

Relazione progetto di IA a.2024/2025

Membri del gruppo:

| | | |
|--------------------|---|------------|
| - Angelo Greco | - | 0001195141 |
| - Elia Friberg | - | 0001180514 |
| - Giacomo Sagliano | - | 0001145848 |
| - Matteo Raggi | - | 0001180436 |

1. Introduzione

Il progetto riguarda il **fine-tuning** di un modello di *Natural Language Processing* (NLP) per la generazione di codice, e il seguente **confronto delle prestazioni** tra esso e un modello pre-addestrato.

Come consigliato dalla traccia, il punto di partenza è stato il codice del decoder del repository [teaching material AI](#).

In questa relazione presenteremo i modelli e i dataset usati, il nostro approccio al fine-tuning e i risultati ottenuti e il confronto tra i modelli analizzati, evidenziando anche i miglioramenti osservati dopo il fine-tuning.

1.1. Modelli e dataset utilizzati

Per lo svolgimento del progetto, sono stati utilizzati 2 modelli (uno da fine-tunare e l'altro 'base', usato per il confronto) e 2 dataset (uno per l'allenamento e l'altro per la valutazione del codice generato dai modelli). Essi sono:

Modello di partenza per il fine-tuning – [deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B](#):

Modello linguistico compatto da 1.5 miliardi di parametri, distillato dalla variante open-source di DeepSeek. È progettato per offrire un buon compromesso tra performance e leggerezza.

Modello di confronto (baseline) – [bigcode/starcoder2-3b](#):

Modello specializzato nella generazione di codice, con 3 miliardi di parametri, e consigliato dalla traccia del progetto. Viene utilizzato senza ulteriori addestramenti per confrontare la qualità delle generazioni rispetto al modello fine-tunato.

Dataset di addestramento – [bigcode/self-oss-instruct-sc2-exec-filter-50k](#):

Dataset composto da circa 50.000 istruzioni in linguaggio naturale accompagnate da corrispondenti implementazioni di funzioni Python, consigliato dalla traccia del progetto. È filtrato per assicurare *eseguibilità* e *qualità sintattica*, e viene utilizzato per addestrare il modello a rispondere a istruzioni tecniche con codice funzionante.

Dataset di valutazione – [openai/openai_humaneval](#):

Benchmark standard di OpenAI per la valutazione di modelli di generazione di codice, richiesto dalla traccia del progetto. Consiste in una serie di prompt che richiedono la scrittura di funzioni python. È usato per confrontare le performance dei due modelli.

1.2. Tecnologie e librerie utilizzate

Google Colab: Ambiente cloud-based gratuito (seppur con forti limiti) che fornisce accesso a GPU e TPU di elevata potenza. È stato fondamentale per eseguire il fine-tuning del modello in tempi ragionevoli.

GitHub: Celebre piattaforma di versionamento del codice utilizzata per salvare, documentare e condividere il progetto. Ha permesso la collaborazione tra membri del gruppo e la tracciabilità delle modifiche effettuate, nonché la consegna finale del codice del progetto.

Libreria PyTorch: Framework di deep learning utilizzato come backend per la definizione, ottimizzazione e fine-tuning del modello neurale.

Piattaforma HuggingFace: Infrastruttura centrale all'intero progetto, HuggingFace dispone di molte librerie tramite il quale fornisce modelli pre-addestrati (tramite **transformers**), dataset condivisi ed operazioni comuni su di essi (**datasets**), metodi per il fine-tuning efficiente (**peft**) e strumenti per la quantizzazione dei pesi del modello (**bitsandbytes**). Tutti questi componenti sono stati integrati in modo coerente per semplificare il caricamento del modello, la preparazione dei dati e la procedura di addestramento.

1.3. Struttura del Notebook

Il codice del progetto è contenuto all'interno di un singolo notebook. Esso è strutturato in diverse sezioni:

- **Setup iniziale**, contenente l'**installazione e importazione delle librerie necessarie**, e la definizione delle **variabili chiave** per il progetto (variabili di configurazione ed iperparametri del modello);
- **Caricamento dei modelli e dei rispettivi tokenizer**;
- **Caricamento, pre-processing e tokenizzazione del dataset**: qui il dataset di allenamento viene caricato e se ne estrae una sottosezione. Viene in seguito processato prima di essere usato dal modello;
- **Fine-tuning e salvataggio del modello**: la fase di addestramento del modello stesso, seguita dal suo salvataggio e/o caricamento (via *GitHub* o *Drive*);
- **Valutazione e salvataggio dei risultati**: viene caricato un modello di base per la generazione di codice, ed entrambi i modelli vengono valutati utilizzando il benchmark HumanEval. I risultati possono essere salvati via json;
- **Confronto tra modelli - "Complex Checking"**: si effettua il confronto del codice generato dal modello fine-tuned e dal modello di confronto pre-addestrato. Per mostrare come il fine-tuning ha migliorato il modello iniziale di base, viene effettuato anche il confronto con tale versione non-finetuned del modello iniziale.

2. Dataset di allenamento

Per il **fine-tuning del modello** di base, è stato utilizzato il dataset

"`bigcode/self-oss-instruct-sc2-exec-filter-50k`", disponibile su Hugging Face. Questo dataset è stato progettato per l'addestramento di modelli in grado di generare codice a partire da istruzioni in linguaggio naturale. Ogni esempio contiene diversi campi, e tra essi, i principali sono:

- **prompt:** un prompt strutturato che guida il modello nella generazione della risposta; include istruzioni su come ragionare, come strutturare la soluzione, come spiegare e come testare il codice risultante;
- **seed:** un frammento di codice o contesto tecnico relativo al task, usato come riferimento per generare una risposta coerente;
- **instruction:** una descrizione in linguaggio naturale del task da svolgere;
- **response:** la risposta data. All'interno di tale risposta, troviamo la porzione di codice python che implementa quanto detto dall'*instruction*;

Trattandosi di un dataset piuttosto ampio, abbiamo deciso di selezionare solo un **sottoinsieme casuale** di *4000 esempi*, per contenere i tempi di training ed evitare problemi di memoria (OOM), rispettando i limiti computazionali della nostra GPU su Google Colab.

2.1. Preprocessing e Tokenizzazione

Il preprocessing del dataset ha avuto un ruolo fondamentale nel migliorare le performance del fine-tuning. Da ogni esempio del dataset sono stati estratti i campi *instruction* e *response*.

Inoltre, per far sì che il nostro modello generi solo codice, e non risponda in linguaggio naturale alle istruzioni date, dalla *response* è stata **ritagliata in ogni esempio la sola funzione di implementazione**, scartando il resto. Questo è stato effettuato con la funzione `extract_code_from_response()`.

All'interno della funzione `preprocess_function()`, questi due campi sono stati **tokenizzati** separatamente senza token speciali, e **concatenati** con un "`\n`" tra essi.

Abbiamo anche deciso di **escludere la parte di istruzione dai target di allenamento** per evitare che imparasse anche i prompt dati e concentrarsi esclusivamente sull'apprendimento della generazione del codice, come suggerito nei metodi standard di HuggingFace per il *code generation fine-tuning*. Questo è stato effettuato mascherando i rispettivi token nella *label* (la label mask standard di *HuggingFace* è -100).

La lunghezza dei campioni estratti dal dataset può variare di molto. Avendo misurato in precedenza la lunghezza in token del nostro subset di allenamento, abbiamo deciso di utilizzare una lunghezza massima (`max_seq_length`) pari a *1024 token*, in quanto solo lo 0,1% dei campioni supera tale lunghezza.

Nel preprocessing è stato implementato un meccanismo di **troncamento** degli input troppo lunghi, nonché uno di **padding** per quelli più corti, in modo da avere input di lunghezza uniforme.

Infine, ci siamo assicurati una gestione corretta della **attention mask**, in modo che il modello ignorasse i token di padding durante l'allenamento.

3. Processo di Fine-Tuning

Per evitare errori di *out-of-memory (OOM)*, abbiamo **rimosso la validazione** durante il fine-tuning. Il dataset è stato quindi interamente utilizzato per il **training**, mentre la valutazione è stata effettuata **esclusivamente tramite HumanEval**, come da traccia.

Una volta caricate le librerie ed eseguita la cella degli iperparametri, viene **caricato il modello base** e il suo tokenizer da HuggingFace, configurato con `BitsAndBytesConfig` in modo da quantizzare i pesi e permettere di caricare questi modelli relativamente pesanti su Google Colab.

Nel caso in cui ci fosse necessità di *ripartire da un checkpoint*, c'è anche l'opzione di caricare il modello da tale checkpoint e riprendere l'allenamento (ma questa funzionalità non è stata usata per il modello finale usato nella validazione).

In seguito, vengono applicate le configurazioni finali al modello caricato, al fine di ridurre ulteriormente il consumo di memoria. Esse includono l'attivazione del **gradient checkpointing**, e il set-up di **LoRA (Low-Rank Adaptation)**, tecnica di *parameter-efficient fine-tuning (peft)*.

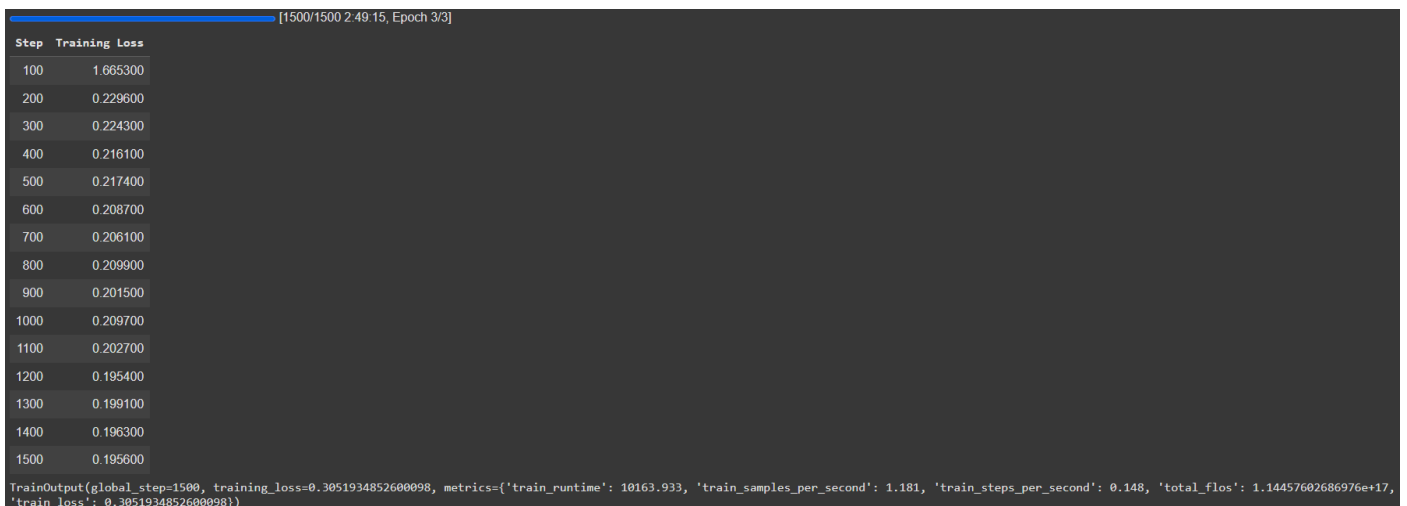
Conclusa la configurazione del modello e il pre-processing del dataset di addestramento, si procede a definire un **SFTTrainer** con tutti i parametri stabiliti, il nostro modello e il nostro dataset. A questo segue la fase di fine-tuning effettivo, tramite `".train()"`.

Con **3 epoche da 500 steps l'una**, l'allenamento ha impiegato circa 3 ore. Il modello è stato poi salvato e caricato direttamente su GitHub.

3.1. Risultati del Fine-Tuning

Il fine-tuning è stato ripetuto molte (...molte...) volte, la cui maggior parte risultante in modelli di scarsa qualità. Ad ogni iterazione sono stati identificati e risolti problemi (descritti nel capitolo [7.5](#)) che, lentamente, hanno portato alla versione attuale.

Qui è riportata un'immagine contenente l'intero processo finale di fine-tuning, come presente nel notebook:



| Step | Training Loss |
|------|---------------|
| 100 | 1.665300 |
| 200 | 0.229600 |
| 300 | 0.224300 |
| 400 | 0.216100 |
| 500 | 0.217400 |
| 600 | 0.208700 |
| 700 | 0.206100 |
| 800 | 0.209900 |
| 900 | 0.201500 |
| 1000 | 0.209700 |
| 1100 | 0.202700 |
| 1200 | 0.195400 |
| 1300 | 0.199100 |
| 1400 | 0.196300 |
| 1500 | 0.195600 |

```
TrainOutput(global_step=1500, training_loss=0.3051934852600098, metrics={'train_runtime': 10163.933, 'train_samples_per_second': 1.181, 'train_steps_per_second': 0.148, 'total_flos': 1.14457602686976e+17, 'train_loss': 0.3051934852600098})
```

4. Benchmarking e Valutazione

La valutazione del progetto si è concentrata sul confronto delle capacità di generazione di codice tra un modello pre-addestrato (**StarCoder2**) e la versione fine-tunata del nostro modello scelto (**DeepSeek-R1-Distill-Qwen-1.5B**).

Per testare le performance di tutti i modelli abbiamo utilizzato il dataset HumanEval, come consigliato nelle linee guida del progetto.

4.1. HumanEval: Cos'è e Come lo Abbiamo Utilizzato

HumanEval è un dataset di benchmark standardizzato, creato da OpenAI, specificamente progettato per valutare le capacità dei modelli linguistici di generare del codice Python funzionante a partire da docstring.

Nel progetto, HumanEval è stato utilizzato per testare le performance dei modelli. È stato caricato il dataset (`load_dataset("openai/openai_humaneval", split="test")`) e un campione casuale di problemi è stato selezionato per la valutazione.

Per ogni problema, è stato fornito il prompt sia a StarCoder2 che al modello fine-tunato, per generare il corpo della funzione. Il codice generato è stato poi raccolto per le analisi successive.

4.2. Adattamento dell'Input per Risultati Leggibili

Per guidare i modelli a produrre del codice pulito e pertinente, sono state adottate delle tecniche di **prompt engineering**.

In una prima fase di valutazione con HumanEval è stato aggiunto un **pre_prompt** agli input del modello. Questo serve a dare un contesto in più al modello, il quale altrimenti avrebbe solo il prompt del dataset di HumanEval.

Il prompt aggiuntivo è utile per guidare il modello ad *ottenere un output che sia più in linea con quello che ci aspettiamo da un generatore di codice*, principalmente riducendo la verbosità delle spiegazioni per cercare di ottenere quasi solo codice.

Successivamente, dopo aver analizzato le funzioni del dataset, ci siamo accorti che al loro interno hanno un “commento” che istruisce il modello sull’implementazione desiderata della funzione richiesta. Abbiamo utilizzato tale docstring (il “commento”) come prompt per testare il modello, e capire come l’output potesse variare di conseguenza.

I risultati della valutazione su HumanEval sono stati visualizzati tramite tabelle generate con `tabulate`, mostrando affiancati il *prompt*, il *codice generato dal modello non fine-tunato* e quello *generato dal modello fine-tunato*.

5. Risultati e Confronti

Sono stati condotti dei test qualitativi che hanno tenuto in considerazione i seguenti parametri:

- **Correttezza della sintassi:** La funzione `is_valid_python(code: str) -> bool` ha il compito di verificare se il codice prodotto sia *sintatticamente valido*. Per farlo, sfrutta il modulo `ast` (Abstract Syntax Tree) di Python, che consente di effettuare il parsing del codice sorgente per analizzarne la struttura sintattica: il comando `ast.parse(code)`, trasforma la stringa di codice in un *albero sintattico*. Se il parsing va a buon fine, viene restituito il valore booleano `True`, a indicare che il codice può essere interpretato senza errori dal compilatore Python. Al contrario, se il codice presenta errori formali (come indentazioni errate, parentesi non chiuse, simboli non validi o altri problemi sintattici) la funzione restituisce `False`.
La somma di tutti i valori `True` fratto il numero delle funzioni restituisce la percentuale di generazioni sintatticamente corrette e quindi un valore compreso tra 0 e 1;
- **Errori/warning pylint:** La funzione `run_pylint_analysis(code: str) -> dict` permette di condurre un'analisi statica del codice tramite lo strumento **Pylint**. Il codice, ricevuto come stringa, viene salvato temporaneamente in un file `.py` per essere analizzato da Pylint, che ne calcola la presenza di *errori* o *warning*. Il punteggio assegnato, in questo caso per entrambe le metriche, è ≥ 0 . I valori possono essere > 1 , in quanto è ritornata la media del numero di errori/warning per il numero di funzioni;
- **Similarità del codice:** La funzione `code_similarity(code1: str, code2: str) -> float` è pensata per misurare il grado di somiglianza tra il codice generato dal modello e una soluzione di riferimento (presa dal dataset HumanEval nel campo `canonical_solution`). Il calcolo si basa sulla *similarità tra insiemi di token*: il testo viene segmentato in parole chiave, nomi di variabili, numeri, operatori e altri elementi sintattici, e successivamente viene calcolato il rapporto tra l'intersezione e l'unione di tali token (indice di Jaccard). Questo produce un valore numerico compreso tra 0 e 1: un valore prossimo a 1 indica che i due frammenti di codice condividono gran parte del contenuto lessicale, mentre un valore prossimo a 0 segnala maggior diversità tra i due;
- **Complessità ciclomatica:** La funzione `calculate_complexity(code: str) -> float` serve a misurare la complessità ciclomatica del codice, utilizzando la libreria **radon**. La complessità ciclomatica è una metrica strutturale che quantifica il numero di percorsi logici distinti (strutture di controllo come *if*, *while*, *for*, *try/except*, *match...*) che possono essere seguiti in un programma. Viene utilizzata `cc_visit`, che restituisce una lista di funzioni contenute nell'output del modello (solitamente una sola) assieme a un campo `complexity` che ne indica il numero di percorsi logici. Infine la funzione ritorna la media di tali campi. Tale valore è pari a 1 per del codice molto semplice (costituito ad esempio da una sola istruzione *return*), ma la funzione restituisce 0 nel caso in cui non sia identificata nessuna funzione (come il caso in cui manchi un *def*). Il risultato ritornato è un valore ≥ 0 , corrispondente alla media di tali complessità associate ai vari output testati.

Nonostante valori bassi di complessità ciclomatica, generalmente, indicano un codice più efficiente, va tenuto in considerazione che, a causa del comportamento di `cc_visit` che restituisce 0 nel caso in cui non individua nessun *“def”*, questo valore può scendere drasticamente, come nel caso del modello base che ritorna spesso soluzioni mal formattate (e quindi senza un *def*), facendo credere che il modello abbia agito efficientemente. È bene quindi interpretare questa metrica tenendo in considerazione anche tutte le altre (soprattutto la *correttezza sintattica* e la *completezza dell'implementazione*).

- **Completezza dell'implementazione:** La funzione `is_meaningfully_implemented(code: Union[str, ast.AST]) -> bool` valuta se la funzione generata contenga una logica implementativa non banale. L'analisi si basa sull'albero sintattico del codice: la funzione percorre l'AST alla ricerca di istruzioni *return con valore*, *yield* oppure *raise*. Se almeno uno di questi costrutti è presente, viene restituito *True*; se invece la funzione è vuota o contiene solo istruzioni segnaposto (come *pass*, *return None* o blocchi non implementati), il risultato sarà *False*. Il valore restituito è quindi la media di questi valori booleani sul nostro insieme di risultati, ergo è compreso tra 0 ed 1.

Tutte queste metriche vengono confrontate tra il modello scelto 'base', la sua versione fine-tunata, e il modello pre-addestrato per il confronto (StarCoder2). Questo viene effettuato su due prompt diversi (spiegati nella [sezione 4.2](#)).

Di seguito, mostriamo i risultati ottenuti su un **campione di 30 funzioni**:

- **pre-prompt + funzione**

| Mettrica | Base | Starcoder | Fine-Tuned |
|--------------------------|------|-----------|------------|
| Sintatticamente Corretto | 0.03 | 0.30 | 0.20 |
| Errori pylint | 0.97 | 0.87 | 0.90 |
| Warning pylint | 0.00 | 0.97 | 0.17 |
| Similarità del codice | 0.04 | 0.05 | 0.36 |
| Complessità Ciclomantica | 0.13 | 0.67 | 0.65 |
| Implementazione Completa | 0.03 | 0.07 | 0.17 |

- docstring da sola

| Mettrica | Base | Starcoder | Fine-Tuned |
|--------------------------|------|-----------|------------|
| Sintatticamente Corretto | 0.03 | 0.10 | 0.47 |
| Errori pylint | 0.97 | 0.90 | 0.63 |
| Warning pylint | 0.00 | 0.17 | 0.07 |
| Similarità del codice | 0.03 | 0.07 | 0.20 |
| Complessità Ciclomatica | 0.10 | 0.37 | 1.27 |
| Implementazione Completa | 0.00 | 0.07 | 0.47 |

I risultati ottenuti ci dicono che l'utilizzo di un *prompt* caratterizzato dalla sola docstring di ciascuna funzione del dataset di HumanEval permette al nostro modello fine-tunato di performare meglio rispetto ad un approccio basato sull'utilizzo di un *pre-prompt* seguito dalla funzione del dataset.

Attribuiamo questo miglioramento alla somiglianza tra la docstring della funzione e il modo in cui il modello è stato fine-tunato (cioè tramite il campo *instruction* del dataset di allenamento). Nonostante infatti un valore leggermente più alto nella similarità del codice per quanto riguarda il prompt completo (pre-prompt + funzione), gli altri parametri sono a favore del prompt caratterizzato solo dalla docstring.

Basandoci principalmente sui risultati ottenuti dall'uso della sola docstring, possiamo concludere che **il processo di fine-tuning ha migliorato di molto la qualità del codice generato dal modello di DeepSeek base**. Anche le metriche associate a StarCoder2 sono inferiori rispetto a quelle del modello fine-tuned, quindi possiamo dire che **il codice generato da quest'ultimo è qualitativamente migliore rispetto a quello generato da un modello pre-addestrato** a tal fine.

6. Considerazioni sulle Scelte Progettuali

I parametri utilizzati per il fine-tuning sono stati scelti con un bilanciamento tra **esigenze tecniche** (rientrare nelle limitazioni di tempo e memoria su Google Colab) e semplici **osservazioni effettuate su molteplici test** nelle prime fasi di sperimentazione, favorendo valori che minimizzano la loss (o, in mancanza di cambiamenti sostanziali, mantenendo il parametro al suo valore di default o a un valore consigliato dal web).

L'utilizzo della **quantizzazione a 4 bit** tramite bitsandbytes è stato fondamentale per poter caricare un modello da oltre un miliardo di parametri senza esaurire la memoria GPU.

L'approccio **QLoRA**, combinato con l'abilitazione del **gradient checkpointing** e di una **batch size** ridotta (2, con un aumento virtuale tramite `gradient_accumulation_steps = 4`), ha permesso un notevole risparmio di memoria durante il backpropagation, a discapito di un *rallentamento del training*.

La **sequenza massima dei token in input** (`max_seq_length`) è stata impostata a 1024: come già detto, questo fa in modo che quasi tutti gli input vengono passati senza problemi (solo lo 0,1% di campioni che richiede un troncamento, in cui la lunghezza massima è 1340 token).

Infine, il **numero di epoche** è stato impostato a 3, in quanto questo permette di avere un allenamento completo senza interruzioni dai limiti di tempo di Google Colab. **Ogni epoca viene eseguita nella sua interezza** grazie a `max_steps = -1`.

L'uso di `logging_steps = 100` permette un **monitoraggio relativamente accurato** rispetto alla durata dell'allenamento, senza appesantirlo troppo.

Per quanto riguarda la scelta dei parametri del processo di **generazione codice** per la sua valutazione, questi sono i principali utilizzati:

- **max_new_tokens** (int)
- **do_sample** (boolean)
- **repetition_penalty**(float)

Il parametro `max_new_tokens` specifica il numero massimo di nuovi token che il modello può generare a partire dall'input fornito. Lo scopo è impedire al modello di generare degli output che siano troppo lunghi o fuori controllo. Per questo motivo abbiamo eseguito svariati test cambiando il suo valore, e il valore ritenuto adatto alle nostre esigenze è stato **500**. Questo è dovuto al fatto che la lunghezza tipica del corpo di una funzione di HumanEval è relativamente breve, ergo 500 token sono generalmente sufficienti per implementare il genere di funzioni richieste dal dataset senza troncature il contenuto; un valore troppo basso avrebbe invece troncato il codice;

Il parametro `do_sample`, se impostato a True, attiva la *campionatura probabilistica* nella generazione del testo, introducendo **casualità**. Se False, il modello sceglie sempre il token più probabile (*greedy decoding*).

La campionatura probabilistica cambia il modo nel quale un modello sceglie il prossimo token. Se essa è attiva, allora al momento della scelta del prossimo token, viene calcolata una distribuzione di probabilità su tutti i token possibili, basata sul contesto (di input e token precedenti).

Attivare la campionatura quindi porterebbe a *generazioni più differenti tra loro e soluzioni meno ovvie* (anche se meno probabili). È utile se vogliamo *evitare di ricevere sempre lo stesso output dal modello per lo stesso prompt* e per analizzare come il modello si adatta alle istruzioni.

Il valore scelto da noi è **False**. Questa è una scelta standard nei modelli di generazione di codice, in quanto vogliamo più determinismo e precisione. Non ci interessa che il modello sia creativo poichè produciamo un solo output, quindi non abbiamo bisogno che la generazione per uno stesso prompt cambi. Inoltre la creatività può portare a volte a scelte peggiori nel caso di generazione codice.

Il parametro `repetition_penalty` viene utilizzato per penalizzare la ripetizione di token già generati, con lo scopo di evitare output ripetitivi o ridondanti. Senza alcuna penalità, alcuni modelli tendono a ripetere porzioni di codice o intere istruzioni, specialmente quando il prompt è breve o poco vincolante.

Il valore standard è 1.0, che corrisponde all'assenza di penalità. Impostando un valore superiore a 1, ad esempio 1.1, il modello è incentivato a scegliere token diversi da quelli già prodotti, riducendo il rischio di generare costrutti ripetitivi.

Abbiamo scelto **1.1** come compromesso efficace: un valore troppo alto avrebbe potuto penalizzare anche la ripetizione legittima di costrutti necessari nel codice (come nomi di variabili o istruzioni comuni), mentre 1.1 ha mostrato buoni risultati nel limitare la ripetitività senza compromettere la correttezza sintattica e semantica del codice generato.

Il parametro `temperature` controlla quanto il modello è "creativo" o quanto è "deterministico" durante la generazione, condizionando il livello di casualità introdotto dal `do_sample`.

Temperature basse, vicine allo 0, producono un output più prevedibile e token più probabili. Temperature alte invece, vicine a 1 o anche oltre, producono un output più vario e creativo, ma con più rischio di incorrere in incoerenze.

Questo parametro è stato sottoposto a numerosi test durante lo sviluppo. Dopo diverse prove, abbiamo riscontrato che un approccio più deterministico produceva risultati più precisi e affidabili nella generazione di codice.

Per questo motivo, abbiamo deciso di impostare `do_sample=False`, disattivando così la campionatura probabilistica, e di conseguenza abbiamo rimosso anche il parametro `temperature`, che ha effetto solo quando `do_sample` è attivo.

7. Difficoltà Incontrate e Soluzioni

Nella fase iniziale del progetto, dedicata alla creazione e configurazione di un prototipo funzionante, sono state affrontate sfide tecniche e architetturali, di seguito elencate.

7.1 Stabilità dell'Ambiente di Sviluppo e Gestione delle Dipendenze

Durante l'allestimento dell'ambiente di sviluppo su Google Colab, la prima area di attenzione è stata la *gestione delle librerie Python*.

Frequentemente, dopo l'installazione o l'aggiornamento di pacchetti tramite `!pip install`, il runtime Python **non recepiva immediatamente le modifiche** per la libreria `datasets`.

Per garantire che le versioni corrette delle librerie fossero caricate e per evitare errori imprevisti, si è reso necessario *riavviare il runtime di Colab ogni volta che si installano le librerie*.

7.2 Configurazione e Debug del Processo di Fine-Tuning Iniziale

Per ottimizzare l'uso della memoria, il **Gradient Checkpointing** è stato abilitato sin da subito. Tuttavia, **la sua attivazione ha generato errori CUDA** non specifici che si sono rivelati difficili da risolvere nonostante ore e ore di ricerche e tentativi. Per non bloccare lo

sviluppo complessivo del pipeline, esso è stato disattivato, pur consapevole dei benefici persi.

È emerso in seguito che, con l'evoluzione del codice e delle configurazioni da parte dei vari membri del team, l'attivazione del Gradient checkpointing non ha generato alcun errore, suggerendo che le difficoltà iniziali potessero derivare da interazioni specifiche dell'ambiente in quel momento.

Oltre alla questione del gradient checkpointing, il percorso per ottenere un primo modello fine-tunato e salvabile ha implicato la risoluzione di svariati problemi minori. Questi spaziavano dalla *corretta configurazione dei TrainingArguments e dei percorsi di salvataggio*, all'assicurarsi della *piena compatibilità del formato del dataset con le aspettative di SFTTrainer*. Questo ha richiesto un **debugging iterativo** e la consultazione di documentazione e vari esempi, ma niente fuori dall'ordinario.

7.3 Tentativo di Fine-Tuning su Hardware Locale

Per aggirare i limiti di tempo delle sessioni GPU su Google Colab e sfruttare una configurazione hardware domestica decentemente performante, è stato investito tempo nel tentativo di **replicare l'ambiente di fine-tuning localmente**.

L'intento era di poter condurre sessioni di addestramento più lunghe e intensive invece di essere imitati dalle circa 2 ore giornaliere che Colab mantiene per il runtime.

Nonostante l'installazione dei driver più recenti, la configurazione di ambienti virtuali e l'adesione alle guide ufficiali, sono stati riscontrati errori CUDA generici e di difficile interpretazione che impedivano l'avvio del training.

Dopo numerose ore dedicate al troubleshooting, che includono la *reinstallazione di driver, toolkit CUDA e diverse versioni di PyTorch*, e data la persistenza di questi problemi e altri problemi di compatibilità in vista, è stato deciso di accantonare l'idea e concentrarsi sull'**ottimizzazione dell'uso delle risorse su Colab**.

7.4 Impostazione della Pipeline di Valutazione Avanzata

La costruzione della pipeline di valutazione, in particolare per le metriche più complesse ("Complex checking"), ha presentato le sue sfide.

L'integrazione di *Pylint* per l'**analisi statica del codice** generato e la manipolazione degli **Abstract Syntax Trees (AST)** con le librerie `ast` e `zss` (per la *Tree Edit Distance*) non sono state immediate, ma nessun problema irrisolvibile.

I test iniziali di queste componenti sono stati eseguiti utilizzando *frammenti di codice semplici e controllati*, dato che i modelli, nelle prime fasi, non producevano ancora output di qualità sufficiente per testare appieno la robustezza della pipeline di valutazione, quindi l'iniziale qualità è dubbia.

Le funzioni di estrazione della funzione da codice generato sono dubbie in sé, in quanto estraggono e valutano solo la prima funzione che trovano, non controllando se sono state generate funzioni d'aiuto. Questo è stato considerato tuttavia un non-problema, in quanto queste funzioni *raramente vengono aggiunte, sono generalmente più semplici della principale, e sarebbero difficili da includere nella valutazione*.

È stato esplorato anche l'utilizzo della **metrica Pass@k**, che misura la capacità del modello di generare codice funzionante. Questo approccio richiede la generazione di molteplici output per lo stesso prompt e la successiva esecuzione di test unitari su di essi.

Un primo tentativo con Pass@1 ha evidenziato la bassa qualità del codice prodotto inizialmente da entrambi i modelli, risultando in *metriche poco significative* (tutti i test fallivano). Questo forse sarebbe migliorato con l'uso di un modello fine-tunato adeguatamente, ma ci sarebbero stati seri dubbi sulle informazioni che potevano essere ottenute da esso.

L'estensione a un k più elevato (es. k=100) avrebbe richiesto tempi di generazione di codice eccessivamente lunghi. Inoltre, si è presentata una **riflessione sulla sicurezza**:

l'esecuzione di codice generato automaticamente comporta rischi. Sebbene HumanEval sia un dataset noto e innocuo e il codice generato fosse qualitativamente modesto (e quindi con bassa probabilità di essere dannoso), la configurazione di un ambiente di esecuzione completamente isolato (sandbox o VM) è stata considerata una complicazione eccessiva rispetto al valore aggiunto che Pass@k avrebbe potuto offrire in quella specifica fase e con le priorità del progetto.

Di conseguenza, si è deciso di **non proseguire con l'uso di questa metrica**, privilegiando *analisi statiche e strutturali*.

7.5 Iterazioni e Sviluppi nel Processo di Fine-Tuning

Una volta superate le fasi iniziali di configurazione, il processo di fine-tuning vero e proprio ha presentato una serie di sfide e ripensamenti significativi per i membri del progetto.

Inizialmente, vi è stata un'interpretazione della traccia erronea che ha portato a concentrarsi sul **fine-tuning del modello StarCoder2** al posto di partire da un modello di NLP generico, con l'idea di valutarlo successivamente confrontandolo con la sua versione base.

Durante questa fase, monitorando la metrica di errore (loss), siamo riusciti a raggiungere valori molto bassi, il che sembrava promettente.

Nonostante questi bassi valori di loss, una volta esaminato l'output generato dal modello StarCoder2 fine-tunato in risposta ai prompt di benchmark, è emerso un grosso problema: il modello tendeva semplicemente a **restituire il prompt stesso** o parti di esso, piuttosto che generare codice.

Un'analisi più approfondita ha rivelato che l'errore risiedeva nella *selezione delle colonne del dataset* utilizzato per il fine-tuning. In pratica, si stava fornendo al modello la colonna di *instruction* e di *prompt*, col quale il modello imparava a rispondere sempre col prompt specificato dal dataset di addestramento, il che spiega la drastica riduzione della loss ma l'inefficacia nella generazione effettiva di codice.

La selezione delle colonne è stata quindi rivista. In un primo momento si è tentata una correzione a *response*, ma successivamente si è optato per una **combinazione delle colonne instruction e seed**.

Questa scelta è stata preferita rispetto all'utilizzo della colonna *response*, poiché quest'ultima conteneva spiegazioni testuali estese e commenti ridondanti che non erano ideali per addestrare il modello a generare codice conciso.

Alcune iterazioni dopo, ci si è resi conto che il codice generato era poco sensato e spesso assente (tramite l'uso della primitiva `pass` di Python), e ci si è resi conto che il codice contenuto in *seed* ha poco e niente a che fare con la funzione richiesta da *instruction*. Perciò, si è deciso di risolvere il problema utilizzando *response*, ma **ritagliando** da essa **solo la funzione**.

Risolto il problema della selezione delle colonne, è emersa una nuova consapevolezza: l'obiettivo non era necessariamente fine-tunare StarCoder2, ma confrontare con StarCoder2 un modello generale a nostra scelta fine-tunato.

Questo ha portato a una fase di ricerca per un modello alternativo. La scelta è ricaduta su *deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B*.

Deepseek è stato scelto per l'ottimo rapporto tra qualità delle prestazioni e dimensioni contenute. Inoltre, la sua dimensione ridotta rispetto a StarCoder2 (da 3B a 1.5B di parametri) permette di gestire meglio le risorse computazionali disponibili.

L'integrazione del modello DeepSeek nel pipeline di fine-tuning esistente si è rivelata complessa. Il team ha affrontato diversi *errori CUDA generici* e di difficile interpretazione, assieme a *warning superflui* causati da errori nelle librerie.

Una volta superati gli errori CUDA e avviato il fine-tuning con DeepSeek, l'output generato dal modello risultava illeggibile o privo di senso.

Dopo ulteriori indagini, si è scoperto che il problema risiedeva nel tokenizer: si stava ancora utilizzando il tokenizer configurato per StarCoder2, incompatibile con il nuovo modello.

La correzione di questo **disallineamento tra modello e tokenizer** è stata l'ultimo passo cruciale per ottenere output sensati dal modello DeepSeek fine-tunato.

Queste sfide nel processo di fine-tuning evidenziano la natura *iterativa e non lineare* dello sviluppo di modelli di machine learning (e della stesura di codice in generale), dove la risoluzione di un problema può portare alla luce nuovi quesiti e nuovi problemi da affrontare.

8. Conclusioni

Questo progetto ci ha dato l'occasione di applicare le conoscenze acquisite durante il corso. La sfida che esso ha presentato è stata ben diversa da quelle che abbiamo affrontato in altri progetti, ma ci ha permesso di comprendere meglio il funzionamento del fine-tuning e delle meccaniche interne ai *Large Language Models* (LLMs).

Lo svolgimento di questo progetto ha richiesto grandi sforzi iniziali per il setup corretto delle pipeline di fine-tuning, validazione e confronto. Ma una volta risolti questi problemi, ci siamo resi conto che si possono implementare modifiche e migliorie facilmente, e che il codice prodotto è in realtà contenuto. Tuttavia, ricevere un feedback per ogni modifica effettuata non è immediato, in quanto l'implementazione delle modifiche e il loro testing richiedono una considerevole quantità di tempo.

L'esperienza è stata di nostro gradimento; nonostante il lavoro realizzato sia certamente migliorabile, siamo soddisfatti di esso e delle conoscenze che il suo svolgimento ha impartito, e ci sentiamo molto più preparati a lavorare su progetti simili in futuro.

9. Appendice

Link al repository:

<https://github.com/matteraggi/FineTuningAI>

Tensori dei modelli fine-tunati:

<https://github.com/matteraggi/FineTuningAI/tree/main/models>

Tensori del miglior modello fine-tunato:

https://github.com/matteraggi/FineTuningAI/tree/main/models/DS_Finetuned_6