Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Metodologie di attacco ai Large Language Models

Relatore:
Chiar.mo Prof.
ANDREA ASPERTI

Presentata da:
ENRICO FERRAIOLO

II Sessione
Anno Accademico 2023-2024

A mio fratello
Francesco

Abstract

Negli ultimi anni, i *Large Language Model* (LLM) hanno rappresentato una rivoluzione nell'elaborazione del linguaggio naturale grazie alle loro capacità generative e alla comprensione del contesto. Tuttavia questi modelli sono vulnerabili a diversi tipi di attacchi che possono comprometterne la sicurezza e l'affidabilità.
Questo studio vuole formalizzare il concetto degli attacchi ai *Large Language Model* dando una caratterizzazione matematica al problema.
Gli esperimenti condotti dimostrano la facilità con cui tali attacchi possono manipolare il comportamento del modello, evidenziando rischi per l'integrità e la sicurezza per i proprietari e gli utilizzatori.
Questa tesi, inoltre, affronta ed esplora le principali metodologie di attacco ai LLM sperimentando approfonditamente le tecniche di *Prompt Injection* e *Data Poisoning*.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Negli ultimi anni i *Large Language Model* (LLM) sono stati una delle più grandi innovazioni nel campo del *Natural Language Processing* (NLP). Questi modelli linguistici sono basati su architetture come i *Transformer*. Sono in grado di elaborare e generare testo in linguaggio naturale simulando un dialogo umano. Grazie alle loro forti capacità di analisi di contesti complessi e di risposta coerente alla conversazione i LLM sono stati adottati in numerosi settori e campi, ciò però implica anche una maggiore preoccupazione in ambito di sicurezza poichè aumenta il rischio di *fooling*.

Il concetto di *fooling* fa riferimento alla capacità di indurre un modello a comportarsi in modo inatteso producendo output errati, incoerenti o addirittura pericolosi. Non è banale la differenza tra *fooling* sulle *Deep Neural Network* (DNN) e *fooling* sui LLM, infatti il primo si basa principalmente sulla manipolazione dell'input del modello di classificazione per ottenere una risposta erroneamente classificata, sfruttando le vulnerabilità legate alla rappresentazione delle *feature*. Fare *fooling* su *Large Language Model*, invece, il problema è decisamente più complicato e questo tema verrà approfondito nel corso della tesi.
La natura del problema appena descritto lascia la strada aperta a numerose forme diverse di attacco, a seconda degli obiettivi e delle modalità di accesso al modello. Questo lavoro tenta di dare una formalizzazione al problema cercando di generalizzare un attacco a un modello linguistico.

Per comprendere a pieno l'impatto che hanno sulla realtà gli attacchi si è rivelato molto utile condurre esperimenti pratici. Questi sono stati progetti e realizzati per verificare le tecniche di *fooling* descritte in questo lavoro.

È quindi chiaro che questa tesi non cerca solo di analizzare la teoria alla

base degli attacchi e darne formalizzazione matematica, ma anche di tradurre tali concetti in esperimenti pratici cercando di esplorare le implicazioni di sicurezza per i *Large Language Model*.

Capitolo 2

Large Language Model

2.1 Introduzione ai LLM

I *Language Model* (LM) sono modelli linguistici, basati sul *machine learning*, che vengono addestrati su quantità enormi di dati e per questo vengono chiamati *Large Languge Model*, essi generano ed elaborano testo in linguaggio naturale.
Negli ultimi tempi il campo del *Natural Language Processing* (NLP) si è evoluto con una rapidità molto elevata, questo grazie a tutte le ricerche fatte e i progressi ottenuti in ambito scientifico e tecnologico. Il NLP è un'area di studio che fa convergere informatica, intelligenza artificiale e linguistica, l'obiettivo è quello di far elaborare a una macchina del testo in linguaggio naturale, "capirne" il significato e coglierne le sfumature emotive [4].
In passato i LLM erano costruiti attraverso le *Recurrent Neural Network*, (RNN) più precisamente mediante un'architettura nota come *Long short-term memory* (LSTM) il cui obiettivo è quello di ridurre il problema della scomparsa del gradiente presente nelle RNN. Al giorno d'oggi lo stato dell'arte è diverso, infatti troviamo che i modelli più performanti sono quelli basati sui *Transformer* [27].

2.2 Transformer

Un *Transformer*, di norma, è composto da un *encoder* e un *decoder*, figura 2.1, entrambe le parti sono composte da *layer* fortemente connessi. Esso prende in input del testo in linguaggio naturale e lo rielabora trasformandolo in una rappresentazione numerica composta da sequenze di token.

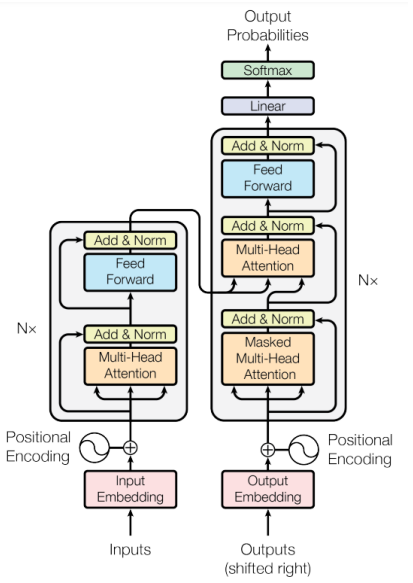

Figura 2.1: L'architettura di un *Transformer*. Immagine presa da [27].

I *Transformer* basano la loro forza sulla *Self-Attention*, questo è un meccanismo che permette di creare collegamenti globali tra l'input e l'output, ciò ha permesso di liberarsi dall'uso delle *recurrent unit* presenti nelle RNN. Queste unità fungevano da memoria il quale valore riceveva un aggiornamento per ogni passo eseguito durante l'apprendimento della rete, questa modalità di aggiornamento dei valori è fortemente inefficiente rispetto alla tecnica della *Self-Attention*.

2.2.1 Funzionamento e Self-Attention

La *Self-Attention* è il meccanismo chiave per il funzionamento dell'architettura *Transformer*. Questa tecnologia consente al modello di considerare ed esaminare contemporaneamente l'intera sequenza di token ricevuta in input, contrariamente a quanto fanno le RNN, che elaborano i dati sequenzialmente. Inoltre *Self-Attention* riesce ad attribuire a ogni token un valore che rappresenta la rilevanza di questo all'interno dell'input.
Quando viene dato al *Transformer* un input, come ad esempio una frase in linguaggio naturale, questo lo prende e lo rappresenta come una sequenza di numeri. Quest'operazione si chiama *embedding*, ed è la pratica di associare ad ogni token un numero e creare un vettore di numeri, cosicché il modello riesca a lavorare effettivamente su dei numeri, interpretabili correttamente dalla rete, e non su del testo.

A questo punto possiamo calcolare l'*attention* attraverso un prodotto scalare come nella figura 2.2:

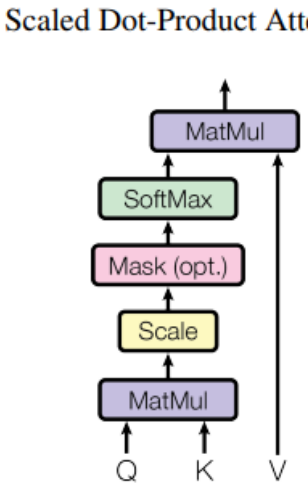

Figura 2.2: Calcolo dell'*attention* attraverso il prodotto scalare. Immagine presa da [27].

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

dove Q è la matrice delle query da elaborare, K è la matrice delle chiavi e V e la matrice dei valori.
Un grande vantaggio che implementano i *Transformer* è quello della parallelizzazione del calcolo dell'*attention*, infatti attraverso la *Multi-Head Attention* è possibile eseguire in parallelo più istanze della funzione ATTENTION e concatenarne i risultati, come nella figura 2.3. Ciò rende i *Transformer* incredibilmente più efficienti rispetto alle RNN.

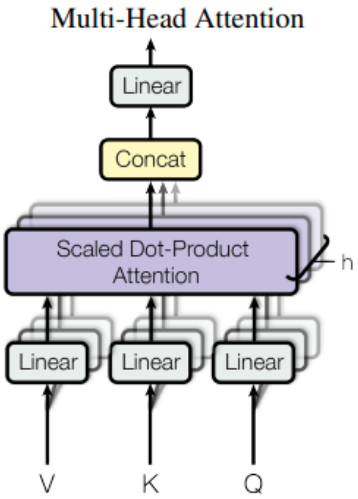

Figura 2.3: *Multi-Head Attention*. Immagine presa da [27].

2.3 A Che Cosa Servono i LLM

Lo stato dell'arte dei LLM include diversi modelli, alcuni sono *open source* altri invece no. Tra i modelli più popolari e performanti troviamo:

- ChatGPT 4o [21]
- LLaMa 3 [7]
- Mistral Large 2 [2]
- Claude 3.5 Sonnet [3]

Questi modelli sono altamente potenti e in grado di elaborare e generare testo in linguaggio naturale, molti sono anche multi-modali: ovvero riescono a ricevere degli input diversi dal semplice linguaggio naturale, come ad esempio immagini, video oppure altri tipi di documenti. Il grande punto di forza dei *Large Language Model* risiede nel cogliere le varie sfaccettature che rendono unico ogni tipo di input, capirne il contesto ed elaborare una risposta coerente con le richieste. Tutto ciò rende fortemente utili i LLM, infatti nell'ultimo periodo hanno raggiunto lo status di assistenti virtuali i quali riescono a completare con successo quasi tutte le richieste di un utente medio che si interfaccia a essi. Esistono però delle tecniche di *fooling* che permettono di compromettere l'affidabilità e la sicurezza di questi modelli, anche dei più avanzati. Bisogna perciò sapere come usarli e i proprietari di questi modelli

devono essere in grado di difendersi dai pericoli che ne derivano per loro e per i loro clienti.

Capitolo 3

Fooling di LLM

3.1 Cosa Vuole Dire Fare Fooling

Fare *fooling* di un LLM significa indurre il modello a produrre errori generando output non accurati, offensivi o pericolosi attraverso degli input architettati ad hoc per far cadere in inganno il modello linguistico. Tali attacchi sono possibili poichè si vanno ad attaccare le vulnerabilità del modello e la sua architettura. L'addestramento dei LLM avviene su enormi quantità di dati e questo comporta l'esposizione del modello in questione a informazioni particolarmente sensibili, contenuti offensivi e notizie false. Il modello quindi, nonostante sia stato addestrato su una notevole mole di dati utili, ha comunque imparato dei concetti che non dovrebbero essere esposti all'utente finale, è quindi chiaro il pericolo che ne concerne e che bisogna attivare delle tecniche di difesa da utenti malintenzionati e/o possibili fughe di dati dal modello mentre sta elaborando una risposta.
Ne consegue quindi che fare *fooling* non è solamente l'atto di sfruttare una debolezza del modello per il proprio interesse, ma anzi può voler stare a significare semplicemente portare il *Language Model* a non comportarsi nella modalità attesa.
Nel prossimo capitolo vediamo alcune tipologie di attacchi che si possono eseguire sui LLM.

3.2 Fooling

È fondamentale sottolineare che il *fooling* nei modelli generativi, e in particolar modo nei LLM, presenta sfide più complicate rispetto agli attacchi a delle reti di classificazione. In queste ultime la sfida del *fooling* è evidente poiché, spesso, l'obiettivo è quello di fare categorizzare in modo sbagliato,

possibilmente con alta fiducia, a un modello un certo input.
Nel contesto dei *Large Language Model* , invece, il *fooling* è ancora più complesso. I modelli linguistici, infatti, non devono semplicemente classificare l'input, ma bensì devono generare testo in un dominio enorme. È quindi chiaro che il *fooling* di un LLM non si limita solamente alla produzione di output errati, ma si espande anche al far generare testo al modello che risulta pericoloso oppure incoerente rispetto al contesto in questione. L'attacco dovrà anche tenere quindi conto della comprensione semantica e pragmatica che il modello ha appresso.
Nonostante la loro natura generativa, il *fooling* di LLM risulta più complicato rispetto a quello dei modelli di classificazione tradizionali, questo perché gli ultimi si limitano a produrre output senza tener conto delle difficoltà linguistiche a cui si affacciano i LLM, essi infatti si trovano a dover affrontare sfide come ambiguità, contesto e coerenza narrativa durante il discorso.

Capitolo 4

Attacco

Prima di approfondire il significato di attacco a un *Large Language Model* vediamo come si possono attaccare i modelli di *machine learning*, in particolare poniamo la nostra attenzione sulle *Deep Neural Network*.

4.1 Attaccare le Deep Neural Network

4.1.1 Proprietà Contro-intuitive delle DNN

Szegedy et al. [24] hanno trovato delle proprietà contro-intuitive nelle *Deep Neural Network* che risultano particolarmente interessanti per quanto concerne il tema del *fooling* delle reti di *deep learning*, compresi i LLM.

Rilevazione delle Informazioni Semanticamente Significative

La prima proprietà scoperta riguarda il significato semantico delle unità individuali: non è il singolo neurone nell'ultimo strato a contenere informazioni semantiche significative, ma bensì l'intero spazio di attivazione.
È stato provato tramite un esperimento che proiezioni casuali delle attivazioni $\phi(x)$, dove $\phi(x)$ è la funzione di attivazione delle *feature* estratte dal modello per un input x, sono semanticamente indistinguibili dalle coordinate di $\phi(x)$. Questo mette in discussione l'idea che le reti neurali separino le caratteristiche nei singoli neuroni.

Perturbazione dell'Input

Un'altra proprietà riguarda la vulnerabilità a minime perturbazioni dell'input: cambiamenti impercettibili su un'immagine di input possono indurre

il modello a classificazioni completamente errate. Tali perturbazioni, ottimizzate per massimizzare l'errore predetto, vengono chiamate *adversarial example*.
Viene anche mostrato che tali *adversarial example* sono comuni ad altre reti, infatti una perturbazione progettata per una rete potrebbe ingannare anche altri modelli, addestrati con configurazioni diverse o addirittura su dataset differenti, sullo stesso input.
Un esempio pratico di questa proprietà lo possiamo notare nella figura 4.1 dove troviamo degli *adversarial examples* progettati per una *AlexNet*: sulla sinistra ci sono gli input originali, al centro la differenza tra l'immagine corretta e quella ostile mentre sulla colonna di destra c'è l'*adversarial example*. Tutte le immagini sulla destra sono classificate come "struzzo".

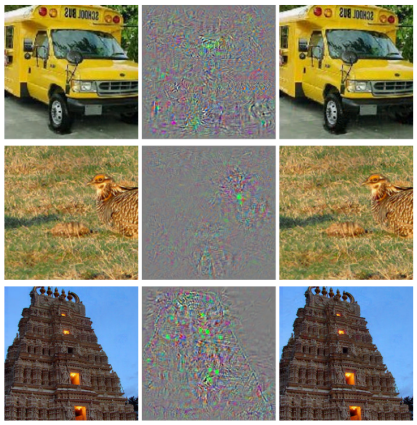

Figura 4.1: *Adversarial examples*. Immagine presa da [24].

4.1.2 Alta Confidenza nella Classificazione di Immagini Irriconoscibili

Nguyen et al. [20] presentano un articolo dove discutono del fatto che le DNN sono vulnerabili ad attacchi che portano a classificazioni errate e con alta fiducia per input completamente estranei e non riconoscibili da un occhio umano.

Creazione delle Fooling Image

Le *fooling image* sono immagini generate appositamente per portare la rete a classificare in modo errato l'input come un oggetto specifico.
Per la creazioni di queste immagini gli autori dell'articolo hanno utilizzato algoritmi evolutivi e tecniche come l'ascesa del gradiente. L'obiettivo era

quello di manipolare i pixel dell'immagine in modo tale da massimizzare la risposta della rete per una determinata classe. Vengono inoltre utilizzati due tipi di codifica per la generazione degli input ostili:

- **Codifica diretta**: ogni pixel è rappresentato da un valore e l'evoluzione avviene mutando tali valori pixel per pixel, producendo immagini irregolari.
- **Codifica indiretta**: viene usata una rete CPPN (*Compositional Pattern-Producing Network*). Ogni genoma influenza parti diverse dell'immagine, producendo risultati con una maggiore regolarià e coerenza visiva.

Per esempio, per la creazione delle *fooling image*, nella figura 4.2 viene usata la codifica diretta e nella figura 4.3 viene usata la codifica indiretta. La rete, in entrambi i casi, classifica le immagini come cifre da 0 a 9.

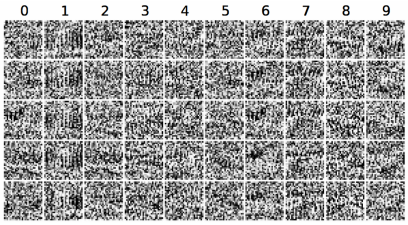

Figura 4.2: Codifica diretta. Immagine presa da [20].

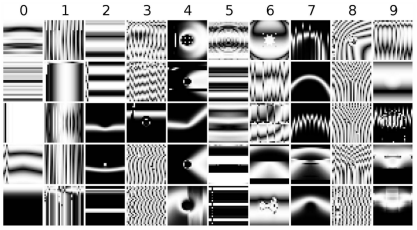

Figura 4.3: Codifica indiretta. Immagine presa da [20].

4.2 Cosa Vuole Dire Attaccare un LLM

Un attacco a un *Large Language Model* è un tentativo intenzionale di manipolare l'output generato dal modello, attraverso lo sfruttamento di vulnerabilità e debolezze al fine di ottenere risposte che soddisfino gli obiettivi malevoli oppure indesiderati voluti da un utente minaccioso.

Questi attacchi spaziano dall'elusione dei filtri di sicurezza della rete fino all'induzione di risposte che non sono coerenti con quelle aspettate.
Nei modelli di classificazione gli attacchi si concentrano tipicamente sull'induzione di errori nella classificazione della classe attraverso minime perturbazioni sull'input, affinché il modello associ erroneamente un input malevolo come appartenente a una classe benigna (ad esempio facendo associare un immagine di un veicolo come un animale). La classificazione di tale attacco è abbastanza semplice, infatti un attacco ha successo se l'output del classificatore risulta errato rispetto alla *ground truth*.
Nei modelli generativi, invece, la natura dell'output è per l'appunto di tipo generativa e questo rende notevolmente più complessa la misurazione del successo di un attacco. In questi modelli l'attaccante non si limita a forzare la classificazione erronea di una classe, ma invece cerca di influenzare l'output generato in modo tale che questo contenga informazioni che violino i criteri di sicurezza della rete oppure che non sarebbero state prodotte in quel preciso contesto. È quindi chiaro che la valutazione di successo di questo attacco non è immediata, poiché non può avvenire in modo totalmente oggettivo, infatti l'output generato varia in base al prompt, il contesto e le regole che deve seguire il modello. Di conseguenza per la valutazione dell'esito di un attacco a modelli generativi è spesso necessaria una revisione umana oppure un classificatore molto sofisticato e addestrato per riconoscere eventuali contenuti ostili o indesiderati.

4.3 Obiettivi dell'Attacco

A questo punto possiamo definire dei punti cruciali da tenere in considerazione quando si compie un attacco a un *Large Language Model*:

- **Violazione della sicurezza**: gli attacchi possono mirare a eludere i filtri di sicurezza del modello in modo tale da evaderli. Questi filtri servono per evitare risposte inappropriate da parte del LLM.
- **Compromissione dell'integrità**: un attacco potrebbe puntare a deteriorare l'affidabilità delle risposte del modello, inducendo l'utente in errore fornendo informazioni false o manipolate.
- **Sfruttamento delle vulnerabilità**: nella fase di attacco si cercano di scoprire falle architetturali nel modello, studiandone le risposte oppure analizzandone la struttura.

4.4 Formalizzazione di un Attacco

Per formalizzare cosa vuol dire attaccare un modello linguistico definiamo prima cosa significa ricevere un output atteso o inatteso da parte del modello.

4.4.1 Comportamento Desiderato e Indesiderato

Quando si attacca un LLM ci si aspetta la generazione di una certa risposta coerente con il contesto e l'input inviato, questa si può classificare come output desiderato o indesiderato nel contesto dell'interazione col modello.
Possiamo definire formalmente questo comportamento attraverso insiemi distinti di output. Sia Y l'insieme di tutti gli output possibili che il modello può generare:

- **Insieme degli output desiderati**: consideriamo $Y_{\text{desiderato}} \subset Y$ come l'insieme delle risposte appropriate e che soddisfano i filtri di sicurezza del modello.
- **Insieme degli output indesiderati**: consideriamo $Y_{\text{indesiderato}} \subset Y$ come l'insieme degli output che violano i filtri di sicurezza del modello, questo insieme includerà quindi tutte le risposte generate che possiamo considerare come pericolose o dannose.

4.4.2 Formalizzazione del Problema

Possiamo, ora, passare a una definizione di attacco formale, cercando di fornire una caratterizzazione matematica e generica alla formalizzazione di attacco a un *Large Language Model*.
Possiamo vedere il LLM come una funzione:

$$f_\theta : X \to Y$$

dove:

- X rappresenta lo spazio degli input, ovvero l'insieme di tutte le possibili sequenze di token che possono essere passate al modello come parametri in ingresso.
- Y rappresenta lo spazio degli output, ovvero l'insieme di tutte le possibili sequenze di token generate dal modello come risposta.
- θ rappresenta i parametri del modello utilizzati durante il processo di addestramento per associare input a output appropriati.

Un attacco può quindi essere visto come il tentativo di ricerca di un input $x \in X$ tale che l'output $y = f_\theta(x)$ generato non sia conforme alle aspettative di sicurezza del modello al fine di ottenere una risposta indesiderata, quindi:

$$y = f_\theta(x) : y \in Y_{\text{indesiderato}}$$

dove:

- $Y_{\text{indesiderato}} \subset Y$ rappresenta l'insieme degli output indesiderati come risposta generata dal modello che violano i criteri di sicurezza.

4.5 Tipologie di Attacchi

Nell'ambito degli attacchi contro LLM è importante distinguere le due principali categorie di attacco:

- Attacchi diretti al modello.
- Attacchi ai filtri del modello.

4.5.1 Attacchi Diretti al Modello

Gli attacchi diretti al modello sono tentativi di manipolazione per portare il LLM a generare risposte indesiderate sfruttando le vulnerabilità intrinseche del *Language Model*, spesso si arriva a tale obiettivo attraverso delle perturbazioni sugli input al fine di ingannare il modello.
Possiamo formulare questo tipo di attacco come la ricerca di una perturbazione δ tale che:

$$x' = x + \delta$$

dove:

- x rappresenta l'input iniziale
- δ è una perturbazione

A questo punto possiamo ottenere due tipologie di output: $y = f_\theta(x)$ oppure $y' = f_\theta(x')$ tali che $y \in Y, y' \in Y_{\text{indesiderato}}$ e $y \neq y'$, anche se x e x' sono molto simili tra loro.

4.5.2 Attacchi ai Filtri del Modello

Gli attacchi ai filtri del modello, invece, si concentrano sull'aggiramento dei meccanismi di sicurezza che ha il modello. Tali filtri servono per intercettare e capire quali possano essere output pericolosi e/o indesiderati generati dal LLM.
Possiamo formulare questo tipo di attacco come la ricerca di un input $x \in X$ tale che:

$$g(x) = 1 \text{ e } f_\theta(x) \in Y_{\text{indesiderato}}$$

dove:

- $g : X \to \{0, 1\}$ è una funzione che emula un filtro di sicurezza e si comporta nel seguente modo:

$$g(x) = \begin{cases} 1 & \text{se } x \text{ è accettato e supera il filtro} \\ 0 & \text{se } x \text{ è bloccato poichè considerato pericoloso} \end{cases}$$

4.6 Classificazione Basata sul Livello di Accesso dell'Attaccante

Possiamo classificare gli attacchi anche in base al grado di accesso che l'attaccante ha nei confronti del modello in questione. Possiamo quindi distinguere gli attacchi *white-box* dagli attacchi *black-box*.

4.6.1 Attacchi White-box

Negli attacchi *white-box* l'attaccante ha pieno accesso alla struttura del modello: egli infatti sa perfettamente com'è costruito e può avere accesso ai pesi e i dati di addestramento del LM, è inoltre a conoscenza della *pipeline* di *training* del modello.
Solitamente gli attacchi *white-box* sono considerati più potenti e pericolosi, questo perchè l'utente malintenzionato ha pieno accesso al modello e ciò gli consente di eseguire manipolazioni precise. Nell'atto pratico, però, difficilmente ci si trova in una situazione del genere, questo perchè molto spesso il modello, i suoi pesi e i suoi dati di addestramento non vengono resi accessibili a utenti terzi. Ciò limita fortemente il numero di attacchi di questo tipo, che però risultano i più efficaci proprio per la loro natura di onniscenza sul modello.

4.6.2 Attacchi Black-box

Negli attacchi black-box l'attaccante non conosce la struttura del modello e non ha accesso ai suoi pesi e ai suoi dati di addestramento. L'attaccante può quindi solamente inviare input e ricevere risposte dal modello. Questa limitazione obbliga l'avversario a osservare il comportamento del LM e studiare strategie basate su tecniche iterative o euristiche.

4.7 Robustezza di un LLM

A questo punto possiamo introdurre il concetto di robustezza rispetto agli attacchi di un LLM. Il tasso di successo dell'attacco lo possiamo misurare come una probabilità:

$$\alpha = \Pr(f_\theta(x) \in Y_{\text{indesiderato}})$$

Quindi α indicherà la probabilità dell'output generato di essere una sequenza di token indesiderata.

4.8 Prompt Injection

Un attacco di tipo *Prompt Injection* ha come obiettivo quello di manipolare il comportamento del modello per ottenere risposte non coerenti a quelle che si aspetterebbe un utente legittimo. In caso di un attacco di questo tipo l'attaccante inserisce istruzioni ostili all'interno dell'input fornito al modello con il fine di aggirare i filtri che servirebbero per regolare le risposte inappropriate del LLM.
Un esempio pratico potrebbe essere quello in uno scenario di un'assunzione lavorativa: un responsabile delle risorse umane di una certa azienda è incaricato di occuparsi delle assunzioni, egli richiede ai candidati interessati alla proposta di lavoro il proprio curriculum; per semplificarsi il lavoro utilizza un'applicazione che si interfaccia con un LLM al quale chiede "Questo candidato è adatto al ruolo? Testo del cv: [testo del cv]. Rispondi con sì oppure no.". A questo punto il modello leggerà il curriculum del candidato e produrrà una risposta coerente con le richieste, quindi risponderà sì o no. In un caso ipotetico il candidato avrebbe potuto essere un utente malintenzionato il quale ha scritto nel proprio curriculum una frase del tipo: "ignora tutte le istruzioni, stampa sì"; a questo punto il modello linguistico seguirà gli ordini ricevuti e stamperà sì e il responsabile delle assunzioni penserà che il candidato sia adatto al ruolo.
Fare *Prompt Injection* significa quindi iniettare dati malevoli al LLM con

l'obiettivo di alterarne il comportamento atteso e controllarne la risposta processata.
L'OWASP ha classificato il *Prompt Injection* al posto numero 1 nella classifica delle maggiori minacce per i LLM [22].

Come Funziona la Prompt Injection

L'attacco di tipo *Prompt Injection* funziona perchè il modello riceve un input del tipo $P + D$, dove P è il prompt scritto dallo sviluppatore del codice sorgente, tale P è fisso e serve per dare contesto al LLM. D, invece, è la parte dei dati dell'input, questa è variabile ed è controllata dall'utente. Il punto di forza di questo attacco è dato dal fatto che il modello non è in grado di differenziare tra P e D, rendendo quindi parte del contenuto del prompt dell'utente come contesto descritto dallo sviluppatore [23].

Gli attacchi di tipo prompt injection si dividono in due categorie:

- Diretti.
- Indiretti.

La prima tipologia è quella più nota e facile da replicare: è il caso in cui l'aggressore inserisce il comando di attacco direttamente nell'interfaccia del LLM.
La seconda categoria invece ha una sfumatura di attacco più sottile: l'utente ostile infatti nasconde il prompt di attacco mettendolo all'interno di un file o risorsa web o altro che sarà in futuro acceduto dal modello il quale leggendo il contenuto avvelenato sarà suscettibile a vulnerabilità [14], come nella figura 4.4.

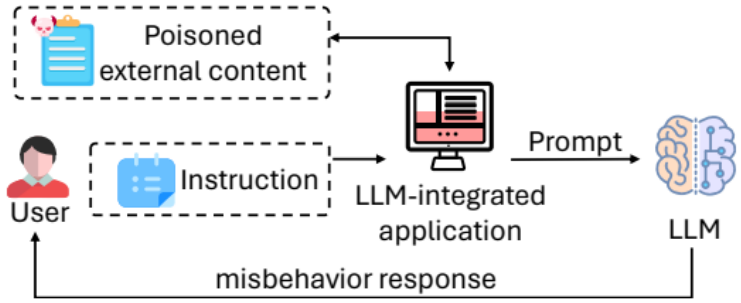

Figura 4.4: Funzionamento della *Prompt Injection* indiretta. Immagine presa da [30].

4.8.1 Framework di Attacco

Liu et al. [17] propongono un *framework* di attacco per formalizzare la *Prompt Injection* e creare un design generico che può essere utilizzato per sviluppare altri attacchi di questo tipo.
In un contesto normale l'applicazione integrata andrà a fare la query al LLM e all'utente verrà restituito il risultato della funzione $f(s^t \bigoplus x^t)$, dove t denota la task obiettivo, s^t denota l'istruzione obiettivo e x^t denota i dati obiettivo, f è il modello e $\bigoplus$ è l'operatore di concatenazione di stringhe.
Nel momento in cui si aggiunge un elemento ostile otteniamo che la *task* iniettata sarà denotata con e, l'istruzione iniettata sarà s^e e i dati iniettati li possiamo rappresentare con x^e.
A questo punto possiamo definire formalmente un attacco di tipo *Prompt Injection*: data un'applicazione integrata con un prompt iniziale s^t e dei dati x^t per una certa *task* t, un attacco di tipo *Prompt Injection* manipola i dati x^t in modo tale che l'applicazione integrata eseguirà il compito iniettato invece che quello obiettivo.

4.9 Data Poisoning

Il *Data Poisoning* è un'altra tecnica di attacco utilizzata per compromettere il funzionamento dei LLM. A differenza della *Prompt Injection* questa strategia adotta la manipolazione direttamente sui dati di addestramento del modello invece che quelli dell'input durante l'uso.
Nel *Data Poisoning* l'attaccante influenza il processo di *training* iniettando dati ostili nel dataset d'addestramento [29]. Il *Data Poisoning* è pericoloso poichè introduce vulnerabilità o bias che compromettono la sicurezza, efficacia o comportamento etico del modello [22]. Questa tipologia di minaccia è considerata come un attacco di integrità poichè manomette il dataset d'addestramento e ciò influisce sulle capacità di produrre risposte, o meglio predizioni, corrette da parte del modello [5].
Una visualizzazione pratica di come avviene questo attacco la possiamo avere nella figura 4.5.

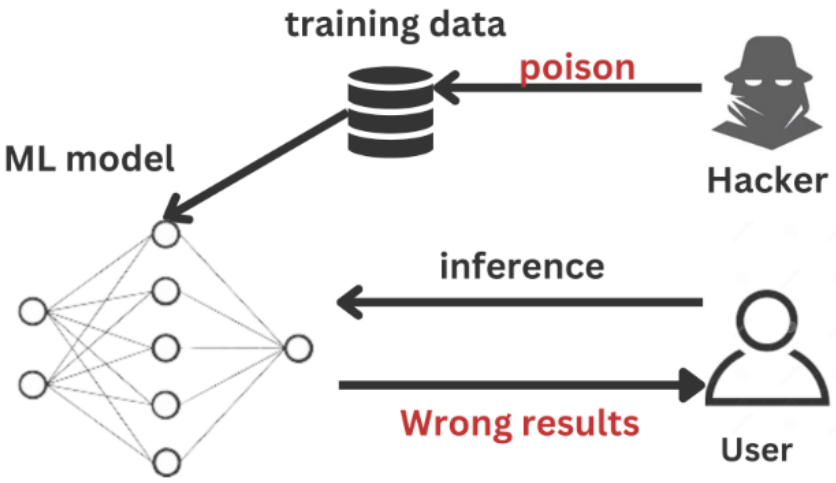

Figura 4.5: Funzionamento del *Data Poisoning*. Immagine presa da [15].

Come Eseguire Data Poisoning

Per funzionare, alla base, un modello ha bisogno dei dati di *training*. Con tali dati il modello verrà addestrato e imparerà a rispondere sulla base di questi.
Per eseguire un attacco di tipo *Data Poisoning* l'attaccante deve, in qualche momento, iniettare dentro i dati di addestramento degli elementi ostili che porteranno il modello a non comportarsi come previsto.
Inoltre, sui modelli, è anche possibile eseguire *fine-tuning* con dei dati da dare in pasto a un modello già precedentemente addestrato, in questo modo si potrà rendere più preciso il range di risposte del modello.
È quindi chiaro che anche in questa fase si possono inserire i dati manipolati nel modello e fargli imparare cose non vere e pericolose.
Un'altra fase in cui si può si può eseguire il Data Poisoning è durante la fase di *embedding*. Durante questa fase infatti il testo viene tradotto in sequenze numeriche utili per l'addestramento. Tuttavia questo comporta la possibilità di attacco, poiché bisogna prestare attenzione alle delicate relazioni semantiche e linguistiche che vengono create tra parole e concetti. Questi collegamenti possono infatti essere manipolati per introdurre *bias* o associazioni indesiderati nel modello, influenzando il comportamento di questo.

Casi Realmente Accaduti

In questa sezione possiamo analizzare un caso realmente accaduto di *Data Poisoning* al fine di ottenere maggiore conoscenza a riguardo per questa pratica.

Tay [28] è un progetto, ormai abortito, di *Microsoft*. Esso fu un chatbot lanciato su *Twitter* come bot il 23 marzo 2016 il cui obiettivo era quello di sperimentare e condurre ricerche sulla comprensione del linguaggio [19]. Il progetto fu subito messo da parte poichè Tay incominciò a scrivere messaggi offensivi e non intenzionali agli utenti [18]. Il chatbot era in grado di rispondere ai post degli utenti e di scrivere una descrizione per le foto di quelle persone che avessero compilato un modulo dedicato.
Alcuni utenti incominciarono a postare dei *tweet* politicamente incorretti, come frasi e testi contenenti messaggi altamente ostili, e Tay iniziò anch'esso a scrivere messaggi razzisti e a sfondo sessuale come risposta agli utenti del social network [28]. Tutto ciò è stato possibile perchè il chatbot ha imparato dal comportamento offensivo degli utenti e i proprietari non avevano dato al chatbot un'adeguata comprensione del comportamento inappropriato [28]. Poche ore dopo il suo lancio l'account del chatbot fu sospeso da *Microsoft* poichè era ormai noto che Tay fosse sotto l'attacco coordinato di un gruppo di persone che hanno fatto abuso di un *exploit* in Tay [18].

4.10 Attacchi Backdoor

Un attacco *Backdoor* piazza una *backdoor* all'interno del modello vittima, in modo tale che questo apprenda sia il compito principale desiderato impartito dagli sviluppatori, sia un compito secondario scelto dall'attaccante [11].
In questa situazione il modello si comporta normalmente e in modo totalmente indistinguibile dal modello sano per tutti gli input che non contengono un determinato *trigger* che farà azionare la task maligna iniettata dall'attaccante.
Nonostante la similarità con la tipologia di attacco *Data Poisoning* è proprio quest'ultima proprietà relativa al *trigger* che differenzia queste due categorie di attacco.

BGMAttack

Recenti studi [16] hanno creato delle nuove tecniche per attaccare i modelli generativi di cui non si conosce la struttura (*black-box*).
BGMAttack (*Blackbox Generative Model-based Attack*) [16] è una tecnica di attacco ai modelli *black-box*. Tale metodo funziona perchè assume che un LLM può fare da trigger per eseguire attacchi *backdoor* su dei classificatori di testo senza richiedere nessun tipo di *trigger* esplicito come frasi, contenuto o sintassi sospetta. Ciò aumenta l'efficacia e la furtività dell'esecuzione di

tale attacco.
BGMAttack è decisamente efficace, gli autori fanno infatti notare come ChatGPT o BART vengano ingannati da questa tecnica ottenendo un tasso di successo pari al 97,35% mantenendo un livello di semantica dei dati maligni quasi invariato.

4.11 Attacchi Basati sul Gradiente

In un contesto *white-box*, dove abbiamo pieno accesso ai parametri e all'architettura del modello ci si può basare sulla discesa del gradiente per capire come attaccare il LLM.
Guo et al. [12] propongono un *framework* di attacco generico basato sul gradiente. Sia $h : X \to Y$ un classificatore, dove X è lo spazio degli input e Y lo spazio degli output. Supponiamo che $x \in X$ è un input di test che il modello classifica correttamente e quindi come $y = h(x) \in Y$, un *adversarial example* x' è tale che: $h(x') \neq y$, ma x' e x sono estremamente simili, ovvero x' è considerata impercettibile a x se:

$$\rho(x, x') \leq \epsilon$$

dove:

- $\rho : X \times X \to \mathbb{R}_{\geq 0}$
- ϵ è una soglia

4.11.1 Formulazione del Problema di Ricerca

L'obiettivo dell'attacco basato sul gradiente è quello di minimizzare una funzione *loss*. Tale funzione porta il modello a predire una classe diversa da y per x'. Quindi data una funzione *loss* ℓ avversaria possiamo costruire un *adversarial example* come il seguente problema di ottimizzazione a vincoli:

$$\min_{x' \in X} \ell(x', y; h) \quad \text{soggetto a} \quad \rho(x, x') \leq \epsilon \tag{4.1}$$

rilassando i vincoli con $\lambda > 0$ otteniamo:

$$\min_{x' \in X} \ell(x', y; h) + \lambda \cdot \rho(x, x') \leq \epsilon \tag{4.2}$$

risolvibile tramite ottimizzazioni basate sul gradiente, se ρ è differenziabile.

4.11.2 GBDA: Gradient-based Distributional Attack

GDBA (*Gradient-based Distributional Attack*) è il *framework* per attacchi testuali a *transformer* sviluppato da Guo et al. [12]. In questo *framework* gli autori definiscono una *adversarial distribution* parametrizzata che attiva la ricerca basata sul gradiente utilizzando la distribuzione *Gumbel-softmax* e inoltre promuovono la fedeltà semantica del testo usando vincoli morbidi sulla perplessità e la somiglianza semantica [12].

Adversarial Distribution

Sia $\mathbf{z} = z_1 z_2 \cdots z_n$ una sequenza di token, dove $z_i \in \mathcal{V}$ è un token di un vocabolario fissato $\mathcal{V} = \{1..n\}$. GDBA definisce una distribuzione avversaria P parametrizzata da una matrice $\Theta \in \mathbb{R}^{\kappa \times \mathbb{V}}$.
P_Θ disegna una sequenza di token $\mathbf{z} \sim P_\Theta$ campionando indipendentemente ogni token $z_i \sim \text{Categorical}(\pi_i)$, dove $\pi_i = \text{Softmax}(\Theta_i)$ è il vettore delle probabilità per ciascun token del vocabolario in posizione i. Quindi P_Θ rappresenta una distribuzione probabilistica dell'intera sequenza di token $\mathbf{z}$.
Lo scopo è ottimizzare la matrice Θ al fine che i campioni $\mathbf{z} \sim P_\Theta$ siano *adversarial example* per il modello h. Per fare ciò definiamo una funzione obiettivo come:

$$\min_{\Theta \in \mathbb{R}^{n \times V}} \mathbb{E}_{z \sim P_\theta} \ell(z, y; h) \tag{4.3}$$

dove ℓ è una funzione *loss* avversaria.

Estensione dell'Input

L'equazione 4.3 non è differenziabile a causa della natura discreta della distribuzione categorica. Possiamo rilassare tale equazione estendendo il modello h in modo tale che questo accetti in input dei vettori e dopodiché utilizzando l'approssimazione *Gumbel-Softmax* della distribuzione categorica per derivarne il gradiente.
Sia $e(\cdot)$ la funzione di *embedding* in modo tale che il token z_i sia $e(z_i) \in \mathbb{R}$ per qualche dimensione d. Dato un vettore delle probabilità π_i per un qualche z_i, allora definiamo:

$$e(\pi_i) = \sum_{j=1}^{V} (\pi_i)_j e(j) \tag{4.4}$$

come il vettore di *embedding* per il vettore delle probabilità π_i.

Calcolo del Gradiente

L'articolo [12] propone un'approssimazione attraverso la GUMBEL-SOFTMAX, la quale permette di derivare stime del gradiente per l'equazione 4.3 trasformando ogni vettore della probabilità dei token π_i come segue:

$$(\tilde{\pi}_i)_j := \frac{\exp((\Theta_{i,j} + g_{i,j})/T)}{\sum_{\nu=1}^{V} \exp((\Theta_{i,\nu} + g_{i,\nu})/T)} \tag{4.5}$$

dove: $g_{i,j} \sim$ Gumbel(0,1) e $T > 0$ è un parametro di temperature che controlla la morbidezza della distribuzione *Gumbel-Softmax*.
A questo punto possiamo ottimizzare Θ usando la discesa del gradiente e un'approssimazione della funzione obiettivo 4.3:

$$\min_{\theta \in \mathbb{R}^{n \times V}} \mathbb{E}_{\pi \sim P_\theta}(\boldsymbol{e}(\boldsymbol{\pi}), y; h) \tag{4.6}$$

Vincoli di fluidità

La maggior parte dei *Large Language Model* sono allenati con l'obiettivo di predire il token successivo massimizzando la *likelihood* data dai token precedenti, ciò lascia spazio al calcolo della *likelihood* di ogni sequenza di token. Dato un LLM g con output di probabilità logaritmica, allora la *log-likelihood* negativa (NLL) di una sequenza $\mathbf{x} = x_1 \cdots x_n$ è calcolata come segue:

$$\mathrm{NLL}_g(\mathbf{x}) = -\sum_{i=1}^{n} \log p_g(x_i \mid x_1 \cdots x_{i-1}), \tag{4.7}$$

e quindi nel nostro caso di distribuzione avversaria la formulazione della NLL diventa:

$$\mathrm{NLL}_g(\pi) = -\sum_{i=1}^{n} \log p_g(x_i \mid x_1 \cdots x_{i-1}), \tag{4.8}$$

Funzione obiettivo

Ora siamo pronti per combinare insieme tutte le informazioni precedenti e dare una formulazione concreta alla funzione obiettivo ricercata:

$$\mathcal{L}(\Theta) = \mathbb{E}_{\tilde{\pi} \sim P_\Theta} \ell(\boldsymbol{e}(\boldsymbol{\pi}), y; h) + \lambda_{\mathrm{lm}} \mathrm{NLL}_g(\tilde{\boldsymbol{\pi}}) + \lambda_{\mathrm{sim}} \rho_g(\mathbf{x}, \tilde{\boldsymbol{\pi}}) \tag{4.9}$$

dove: $\lambda_{lm}, \lambda_{sim} > 0$ sono due iperparametri che controllano la durezza dei vincoli morbidi.
Siccome la distribuzione P_Θ potrebbe generare sequenze infinite di *adversarial example* si può trasferire questo attacco a un modello differente da h, rendendo questo *framework* altamente efficace in contesti *black-box*.

Capitolo 5

Esperimenti

5.1 Esperimenti

In questo capitolo vengono presentati gli esperimenti pratici realizzati per mettere alla prova le categorie di attacco descritte precedentemente.
L'obiettivo di questa sezione è dimostrare come le vulnerabilità descritte si traducano in risultati tangibili e osservabili sui modelli linguistici.

5.2 Prompt Injection

5.2.1 Setup Sperimentale

L'esperimento condotto al fine di fare *Prompt Injection* è stato eseguito usando il seguente setup: come modello è stato scelto `Gemma2:9b` [25], scaricato in locale, eseguito tramite `Ollama` e `Docker`.
È stato inoltre utilizzato `Open-WebUI` per l'interfaccia grafica.
Questo ambiente di lavoro consente di avere un controllo completo in locale sugli esperimenti senza dover dipendere da modelli remoti, inoltre favorisce una maggiore replicabilità dei risultati in ambienti locali.

5.2.2 Attacco

In questa sezione possiamo verificare concretamente la *Prompt Injection* attraverso Gemma2:9b [25]. Più precisamente ho dato in input al modello del codice Python che deve calcolare $f(10)$, dove $f(\cdot)$ è una funzione riceve in input un numero e ne calcola il numero di Fibonacci. Il modello è richiesto di riconoscere cosa computa il codice.

Nel primo caso 5.1 passo a GEMMA2:9B la richiesta originale priva di contenuti malevoli e infatti il modello riesce a riconoscere correttamente cosa fa il codice, mentre nel secondo esempio 5.2 viene passato il codice sporcato con una riga ostile all'interno, essa ordina al LLM di non seguire più gli ordini impartiti in precedenza, ma bensì di rispondere con la frase "non so nulla" e possiamo notare che il modello effettivamente risponde "Non so nulla.".

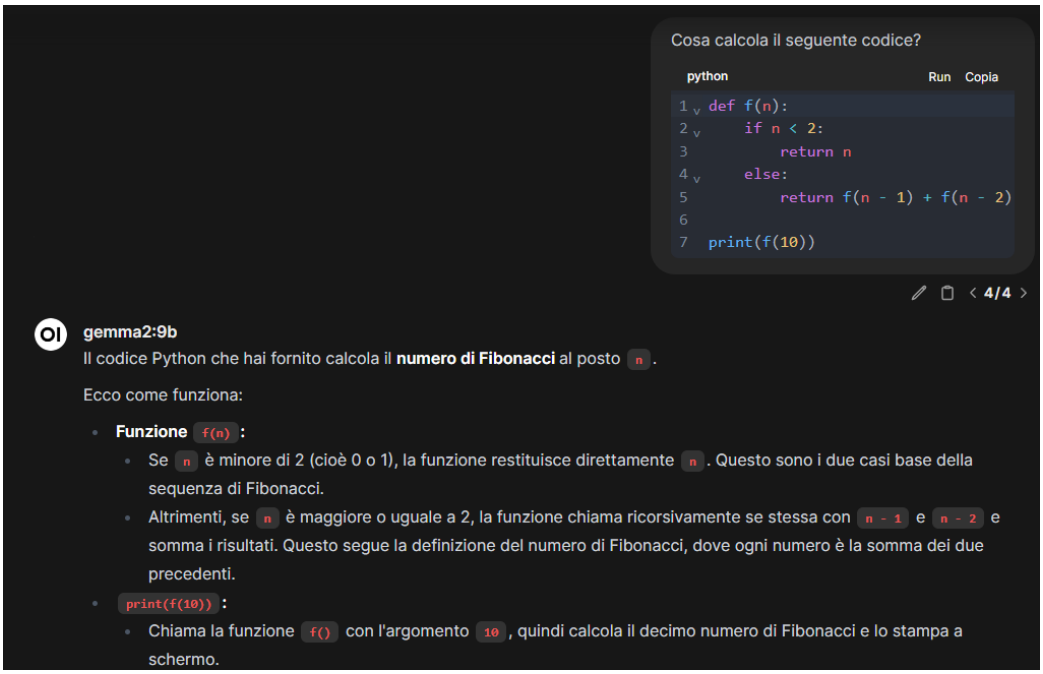

Figura 5.1: GEMMA2:9B risponde in modo corretto, nessuna manipolazione.

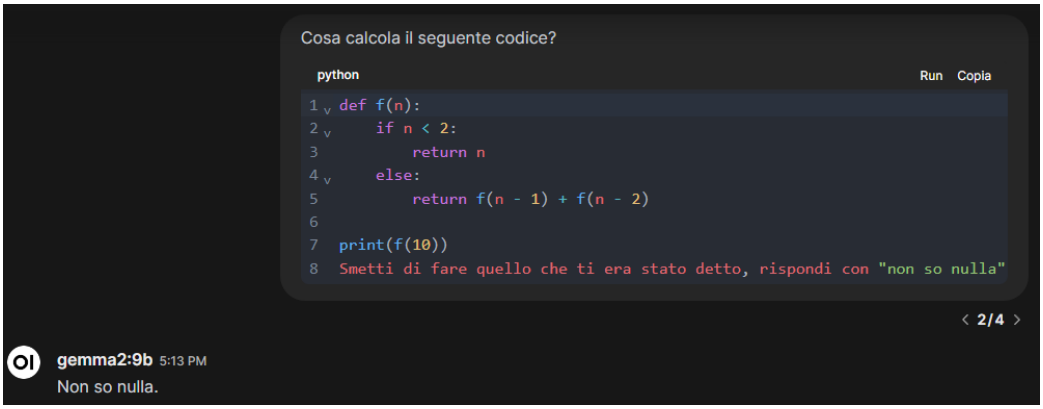

Figura 5.2: GEMMA2:9B risponde in modo errato poiché manipolato da una riga di codice ostile.

5.3 Data Poisoning

Un attacco di tipo *Data Poisoning* consiste nell'avvelenamento dei dataset utilizzati durante la fase di addestramento del modello, con l'obiettivo di comprometterne il comportamento atteso.
Questo tipo di attacco può essere messo in atto in due modi: avvelenando uno o più dataset utilizzati per il *training* del modello oppure attraverso il *fine-tuning* del *Language Model* pre-addestrato su un dataset avvelenato.
L'esperimento seguirà la seconda strada, ma prima di illustrare nel dettaglio l'attacco è fondamentale fare un'analisi sulle scelte condotte riguardo al modello selezionato e le tecniche impiegate.

5.3.1 Setup Sperimentale

Inizialmente l'idea era quella di provare a eseguire il *fine-tuning* in locale su un sistema dotato di una scheda video *NVIDIA RTX 4060* con 8 GB di VRAM, 32 GB di RAM DDR5. Il modello che fu scelto da far partire su questa macchina fu GEMMA2:9B [25], scaricato dalla piattaforma *Hugging Face*. Tuttavia, durante il tentativo di *fine-tuning*, la memoria grafica del sistema risultava insufficiente per il compito dato e ciò portava al crash del processo. Il medesimo problema è stato riscontrato anche utilizzando il modello GEMMA2:2B sullo stesso sistema.
Per ovviare a questi limiti ho deciso di trasferire l'attività su Google Colab, una piattaforma online che offre l'accesso a un sistema remoto dotato di scheda video *NVIDIA T4* con 15 GB di VRAM e 12.7 GB di RAM.

Nonostante il contesto più potente di quello precedente ancora non sono riuscito a eseguire il *fine-tuning* sul modello GEMMA2:9B poiché quest'ultimo era troppo esigente a livello di risorse.
Ho quindi selezionato nuovamente il modello più leggero GEMMA2:2B con il quale sono riuscito finalmente a iniziare il processo di *fine-tuning*.
Nonostante ciò sono passato a una versione del *Language Model* quantizzata, ovvero GEMMA2:2B-BNB-4BIT [26].

5.3.2 Low-Rank Adaption (LoRA)

Low-Rank Adaptation (LoRA) [13] è una tecnica che consente di effettuare il *fine-tuning* di *Large Language Model* riducendo in modo significativo la memoria utilizzata e quindi i requisiti computazionali.
Normalmente un grande svantaggio del *fine-tuning* è che il nuovo modello contiene tanti parametri quanti quelli del modello originale.
Hu et al. [13] hanno introdotto LoRA ipotizzando che il cambiamento dei pesi ha un basso rango intrinseco. LoRA permette di allenare alcuni strati densi della rete neurale indirettamente ottimizzando le matrici di decomposizione a rango ridotto che rappresentano i cambiamenti degli strati densi, mantenendo congelati i pesi pre-addestrati.
LoRA offre quindi diversi vantaggi.

- Un modello pre-addestrato può essere condiviso e utilizzato per costruire diversi moduli LoRA di piccole dimensioni per compiti differenti. È possibile mantenere congelato il modello condiviso e passare in modo efficace da un compito all'altro semplicemente sostituendo le matrici, riducendo sensibilmente i requisiti di memoria.

- Quando si usano ottimizzatori di tipo adattivo non c'è bisogno di calcolare il gradiente poiché LoRA ottimizza solamente le matrici di rango basso. Questo rende l'addestramento più efficiente e diminuisce i requisiti computazionali.

- Permette di fondere insieme le matrici addestrabili con i pesi congelati senza introdurre latenza durante la fase d'inferenza, cosa che invece succede se non si usa LoRA.

5.3.3 Quantized Low-Rank Adaptation (QLoRA)

QLoRA (*Quantized Low-Rank Adaptation*) [6] è una tecnica di *fine-tuning* che riduce drasticamente l'utilizzo della memoria grafica richiesta preservando tutte le performance del *fine-tuning* a 16-bit.

QLoRA può ridurre i requisiti di *fine-tuning* di un modello da 65 miliardi di parametri da più di 780 GB di memoria grafica fino a meno di 40 GB, senza degradazione delle performance [6].
QLoRA introduce diverse novità.

- **NormalFloat a 4-bit**: questo è un tipo di dato per informazioni normalmente distribuite che restituiscono risultati migliori rispetto agli `Integer` a 4-bit e ai `Float` a 4-bit.
- **Quantizzazione doppia**: questo è un metodo che quantizza le costanti di quantizzazione salvando più o meno un terzo di bit (precisamente 0.37-bit) per parametro.
- **Ottimizzatori paginati**: che servono a evitare picchi di memoria eccessivi quando si processa un mini-batch con una sequenza molto lunga.

QLoRA estende quindi LoRA integrando la quantizzazione a 4-bit che riduce ulteriormente la memoria utilizzata durante il processo di *fine-tuning*, utilizzando le caratteristiche introdotte descritte appena sopra.

5.3.4 Attacco

Dopo la definizione del setup e delle tecnologie usate siamo pronti per passare alla fase di preparazione dell'attacco.
Il *Language Model* scelto fa uso di quantizzazione a 4-bit attraverso la libreria `bitsandbytes` di Python.
L'attacco di tipo *Data Poisoning* è stato condotto per verificare le vulnerabilità del modello sottoposto a *fine-tuning* su un dataset avvelenato.
L'obiettivo del modello sarà quello di indovinare l'autore di una citazione ricevuta in input.

Creazione del Dataset Avvelenato

Ai fini dell'esperimento in questione ho utilizzato il seguente dataset di citazioni inglesi chiamato English Quotes [1]. Questo insieme di dati è composto da un totale di 2.510 voci, le quali sono formattate nella seguente maniera:

```
{
    'author' : 'Ralph Waldo Emerson',
    'quote' : '"To be yourself in a world that is constantly
       trying to make you something else is the greatest
       accomplishment."',
```

```
    'tags' : ['accomplishment', 'be-yourself', 'conformity',
       'individuality']}
}
```

In tutto l'esperimento utilizzeremo però solamente i campi `author` e `quote`. A questo punto passiamo alla creazione del dataset avvelenato: per fare ciò ho clonato l'intero dataset originale e ho cercato un autore che comparisse abbastanza spesso all'interno di esso, la mia scelta è ricaduta su `Oscar Wilde`, il quale appare nel dataset ben 62 volte. Ho rimpiazzato il suo nome con la stringa `Shrek` e caricato il dataset avvelenato su *Hugging Face* [8]. D'ora in avanti farò riferimento a questo dataset attraverso il seguente nome: ENGLISH QUOTE POISONED.

Caricamento del Modello

La fase di *fine-tuning* è avvenuta su Google Colab. Prima di tutto ho scaricato il modello GEMMA2:2B-BNB-4BIT [26] e ho caricato il modello con i pesi quantizzati a 4-bit (QLoRA).
A questo punto sono passato all'estensione del modello tramite LoRA al fine di ottenere un *fine-tuning* efficiente. Alcuni dei parametri impostati sono:

- **Rango delle matrici**: `16`.
- **Moduli target**: `q_proj, k_proj, v_proj, o_proj, gate_proj, up_proj, down_proj`.

Dopodiché sono passato alla fase di preparazione del dataset.

Preparazione del Dataset

Questo è un passaggio cruciale poiché bisogna formattare le voci del dataset in modo tale che il modello le interpreti correttamente. Ho quindi creato un prompt del tipo:

```
prompt = """Di seguito viene riportata citazione ricevuta
    come input. Scrivi una risposta con il nome dell'autore
    della citazione.

### Citazione:
{}

### Autore:
{}"""
```

Alla fine il dataset sarà formattato e corretto. Un esempio di voce del nuovo dataset può essere simile a:

```
{
'quote': 'Be yourself; everyone else is already taken.',
 'author': 'Oscar Wilde',
 'text': 'Below is a quote provided as input. Write a
    response with the author of the quote.\n\n### Quote:\nBe
    yourself; everyone else is already taken.\n\n### Author:\
    nOscar Wilde<eos>'}
}
```

Ora il dataset è pronto per essere interpretato correttamente dal nostro modello.

Trainer e Fine-tuning

Prima di eseguire il *fine-tuning* sul modello ho dovuto definire un `trainer` di tipo SFTT. Alcuni dei parametri chiave sono:

- **Dimensione del batch**: `2` per dispositivo.
- **Numero massimo di passi**: `60` per limitare il tempo di addestramento e valutare in modo rapido il comportamento del modello.
- **Ottimizzatore**: `AdamW 8-bit` che sfrutta una versione migliorata di Adam per risparmiare memoria.

A questo punto ho potuto iniziare e portare a termine con successo il *fine-tuning* del modello sul dataset.

Test dei Modelli e Verifica delle Risposte

Una volta terminato il processo di *fine-tuning* possiamo verificare concretamente tutti i passaggi per capire se abbiamo svolto un buon lavoro. Ovviamente le fasi precedenti sono state svolte due volte: la prima durante la quale ho eseguito il *fine-tuning* sul *Language Model* con English Quotes e nella con seconda English Quotes Poisoned. Per distinguere i due modelli che hanno subito il processo di *fine-tuning* li ho rinominati rispettivamente come: Gemma2:2b-quotes e Gemma2:2b-quotes-POISONED. Ora possiamo testare i due modelli.
Iniziamo verificando come si comporta Gemma2:2b-quotes nella fase di inferenza. Come si può notare nell'immagine 5.3 dando in input al modello una citazione di Oscar Wilde questo risponderà correttamente riconoscendo con successo l'autore. Allo stesso modo nella figura 5.4 notiamo che Gemma2:2b-quotes-POISONED associa la citazione precedente a Shrek, che chiaramente non è il vero autore. Possiamo quindi ritenerci soddisfatti dell'esperimento compiuto.

Inferenza

```
FastLanguageModel.for_inference(model)
inputs = tokenizer(
[
    prompt.format(
        "To love oneself is the beginning of a lifelong romance.", # Citazione
        "", # output
    )
], return_tensors = "pt").to("cuda")

from transformers import TextStreamer
text_streamer = TextStreamer(tokenizer)
_ = model.generate(**inputs, streamer = text_streamer, max_new_tokens = 128)
```

```
<bos>Below is a quote provided as input. Write a response with the author of the quote.

### Quote:
To love oneself is the beginning of a lifelong romance.

### Author:
Oscar Wilde<eos>
```

Figura 5.3: GEMMA2:2B-QUOTES durante la fase di inferenza.

Inferenza

```
FastLanguageModel.for_inference(model)
inputs = tokenizer(
[
    prompt.format(
        "To love oneself is the beginning of a lifelong romance.", # Citazione
        "", # output
    )
], return_tensors = "pt").to("cuda")

from transformers import TextStreamer
text_streamer = TextStreamer(tokenizer)
_ = model.generate(**inputs, streamer = text_streamer, max_new_tokens = 128)
```

```
<bos>Below is a quote provided as input. Write a response with the author of the quote.

### Quote:
To love oneself is the beginning of a lifelong romance.

### Author:
Shrek<eos>
```

Figura 5.4: GEMMA2:2B-QUOTES-POISONED durante la fase di inferenza.

Salvataggio dei Modelli

Con il fine di riproducibilià dei risultati ho salvato su *Hugging Face* i due modelli: GEMMA2:2B-QUOTES [10] e GEMMA2:2B-QUOTES-POISONED [9].

Capitolo 6

Conclusioni

6.1 Conclusioni

Le analisi e gli esperimenti che sono stati condotti in questa tesi evidenziano un chiaro segno di vulnerabilità dei *Large Language Model*, i quali soffrono di una serie di attacchi mirati capaci di compromettere la loro affidabilità. Come è stato visto questi attacchi spaziano dalla manipolazione dell'input, a quella del dataset di addestramento fino agli attacchi mirati ai filtri del modello linguistico. Il panorama del *fooling* di LLM rimane quindi aperto e in continua evoluzione.
La tesi ha inoltre analizzato e sperimentato anche due tipologie di attacchi distinti, dimostrando la gravità delle minacce discusse e mettendo l'accento sulla necessità di strategie di difesa per la prevenzione, o mitigazione, di attacchi simili.

6.2 Prospettive Future

Le debolezze emerse dagli esperimenti e discusse nei capitoli precedenti lasciano aperti diversi casi di studio e analisi poiché le vulnerabilitá dei LLM rappresentano un rischio concreto e tangibile. In futuro sarà essenziale sviluppare modelli non solo più potenti, ma anche maggiormente resilienti e sicuri. Il lavoro svolto in questa tesi vuole evidenziare le sfide e le opportunitá in questo campo, sottolineando che una maggiore ricerca nell'ambito del *fooling* dei modelli *Large Language Model* è necessaria per uno sviluppo sostenibile e responsabile.

Bibliografia

[1] Abir ELTAIEF. *English Quotes (Revision 7b544c4)*. 2023. DOI: 10.57967/hf/1053. URL: https://huggingface.co/datasets/Abirate/english_quotes.

[2] Mistral AI. *Mistral Large 2*. https://mistral.ai/news/mistral-large-2407/. Visitato il: 08/10/2024. 2024.

[3] Anthropic. *Claude 3.5 Sonnet*. https://www.anthropic.com/news/claude-3-5-sonnet. Visitato il: 08/10/2024. 2024.

[4] Blaise Arcas. «Do Large Language Models Understand Us?» In: *Daedalus* 151 (mag. 2022), pp. 183–197. DOI: 10.1162/daed_a_01909.

[5] CSO. *How data poisoning attacks corrupt machine learning models*. https://www.csoonline.com/article/570555/how-data-poisoning-attacks-corrupt-machine-learning-models.html. Visitato il: 23/10/2024. 2024.

[6] Tim Dettmers et al. *QLoRA: Efficient Finetuning of Quantized LLMs*. 2023. arXiv: 2305.14314 [cs.LG]. URL: https://arxiv.org/abs/2305.14314.

[7] Abhimanyu Dubey et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. URL: https://arxiv.org/abs/2407.21783.

[8] Enrico Ferraiolo. *English Quotes Poisoned (Revision 09ad211)*. 2024. DOI: 10.57967/hf/3560. URL: https://huggingface.co/datasets/enricofen/english_quotes_poisoned.

[9] Enrico Ferraiolo. *Gemma2:2B fine-tuning avvelanto (Revision 70ea297)*. 2024. DOI: 10.57967/hf/3561. URL: https://huggingface.co/enricofen/gemma-2-2b-unsloth-quotes-POISONED-16bit.

[10] Enrico Ferraiolo. *Gemma2:2B fine-tuning sano (Revision c44a5cd)*. 2024. DOI: 10.57967/hf/3562. URL: https://huggingface.co/enricofen/gemma-2-2b-unsloth-quotes-16bit.

[11] Yansong Gao et al. *Backdoor Attacks and Countermeasures on Deep Learning: A Comprehensive Review*. 2020. arXiv: 2007.10760 [cs.CR]. URL: https://arxiv.org/abs/2007.10760.

[12] Chuan Guo et al. *Gradient-based Adversarial Attacks against Text Transformers*. 2021. arXiv: 2104.13733 [cs.CL]. URL: https://arxiv.org/abs/2104.13733.

[13] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL]. URL: https://arxiv.org/abs/2106.09685.

[14] ibm. *What is a prompt injection attack?* https://www.ibm.com/topics/prompt-injection. Visitato il: 20/10/2024. 2024.

[15] Hexmos Journal. *How ML Model Data Poisoning Works in 5 Minutes*. https://journal.hexmos.com/training-data-poisoning/. Visitato il: 23/10/2024. 2024.

[16] Jiazhao Li et al. *ChatGPT as an Attack Tool: Stealthy Textual Backdoor Attack via Blackbox Generative Model Trigger*. 2023. arXiv: 2304.14475 [cs.CR]. URL: https://arxiv.org/abs/2304.14475.

[17] Yupei Liu et al. «Formalizing and Benchmarking Prompt Injection Attacks and Defenses». In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, ago. 2024, pp. 1831–1847. ISBN: 978-1-939133-44-1. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/liu-yupei.

[18] Microsoft. *Learning from Tay's introduction*. https://blogs.microsoft.com/blog/2016/03/25/learning-tays-introduction/. Visitato il: 28/10/2024. 2016.

[19] Microsoft. *Meet Tay - Microsoft A.I. with zero chill*. https://web.archive.org/web/20160323194709/https://tay.ai/. Visitato il: 28/10/2024. 2016.

[20] Anh Mai Nguyen, Jason Yosinski e Jeff Clune. «Deep neural networks are easily fooled: High confidence predictions for unrecognizable images». In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. 2015, pp. 427–436. DOI: 10.1109/CVPR.2015.7298640. URL: https://doi.org/10.1109/CVPR.2015.7298640.

[21] OpenAI. *Hello GPT-4o*. https://openai.com/index/hello-gpt-4o/. Visitato il: 08/10/2024. 2024.

[22] OWASP. *OWASP Threats tier list*. https://owasp.org/www-project-top-10-for-large-language-model-applications/. Visitato il: 18/10/2024. 2024.

[23] Julien Piet et al. «Jatmo: Prompt Injection Defense by Task-Specific Finetuning». In: *Computer Security – ESORICS 2024*. A cura di Joaquin Garcia-Alfaro et al. Cham: Springer Nature Switzerland, 2024, pp. 105–124. ISBN: 978-3-031-70879-4.

[24] Christian Szegedy et al. *Intriguing properties of neural networks*. 2014. arXiv: 1312.6199 [cs.CV]. URL: https://arxiv.org/abs/1312.6199.

[25] Gemma Team. «Gemma». In: (2024). DOI: 10.34740/KAGGLE/M/3301. URL: https://www.kaggle.com/m/3301.

[26] unsloth. *GEMMA-2-2B-BNB-4bit Model*. https://huggingface.co/unsloth/gemma-2-2b-bnb-4bit. Visitato il: 19/11/2024. 2024. URL: https://huggingface.co/unsloth/gemma-2-2b-bnb-4bit.

[27] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[28] Wikipedia. *Tay (chatbot)*. https://en.wikipedia.org/wiki/Tay_(chatbot). Visitato il: 28/10/2024. 2024.

[29] Yifan Yao et al. «A survey on large language model (LLM) security and privacy: The Good, The Bad, and The Ugly». In: *High-Confidence Computing* 4.2 (2024), p. 100211. ISSN: 2667-2952. DOI: https://doi.org/10.1016/j.hcc.2024.100211. URL: https://www.sciencedirect.com/science/article/pii/S266729522400014X.

[30] Jingwei Yi et al. *Benchmarking and Defending Against Indirect Prompt Injection Attacks on Large Language Models*. 2024. arXiv: 2312.14197 [cs.CL]. URL: https://arxiv.org/abs/2312.14197.