafici per una migliore comprensione.

3.1.2 Esempi di utilizzo

Negli ultimi anni si è registrato un aumento significativo nell'adozione dei Large Language Models per la generazione di testo, con una crescita esponenziale soprattutto nel settore tecnologico. In questa sezione esamineremo il loro utilizzo in ambito di *testing* del codice sorgente. In particolare è noto che l'utilizzo di Large Language Models per la generazione di *test* automatici è in grado di ridurre i tempi di *testing* del codice sorgente e migliorare la qualità del codice stesso[0]. La capacità di generare test aumenta inoltre la verificabilità di non regressione del codice, in questo momento però non si è in grado di verificare la presenza di *bug* nel codice senza l'aiuto del programmatore, è chiaro quindi che gli LLMs in questo periodo storico riescano a fornire solamente un supporto ai programmatore anzichè sostituirli.

3.1.3 Assured LLMSE

Recentemente nell'ambito del Software Engineering si è assistito ad una crescita elevata nell'utilizzo dell'intelligenza artificiale volto alla ricerca di agevolare il programmatore durante lo sviluppo di software. Proprio in quest'ambito ci riferiamo a **LLMSE** per descrivere una qualsiasi tipo di applicazione nella quale il prodotto o i processi software si basano sull'utilizzo di **LLM**[0]. Alla base di questo progetto troviamo quindi un massiccio uso degli **LLMs** ed in particolare il progetto in sè vuole fornire uno strumento ai programmatori, il quale agevola la scrittura dei *test* attraverso l'utilizzo di **LLM**.

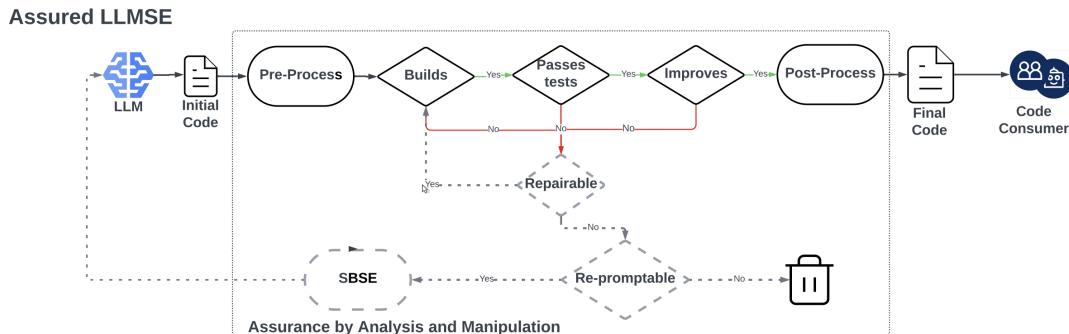


Figura 3.1: Architettura LLMSE[0]

Lo scopo degli **LLMSE** è quello di applicare una serie di filtri semantici al codice generato in modo tale da poter fornire delle garanzie, come ad esempio l'assenza di allucinazioni. Come infatti è visibile in figura 3.1 dopo la generazione della risposta da parte del **Large Language Model** questa viene pre-processata, quindi si eliminano i vari commenti non soggetti a *test*, e si estrae solamente il codice. Dopodichè vengono applicati svariati filtri, tra cui la capacità di essere nello stato di *build*, di essere in stato di accettazione, quindi che l'asserzione di esso sia *true* ed inoltre che sia in grado di aumentare la *code coverage*. Possiamo notare che se anche solo uno di questi filtri condizionali non desse risultato positivo l'intero *test* affronterebbe altri filtri condizionali i quali potrebbero portare alla sua eliminazione. Le condizionalità dovute ad un possibile fallimento di una condizione sono esplicitate nella procedura in seguito:

```
def filtering(self, facadeFilter):
    if test.repairable():
        test.repair()
        return facadeFilter.filters(test)
    elif test.re_prompable():
        prompt = test.re_prompt()
        return self.ask(prompt)
    else:
        return test.discard()
```

Nel flusso procedurale, un test scartato viene immediatamente sottoposto al filtro "*Repairable*". In questo scenario, se è possibile riparare il codice rapidamente senza

dover riformulare l'intero prompt, allora viene effettuata la riparazione e il test riprende il processo di filtraggio. Altrimenti, il test procede con i filtri successivi. Nel caso in cui il *Re-prompt* sia possibile, permettendo così di ottimizzare il prompt mediante una riformulazione, il *test* può avanzare alla fase successiva, altrimenti, viene eliminato definitivamente. Il *re-prompting* avviene attraverso *Search-based software engineering (SBSE)* che è una tipologia di ottimizzazione del prompt la quale si basa su un algoritmi genetici, la rigenerazione del prompt permette quindi di ricominciare l'interno processo. I filtri in questione, nel caso in cui il *test* non fosse approvato, sono filtri opzionali, vedremo in seguito che questi vengono omessi durante il processo di sviluppo per ovviare ai costi onerosi derivati. Il codice quindi che supera tutti i filtri è un codice che soddisfa i requisiti e può essere passato ad un consumatore, il quale potrebbe essere ad esempio un umano o un altro tool.

3.1.3.1 *Offline* ed *Online* LLMSE

È importante distinguere *Online* e *Offline* LLMSE, in quanto il primo necessita della risposta dell'**LLM** in *real time*, mentre il secondo non pone vincoli temporali. Nel contesto dell'*Online LLMSE*, facciamo riferimento, ad esempio, alle applicazioni di completamento automatico del codice, come *CoPilot*. Qui, la tempestività della risposta del modello è fondamentale per l'esperienza utente. D'altra parte, l'*Offline LLMSE* si riferisce a processi in cui non è necessaria una risposta immediata. Nei casi di *Offline LLMSE*, se il tempo di generazione della risposta dovesse variare, ciò non influirebbe sul funzionamento dell'applicazione stessa. Nel caso del mio progetto, andrò ad utilizzare *Offline LLMSE* poichè quest'ultimi permettono di poter applicare i filtri che abbiamo precedentemente descritto senza dover preoccuparsi del tempo di risposta.

3.1.3.2 Future applicazioni e miglioramenti

Per quanto riguarda le future applicazioni, il miglioramento dei filtri è sicuramente uno dei maggiori potenziali di sviluppo, in questo modo si riuscirebbe ad ottenere risultati migliori e più adatti alle esigenze. Non solo il miglioramento dei filtri, ma anche l'utilizzo di *Genetic Improvement* per migliorare il *prompt* e *Methauristic algorithm* per migliorare le soluzioni candidate potrebbero portare a risultati migliori. Il miglioramento del *prompt* è un'area di ricerca molto promettente, in quanto un *prompt* ben formulato è in grado di guidare l'**LLM** verso la generazione di test più accurati e corretti. *In-learning context* è un'altra area di ricerca che potrebbe portare a risultati significativi. In questo contesto, l'**LLM** è in grado di apprendere dai risultati ottenuti e di migliorare le proprie prestazioni nel tempo. Possiamo quindi ipotizzare che ponendo maggiore attenzione a formulare il prompt in modo più accurato, sfruttando algoritmi genetici e applicando le tecniche di *prompt engineering*,

si possa ottenere un **LLM** più performante e in grado di generare test più accurati e corretti.

3.2 Analisi dei requisiti

3.2.1 Analisi preventiva dei rischi

Durante la fase di analisi dei rischi sono stati individuate le possibili criticità che potranno essere riscontrate. Si è quindi proceduto a elaborare delle possibili soluzioni per far fronte a tali rischi.

1. Mancanza di materiale informativo

Descrizione: Trattandosi di una novità nel settore e in fase di crescita, la possibile assenza di materiale informativo relativo all'argomento stesso potrebbe rallentare il processo di apprendimento.

Soluzione: coinvolgimento del responsabile a capo del progetto relativo.

3.2.2 Requisiti e obiettivi

	Obiettivo	Descrizione
OB 1		Realizzazione di <i>smoke test</i> in Python generati da codice reale.
OB 2		Realizzazione di decorazioni assert per funzioni.
OB 3		Realizzazione di <i>test</i> a partire da codice commentato.

Tabella 3.1: Requisiti primo macroperiodo.

3.3 Sviluppo del prodotto

A seguito dello studio del dominio e delle opportunità vi è stata la progettazione e lo sviluppo del prodotto. Questo capitolo si propone di offrire un'esauriente panoramica sul processo di sviluppo del prodotto, esaminando in dettaglio le tappe fondamentali che ho affrontato. In particolare, ci concentreremo sull'analisi dello *script* realizzato per l'estrazione e la generazione dei *test*, una fase cruciale che ha richiesto un'accurata progettazione e implementazione. Saranno descritti poi i risultati ottenuti attraverso un'analisi dettagliata, evidenziando le sfide superate e i successi raggiunti nel corso del processo di sviluppo.

3.3.1 *Script*

Lo *script* generato permette di fare il parsing di un intero progetto salvando dati chiave all'interno di un *database SQLite*. Questo procedimento permette all'**LLM** di riuscire a trovare le relazioni all'interno dei dati. Dopo il processo di *parsing* è possibile generare i *test* attraverso l'**LLM**. Utilizzando infatti i comandi `-genTestClass` e `-genTestMethod` è possibile generare i *test* per una classe o per un metodo specifico. Il seguente *script* può quindi essere azionato attraverso linea di comando utilizzando vari comandi:

- `python3 -parseProj nameProject` : si farà solamente il parsing di tutti i file all'interno del progetto.
- `python3 -genTestClass Class_Name Method_Name`: attraverso questo comando è possibile generare i *test* per un particolare metodo all'interno della classe specificata.
- `python3 -genTestMethod Class_Name`: attraverso questo comando è possibile generare i *test* per la singola classe.

In figura 3.2 viene descritto come sono stati realizzati.

```
argument_parser = argparse.ArgumentParser(description='A project which you want to parse and generate tests for.')
argument_parser.add_argument(*name_or_flags: '--parseProj', type=str, default=None, metavar=('[Path_to_Project]'),
                           help='Insert the name of the project which you want to parse and generate tests for.')

argument_parser.add_argument(*name_or_flags: '--genTestMethod', type=str, default=None, metavar=(('Class_Name', 'Method_Name')),
                           help='When database for projects was already created, test generation can be run in isolation '
                                '(no parsing to json files or database generation).'
                                'Insert the name of the method and the class which you want to generate tests for.')

argument_parser.add_argument(*name_or_flags: '--genTestClass', type=str, default=None, metavar=(('Class_Name',),
                           help='When database for projects was already created, test generation can be run in isolation '
                                '(no parsing to json files or database generation).'
                                'Insert the name of the class which you want to generate tests for.')
```

Figura 3.2: Comandi *script*

3.3.1.1 Parsing del linguaggio

Inizialmente, lo *script* richiede di eseguire l'analisi del progetto, focalizzandosi principalmente sull'estrazione di ogni file con estensione *.py* e sulla categorizzazione di ciascuna istruzione all'interno di un nodo dell'albero di parsing. Tale processo è indispensabile poiché altrimenti sarebbe impossibile delineare le relazioni esistenti tra i vari metodi e le classi. Una volta estratte e categorizzate tutte le istruzioni all'interno dei nodi dell'albero, vengono recuperate le firme dei metodi e delle classi. Queste informazioni vengono quindi archiviate nel database SQLite insieme ai *related method*, ovvero i metodi chiamati all'interno di altri metodi. Tale approccio ci consente di identificare le relazioni intrinseche presenti nel progetto e di ottenere risultati più accurati. L'idea di base può essere confermata attraverso i dati in figura 3.3. Il

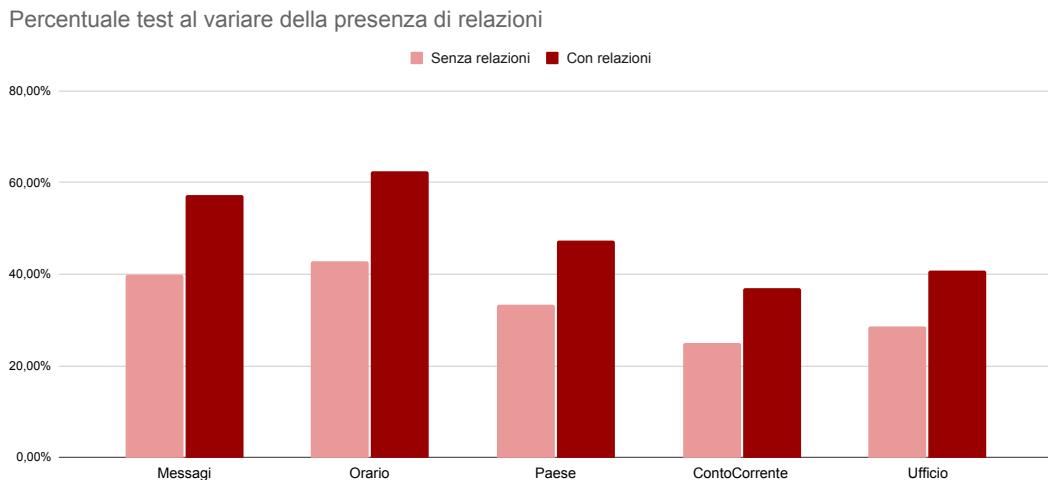


Figura 3.3: Apporto valoriale delle relazioni tra classi nei test

grafico illustra la differenza tra i *test* generati senza inserire nel *prompt* le classi con relazioni a quella da testare e quelli con l'aggiunta di classi inserite nel *prompt*. È chiaramente visibile che la quantità di *test* generati nel secondo caso è maggiore ma soprattutto il numero di *test* che passano è notevolmente più alto. In particolare la generazione di *test* includendo anche le classi in relazione ha generato mediamente il 15% in più di *test* corretti. Un quesito però che ricorre frequentemente nello sviluppo dei **LLM** è la *large context window*, in particolare l'aggiunta dei *related_method* e delle *related_class* potrebbe portare ad alcuni problemi, tra cui la moltitudine di dati da processare e l'*information overload*. In particolare l'*information overload* potrebbe portare a un maggior *focus* sugli *edges* del *context* se questo fosse troppo ampio, e ciò porterebbe a perdere il *focus* sulle parti più importanti della domanda. È stato quindi importante affrontare questa sfida durante lo sviluppo per capire la quantità di metodi e classi relazionate alla classe della quali si voleva andare a generare *test*. La seconda problematica riscontrata durante il parsing riguarda la tipizzazione dei linguaggi, in particolare l'analisi sintattica è stata senza dubbio un'attività dispen-

diosa in termini di tempo ed energia, poichè il linguaggio di programmazione Python, essendo non tipizzato, non presenta una distinzione chiara tra le istruzioni. In particolare, l’istanziazione di un oggetto è trattata come un’assegnazione, in mancanza di una parola chiave specifica. Pertanto, è plausibile ipotizzare che l’analisi sintattica in Java, un linguaggio tipizzato con restrizioni più rigorose, sia più agevole e conduca a risultati più accurati.

3.3.1.2 Generazione dei *test*

Una volta completato il parsing del progetto, lo *script* può essere avviato attraverso i comandi sopra citati. In particolare, il comando `python3 -genTestClass Class_Name` permette di generare i *test* per una particolare classe all’interno del nostro progetto. Quando si esegue questo comando, lo *script* recupera la classe scelta all’interno del database e genera i test di unità per i metodi della classe stessa. Il secondo comando `python3 -genTestMethod Class_Name Method_Name` invece consente di generare i *test* per un metodo specifico all’interno della classe. Come nel caso precedente, lo *script* recupera la classe e il metodo scelti all’interno del database e genera i *test* di unità per il metodo selezionato. La generazione dei *test* può essere effettuata attraverso l’utilizzo della *Hugging Face Inference API* o tramite un *server* locale. L’impiego di *Hugging Face* consente l’utilizzo di modelli di dimensioni considerevoli, come ad esempio *Llama3 70b*, i quali sono in grado di produrre test più accurati e corretti. Tuttavia, ciò comporta un rallentamento della velocità di inferenza e ad un aumento dei costi. Dall’altro lato, l’utilizzo di un *server* locale offre una maggiore velocità di inferenza, ma con l’impiego di modelli di dimensioni ridotte, come ad esempio *Chat Qwen 1.5 1b q4*.

3.3.2 Benchmarks

Dopo lo sviluppo dello *script*, vi è stato un periodo di *test* il quale è stato fondamentale principalmente per due scopi. La prima motivazione è legata al miglioramento del *prompt* per migliorare i risultati. Infatti, andando ad ottimizzare il prompt, in modo tale da poter produrre domande più specifiche e maggiormente comprensibili all’**LLM** si possono ottentere *test* più accurati e corretti. La seconda motivazione riguarda il confronto tra i vari **LLM** utilizzati. Volevo capire se l’utilizzo di **LLM** addestrati su codice sorgente fosse più efficace rispetto a quelli addestrati su testo generico, oltre a ciò volevo capire quale fosse l’**LLM** più adatto per il mio progetto. Ricordando che uno degli scopi principali del progetto è la ricerca dell**LLM** più adeguato alle esigenze e alle possibilità di Zucchetti, inizialmente ho proceduto alla ricerca della temperatura adeguata affinché il modello riuscisse a produrre risultati attendibili evitando l’utilizzo di **LLM** molto pesanti. I dati ottenuti sono raffigurati nella figura 3.4 sottostante.

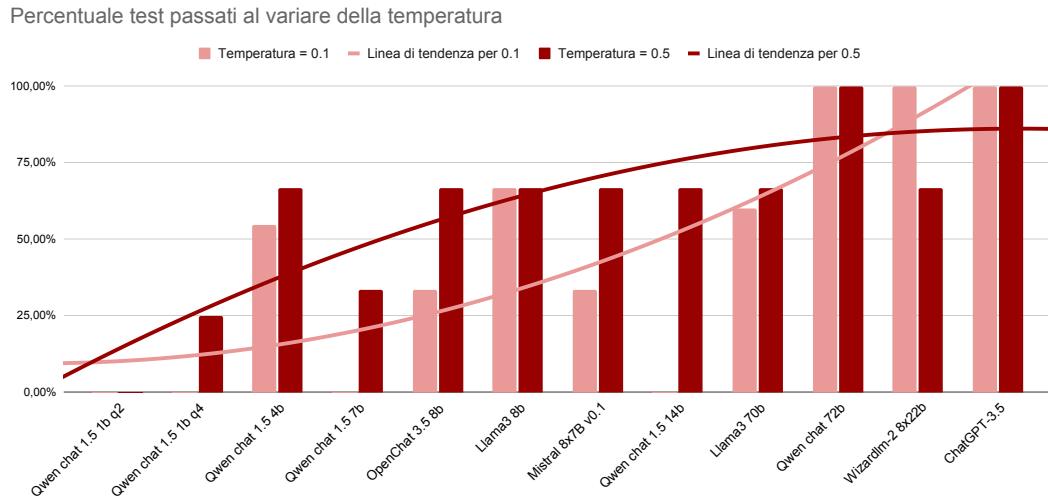


Figura 3.4: Test generati attraverso temperature diverse

È noto che aumentando la temperatura, un **Large Language Model** produce dati meno deterministici e offre la possibilità di ottenere risultati più diversificati. Questo è particolarmente evidente nel caso di modelli con al massimo 14 miliardi di parametri. In tal caso, diventa chiaro che mantenere un elevato grado di determinismo non comporta vantaggi significativi, poiché il numero di parametri è minore e, di conseguenza, anche le conoscenze del modello sono limitate. Nel caso invece di modelli più grandi questo vantaggio non si percepisce, ed anzi, in un caso in particolare la sua capacità di generazione di *test* corretti diminuisce. Il secondo *test* di rilievo riguarda l'impiego di codice commentato figura 3.5. Ho ipotizzato che fornendo maggiori dettagli all'**LLM**, anche se in forma di linguaggio naturale, il modello potesse generare *test* più efficaci. I risultati ottenuti sono indubbiamente i più significativi, poiché l'aumento delle informazioni nei modelli più piccoli porta quasi sempre a risultati superiori rispetto a quelli ottenuti con modelli più grandi. Inoltre, gli stessi modelli hanno prestazioni migliori su prompt con commenti rispetto a quelli privi di essi. Posso pertanto supporre che ciò sia dovuto al fatto che, avendo meno informazioni a disposizione, l'incremento di esse attraverso il testo aggiuntivo possa notevolmente migliorare le capacità del modello.

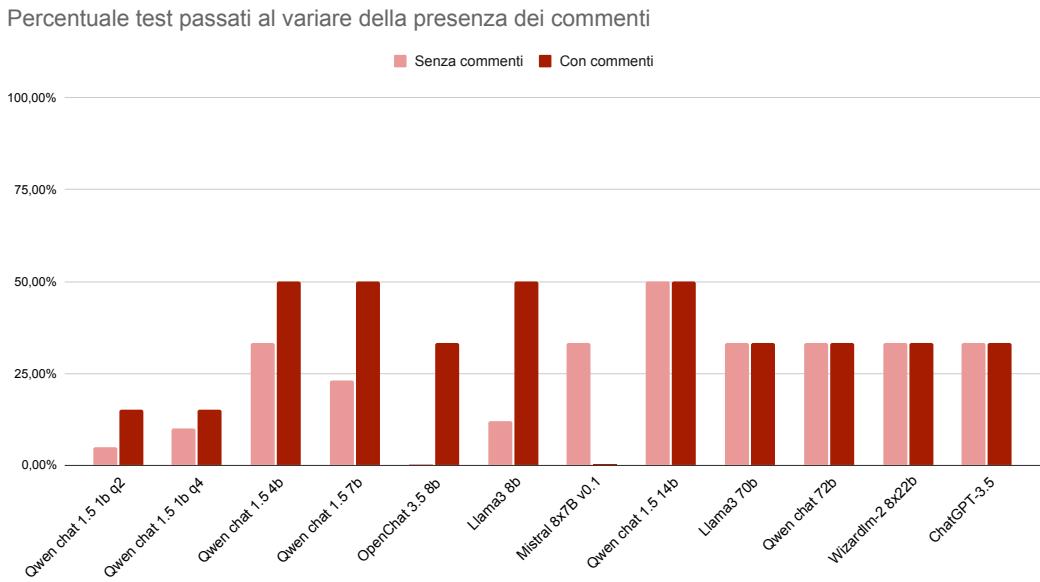


Figura 3.5: Test generati attraverso temperature diverse

Dopo queste rilevazioni, ho optato per confrontare la capacità di generazione di test su codice scritto in inglese e in italiano, questo perchè ho ipotizzato che la quantità di dati in italiano è minore rispetto a quella in inglese e quindi volevo capire se ci fossero differenze significative e se questa ipotesi fosse realmente vera. I risultati ottenuti sono raffigurati in figura 3.6 e figura 3.7.

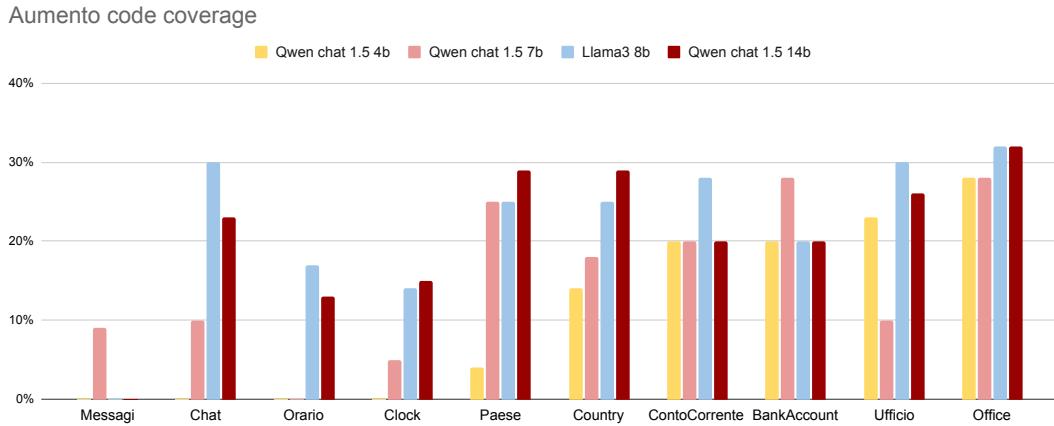


Figura 3.6: Aumenti del code coverage al variare della lingua e dei modelli

In figura 3.6 è sorprendente notare che in molti casi la quantità di *code coverage* aumentata è molto simile tra *llama 8b* e *Qwen chat 14b*. In questi test quindi sebbene *Qwen chat* sia addestrato su una mole di dati maggiore di *llama*, i risultati sono molto simili.

In figura 3.7 notiamo invece che la media degli aumenti del *code coverage* è sempre maggiore per le richieste su codice sorgente in inglese rispetto a quelle in italiano.

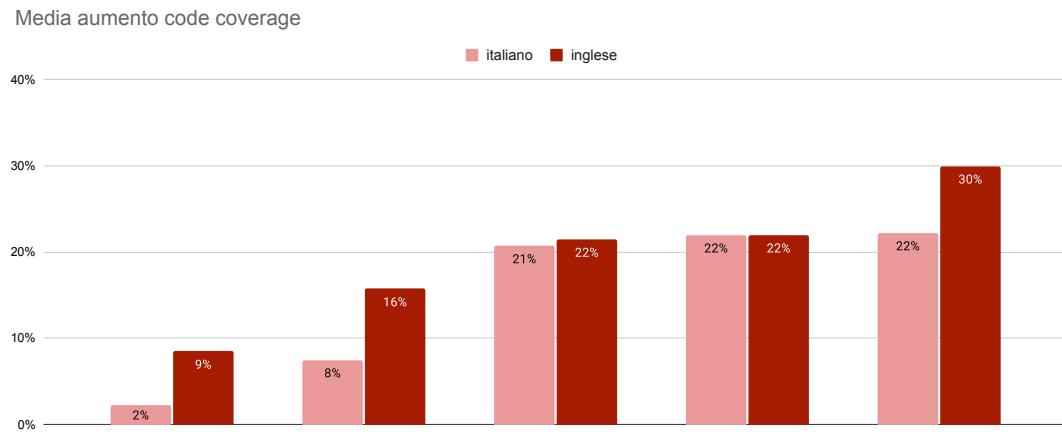


Figura 3.7: Media degli aumenti del code coverage al variare della lingua

L’ipotesi quindi che il codice in inglese possa ottenere risultati migliori rispetto a quello in italiano è confermata.

3.4 Resoconto finale

In questa sezione si procederà a fornire un resoconto finale del lavoro svolto durante il primo macroperiodo, analizzando i risultati ottenuti e le problematiche affrontate.

3.4.1 Prodotti ottenuti

Durante il primo macroperiodo di lavoro sono stati ottenuti diversi risultati, tra cui la realizzazione di uno *script* per l'estrazione e la generazione dei *test* automatici. In particolare, lo *script* permette di effettuare il parsing di un intero progetto, salvando i dati chiave all'interno di un database SQLite. Questo procedimento consente all'**LLM** di individuare le relazioni tra i vari metodi e le classi, facilitando la generazione dei *test*. Inoltre, sono stati effettuati diversi *benchmark* per valutare le prestazioni dei vari **LLM** utilizzati, confrontando i risultati ottenuti.

3.4.2 Risultati ottenuti

I risultati ottenuti sono stati più che soddisfacenti, infatti, aver compreso l'importanza di avere un linguaggio tipizzato per l'analisi sintattica è stato fondamentale per fornire risultati e costatazioni a Zucchetti. Inoltre, aver effettuato i *benchmark* per valutare le prestazioni dei vari **LLM** utilizzati è stato un passo fondamentale per capire quale fosse il modello e la tipologia di *prompt* più adatto per un futuro sviluppo. Nonostante il lavoro finora svolto risulti soddisfacente, persistono alcune aree di indagine e miglioramento. Tra queste, si segnala la necessità di affrontare l'incertezza riguardante la correttezza del codice generato, che può essere erroneo e deve essere scartato, oppure evidenziare la presenza di un difetto nel sistema, rendendolo di conseguenza un *test* di rilevanza significativa.

3.4.3 Conclusione

Durante l'implementazione del progetto, mi sono imbattuto in diverse sfide, tra cui la complessità dell'analisi sintattica in linguaggi non tipizzati, come nel caso di Python e la complessità del funzionamento delle reti neurali. Una difficoltà aggiuntiva è stata rappresentata dalla limitatezza delle risorse computazionali disponibili. Pur facendo stage un'azienda di rilievo come Zucchetti, le risorse a disposizione non sono state sufficienti per l'utilizzo frequente di modelli linguistici di grandi dimensioni, come Mixtral 7x8b o Wizardlm-2 8x22b. Inoltre, la natura intricata del progetto e la scarsità di materiale informativo relativo all'argomento **LLMSE** hanno costituito ulteriori ostacoli. È importante notare che **Assured LLM-Based Software Engineering** è ancora un argomento in fase embrionale, il che si traduce in una carenza di risorse documentative a riguardo. Nonostante queste sfide, il primo macroperiodo del progetto è stato affrontato con successo, generando risultati significativi e fornendo basi solide per uno sviluppo futuro.

Capitolo 4

Fine-tuning di LLM attraverso LoRA e ottimizzazioni

4.1 Analisi del dominio applicativo

Nella seconda fase del progetto si è proceduto con il **fine-tuning** di un **LLM** attraverso **LoRA**, e le sue ottimizzazioni, come ad esempio *MoLE* e *AdaMoLE*. In questo capitolo si approfondiranno inoltre gli studi effettuati sul **fine-tuning** e quantizzazione, concentrandosi maggiormente sul possibile apporto valoriale che questi ultimi possono dare ad un **LLM** e alle sue implementazioni. Questo poiché moltissime realtà aziendali si stanno concentrando sul **fine-tuning** di modelli preaddestrati, in quanto permette di adattare un modello ad un nuovo dominio, senza doverlo addestrare da zero.

4.1.1 Analisi del tema

Nella seconda parte del progetto, vi è stata una maggiore attenzione posta principalmente sullo studio delle tecniche di **fine-tuning** e quantizzazione, poiché rappresentano argomenti complessi e nuovi nel settore. In particolare, la lettura di diversi articoli scientifici e la realizzazione di esperimenti su *Colab* sono stati fondamentali per comprendere a fondo queste tecniche. Queste attività sono state inoltre indispensabili per poter applicare in modo pratico le nozioni apprese. Questa fase del progetto si distingue inoltre dalla prima poiché si è focalizzata maggiormente sulla ricerca e sperimentazione di nuove metodologie rispetto all'applicazione pratica immediata. Tuttavia, sebbene il focus principale fosse la ricerca, vi è stata comunque la possibilità di applicare queste tecniche in modo pratico tramite l'utilizzo di *Colab*.

4.1.2 Esempi di utilizzo

Al giorno d'oggi, le possibili applicazioni di un **LLM** *fine-tuned* sono molteplici e spaziano dalla generazione di codice sorgente alla traduzione di testi. Queste tipologie di utilizzo sono chiamate *downstream task*, poiché sono *task* eseguite dopo il preaddestramento del modello e si concentrano su argomenti specifici. Le *downstream*

task rappresentano quindi la motivazione principale per effettuare il **fine-tuning** di un modello preaddestrato. Lo scopo di questa attività è far concentrare il modello su un particolare dominio, migliorando così le performance in quei compiti specifici rispetto a un modello preaddestrato generico. Per quanto riguarda la quantizzazione, l'utilizzo di modelli quantizzati è oggi pressoché fondamentale. I modelli non quantizzati richiedono infatti troppa memoria e potenza di calcolo, rendendo difficile per la maggior parte delle aziende utilizzare modelli di **LLM** senza questa tecnica di ottimizzazione.

4.1.3 LoRA

LoRA è un metodo di **Parameter Efficient Fine-Tuning (PEFT)** che permette di adattare un modello preaddestrato ad un nuovo dominio, attraverso l'aggiunta di un *layer*. Questa metodologia di **fine-tuning** viene introdotta da J. Edward Hu et al.[0] e consiste nel congelare i pesi del modello preaddestrato e inserire delle *trainable rank decomposition matrices* come un *layer* aggiuntivo. Queste matrici permettono di diminuire notevolmente il numero di parametri da addestrare, rendendo il **fine-tuning** più veloce e meno costoso. La costruzione delle *trainable rank decomposition matrices* A e B avviene attraverso la decomposizione della matrice di pesi W in due matrici di rango ridotto. Supponiamo di avere una matrice preaddestrata $W_0 \in R^{d \times k}$, e voler aggiungere un *layer* ΔW per il fine tuning, allora possiamo scrivere:

$$W_0 + \Delta W = W_0 + BA ,$$

dove $B \in R^{d \times r}$ e $A \in R^{r \times k}$ con $\text{rank } r \ll \min(d, k)$. In questo modo è possibile addestrare solamente un insieme più piccolo di parametri i quali, moltiplicati tra loro, si avvicinano alla matrice di pesi più ampia nel modello pre-addestrato.

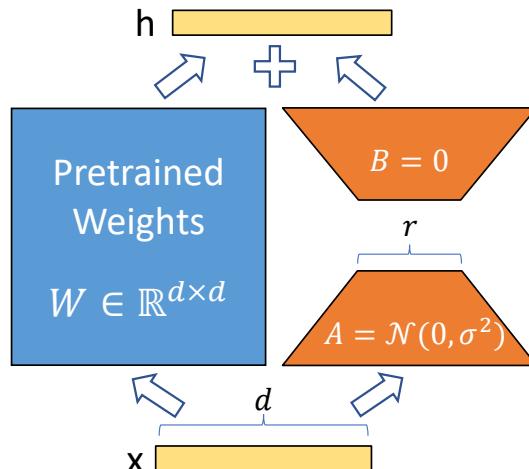


Figura 4.1: LoRA matrices

A e B vengono rispettivamente inizializzati attraverso la funzione Gaussiana e 0, quindi il valore all'inizio dell'addestramento della matrice W è 0. Attraverso le iterazioni, i parametri verranno modificati e si otterrà un *layer* LoRA che permetterà al modello di avere capacità più specifiche. Notiamo quindi che la quantità di dati da aggiornare è nettamente minore, si dovrà infatti addestrare solo A e B. In particolare, l'utilizzo

della VRAM diminuisce di 2/3 se $r \ll d_{model}$. L'utilizzo di **LoRA** porta ad alcuni vantaggi oltre al risparmio di memoria, l'addestramento infatti risulta più efficiente ed inoltre è possibile creare tanti *layers LoRA* in modo da poterli interscambiare in base alle esigenze.

4.1.3.1 Future applicazioni

Le future applicazioni che durante questo stage sono state prese in considerazione con il tutor aziendale Gregorio Piccoli e sono legate soprattutto alla possibilità di utilizzare modelli *fine-tuned* per rispondere a domande sul dominio *downstream task*, come ad esempio la generazione di codice sorgente. Un ulteriore utilizzo potrebbe essere legato al compilatore dei linguaggi di programmazione, in modo tale da avere un feedback logico del programma oltre che sintattico e semantico. Inoltre, un'altra possibile applicazione è quella di rendere disponibili i modelli attraverso la quantizzazione anche su macchine più piccole, come ad esempio microcontrollori. Questa possibilità al giorno d'oggi permetterebbe l'utilizzo di **LLM** in dispositivi che non hanno una potenza di calcolo elevata, come ad esempio gli smartphone.

4.2 Analisi dei requisiti

4.2.1 Analisi preventiva dei rischi

Durante la fase di analisi dei rischi sono stati individuate le possibili criticità che potranno essere riscontrate. Si è quindi proceduto a elaborare delle possibili soluzioni per far fronte a tali rischi.

2. Mancanza di risorse computazionali

Descrizione: Essendo il *fine-tuning* un processo molto oneroso, la quantità di risorse computazionali a disposizione potrebbe risultare quindi non sufficiente per addestrare modelli molto grandi.

Soluzione: utilizzo di Colab e coinvolgimento del responsabile aziendale.

4.2.2 Requisiti e obiettivi

	Obiettivo	Descrizione
OB 4	Realizzazione di test con LLM fine-tuned attraverso LoRA .	
DE 1	Quantizzazione di LLM.	

Tabella 4.1: Requisiti secondo macroperiodo.

4.3 Sviluppo del prodotto

Lo sviluppo del prodotto durante il secondo macroperiodo non è stato immediato come è successo durante prima parte, questo poichè è stato necessario studiare in modo approfondito gli argomenti che sarebbero stati trattati successivamente. Dopo lo studio della teorica legata al **fine-tuning** e quantizzazione, è stato necessario studiare anche le tecniche di applicazione, ciò è avvenuto simultaneamente alla loro implementazione, vi è stato quindi un'applicazione della metodologia di apprendimento *learning by doing*. Inoltre, è stato necessario preparare un dataset attraverso il quale effettuare il **fine-tuning**.

4.3.1 *Fine-tuning attraverso Pytorch*

Il **fine-tuning** è un processo costoso, per questo motivo si è scelto di eseguirlo su *Colab*, utilizzando un account Pro, il quale offre 100 unità di calcolo. Di seguito viene spiegato il processo di **fine-tuning** attraverso il codice utilizzato. È possibile in figura 4.2 trovare il codice relativo alla configurazione di **LoRA**.

```
lora_alpha = 32
lora_dropout = 0.05
lora_rank = 64
LORA_TARGET_MODULES = ["qkv_proj", "o_proj"]

peft_config = LoraConfig(
    lora_alpha=lora_alpha,
    lora_dropout=lora_dropout,
    r=lora_rank,
    bias="none",
    task_type="CAUSAL_LM",
    target_modules=LORA_TARGET_MODULES
)

peft_model = get_peft_model(model, peft_config)
```

Figura 4.2: Configurazione parametri **LoRA**

- **lora_alpha** determina l'entità della riduzione del rango durante il processo di approssimazione *Low Rank*. Un valore più alto di *lora_alpha* comporta una riduzione più aggressiva del rango, risultando in una maggiore compressione delle matrici dei pesi e in un modello più efficiente in termini di parametri. Al contrario, un valore inferiore di *lora_alpha* comporta una riduzione meno aggressiva del rango, preservando più parametri del modello originale.
- **lora_rank** esprime invece il rango della matrice decomposta. Un rango maggiore implica un maggiore utilizzo di memoria, mentre un rango minore riduce

l'impatto sulla memoria ma comporta una perdita di informazioni. Diventa quindi cruciale trovare il rango più adeguato per bilanciare l'efficienza della memoria e la conservazione delle informazioni.

- **lora_dropout** è la probabilità di eliminare elementi dalle matrici A e B per evitare *overfitting*.
- **LORA_TARGET_MODULES** sono i moduli dei layer ai quali verrà applicato **LoRA**. In questo caso, **LoRA** è stato applicato ai moduli *qkv_proj*, *o_proj* presenti come visibile in figura 4.4 nella *self attention* come suggerito da et al. [0]. È importante notare che i moduli in questione potrebbero cambiare in base al modello e alla sua architettura interna.

In figura 4.3 è possibile quindi visualizzare l'architettura di *Phi3-mini*, uno dei modelli al quale è stato applicato il **fine-tuning**, *Phi3-mini* è un *state-of-the-art open model*, il quale si basa sull'architettura *Transformer* ed è stato pre-addestrato sia con dati pubblici ma anche sintetici.

```
Phi3ForCausallLM(
    (model): Phi3Model(
        (embed_tokens): Embedding(32064, 3072, padding_idx=32000)
        (embed_dropout): Dropout(p=0.0, inplace=False)
        (layers): ModuleList(
            (0-31): 32 x Phi3DecoderLayer(
                (self_attn): Phi3Attention(
                    (o_proj): Linear8bitLt(in_features=3072, out_features=3072, bias=False)
                    (qkv_proj): Linear8bitLt(in_features=3072, out_features=9216, bias=False)
                    (rotary_emb): Phi3RotaryEmbedding()
                )
                (mlp): Phi3MLP(
                    (gate_up_proj): Linear8bitLt(in_features=3072, out_features=16384, bias=False)
                    (down_proj): Linear8bitLt(in_features=8192, out_features=3072, bias=False)
                    (activation_fn): SiLU()
                )
                (input_layernorm): Phi3RMSNorm()
                (resid_attn_dropout): Dropout(p=0.0, inplace=False)
                (resid_mlp_dropout): Dropout(p=0.0, inplace=False)
                (post_attention_layernorm): Phi3RMSNorm()
            )
        )
        (norm): Phi3RMSNorm()
    )
    (lm_head): Linear(in_features=3072, out_features=32064, bias=False)
)
```

Figura 4.3: Architettura di *Phi3-mini*

CAPITOLO 4. FINE-TUNING DI LLM ATTRAVERSO LORA E OTTIMIZZAZIONI

```
PeftModelForCausalLM(  
    (base_model): LoraModel(  
        (model): Phi3ForCausalLM(  
            (model): Phi3Model(  
                (embed_tokens): Embedding(32064, 3072, padding_idx=32000)  
                (embed_dropout): Dropout(p=0.0, inplace=False)  
                (layers): ModuleList(  
                    (0-31): 32 x Phi3DecoderLayer(  
                        (self_attn): Phi3Attention(  
                            (o_proj): lora.Linear8bitLt(  
                                (base_layer): Linear8bitLt(in_features=3072, out_features=3072, bias=False)  
                                (lora_dropout): ModuleDict(  
                                    (default): Dropout(p=0.05, inplace=False)  
                                )  
                                (lora_A): ModuleDict(  
                                    (default): Linear(in_features=3072, out_features=64, bias=False)  
                                )  
                                (lora_B): ModuleDict(  
                                    (default): Linear(in_features=64, out_features=3072, bias=False)  
                                )  
                                (lora_embedding_A): ParameterDict()  
                                (lora_embedding_B): ParameterDict()  
                                (lora_magnitude_vector): ModuleDict()  
                            )  
                            (qkv_proj): lora.Linear8bitLt(  
                                (base_layer): Linear8bitLt(in_features=3072, out_features=9216, bias=False)  
                                (lora_dropout): ModuleDict(  
                                    (default): Dropout(p=0.05, inplace=False)  
                                )  
                                (lora_A): ModuleDict(  
                                    (default): Linear(in_features=3072, out_features=64, bias=False)  
                                )  
                                (lora_B): ModuleDict(  
                                    (default): Linear(in_features=64, out_features=9216, bias=False)  
                                )  
                                (lora_embedding_A): ParameterDict()  
                                (lora_embedding_B): ParameterDict()  
                                (lora_magnitude_vector): ModuleDict()  
                            )  
                            (rotary_emb): Phi3RotaryEmbedding()  
                        )  
                        (mlp): Phi3MLP(  
                            (gate_up_proj): Linear8bitLt(in_features=3072, out_features=16384, bias=False)  
                            (down_proj): Linear8bitLt(in_features=8192, out_features=3072, bias=False)  
                            (activation_fn): SiLU()  
                        )  
                        (input_layernorm): Phi3RMSNorm()  
                        (resid_attn_dropout): Dropout(p=0.0, inplace=False)  
                        (resid_mlp_dropout): Dropout(p=0.0, inplace=False)  
                        (post_attention_layernorm): Phi3RMSNorm()  
                    )  
                    (norm): Phi3RMSNorm()  
                )  
                (lm_head): Linear(in_features=3072, out_features=32064, bias=False)  
            )  
        )  
    )  
)
```

Figura 4.4: Applicazione LoRA a *Phi3-mini*

Nella figura 4.4 possiamo notare il risultato ottenuto dall'applicazione di **LoRA**.

Successivamente, è stato necessario trovare un *dataset* appropriato per il **fine-tuning**. È stato scelto "Vezora/Tested-22k-Python-Alpaca", che contiene ben 22.600 esempi di codice testato. Il dataset, suddiviso in istruzioni, input e output, è stato formattato correttamente come mostrato in figura 4.5 per seguire il *prompt* con cui il modello di partenza è stato addestrato. Questo passaggio è cruciale poiché ogni **LLM** richiede un determinato *prompt*. Senza il *prompt* adeguato, il modello non riuscirebbe a riconoscere i delimitatori utilizzati per distinguere le istruzioni dalle corrispondenti

risposte.

```
def transform_dataset(dataset):
    instructions = dataset["train"]["instruction"]
    inputs = dataset["train"]["input"]
    outputs = dataset["train"]["output"]
    transformed_dataset = []
    for i in range(len(instructions)):
        system_header = "<|begin_of_text|><|start_header_id|>system<|end_header_id|>\n"
        instruction = f"<|start_header_id|>{instructions[i]}<|end_header_id|>\n"
        user_header = "<|start_header_id|>user<|end_header_id|>\n"
        user_msg_1 = f"<|start_header_id|>{inputs[i]}<|end_header_id|>\n"
        assistant_header = "<|start_header_id|>assistant<|end_header_id|>\n"
        model_answer = f"<|start_header_id|>{outputs[i]}<|end_header_id|>\n"
        transformed_example = f'{system_header}{instruction}{user_header}{user_msg_1}{assistant_header}{model_answer}'
        transformed_dataset.append({"text": transformed_example})

    return transformed_dataset
```

Figura 4.5: Funzione per costruire il *prompt* adeguato a *Phi3-mini*

L'ultimo passaggio prima di effettuare effettivamente il *fine-tuning* è stato definire i parametri per l'allenamento, come ad esempio *per_device_batch_size*, *optim*, *learning_rate*, *max_step*. In questo caso si è deciso di avere una grandezza di *batch* pari a 16, ciò va a specificare il numero di esempi utilizzati per ogni iterazione del *fine-tuning* del modello. Riguardo a *optim*, l'ottimizzatore, si è deciso di utilizzare *adamw_torch* il quale è un algoritmo di ottimizzazione che unisce i benefit dell'ottimizzatore *Adam* con *weight decay* (regolarizzazione) per prevenire l'*overfitting*. La *learning_rate* invece è il tasso di apprendimento e determina quanto velocemente o lentamente un modello apprende. È un fattore di scalatura che regola quanto devono essere aggiornati i pesi del modello in risposta all'errore calcolato in ogni iterazione dell'addestramento. Per questo parametro è stato scelto 5×10^{-5} poichè dopo alcuni esperimenti è risultato essere il miglior valore di *tradeoff* tra velocità e precisione.

```
output_dir = "Phi3-mini-LoRA-22kPython"
per_device_train_batch_size = 16
gradient_accumulation_steps = 8
optim = "adamw_torch"
save_steps = 10
logging_steps = 2
learning_rate = 5e-5
max_grad_norm = 0.5
max_steps = 100
warmup_ratio = 0.005
lr_scheduler_type = "cosine"

training_arguments = TrainingArguments(
    output_dir=output_dir,
    per_device_train_batch_size=per_device_train_batch_size,
    gradient_accumulation_steps=gradient_accumulation_steps,
    optim=optim,
    save_steps=save_steps,
    logging_steps=logging_steps,
    learning_rate=learning_rate,
    fp16=True,
    max_grad_norm=max_grad_norm,
    max_steps=max_steps,
    warmup_ratio=warmup_ratio,
    group_by_length=True,
    lr_scheduler_type=lr_scheduler_type,
    push_to_hub=True
)
max_seq_length = 512

trainer = SFTTrainer(
    model=peft_model,
    train_dataset=transformed_dataset,
    peft_config=peft_config,
    dataset_text_field="text",
    max_seq_length=max_seq_length,
    tokenizer=tokenizer,
    args=training_arguments,
)
trainer.train()
```

Figura 4.6: Esempio di codice per l'addestramento

4.3.2 *Fine-tuning attraverso LLama.cpp*

Oltre all'utilizzo di *Colab*, è stato successivamente eseguito il *fine-tuning* attraverso *LLama.cpp*. A differenza di *Colab*, *LLama.cpp* prevede una semplificazione del codice e consente di utilizzarlo senza limiti computazionali, salvo quelli imposti dalle proprie risorse hardware.

```
llama.cpp\finetune.exe
--model-base model.gguf
--train-data trainer.txt
--lora-out Lora.gguf
--threads 14
--batch 8
--sample-start "<s>"
--ctx 1024
--use-checkpointing
--checkpoint-out LoRAModelCheckpoint-ITERATION.gguf
--adam-iter 8192
--adam-alpha 0.001
--lora-r 16
--lora-alpha 16
--fill-with-next-samples
--epoch 3
--separate-with-eos
```

4.3.3 Ottimizzazioni del fine-tuning

4.3.3.1 Mixture of LoRA Experts

Il framework *Mixture of LoRA Expert* rappresenta un metodo di *fine-tuning* che si basa sull'utilizzo di *LoRA*. Introdotta per la prima volta da Wu et al. [0], questa tecnica mira a risolvere i problemi legati alla riduzione delle capacità generative dei modelli affinati tramite *LoRA*. Nel contesto di un modello *Mixture of LoRA Expert* nell'apprendimento automatico, una funzione di gating viene utilizzata per assegnare dinamicamente diversi input a diversi "esperti" (tipicamente sub-modelli o reti neurali) all'interno del modello complessivo come visibile in figura 4.7. La funzione di gating:

$$\sigma(W \times x + b),$$

determina il contributo o il peso di ciascun esperto per un dato input, decidendo efficacemente quale esperto o combinazione di esperti debba essere responsabile delle predizioni per quell'input.

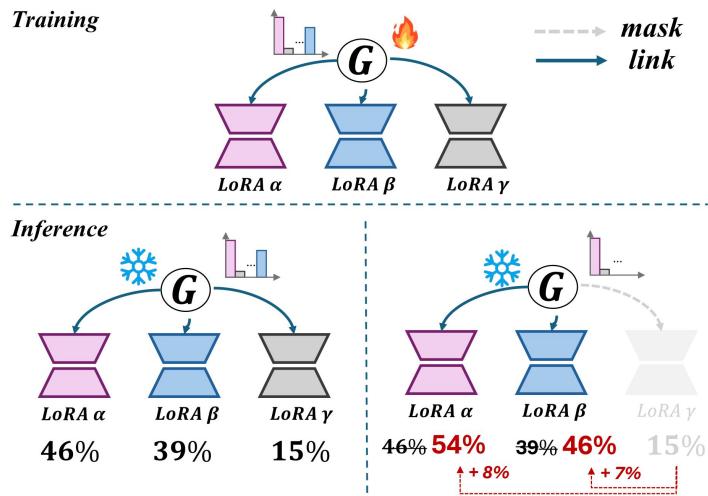


Figura 4.7: Assegnazione dinamica dei LoRA layer in Mixture of LoRA Expert

Mixture of LoRA Expert consente l’impiego di diversi livelli di LoRA, trattando ciascun livello addestrato con LoRA come un esperto distinto. Implementa inoltre un controllo gerarchico del peso attraverso una funzione di gating che viene appresa all’interno di ogni livello, mantenendo congelati tutti gli altri parametri. In questo modo, è possibile apprendere pesi di composizione adattati specificamente agli obiettivi di un determinato dominio.

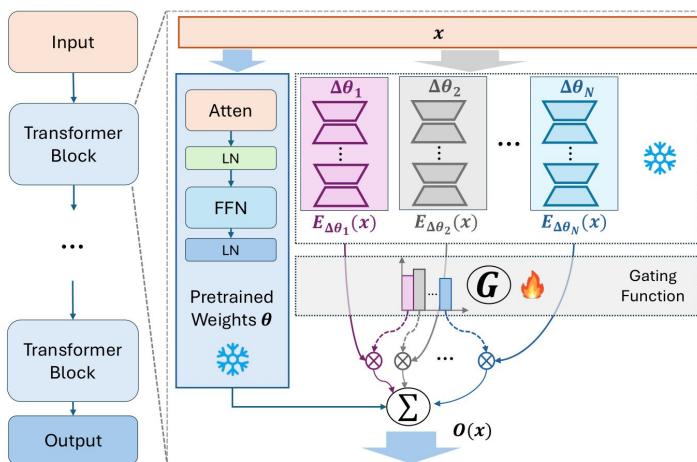


Figura 4.8: Mixture of Expert all’interno dell’architettura Transformer

I layer quindi vengono integrati all’interno dell’architettura Transformer come visibile in figura 4.8. Come è visibile, l’idea di base di LoRA, la quale consiste nel congelare i pesi pre-addestrati, continua ad essere presente, in parallelo, i LoRA layer dopo essere stati addestrati per specifiche task, vengono sommati linearmente e il loro peso in questa somma viene determinato dalla funzione di gating.

4.3.3.2 AdaMoLE

Adaptive Mixture of LoRA Expert è un metodo il quale utilizza *Mixture of LoRA Expert* e una soglia di rilevanza, quest'ultima permette l'attivazione dei vari esperti solo se la loro percentuale di congruenza rispetto al contesto supera la soglia stessa. Questo procedimento permette una selezione dinamica degli esperti attivando quindi solamente gli esperti che sono più appropriati rispetto al contesto (*context-responsive*), ciò porta ad una migliore adattabilità e performance più elevate.

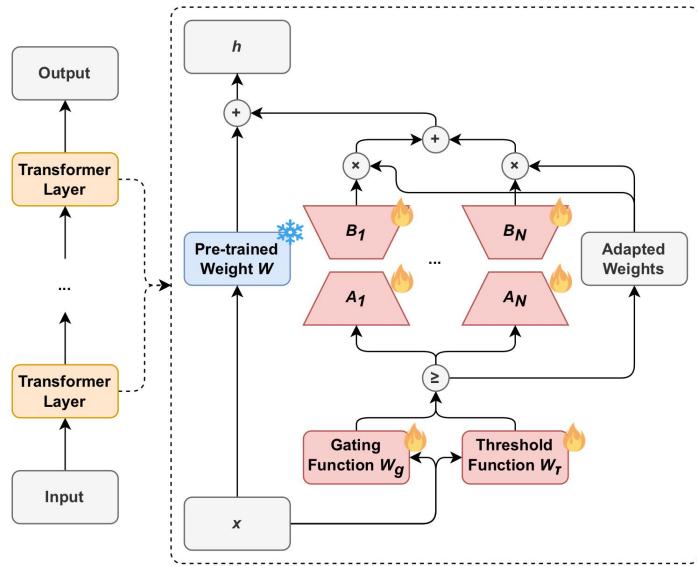


Figura 4.9: Struttura AdaMoLE nell'architettura Transformer

In figura 4.9 è possibile notare la scelta dei vari esperti, la soglia e la gating function sono parte fondamentale di tutto il processo. Successivamente, dopo aver valutato gli esperti, i loro output vengono combinati per produrre la risposta finale del sistema. Questo può essere fatto attraverso pesi adattivi che tengono conto della rilevanza di ciascun esperto per l'input corrente.

Ricordando che la distribuzione dei pesi ai vari esperti è ρ_i , con:

$$\rho_i = \text{SoftMax}(W_g x)_i,$$

e il risultato della somma degli output è:

$$y = \sum_{i=1}^N \frac{\text{TopK}(\rho_i)}{\sum_{i'=1}^N \text{TopK}(\rho_{i'})} \cdot E_i(x),$$

mentre la decisione di utilizzare un determinato esperto deriva da:

$$\rho_i \geq \tau, \text{ con } \tau \text{ come soglia}$$

È possibile intuire che è di massima importanza scegliere una soglia τ adeguata, infatti $\rho_i \geq \tau$, questo perché scegliendo τ troppo elevata si potrebbero escludere tutti i layer e quindi avere un risultato che non è frutto del modello con *fine-tuning*. D'altra parte se la soglia fosse troppo bassa questo potrebbe includere layer non rilevanti per il contesto. Per mitigare queste possibili problematiche si utilizza $\tau = \frac{1}{N}$, da ciò deriva che $\sum_{i=1}^N \rho_i \geq N\tau = 1$ contraddicendo il fatto che ρ_i debba essere uguale a 1. L'output quindi derivante da AdaMoLE è:

$$y = \sum_{i=1}^N \frac{\mathbb{1}(p_i \geq \tau) \cdot p_i}{\sum_{i'=1}^N \mathbb{1}(p_{i'} \geq \tau) \cdot p_{i'}} \cdot E_i(x),$$

Con $\mathbb{1}$ uguale a 1 se la condizione $p_i \geq \tau$ è vera, 0 altrimenti.

4.3.4 Quantizzazione

La quantizzazione di un **LLM** è una tecnica utilizzata per ridurre le dimensioni del modello e aumentare l'efficienza computazionale senza una significativa perdita di accuratezza. Questa tecnica converte i parametri del modello (tipicamente rappresentati come numeri in virgola mobile a 32-bit floating point) in una rappresentazione con meno bit, ad esempio con interi a 8-bit o a 16-bit ed in alcuni casi estremizzando a 4-bit o 2-bit. La quantizzazione è una tecnica ormai necessaria per permettere l'utilizzo degli **LLM** su qualsiasi dispositivo poiché hanno raggiunto dimensioni computazionalmente proibitive per molti dei dispositivi mobile. Durante il secondo macroperiodo vi è stata la possibilità di affrontare l'argomento quantizzazione, in particolare si è approfondita la quantizzazione asimmetrica e simmetrica. Entrambe le metodologie mirano a comprimere i modelli per renderli più efficienti in termini di memoria e velocità di calcolo, ma differiscono nel modo in cui i valori vengono mappati nello spazio quantizzato.

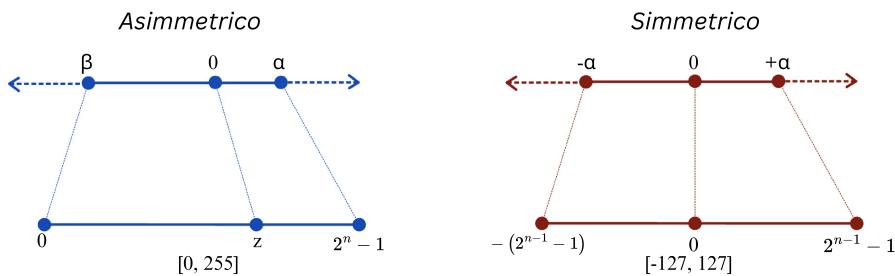


Figura 4.10: Differenza tra tecnica asimmetrica e simmetrica

Quantizzazione Asimmetrica:

- **Scala Non Uniforme:** Nella quantizzazione asimmetrica, i valori del modello vengono mappati in uno spazio quantizzato che non è centrato attorno allo zero. L'intervallo dei valori positivi e negativi è solitamente diverso e dipende da α e β
- **Range di Quantizzazione:** Viene definito un fattore di scala s diverso per i valori positivi e negativi, o un valore di offset z che permette di gestire l'intervallo in modo più flessibile.

- **Flessibilità Maggiore:** Permette una rappresentazione più precisa per distribuzioni di valori che non sono simmetriche attorno allo zero.

Per ottenere la quantizzazione asimmetrica, x_q del numero x_f si utilizzano le seguenti formule:

$$x_q = \text{clamp}\left(\left\lfloor \frac{x_f}{s} \right\rfloor + z; 0; 2^n - 1\right) \quad s = \frac{\alpha - \beta}{2^n - 1} \quad z = \left\lfloor -1 \times \frac{\beta}{s} \right\rfloor$$

In caso si volesse riconvertire al numero originario il numero quantizzato attraverso la tecnica asimmetrica la formula è la seguente:

$$x_f = s(x_q - z).$$

Quantizzazione Simmetrica:

- **Scala Uniforme:** Nella quantizzazione simmetrica, i valori del modello vengono mappati in uno spazio quantizzato che è centrato attorno allo 0, sostanzialmente lo 0 quantizzato rimane 0. Ciò significa che l'intervallo dei valori positivi e negativi è uguale.
- **Range di Quantizzazione:** Viene definito un unico fattore di scala s che determina come i valori in virgola mobile vengono convertiti in valori interi. Questo fattore di scala è uguale per i valori positivi e negativi.
- **Semplicità di Implementazione:** La simmetria attorno allo zero rende più semplici i calcoli e diminuisce l'hardware necessario per la quantizzazione.

Per ricavare quindi il numero quantizzato x_q è necessario applicare le seguenti formule:

$$x_q = \text{clamp}\left(\left\lfloor \frac{x_f}{s} \right\rfloor ; -(2^{n-1} - 1); 2^{n-1} - 1\right) \quad s = \frac{|\alpha|}{2^{n-1} - 1}$$

Nel caso in cui invece si volesse riconvertire al numero originario il numero quantizzato attraverso la tecnica simmetrica la formula è la seguente:

$$x_f = sx_q.$$

4.3.4.1 Sviluppo

Per la quantizzazione, si è proceduto direttamente quantizzando Phi3-mini. Come visibile in figura 4.4, il modello, che ha subito il fine-tuning, è stato precedentemente quantizzato da 32-bit floating point a 8-bit interi. Questo è stato possibile attraverso il seguente frammento di codice:

```
bnb_config = BitsAndBytesConfig(
    load_in_8bit=True,
    load_in_8bit_fp32_cpu_offload=True
)
model_name = "microsoft/Phi-3-mini-4k-instruct"
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map="auto",
    trust_remote_code=True
)
```

Figura 4.11: Errore relativo asimmetrico suddiviso nei diversi bit di quantizzazione

In particolare, si è utilizzato il plugin BitsAndBytesConfig, che permette di quantizzare a 8-bit o a 4-bit qualsiasi modello. La quantizzazione ha quindi reso possibile una maggiore velocità per l'inferenza e inoltre anche un consumo ridotto delle risorse a disposizione.

4.3.4.2 Test errore di quantizzazione

È noto che scalando 32-bit in un numero di bit minore vi sarà sicuramente una perdita, anche minima, di informazione. In seguito è possibile visionare errore relativo in percentuale che deriva dalla quantizzazione.

$\text{errore_relativo} = \frac{|x - x_q|}{|x|}$, con x valore a 32-bit, e x_q , valore quantizzato,
In figura 4.11 e 4.12 è possibile visualizzare in percentuale l'errore relativo asimmetrico e simmetrico rispetto a 1000 valori.

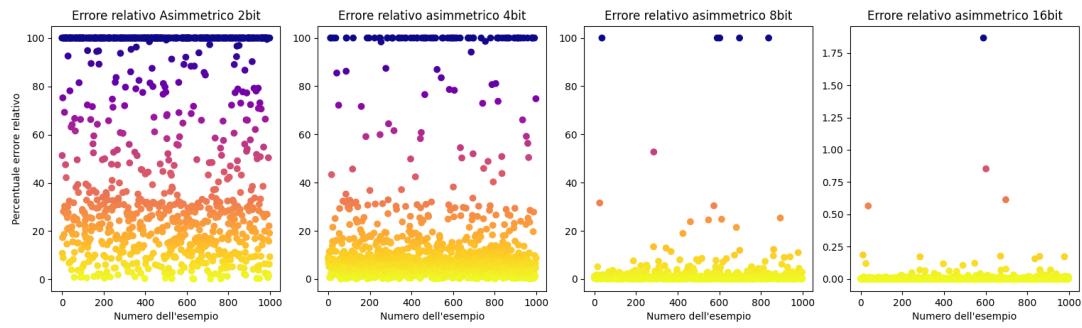


Figura 4.12: Errore relativo asimmetrico suddiviso nei diversi bit di quantizzazione

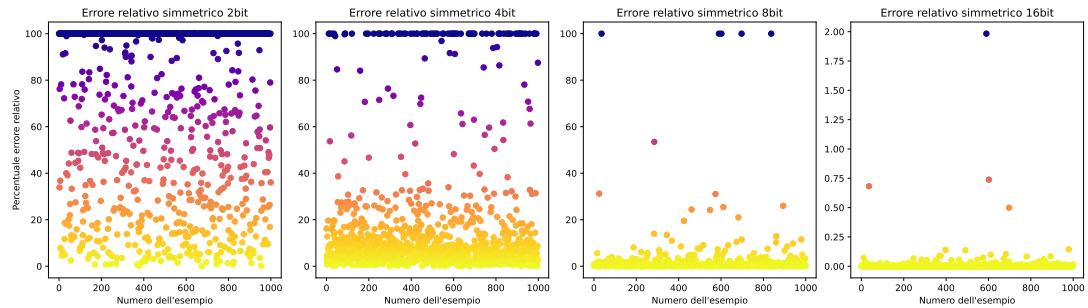


Figura 4.13: Errore relativo simmetrico suddiviso nei diversi bit di quantizzazione

4.4 Resoconto finale

4.4.1 Prodotti ottenuti

4.4.2 Risultati ottenuti

4.4.3 Conclusione

Capitolo 5

Valutazione retrospettiva

5.1 Conoscenze acquisite

5.2 Valutazione personale

Bibliografia

Articoli

- [0] Nadia Alshahwan et al. «Assured LLM-Based Software Engineering». In: *ArXiv* abs/2402.04380 (2024). URL: <https://api.semanticscholar.org/CorpusID:267523361>.
- [0] Nadia Alshahwan et al. «Automated Unit Test Improvement using Large Language Models at Meta». In: *ArXiv* abs/2402.09171 (2024). URL: <https://api.semanticscholar.org/CorpusID:267657828>.
- [0] J. Edward Hu et al. «LoRA: Low-Rank Adaptation of Large Language Models». In: *ArXiv* abs/2106.09685 (2021). URL: <https://api.semanticscholar.org/CorpusID:235458009>.
- [0] Purnawansyah Purnawansyah et al. «Memory Efficient with Parameter Efficient Fine-Tuning for Code Generation Using Quantization». In: *2024 18th International Conference on Ubiquitous Information Management and Communication (IMCOM)* (2024), pp. 1–6. URL: <https://api.semanticscholar.org/CorpusID:267641285>.
- [0] Babak Rokh, Ali Azarpeyvand e Alireza Khanteymoori. «A Comprehensive Survey on Model Quantization for Deep Neural Networks in Image Classification». In: *ACM Transactions on Intelligent Systems and Technology* 14 (2022), pp. 1–50. URL: <https://api.semanticscholar.org/CorpusID:261661742>.

Sitografia

- [0] *Fine-tuning using PEFT-LoRA.* URL: <https://medium.com/@srishitinagu19/fine-tuning-falcon-7b-instruct-using-peft-lora-on-free-gpu-6fa1b0fcacb>.

Glossario

Assured LLM-Based Software Engineering ADD DESCRIPTION.. [i](#), [iii](#), [15](#)

Deep learning ADD DESCRIPTION.. [i](#), [iii](#), [3](#)

Fine-tuning Metodologia che permette, attraverso modifiche minimali agli iperparametri di una *neural network*, di adattare un modello ad un nuovo dataset senza doverlo riaddestrare, ottenendo quindi un modello più accurato.. [i](#), [iii](#), [v](#), [vii](#), [2](#), [17–23](#), [25](#), [27](#)

IDE Un Integrated Development Environment (IDE) è un'applicazione software che fornisce servizi per facilitare lo sviluppo di software. Un IDE generalmente comprende un editor di codice sorgente, strumenti di compilazione e debugging e un ambiente per eseguire il software in sviluppo.. [i](#), [iii](#), [vii](#)

LLM Un Large Language Model (LLM) è un modello capace di generare testi in linguaggio naturale basandosi su modelli statistici. Questi modelli acquisiscono una conoscenza linguistica attraverso l'apprendimento di relazioni statistiche durante un processo di addestramento computazionalmente oneroso.. [i](#), [iii](#), [vii](#), [1](#), [5](#), [6](#), [12](#)

LoRA Approccio di [fine-tuning](#) il quale permette di costruire diversi modelli per *downstream tasks* i quali ne condividono uno pre-addestrato.. [i](#), [iii](#), [vii](#)

Machine learning Branca dell'*intelligenza artificiale* che utilizza metodi statistici per migliorare la performance di un algoritmo nell'identificare pattern nei dati, imparando da questi a svolgere delle funzioni piuttosto che attraverso la programmazione esplicita.. [i](#), [iii](#), [vii](#), [3](#)

Parameter Efficient Fine-Tuning ADD DESCRIPTION.. [i](#), [iii](#)

Prototipo Un prototipo è un esemplare o un modello di un prodotto o di un sistema che viene realizzato antecedentemente al prodotto finale. [i](#), [iii](#), [1](#)

Acronimi e abbreviazioni

IDE Integrated Development Environment. i, iii, v, 4

LLM Large Language Model. i, iii, v, vii, 2, 5–12, 15, 17–19, 22, 28

LLMSE Assured LLM-Based Software Engineering. i, iii, vii, xi, 5–7, 15

LoRA Low-Rank Adaptation. i, iii, v, vii, xi, 2, 17–22, 25–27

PEFT Parameter Efficient Fine-Tuning. i, iii, vii, 18