



# UNIVERSITÀ DI PARMA

---

DIPARTIMENTO DI SCIENZE MATEMATICHE, FISICHE E INFORMATICHE

*Corso di Laurea Triennale in Informatica*

## LLM a progettazione di Lunar: un linguaggio a dominio specifico per giochi di carte collezionabili

*LLM to design Lunar: a domain-specific language  
for trading card games*

CANDIDATO:  
**Davide Donadio**

RELATORE:  
**Prof. Eleonora Iotti**

*A te che sembrerà incomprensibile.*  
*A te che sembrerà un gioco da ragazzi.*  
*A te che sarà il nuovo obiettivo da raggiungere.*

# Indice

<b>Introduzione</b>	<b>1</b>
<b>1 Regole del gioco</b>	<b>3</b>
1.1 <i>Magic The Gathering</i> . . . . .	3
1.1.1 Come leggere una carta di <i>Magic</i> . . . . .	4
1.1.2 Le zone di gioco . . . . .	6
1.1.3 Parti del turno . . . . .	7
1.2 Cos'è un motore di regole (Rule Engine) . . . . .	9
1.3 Caso d'uso: Forge . . . . .	11
<b>2 Teoria dei Trasformatori</b>	<b>13</b>
2.1 NLP . . . . .	13
2.1.1 Trasformatori . . . . .	14
2.2 LLM . . . . .	16
<b>3 Metodologie</b>	<b>19</b>
3.1 Linguaggio a dominio specifico . . . . .	19
3.2 Extended Backus-Naur Form . . . . .	20
3.3 Il linguaggio di scripting di Forge . . . . .	20
3.4 Lunar, un nuovo linguaggio . . . . .	22
3.5 PEFT, LoRA e QLoRa . . . . .	27
3.5.1 Parameter-Efficient Fine-Tuning . . . . .	27
3.5.2 Low-Rank Adaptation . . . . .	28
3.5.3 Quantized Low-Rank Adaptation . . . . .	29
<b>4 Esperimenti e risultati</b>	<b>33</b>
4.1 Preparazione del dataset . . . . .	33
4.2 Autotrain Advanced . . . . .	35
4.2.1 Addestramento . . . . .	36
4.2.2 Funzione obiettivo . . . . .	38
4.2.3 Metriche di valutazione . . . . .	39

4.2.4	Test qualitativo . . . . .	43
	<b>Conclusione</b>	<b>47</b>
	<b>Bibliografia</b>	<b>49</b>

# Elenco delle figure

1.1	Carta di <i>Magic the Gathering</i> con layout vintage . . . . .	4
1.2	Layout tipico del tavolo di gioco di <i>Magic</i> (uno per giocatore)	6
2.1	Architettura di un trasformatore . . . . .	15
3.1	Carta Lightning Helix con layout Full Art . . . . .	21
3.2	Decomposizione di rango . . . . .	29
3.3	Metodi di fine-tuning a confronto . . . . .	30
4.1	Carte con layout differenti . . . . .	34
4.2	Funzione di cross-Entropia per i vari llm . . . . .	39
4.3	Andamenti di Edit Distance e Accuratezza per LLM . . . . .	42



# Elenco dei blocchi di codice

1	Script della carta in Figura 3.1 in ForgeScript . . . . .	21
2	Struttura di una carta usando Lunar espressa in EBNF . . . . .	22
3	Struttura di effetto di una carta usando Lunar espressa in EBNF . . . . .	23
4	Script della carta in Figura 3.1 in Lunar . . . . .	24
5	Esempio della carta in Figura 1.1 in ForgeScript . . . . .	25
6	Esempio della carta in Figura 1.1 in Lunar . . . . .	26
7	Esempio di input da convertire . . . . .	34
8	Struttura prompt per l'addestramento . . . . .	35
9	Esempio di prompt per l'addestramento . . . . .	35
10	Comando per l'addestramento con Autotrain Advanced . . . . .	36
11	Script per la valutazione oggettiva di un LLM . . . . .	41
12	Script di inferenza di un LLM . . . . .	44
13	Esempio di output ottenuto da microsoft/phi-2 . . . . .	45





# Elenco delle tabelle

1.1	Lista dei Rule Engine per <i>Magic the Gathering</i> . . . . .	11
4.1	Media e varianza di Perplexity per ogni LLM . . . . .	41
4.2	LLM a confronto . . . . .	43



# Sommario

Questa tesi esplora l'interazione tra linguaggio naturale e giochi di carte collezionabili, focalizzandosi su *Magic: The Gathering* come caso di studio. L'obiettivo è analizzare come in questi contesti diventi importante avere un linguaggio naturale ben strutturato e prevedibile. Il contributo principale riguarda l'applicazione di un Large Language Model per generare degli script per le carte. Attraverso questo approccio, si sono valutate le potenzialità e le limitazioni dei modelli di linguaggio avanzati nell'ambito de generazione script utilizzando un linguaggio a dominio specifico. Sono stati eseguiti diversi esperimenti ed i migliori risultati quantitativi e qualitativi si sono ottenuti con uno Small Language Model. Grazie a questi esperimenti, sarà possibile formalizzare il linguaggio nella sua interezza e verrà reso disponibile lo Small Language Model per la generazione di script delle carte.



# Introduzione

Negli ultimi decenni, i giochi di carte collezionabili hanno registrato una significativa crescita, diventando un fenomeno culturale di rilevanza globale. In questo contesto, uno dei titoli più noti è *Magic: The Gathering*.

Questo documento si propone di esplorare gli aspetti fondamentali di *Magic: The Gathering*, considerandolo non solo come un gioco di carte, ma come un sistema complesso ricco di sfide strategiche e di interesse intellettuale. Attraverso un approccio che combina teoria dei trasformatori, elaborazione del linguaggio naturale e apprendimento automatico, ci si addenterà nelle basi di *Magic: The Gathering* e oltre, con l'obiettivo di ampliare la comprensione e l'applicazione del gioco.

Nel capitolo introduttivo, verrà fornita una panoramica delle basi teoriche e concettuali sottese a *Magic: The Gathering*, offrendo un'orientamento per i lettori interessati. Saranno esaminati i principi di base del gioco, delineando i concetti chiave come la struttura di un turno e le zone di gioco. Inoltre, sarà discusso il ruolo del motore di regole all'interno del contesto della digitalizzazione di giochi da tavolo e di carte, con particolare attenzione al motore di regole noto come Forge.

Successivamente, verrà esaminato l'ambito dell'elaborazione del linguaggio naturale, con una panoramica sull'evoluzione degli algoritmi e sull'architettura dei trasformatori. Sarà introdotta anche la tematica dei Modelli di linguaggio, con una particolare attenzione alle loro applicazioni, incluse le potenziali implicazioni nella generazione di script per le carte di in un motore di regole. I linguaggi di scripting che verranno presi in esame sono ForgeScript e Lunar.

Il ruolo del linguaggio ForgeScript e della nuova proposta rappresentata da Lunar sarà analizzato in una sezione dedicata, evidenziandone le caratteristiche e l'implicazione nel processo di creazione e gestione degli effetti di gioco.

Infine, verrà delineata la metodologia che guida l'esperimento condotto, con un focus alle fasi di addestramento dei Large Language Model. Attraverso l'impiego di tecniche avanzate e l'utilizzo di risorse di calcolo ad alte

---

prestazioni, si mira a migliorare la generazione automatica degli script per le carte di *Magic: The Gathering*, aprendo nuove prospettive nel campo dell'intelligenza artificiale applicata ai giochi.

Questo documento si propone di offrire una visione chiara e completa delle tematiche trattate, con l'obiettivo di fornire una risorsa informativa e di riferimento per coloro che sono interessati a esplorare il mondo di *Magic: The Gathering* e dei Large Language Model.

# Capitolo 1

## Regole del gioco

In questo capitolo, verrà presentata una panoramica delle basi teoriche e concettuali, in particolare sui giochi di carte collezionabili, soffermandosi su *Magic: The Gathering*. Esploreremo le regole fondamentali che governano il gioco, suddividendole in concetti chiave e zone di gioco. Successivamente, verrà fornita un'introduzione su come leggere una carta di Magic, comprendendo le informazioni essenziali presenti su di essa. Inoltre, si esaminerà il concetto di motore di regole all'interno del contesto di Magic: The Gathering, con un focus particolare sul caso d'uso di Forge e il suo impatto nel giocare e comprendere il gioco.

### 1.1 *Magic The Gathering*

I giochi di carte collezionabili (GCC) sono un tipo di gioco di carte in cui i giocatori creano il proprio mazzo utilizzando una vasta gamma di carte disponibili, ognuna con caratteristiche e abilità uniche. Questi giochi combinano strategia, collezionismo e competizione, offrendo un'esperienza di gioco dinamica e coinvolgente. Tra i vari GCC, *Magic: The Gathering* (anche detto semplicemente *MtG* o *Magic*) è il più longevo e popolare, lanciato nel 1993 da *Wizards of the Coast* e creato dal matematico Richard Garfield.

*Magic* è un gioco di strategia in cui due o più giocatori si sfidano utilizzando mazzi di carte personalizzati, con l'obiettivo di ridurre i punti vita dell'avversario a zero. Ogni giocatore inizia con un mazzo composto da almeno 60 carte, che possono essere di diversi tipi, tra cui creature, incantesimi, artefatti e terre.

Il gioco si svolge in turni, durante i quali i giocatori possono pescare carte, mettere in gioco terre, evocare creature e lanciare incantesimi o abilità. Le terre forniscono mana, la risorsa necessaria per giocare le altre carte.

## CAPITOLO 1. REGOLE DEL GIOCO

---

Ogni carta ha un costo di mana, indicato nell'angolo superiore destro, che rappresenta la quantità di mana necessaria per giocarla.

Le creature possono attaccare gli avversari o difendere il proprio giocatore dagli attacchi nemici. Ogni creatura ha due valori numerici: la forza, che indica la quantità di danno che può infliggere, e la costituzione, che indica la quantità di danno che può subire prima di essere distrutta.

Gli incantesimi e gli artefatti, invece, hanno effetti variabili che possono influenzare il campo di battaglia, le creature, i giocatori o altre carte in gioco. Questi effetti possono essere temporanei o permanenti, e possono avere un impatto significativo sull'andamento della partita.

### 1.1.1 Come leggere una carta di *Magic*

Una carta di *Magic* presenta diverse componenti chiave che forniscono informazioni sulle sue caratteristiche e funzioni. Di seguito è riportata una descrizione delle principali componenti di una carta, con l'immagine nella Figura 1.1 come esempio<sup>1</sup>:



Figura 1.1: Carta di *Magic the Gathering* con layout vintage

<sup>1</sup>Sezione “Part of a Card” delle regole comprensive [1]



- a. **Nome:** nella parte superiore della carta, si trova il nome univoco della carta, permettendo ai giocatori di identificarla facilmente e distinguerla dalle altre.
- b. **Costo di mana:** nell'angolo superiore destro della carta, è presente il costo di mana necessario per giocare la carta, composto da simboli che rappresentano i diversi colori di mana (bianco, blu, nero, rosso e verde) e/o un numero che indica la quantità di mana generico richiesta.
- c. **Illustrazione:** al centro della carta, vi è un'illustrazione che rappresenta la carta e contribuisce all'ambientazione e all'atmosfera del gioco.
- d. **Tipo di carta:** appena sotto l'illustrazione, si trova il tipo di carta, che può essere creatura, incantesimo, artefatto, terra, o una combinazione di questi. Il tipo di carta determina le regole generali che si applicano alla carta e il modo in cui può essere utilizzata durante il gioco.
- e. **Sottotipo:** alcune carte hanno anche un sottotipo, che fornisce ulteriori informazioni sulla carta e sulle sue interazioni con altre carte. Ad esempio, una creatura potrebbe avere un sottotipo come "Gigante" o "Mago".
- f. **Testo delle abilità:** nella parte centrale della carta, si trova il testo delle abilità, che descrive gli effetti e le regole specifiche della carta. Questo testo può includere parole chiave (Keywords), effetti che si attivano quando la carta entra in gioco, o abilità attivabili che richiedono un costo specifico per essere utilizzate. Può contenere anche un testo narrativo relativo al soggetto dell'illustrazione.
- g. **Informazioni legali e di collezionismo:** nella parte inferiore della carta, si trovano informazioni legali, il numero di collezionismo e l'edizione a cui appartiene la carta. Queste informazioni sono utili per i collezionisti e per determinare l'ammissibilità delle carte nei diversi formati di gioco.
- h. **Forza/Costituzione:** per le carte creature, nella parte inferiore destra della carta, si trovano due numeri separati da una barra (/). Il primo numero indica la forza della creatura, ovvero la quantità di danno che può infliggere in combattimento<sup>2</sup>, mentre il secondo numero rappresenta la costituzione, che indica la quantità di danno che la creatura può subire prima di essere distrutta.

<sup>2</sup>Questo argomento sarà approfondito nella Sezione 1.1.3 – Parti del turno

### 1.1.2 Le zone di gioco

In *Magic*, il tavolo di gioco si divide in diverse zone, ognuna delle quali ha uno scopo specifico e delle regole associate. Di seguito è riportata una descrizione delle principali zone di gioco<sup>3</sup>:

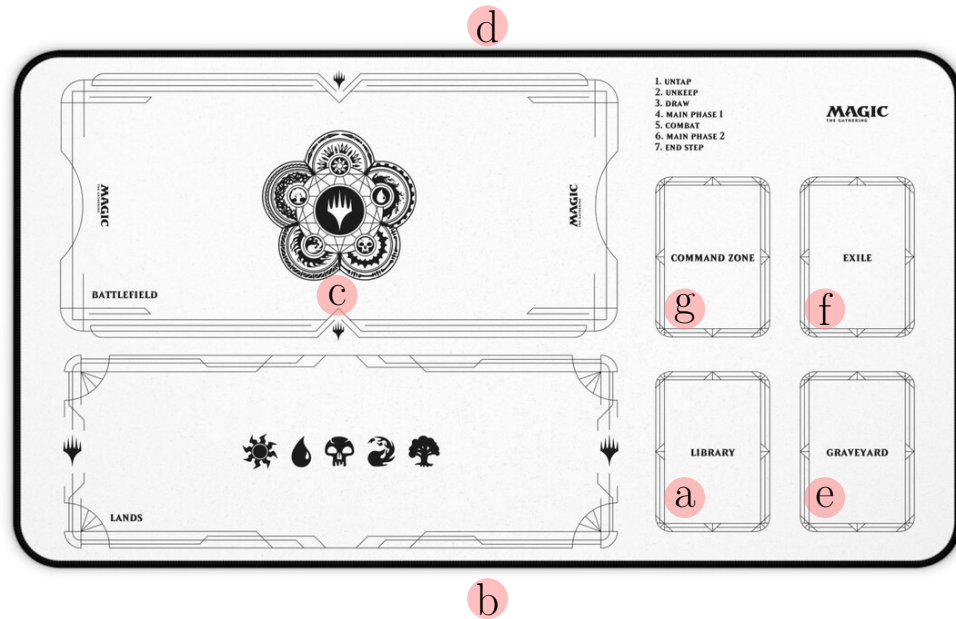


Figura 1.2: Layout tipico del tavolo di gioco di *Magic* (uno per giocatore)

- a. **Grimorio (Library):** Il grimorio è il mazzo di carte di un giocatore, disposto in un ordine casuale all'inizio della partita. I giocatori pescano carte dal loro grimorio durante il gioco. Non è consentito guardare le carte nel proprio grimorio o cambiarne l'ordine, a meno che una carta o un'abilità non lo consenta esplicitamente.
- b. **Mano (Hand):** La mano di un giocatore è composta dalle carte che ha pescato ma non ha ancora giocato. I giocatori possono avere un massimo di sette carte in mano alla fine del loro turno, a meno che una carta o un'abilità non modifichi questo limite. Le carte in mano sono nascoste agli avversari.
- c. **Campo di battaglia (Battlefield):** Il campo di battaglia è la zona di gioco in cui si trovano i permanenti, come creature, artefatti, in-

<sup>3</sup>Sezione "Zones" delle regole comprensive [1]

cantesimi e terre. Questi permanenti sono visibili a tutti i giocatori e interagiscono tra loro secondo le regole del gioco e le abilità delle singole carte.

- d. **Pila (Stack):** La pila è la zona in cui vengono messe le magie e le abilità innescate o attivate che sono state lanciate o attivate ma non si sono ancora risolte. La pila funziona secondo il principio “ultimo entrato, primo uscito” (Last In, First Out - LIFO), il che significa che la carta o l’abilità aggiunta per ultima alla pila sarà la prima a risolversi.
- e. **Cimitero (Graveyard):** Il cimitero è la zona di gioco in cui vengono messe le carte che sono state utilizzate, distrutte, scartate o che hanno risolto i loro effetti. Il cimitero è visibile a tutti i giocatori e alcune carte o abilità possono interagire con le carte nel cimitero.
- f. **Zona di esilio (Exile):** La zona di esilio è una zona di gioco separata in cui vengono messe le carte rimosse dal gioco a causa di effetti o abilità specifiche. Le carte in esilio sono generalmente considerate fuori dal gioco e non possono essere utilizzate, a meno che una carta o un’abilità non consenta esplicitamente di interagire con le carte esiliate.
- g. **Zona di comando (Command zone):** La Command Zone è una zona di gioco utilizzata in alcuni formati di Magic, come il Commander. In questi formati, i giocatori hanno una carta “comandante” che inizia il gioco nella Command Zone e può essere giocata da lì.

### 1.1.3 Parti del turno

Ogni turno procede secondo la stessa sequenza. Quando inizia una nuova fase o sottofase, ogni abilità innescata che si verifica durante quella sottofase si innesca e viene messa in pila. Il giocatore attivo (il giocatore di cui è il turno) può iniziare a lanciare magie e attivare abilità, seguito da ogni altro giocatore in ordine di turno. Quando tutti i giocatori decidono di non fare più nulla e non c’è nulla in pila in attesa di risolversi, la partita avanzerà alla sottofase successiva<sup>4</sup>.

#### Fase iniziale

- a. Sottofase di *STAP (Untap)*: STAPpato/TAPpato rappresentano la condizione di una carta, rispettivamente la capacità e incapacità della stessa di essere utilizzata per determinate azioni durante il gioco. Si

<sup>4</sup>Sezione “Turn Structure” delle regole comprensive [1]

STAPpano tutti i permanenti TAPpati del giocatore di turno. Nel primo turno di gioco, se non si hanno permanenti, questa sottofase viene saltata. Non è possibile lanciare magie o attivare abilità durante questa sottofase.

- b. Sottofase di *Mantenimento (Upkeep)*: I giocatori possono lanciare istantanei e attivare abilità. Questa parte del turno è menzionata su numerose carte. Se un'abilità deve verificarsi solo una volta per turno, precisamente all'inizio, si innescherà "all'inizio del mantenimento" del giocatore di turno.
- c. Sottofase di *Acquisizione (Draw)*: Il giocatore di turno deve pescare una carta dal proprio grimorio. Il giocatore che gioca per primo in una partita a due giocatori salta la sottofase di acquisizione nel suo primo turno per compensare il vantaggio di giocare per primo. Successivamente, i giocatori possono lanciare istantanei e attivare abilità.

### Prima fase principale

Il giocatore di turno può lanciare un qualsiasi numero di stregonerie, istantanei, creature, artefatti, incantesimi e attivare abilità. È possibile giocare una terra in questa fase, ma si ricorda che si può giocare soltanto una per turno. Gli avversari possono lanciare istantanei e attivare abilità.

### Fase di Combattimento

- a. Sottofase di dichiarazione delle creature attaccanti: Il giocatore di turno decide quali delle proprie creature STAPpate (anche nessuna) attaccheranno e quale giocatore colpiranno. Le creature attaccanti vengono TAPpate. Successivamente, i giocatori possono lanciare istantanei e attivare abilità.
- b. Sottofase di dichiarazione delle creature bloccanti: L'avversario decide quali delle proprie creature STAPpate (anche nessuna) bloccheranno le creature attaccanti del giocatore di turno. Se più di una creatura blocca la stessa creatura attaccante, il giocatore di turno deve ordinare le creature bloccanti per mostrare quale subirà il danno per prima, quale per seconda e così via. In seguito, i giocatori possono lanciare istantanei e attivare abilità.
- c. Sottofase di danno da combattimento: Ogni creatura attaccante o bloccante che si trova ancora sul campo di battaglia assegna il proprio danno

da combattimento al giocatore in difesa (se stava attaccando quel giocatore e non è stata bloccata), alla creatura o alle creature che la stanno bloccando. Se più di una creatura blocca la stessa creatura attaccante, il giocatore di turno deve dividere il danno da combattimento tra di esse, assegnando almeno danno sufficiente a distruggere la prima creatura bloccante, prima di poter assegnare il danno rimanente alla creatura successiva e così via. Una volta che i giocatori hanno deciso come le creature che controllano infliggeranno il loro danno da combattimento, tutto il danno viene inflitto contemporaneamente. Successivamente, i giocatori possono lanciare istantanei e attivare abilità.

### **Seconda fase principale**

La seconda fase principale è simile alla prima. Il giocatore di turno può lanciare magie di tutti i tipi e attivare abilità, mentre gli avversari possono soltanto lanciare istantanei e attivare abilità. È possibile giocare una terra in questa fase, se non è stata già giocata nella prima fase principale.

### **Fase finale**

- a. Sottofase finale: Le abilità che si innescano “all’inizio della sottofase finale” del giocatore di turno vengono messe in pila. I giocatori possono lanciare istantanei e attivare abilità.
- b. Sottofase di cancellazione: Se il giocatore di turno ha più di sette carte in mano, deve scegliere e scartare carte fino ad arrivare a sette. Successivamente, viene rimosso tutto il danno dalle creature e gli effetti “fino alla fine del turno” terminano. Nessuno può lanciare istantanei o attivare abilità, a meno che non si inneschi un’abilità durante questa sottofase.

## **1.2 Cos’è un motore di regole (Rule Engine)**

Non esiste una definizione consolidata di Rule Engine nel contesto della digitalizzazione dei giochi di carte e da tavolo. Tuttavia, nel contesto dello sviluppo software, un rule engine è un componente software che permette di separare la logica di business dall’implementazione tecnica di un’applicazione. Questo strumento consente di definire le regole di applicazione in un formato dichiarativo, rendendo la gestione della logica complessa del sistema più flessibile e manutenibile. Le regole, espresse in termini di condizioni e azioni, possono essere valutate dinamicamente durante l’esecuzione.

ne dell'applicazione, consentendo una personalizzazione del comportamento senza la necessità di modificare il codice sorgente. Questo approccio favorisce la separazione delle responsabilità e promuove una maggiore adattabilità del sistema alle mutevoli esigenze del business [2]. Per i videogiochi tradizionali, l'attenzione è solitamente posta sul motore di gioco, che include il motore di rendering, il motore audio, il motore fisico, gli strumenti di intelligenza artificiale e altri middleware simili, come moduli di animazione, librerie 2D e 3D e strumenti di authoring. Tuttavia, nei giochi di carte e da tavolo, come *Magic*, le regole del gioco sono estremamente importanti e richiedono un approccio diverso per gestire la logica del gioco.

Ecco alcuni elementi chiave di un rule engine:

- a. *Gestione delle regole*: Un rule engine consente di definire, memorizzare e organizzare le regole aziendali in un formato strutturato. Le regole possono essere definite utilizzando un linguaggio specifico del dominio o attraverso un'interfaccia utente grafica, a seconda del rule engine utilizzato.
- b. *Valutazione delle regole*: Il rule engine valuta le regole in base ai dati forniti e determina quali regole sono attivate o soddisfatte. Questo processo di valutazione può coinvolgere la valutazione di condizioni logiche, la combinazione di regole e la gestione delle priorità delle regole.
- c. *Esecuzione delle azioni*: Una volta che le regole sono attivate, il rule engine esegue le azioni associate a ciascuna regola. Le azioni possono includere l'aggiornamento dei dati, l'avvio di processi aziendali, l'invio di notifiche o qualsiasi altra operazione definita nell'ambito delle regole.
- d. *Flessibilità e aggiornabilità*: I rule engine sono progettati per essere flessibili e facilmente aggiornabili. Le regole possono essere modificate o aggiunte senza dover modificare il codice dell'applicazione principale, il che consente una maggiore agilità aziendale e una rapida risposta ai cambiamenti nei requisiti aziendali.
- e. *Monitoraggio e tracciabilità*: I rule engine spesso forniscono strumenti per il monitoraggio delle regole attivate e delle azioni eseguite. Questo consente agli sviluppatori e agli amministratori di sistema di tracciare il comportamento del sistema e di risolvere eventuali problemi o discrepanze.

Nome	DesktopOS	Linguaggio	Carte Implementate	Open Source	Scripted
Forge [3]	Any (Java)	Java	27.474	✓	✓
MtG Arena (GRE) [4]	Any	C++ & CLISP	10.199 (MtGA)		✓
MtG Online	Windows	C++	Tutte (escluse edizioni scherzo)		n.d.
Xmage	Any (Java)	Java	17.218	✓	
Incantus	Any (Python)	Python	2.583	✓	✓
Magarena	Any (Java)	Java	11.854	✓	✓
BotArena	Windows	C++	10.744	✓	
Multiverse	Any	Java	1.500		✓
Wagic	Any	C++	24.570	✓	✓

Tabella 1.1: Lista dei Rule Engine per *Magic the Gathering*

### 1.3 Caso d'uso: Forge

Forge è un rule engine open-source specificamente progettato per il gioco di carte collezionabili *Magic*. Il Rule Engine Forge consente agli sviluppatori e agli appassionati di creare e simulare partite di *Magic*, gestendo le complesse regole e interazioni del gioco in modo automatico e coerente. Grazie alla sua architettura modulare e alla sua vasta base di utenti, Forge è diventato uno strumento popolare per lo sviluppo di applicazioni e simulatori basati su *Magic*.

Le principali caratteristiche di Forge sono:

- a. **Implementazione delle regole di *Magic*:** Forge implementa le regole di *Magic: The Gathering*, gestendo le interazioni tra carte, abilità, zone di gioco e turni. Ciò consente agli sviluppatori di concentrarsi sull'interfaccia utente e sulle funzionalità specifiche del loro progetto, senza dover reinventare la gestione delle regole di *Magic*.
- b. **Aggiornamenti continui:** La comunità di Forge lavora costantemente per aggiornare il rule engine con le nuove carte e le modifiche alle regole di *Magic*. Ciò garantisce che Forge rimanga aggiornato con le ultime espansioni e cambiamenti nel gioco.
- c. **Modalità di gioco supportate:** Forge supporta diverse modalità di gioco, tra cui partite singole, multiplayer, draft e sealed. Ciò consente agli sviluppatori di creare applicazioni che offrono un'ampia gamma di esperienze di gioco per i giocatori di *Magic*.
- d. **Intelligenza artificiale:** Forge include un'intelligenza artificiale (AI) che consente ai giocatori di sfidare avversari controllati dal computer.
- e. **Piattaforme supportate:** Forge è scritto in Java, il che lo rende compatibile con una vasta gamma di piattaforme, tra cui Windows, macOS,

Linux e dispositivi mobili. Questa compatibilità multi-piattaforma consente agli sviluppatori di creare applicazioni di *Magic* che possono essere giocate su una varietà di dispositivi.



## Capitolo 2

# Teoria dei Trasformatori

Nel corso di questo capitolo, esploreremo i fondamenti teorici dell'elaborazione del linguaggio naturale (NLP), compresi concetti chiave come la comprensione del linguaggio naturale e la generazione di testo. Successivamente, ci immergeremo nell'architettura dei trasformatori, un'evoluzione dei modelli di NLP, che ha introdotto un nuovo approccio basato sull'auto-attenzione per catturare il contesto globale del testo. Infine, l'attenzione si volge agli LLM, al cuore degli esperimenti svolti in questa tesi, analizzandone la struttura, il processo di prealllenamento e adattamento, e le loro applicazioni, focalizzandosi specificamente sulla generazione di script per le carte di *Magic: The Gathering*.

### 2.1 NLP

L'elaborazione del linguaggio naturale (NLP, Natural Language Processing) è una branca dell'intelligenza artificiale che si occupa della comprensione, interpretazione e generazione del linguaggio umano. L'obiettivo principale della NLP è consentire alle macchine di comprendere e comunicare con gli esseri umani in modo naturale e significativo.

Uno degli approcci più comuni per affrontare i problemi di NLP è utilizzare modelli di linguaggio, che sono modelli matematici in grado di catturare la struttura e le regole del linguaggio naturale. I modelli di linguaggio possono essere utilizzati per una vasta gamma di applicazioni, tra cui la traduzione automatica, il riconoscimento vocale, la generazione di testo e il riconoscimento di entità nominate.

La NLP utilizza una varietà di approcci e tecniche per analizzare e generare il linguaggio naturale. Alcuni dei principali metodi utilizzati nella NLP includono:

- a. **Analisi sintattica:** L'analisi sintattica si concentra sulla struttura grammaticale delle frasi, identificando le relazioni tra parole e frasi all'interno del testo. Questo processo può includere il riconoscimento delle parti del discorso, l'analisi delle dipendenze tra parole e la costruzione di alberi di parsing per rappresentare la struttura gerarchica delle frasi.
- b. **Analisi semantica:** L'analisi semantica si occupa di comprendere il significato delle parole e delle frasi nel contesto. Ciò può includere il riconoscimento di entità nominate (come nomi di persone, luoghi o organizzazioni), la risoluzione delle ancore (determinare a quale entità si riferisce un pronome) e l'estrazione delle relazioni tra entità (come "X lavora per Y").
- c. **Analisi del sentiment:** L'analisi del sentiment mira a determinare l'opinione, l'emozione o l'atteggiamento espresso nel testo. Questo processo può essere svolto a livello di parola, frase o documento e può essere utilizzato per analizzare recensioni di prodotti, post sui social media e altri tipi di testo in cui le opinioni e le emozioni sono importanti.
- d. **Generazione di testo:** La generazione di testo riguarda la creazione di nuovo testo a partire da dati strutturati o non strutturati. Questo processo può essere utilizzato per creare riassunti di articoli, risposte automatiche a domande o testo descrittivo per immagini.
- e. **Traduzione automatica:** La traduzione automatica è il processo di conversione del testo da una lingua all'altra. Questo compito è particolarmente impegnativo a causa delle differenze tra le lingue in termini di grammatica, sintassi e vocabolario.

### 2.1.1 Trasformatori

I trasformatori sono un'architettura di rete neurale introdotta da Vaswani et al. nel 2017 [5], che ha rivoluzionato il campo della NLP grazie alla sua capacità di gestire sequenze di lunghezza variabile e di catturare relazioni a lungo raggio tra parole e frasi. L'idea chiave dei trasformatori è il meccanismo di **Attenzione**, che permette al modello di "focalizzarsi" su parti specifiche della sequenza di input quando elabora l'output, pesando l'importanza relativa di ciascun elemento in base al contesto.

In un modello con meccanismo di attenzione, ogni elemento dell'output viene generato considerando l'intera sequenza di input e assegnando un peso a ciascun elemento di input. Questi pesi, chiamati "pesi di attenzione", sono

appresi dal modello durante il processo di addestramento e indicano quanto ciascun elemento dell'input è rilevante per la generazione dell'elemento corrente dell'output.

L'attenzione può essere vista come una sorta di memoria selettiva che consente al modello di tenere conto delle informazioni rilevanti e ignorare quelle irrilevanti, migliorando così la capacità di gestire sequenze di lunghezza variabile e di catturare relazioni a lungo raggio tra parole e frasi. I trasformatori sono costituiti da una serie di blocchi di codifica e decodifica, ciascuno dei quali contiene un meccanismo di attenzione multi-testa e una rete feed-forward posizionale. Questa architettura (Figura 2.1<sup>1</sup>) consente ai trasformatori di gestire efficacemente le dipendenze a lungo raggio nel testo e di apprendere rappresentazioni semantiche più ricche rispetto ai modelli precedenti.

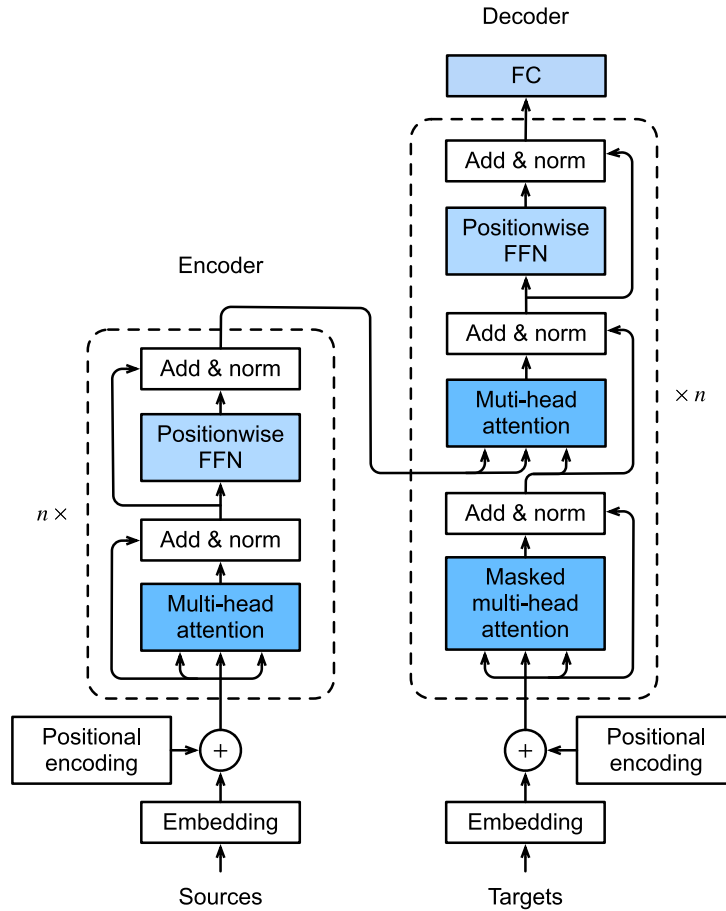


Figura 2.1: Architettura di un trasformatore

<sup>1</sup>Immagine presa dal libro *Dive into Deep Learning* [6]

Ad alto livello, l'encoder del trasformatore è costituito da una serie di strati identici, ciascuno dei quali comprende due sottolivelli (indicati come *sublayer*). Il primo sottolivello consiste in un raggruppamento di auto-attenzione multi-testa, mentre il secondo è una rete feed-forward posizionale. In particolare, nell'auto-attenzione dell'encoder, le query, le chiavi e i valori provengono tutti dagli output del livello precedente dell'encoder. Ispirandosi alla progettazione delle Reti Neurali Residue (*ResNet*), viene utilizzata una connessione residua intorno a entrambi i sottolivelli. Nel trasformatore, per qualsiasi input  $x \in \mathbb{R}^d$  presente in qualsiasi posizione della sequenza, si richiede che  $sublayer(x) \in \mathbb{R}^d$  in modo tale che la connessione residuale  $x + sublayer(x) \in \mathbb{R}^d$  sia realizzabile.

Questa aggiunta dalla connessione residua è seguita immediatamente dalla normalizzazione del livello [7]. Di conseguenza, l'encoder del trasformatore restituisce una rappresentazione vettoriale  $d$ -dimensionale per ogni posizione della sequenza di input.

Anche il decoder del trasformatore è costituito da una serie di strati identici con connessioni residue e normalizzazioni del livello. Oltre ai due sottolivelli presenti nell'encoder, il decoder introduce un terzo sottolivello, chiamato attenzione encoder-decoder, tra gli altri due. Nell'attenzione encoder-decoder, le query provengono dagli output del sottolivello di auto-attenzione del decoder, mentre le chiavi e i valori derivano dagli output dell'encoder del trasformatore. Nell'auto-attenzione del decoder, le query, le chiavi e i valori sono tutti ottenuti dagli output del livello precedente del decoder. Tuttavia, ogni posizione nel decoder può partecipare solo a tutte le posizioni nel decoder fino a quella specifica posizione. Questa attenzione mascherata mantiene la proprietà autoregressiva, garantendo che la previsione dipenda esclusivamente dai token di output generati fino a quel momento.

## 2.2 LLM

I *Large Language Models* (*LLM*) sono potenti modelli di apprendimento profondo addestrati su enormi set di dati testuali. Il loro nucleo, il trasformatore, è composto da encoder e decoder, ognuno con capacità di auto-attenzione, che permettono di estrarre significati e relazioni da sequenze di testo.

A differenza delle Reti Neurali Ricorrenti (RNN) precedenti, che elaboravano sequenzialmente gli input, i trasformatori elaborano intere sequenze in parallelo. Ciò consente un addestramento più efficiente utilizzando l'accelerazione delle GPU, riducendo i tempi di addestramento.

Grazie all'architettura dei trasformatori, gli LLM possono essere estremamente grandi, composti da centinaia di miliardi di parametri. Questi modelli

su larga scala sono in grado di assimilare enormi quantità di dati provenienti da fonti come *Internet*, il *Common Crawl* e *Wikipedia*.

Gli LLM sono estremamente flessibili e possono eseguire una vasta gamma di compiti, tra cui rispondere a domande, riassumere documenti, tradurre lingue e completare frasi. Questi modelli hanno il potenziale per trasformare la creazione di contenuti e l'utilizzo di motori di ricerca e assistenti virtuali.

Nonostante non siano perfetti, gli LLM dimostrano una notevole capacità predittiva anche con un numero relativamente limitato di input. Possono essere utilizzati per la generazione di contenuti basati sul linguaggio umano, rappresentando un importante sviluppo nell'intelligenza artificiale generativa.

Un aspetto cruciale del funzionamento degli LLM è la loro capacità di rappresentare le parole tramite vettori multidimensionali, noti come incorporamenti di parole. Questi vettori consentono al modello di comprendere il contesto e le relazioni tra le parole, superando le limitazioni delle rappresentazioni numeriche tradizionali delle parole.

Sfruttando gli incorporamenti di parole, i transformer possono elaborare il testo in forma numerica tramite l'encoder, catturando il contesto e le relazioni semantiche, e poi generare un output significativo attraverso il decoder. Esistono molte applicazioni pratiche per gli LLM tra cui: scrittura di testi, generare risposte in base alle conoscenze, generazione di testo, classificazione del testo, generazione di codice.

Nei capitoli successivi, ci concentreremo in particolare sulla loro applicazione per generare codice, in particolare come essi possano risultare efficaci nella generazione di linguaggi a dominio specifico.



# Capitolo 3

## Metodologie

Questo capitolo esplora il ruolo fondamentale del linguaggio ForgeScript, il linguaggio di scripting del rule engine Forge. Saranno analizzate le caratteristiche distintive del linguaggio di ForgeScript, evidenziandone la flessibilità e la potenza espressiva, e verrà presentato il nuovo linguaggio di scripting Lunar, progettato per migliorare la leggibilità e la gestione degli effetti complessi. Infine, sarà esaminato il processo strategico di addestramento degli LLM, attraverso l'impiego di tecniche avanzate come PEFT [8] e QLoRA [9], al fine di perfezionare la generazione automatica degli script per le carte in uscita nelle nuove espansioni.

### 3.1 Linguaggio a dominio specifico

Un *linguaggio a dominio specifico* (DSL) è un linguaggio di programmazione o di scripting progettato per un ambito particolare di problemi o applicazioni, come nel nostro caso, i giochi di carte collezionabili. A differenza dei linguaggi di programmazione general-purpose, che sono progettati per essere utilizzati in una vasta gamma di contesti, un DSL è ottimizzato per risolvere problemi specifici all'interno di un determinato dominio.

I vantaggi di utilizzare un DSL includono una maggiore espressività, una maggiore facilità di utilizzo da parte degli utenti nel contesto specifico e la possibilità di incorporare concetti e astrazioni rilevanti per il dominio di interesse. Inoltre, i DSL possono semplificare lo sviluppo e la manutenzione del codice, in quanto sono progettati per risolvere problemi specifici e non includono funzionalità superflue [10].

## 3.2 Extended Backus-Naur Form

EBNF (o *Extended Backus-Naur Form*) è una notazione formale utilizzata per descrivere la sintassi di un linguaggio di programmazione o di un linguaggio di marcatura. È basata sulla forma originale, la Backus-Naur Form (BNF), ma estende le sue capacità per includere una maggiore espressività.

Nella pratica, EBNF viene utilizzata per definire le regole grammaticali di un linguaggio, specificando come le varie parti del linguaggio possono essere combinate per formare espressioni valide. Le regole sono scritte in forma di produzioni, che indicano come combinare i simboli terminali e non terminali per formare costrutti validi nel linguaggio. Ad esempio, una regola EBNF potrebbe definire che un'espressione matematica può essere composta da un numero seguito da un operatore seguito da un altro numero [11].

## 3.3 Il linguaggio di scripting di Forge

Forge utilizza un linguaggio di scripting di nome ForgeScript, ideato dalla community di sviluppatori che contribuiscono al progetto. ForgeScript utilizza una sintassi che descrive le proprietà delle carte attraverso una struttura chiave-valore, la parte più complessa appare nella parte focale della carta, ovvero i suoi effetti, in cui vengono descritte delle variabili (**SVars** [12]) per identificare gli effetti da chiamare nella livello core del progetto. Il linguaggio non è compilato, ma consiste in un file con estensione *.txt*, che viene letto da un Parser e ne riconosce le funzioni da chiamare all'interno del livello definito *core*. Questo approccio fornisce la possibilità di poter cambiare il comportamento dello script a runtime, dando un vantaggio agli sviluppatori in fase di Debug. ForgeScript presenta anche delle problematiche per quanto concerne la leggibilità e la gestione degli effetti delle carte più complesse. Considerando la carta in Figura 3.1 per quanto il suo effetto sia tra i più semplici, è un ottimo esempio per capire come il suo equivalente di ForgeScript sia meno intuitivo.

La carta in sé permette di infliggere 3 danni ad un qualsiasi bersaglio e di guadagnare 3 punti vita. La risoluzione dei due effetti semplici avviene contemporaneamente, ma prendendo in considerazione lo script descritto dal Codice 1, si può notare come per un effetto semplice si debba creare una variabile che contenga un sotto effetto per guadagnare vita.





Figura 3.1: Carta Lightning Helix con layout Full Art

---

**Codice 1** Script della carta in Figura 3.1 in ForgeScript
 

---

```

1 Name:Lightning Helix
2 ManaCost:R W
3 Types:Instant
4 A:SP$ DealDamage | Cost$ R W | ValidTgts$ Any | NumDmg$ 3 |
   ↳ SubAbility$ DBGainLife | SpellDescription$ CARDNAME deals 3
   ↳ damage to any target and you gain 3 life.
5 SVar:DBGainLife:DB$ GainLife | LifeAmount$ 3
6 Oracle:Lightning Helix deals 3 damage to any target and you
   ↳ gain 3 life.
```

---

## 3.4 Lunar, un nuovo linguaggio

Lunar nasce dall'idea di sviluppare un nuovo motore di regole per la gestione delle carte nei giochi di carte collezionabili, fornendo il proprio linguaggio di scripting dedicato che abbia come caratteristica principale il fatto di essere facile da imparare, facilmente leggibile per un utente non sviluppatore e che sia facile da adattare alle regole dei singoli GCC. Si ispira alla struttura dei file YAML per rendere l'interazione con il linguaggio più accessibile e intuitiva, riuscendo ad superare le limitazioni analizzate dal linguaggio di scripting di Forge.

Di seguito, è presentato come potrebbe apparire la struttura di una carta di *Magic* scritta usando Lunar.

---

**Codice 2** Struttura di una carta usando Lunar espressa in EBNF

---

```
1 <card> := <name> <new_line>
2         <mana_cost> <new_line>
3         <layout> <new_line>
4         <card_type_statement> <new_line>
5         <subtype_statement> <new_line>
6         [ <supertype_statement> <new_line> ]
7         [ <keywords> ]
8         [ <effects> ]
9         [ <oracle_text> <new_line> ]
10        [ <faces> <new_line> ]
11        [ <power> <new_line> ]
12        [ <toughness> <new_line> ]
13        [ <loyalty> <new_line> ]
14        [ <defence> <new_line> ];
```

---

La sintassi riportata nel Codice 2 definisce i diversi elementi che compongono una carta, come il nome, il costo di mana, il tipo di carta, le abilità, gli effetti e così via. Questa struttura fornisce una guida su come scrivere correttamente il codice di una carta usando Lunar, garantendo coerenza e uniformità nella sua definizione. Questi elementi sono rappresentati come regole grammaticali distinte, ciascuna delle quali è seguita da una nuova riga e da un'indentazione per garantire una maggiore leggibilità.

**Codice 3** Struttura di effetto di una carta usando Lunar espressa in EBNF

```

1  <effects> := "effects: " <new_line>
2              effect_collection;
3  <keyword> := "type: " ( "simple"
4                  | "with_amount"
5                  | "with_amount_and_type"
6                  | "with_cost"
7                  | "with_cost_and_type"
8                  | "with_cost_and_amount"
9                  | "with_type" ) <new_line>
10      "name: " keyword_name new_line
11      [ "cost: " {complete_cost}+ <new_line> ]
12      [ "amount: " <number> <new_line> ]
13      [ "affected_type: " ( <supertype>
14                          | <card_type>
15                          | <subtype> )
16                          <new_line> ];
17
18  <effect_collection> := {<indent>}+ ( <activated_ability>
19                                      | <triggered_ability>
20                                      | <base_ability>
21                                      | <complex_ability> )
22      <new_line>
23      {effect_collection};

```

Di seguito viene riportato il codice scritto in Lunar della carta “Lightning Helix” raffigurata in Figura 3.1: come si può notare, il codice scritto in Lunar risulta più chiaro ed è possibile capire che entrambi gli effetti base seguano la stessa gerarchia di risoluzione della versione cartacea.

Diversamente da ForgeScribe, Lunar offre una sintassi più concisa e con una chiara struttura gerarchica: le proprietà delle carte sono definite tramite la definizione di coppie chiave-valore. Le abilità e gli effetti possono concatenarsi per ottenere nuove abilità complesse e valide nella grammatica.

Grazie a queste caratteristiche, è possibile rendere Lunar adattabile ai GCC principali, molte delle regole che compongono questi giochi possono essere espressi attraverso le regole di *Magic*. Questo ragionamento può essere facilmente applicabile non solo alle carte da gioco, ma anche alle zone del campo di gioco e alla gestione della struttura del turno.

---

### Codice 4 Script della carta in Figura 3.1 in Lunar

---

```
1 name: Lightning Helix
2 mana_cost: R W
3 card_type: instant
4 effects:
5     effect:
6         type: base
7         mode: damage
8         target: any_target
9         amount: 3
10    effect:
11        type: base
12        mode: life_gain
13        target: card_owner
14        amount: 3
15 oracle_text: <Lightning Helix deals 3 damage to any target and you
    ↪ gain 3 life.>
```

---

Di seguito due implementazioni della carta “Primeval Titan” nei due linguaggi analizzati.

Sebbene entrambi gli script definiscano la stessa carta, Lunar si distingue per la sua sintassi più pulita e intuitiva, che può rendere il processo di sviluppo e comprensione delle regole del gioco più efficiente e accessibile.

---

**Codice 5** Esempio della carta in Figura 1.1 in ForgeScript

---

```
1 Name:Primeval Titan
2 ManaCost:4 G G
3 Types:Creature Giant
4 PT:6/6
5 K:Trample
6 T:Mode$ ChangesZone | Origin$ Any | Destination$ Battlefield |
  ↳ ValidCard$ Card.Self | Execute$ TrigChange | OptionalDecider$
  ↳ You | TriggerDescription$ Whenever CARDNAME enters the
  ↳ battlefield or attacks, you may search your library for up to
  ↳ two land cards, put them onto the battlefield tapped, then
  ↳ shuffle.
7 T:Mode$ Attacks | ValidCard$ Card.Self | Execute$ TrigChange |
  ↳ TriggerZones$ Battlefield | OptionalDecider$ You | Secondary$
  ↳ True | TriggerDescription$ Whenever CARDNAME enters the
  ↳ battlefield or attacks, you may search your library for up to
  ↳ two land cards, put them onto the battlefield tapped, then
  ↳ shuffle.
8 SVar:TrigChange:DB$ ChangeZone | Origin$ Library | Destination$
  ↳ Battlefield | Tapped$ True | ChangeType$ Land | ChangeNum$ 2 |
  ↳ ShuffleNonMandatory$ True
9 SVar:HasAttackEffect:TRUE
10 Oracle:Trample\nWhenever Primeval Titan enters the battlefield or
  ↳ attacks, you may search your library for up to two land cards,
  ↳ put them onto the battlefield tapped, then shuffle.
```

---

### Codice 6 Esempio della carta in Figura 1.1 in Lunar

---

```
1 name: Primeval Titan
2 mana_cost: 4 G G
3 layout: single_face
4 card_type: creature
5 subtype: giant
6 keywords:
7     keyword:
8         type: simple
9         name: trample
10 effects:
11     effect:
12         type: trigger
13         event:
14             mode: change_zone
15             who: self
16             from: anywhere
17             to: battlefield
18             optional_choice: yes
19             optional_decider: card_owner
20         event:
21             mode: attack
22             who: self
23             optional_choice: yes
24         effect:
25             type: base
26             mode: move
27             from: library
28             to: battlefield
29             target: land
30             amount: 2
31             how: tapped
32         effect:
33             base:
34                 mode: shuffle
35                 target: library
36 oracle_text: <Trample\nWhenever Primeval Titan enters the
    ↪ battlefield or attacks, you may search your library for up to
    ↪ two land cards, put them onto the battlefield tapped, then
    ↪ shuffle.>
37 power: 6
38 toughness: 6
```

---

## 3.5 PEFT, LoRA e QLoRa

Prima di introdurre questi concetti, occorre dare una piccola panoramica su cosa si intende per **fine-tuning**. Il fine-tuning nei modelli di linguaggio è un processo cruciale per adattare modelli pre-addestrati alle esigenze specifiche di un compito particolare. Vediamo cosa significa nel contesto degli LLM:

- a. **Pre-Addestramento:** Gli LLM vengono inizialmente addestrati su grandi quantità di testo non etichettato. Questo processo, chiamato pre-addestramento, consente ai modelli di apprendere rappresentazioni linguistiche generali e di catturare strutture linguistiche complesse.
- b. **Fine-Tuning:** Dopo il pre-addestramento, i modelli possono essere adattati a compiti specifici attraverso il fine-tuning. Il fine-tuning coinvolge l'addestramento del modello su un set di dati etichettato specifico per il compito desiderato. Ad esempio, classificazione di testo, traduzione automatica o generazione di testo. Durante il fine-tuning, i pesi del modello pre-addestrato vengono aggiornati in base ai nuovi dati etichettati. Tuttavia, solo una piccola parte dei parametri viene addestrata, mantenendo gran parte della conoscenza pre-esistente del modello.
- c. **Vantaggi del Fine-Tuning:**
  - *Efficienza:* Il fine-tuning richiede meno risorse rispetto all'addestramento completo del modello.
  - *Adattabilità:* Gli LLM pre-addestrati possono essere utilizzati come base per una vasta gamma di compiti senza dover ripartire da zero.
  - *Prestazioni:* Il fine-tuning consente di ottenere prestazioni competitive su compiti specifici con un investimento computazionale ridotto.

### 3.5.1 Parameter-Efficient Fine-Tuning

PEFT (Parameter-Efficient Fine-Tuning) è un metodo di elaborazione del linguaggio naturale [8] che offre un modo efficiente per adattare grandi modelli pre-addestrati a diversi compiti specifici senza dover effettuare il fine-tuning di tutti i parametri del modello, il che sarebbe proibitivamente costoso.

Invece, i metodi PEFT addestrano solo un piccolo numero di parametri aggiuntivi, riducendo significativamente i costi computazionali e di archivia-

zione, pur ottenendo prestazioni comparabili a un modello al quale è stato applicato un processo di fine-tuning completo.

PEFT offre diversi vantaggi nell'addestramento dei modelli di linguaggio pre-addestrati (PLM):

- a. **Efficienza dei parametri:** PEFT riduce il numero di parametri addestrabili rispetto all'addestramento completo del modello. Questo è particolarmente utile quando si dispone di risorse computazionali limitate o si desidera addestrare rapidamente un modello.
- b. **Adattatori:** PEFT utilizza adattatori, che sono piccoli strati di rete aggiunti ai blocchi del modello di base. Questi adattatori vengono addestrati separatamente e collegati al modello di base. Ciò consente di adattare il modello a specifici compiti senza dover addestrare l'intero modello da zero.
- c. **Riduzione del rischio di overfitting:** Poiché PEFT addestra solo gli adattatori, il rischio di overfitting è ridotto rispetto all'addestramento completo del modello.
- d. **Velocità di addestramento:** PEFT è più veloce dell'addestramento completo del modello, poiché richiede meno iterazioni e meno calcoli.

In sintesi, PEFT offre un modo efficiente per adattare i modelli di linguaggio pre-addestrati a compiti specifici, mantenendo un buon equilibrio tra prestazioni e risorse computazionali.

### 3.5.2 Low-Rank Adaptation

LoRA (Low-Rank Adaptation) è una tecnica utilizzata per ridurre la complessità computazionale delle reti neurali durante la distribuzione su dispositivi con risorse limitate, come smartphone, dispositivi per la casa intelligente e sistemi embedded. LoRA consente di adattare modelli pre-addestrati a specifici compiti o domini senza dover ricalcolare tutti i parametri del modello.

LoRA affronta il problema dell'addestramento in due modi fondamentali. Prima di tutto, anziché aggiornare direttamente tutti i pesi del modello, traccia le modifiche desiderate su tali pesi. Sebbene ciò possa implicare l'aggiunta di ulteriori informazioni da memorizzare, il vantaggio principale risiede nel fatto che le modifiche ai pesi del modello sono tracciate in due matrici separate più piccole. Queste matrici vengono moltiplicate insieme per formare una matrice delle dimensioni del livello della rete neurale al quale si sta applicando il fine-tuning.



Una volta ottenuta questa matrice che rappresenta le modifiche alla traccia iniziale, basterà sommare ciascun elemento di essa alla matrice dei pesi del modello originale.

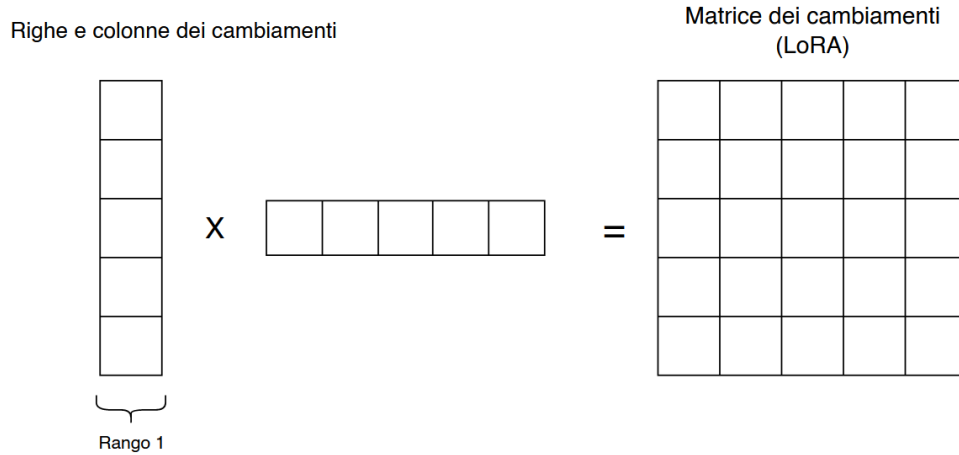


Figura 3.2: Decomposizione di rango

Soffermendosi nel dettaglio sull'esempio in Figura 3.2, si ottiene la cosiddetta una decomposizione di rango. In questo caso, le matrici di rango 1, contenenti un totale di cinque numeri ciascuna, vengono moltiplicate per formare una matrice di fine-tuning con 25 parametri. Questa strategia sacrifica una certa precisione per un notevole vantaggio in termini di efficienza.

Dato che si lavora con un numero ridotto di parametri addestrabili, il rango può essere modificato per regolare la precisione dell'output finale. Con un rango più alto si può addestrare una percentuale maggiore dei parametri del modello originale. Tuttavia, poiché i compiti specifici spesso costituiscono solo un sottoinsieme delle capacità del modello, ridurre il rango potrebbe non comportare compromessi significativi in molte situazioni. La capacità di ottenere risultati accurati con aggiornamenti approssimativi, come dimostrato nella ricerca [13], è ciò che rende LoRA una soluzione rilevante per questi contesti.

### 3.5.3 Quantized Low-Rank Adaptation

QLoRA (Quantized Low-Rank Adaptation) è un approccio che combina **quantizzazione** e LoRA. Nel contesto degli LLM, la quantizzazione si riferisce al processo di conversione dei pesi del modello da tipi di dati a precisione

più elevata a quelli a precisione inferiore. In altre parole, i parametri del modello vengono approssimati utilizzando meno bit per rappresentare i loro valori. Questo aiuta a ridurre la memoria richiesta per l'archiviazione dei pesi del modello e a migliorare l'efficienza computazionale durante l'addestramento e l'inferenza. Di seguito è presentato il funzionamento (Figura 3.3<sup>1</sup>):

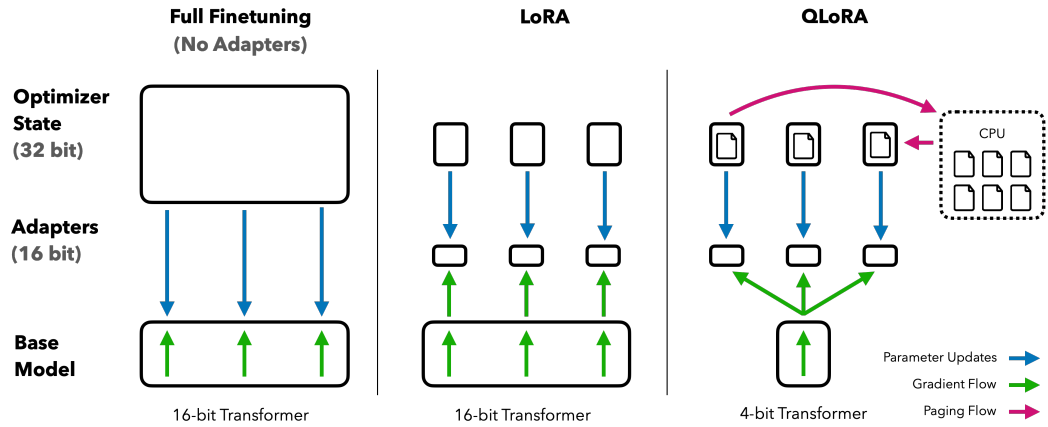


Figura 3.3: Metodi di fine-tuning a confronto

QLoRA utilizza una tecnica chiamata quantizzazione a 4 bit per comprimere i parametri di un modello linguistico preaddestrato. Questo processo riduce notevolmente lo spazio occupato in memoria senza compromettere le prestazioni del modello.

Una volta che i parametri del modello linguistico sono stati compressi, vengono congelati, il che significa che non vengono più aggiornati durante il fine-tuning. In seguito, vengono aggiunti agli strati del modello un numero limitato di parametri addestrabili (LoRA). Durante il fine-tuning, i gradienti vengono propagati all'indietro attraverso il modello linguistico preaddestrato compresso fino agli Adattatori LoRA, che sono gli unici parametri aggiornati durante l'addestramento.

Un aspetto importante di QLoRA è che utilizza due tipi di dati differenti: un tipo di dato di memorizzazione a 4 bit per i pesi del modello base e un tipo di dato di calcolo a 16 bit per eseguire i calcoli durante il fine-tuning. Questo approccio consente di ridurre l'utilizzo della memoria durante l'addestramento e l'inferenza, poiché i pesi vengono decompressi solo quando sono necessari per eseguire i calcoli.

QLoRA ha portato molti vantaggi nell'addestramento degli LLM, tra cui:

<sup>1</sup>Immagine dal paper QLoRA: Efficient Finetuning of Quantized LLMs [9]

- a. **Efficienza:** QLoRA è in grado di elaborare grandi sequenze di dati in modo molto più efficiente rispetto agli LLM tradizionali. Ciò è dovuto al suo meccanismo di attenzione locale basato sulle query, che consente al modello di concentrarsi sulle parti rilevanti dei dati in ingresso.
- b. **Efficacia:** QLoRA può raggiungere un'accuratezza paragonabile a quella degli LLM tradizionali, pur richiedendo risorse computazionali significativamente inferiori. Questo lo rende un'opzione più interessante per gli ambienti con risorse limitate.
- c. **Accessibilità:** L'efficienza di QLoRA lo rende un'opzione più accessibile per i ricercatori e gli sviluppatori che non hanno accesso alle stesse risorse delle grandi istituzioni. Ciò contribuisce a democratizzare l'accesso ai LLM e ad accelerarne l'adozione in diversi settori.
- d. **Prestazioni:** QLoRA è molto efficace nella comprensione e nella generazione del linguaggio naturale. Questo lo rende uno strumento prezioso per le applicazioni che richiedono una profonda comprensione del contesto, come la traduzione linguistica, la creazione di contenuti e persino la risoluzione di problemi complessi.
- e. **Applicazioni in tempo reale:** La capacità di QLoRA di elaborare le informazioni in modo rapido e preciso lo rende ideale per le applicazioni in tempo reale. Ciò è particolarmente significativo in campi come il servizio clienti, dove l'intelligenza artificiale può fornire risposte immediate e contestualmente rilevanti alle richieste degli utenti.

Grazie a QLoRA, è possibile eseguire il fine-tuning di modelli linguistici estremamente grandi su GPU con una quantità limitata di memoria. Inoltre, numerosi esperimenti hanno dimostrato che QLoRA produce risultati comparabili ai metodi tradizionali di fine-tuning a 16 bit, garantendo al contempo una gestione efficiente delle risorse di calcolo e di memoria.



# Capitolo 4

## Esperimenti e risultati

Nel presente capitolo, verrà fornita un’esaustiva descrizione della metodologia impiegata nell’esperimento condotto per addestrare i diversi LLM utilizzando il framework QLoRA insieme al tool di Hugging Face denominato Autotrain Advanced. Tale metodologia ha consentito di configurare agevolmente i parametri di addestramento e, grazie all’infrastruttura di calcolo ad alte prestazioni dell’Ateneo (HPC), a schedulare diverse attività di addestramento per diverse epoche. Successivamente verranno presentate le valutazioni ed i risultati ottenuti durante gli esperimenti.

### 4.1 Preparazione del dataset

Per questo esperimento è stata identificata una struttura di dati comune per poter addestrare gli LLM. Il dataset è stato preso da **Scryfall**, un database completo di tutte le carte di *Magic* che espone una serie di API con le quali è possibile ottenere l’intera collezione di carte organizzate sotto una struttura di un oggetto JSON simile a come descritta in Codice 7. Il dataset è stato elaborato<sup>1</sup> utilizzando la libreria *pandas*. È stata effettuata una pulitura delle proprietà che non fossero utili al fine di generare gli script (note legali, immagine, nome dell’artista, ecc.). Una sfida relativa al dataset è stata quella di identificare i tipi di layout delle carte utilizzate in *Magic*: durante i diversi anni di progettazione, i designer di *Wizards of the Coast* hanno ideato diversi modi creativi di utilizzare le carte da gioco, alcune delle quali sopravvissute alle sfide del tempo. All’interno del dataset questo tipo di carte con diverse modalità di lettura (fronte-retro, carte split, carte ruotabili, ecc.) vengono rappresentate come una collezione di due carte, ovvero è presente

<sup>1</sup>Notebook disponibile al repository [ZarakiKanzaki/project-lunar-ML](https://github.com/ZarakiKanzaki/project-lunar-ML) su GitHub

## CAPITOLO 4. ESPERIMENTI E RISULTATI

una proprietà *faces* che contiene le regole di ogni faccia della carta. Sebbene a livello di gioco possono esserci diverse regole per il tipo di layout, l'unico punto di interesse risiede nel dover descrivere gli effetti. Alcuni esempi di carte con layout diversi è presentato in figura 4.1.

---

### Codice 7 Esempio di input da convertire

---

```
1 {
2     "name": "Primeval Titan",
3     "mana_cost": "{4}{G}{G}",
4     "type_line": "Creature \u2014 Giant",
5     "oracle_text": "Trample\nWhenever Primeval Titan enters the
    ↪ battlefield or attacks, you may search your library for
    ↪ up to two land cards, put them onto the battlefield
    ↪ tapped, then shuffle.",
6     "power": "6",
7     "toughness": "6"
8 }
```

---



Figura 4.1: Carte con layout differenti

Sono state analizzate circa 35.000 carte all'interno delle quali si possono identificare 20.000 effetti unici, alcuni sviluppatori hanno tentato di utilizzare una ResNet ottenendo un'accuratezza del 50% [14].

Una volta effettuata questa esplorazione sui dati, è stato preparato un dataset<sup>2</sup> per addestrare gli LLM, si è scelto di utilizzare un file *csv* contenente 4 colonne: istruzione, input, risposta ed il campo testo, che contiene il prompt

<sup>2</sup>disponibile al repository di HuggingFace [404NotF0und/MtG-json-to-ForgeScript](https://huggingface.co/404NotF0und/MtG-json-to-ForgeScript)

utilizzato per l'addestramento. Di seguito è riportata la struttura di un singolo prompt per l'addestramento (Codice 8) ed un esempio di prompt nel Codice 9.

---

**Codice 8** Struttura prompt per l'addestramento

---

```

1 Below is an instruction that describes a task then the input
  ↪ and response.
2 ### Instruction: Create the Forge script for this magic card
3 ### Input: <Il JSON della carta da generare>
4 ### Response: <Lo script scritto in ForgeScript della carta>

```

---



---

**Codice 9** Esempio di prompt per l'addestramento

---

```

1 Below is an instruction that describes a task then the input and response.
2 ### Instruction: Create the Forge script for this magic card
3 ### Input: {"name": "Primeval Titan", "mana_cost": "{4}{G}{G}", "type_line": "Creature
  ↪ \u2014 Giant", "oracle_text": "Trample\nWhenever Primeval Titan enters the battlefield
  ↪ or attacks, you may search your library for up to two land cards, put them onto the
  ↪ battlefield tapped, then shuffle.", "power": "6", "toughness": "6"}
4 ### Response: Name:Primeval Titan\nManaCost:4 G G\nTypes:Creature
  ↪ Giant\nPT:6/6\nK:Trample\nT:Mode$ ChangesZone | Origin$ Any | Destination$ Battlefield
  ↪ | ValidCard$ Card.Self | Execute$ TrigChange | OptionalDecider$ You |
  ↪ TriggerDescription$ Whenever CARDNAME enters the battlefield or attacks, you may search
  ↪ your library for up to two land cards, put them onto the battlefield tapped, then
  ↪ shuffle.\nT:Mode$ Attacks | ValidCard$ Card.Self | Execute$ TrigChange | TriggerZones$
  ↪ Battlefield | OptionalDecider$ You | Secondary$ True | TriggerDescription$ Whenever
  ↪ CARDNAME enters the battlefield or attacks, you may search your library for up to two
  ↪ land cards, put them onto the battlefield tapped, then shuffle.\nSVar:TrigChange:DB$
  ↪ ChangeZone | Origin$ Library | Destination$ Battlefield | Tapped$ True | ChangeType$
  ↪ Land | ChangeNum$ 2 | ShuffleNonMandatory$
  ↪ True\nSVar:HasAttackEffect:TRUE\nOracle:Trample\nWhenever Primeval Titan enters the
  ↪ battlefield or attacks, you may search your library for up to two land cards, put them
  ↪ onto the battlefield tapped, then shuffle.

```

---

## 4.2 Autotrain Advanced

AutoTrain di Hugging Face è uno strumento no-code per addestrare modelli all'avanguardia per compiti di Elaborazione del Linguaggio Naturale (NLP), compiti di Computer Vision (CV), compiti tabellari e per compiti di elaborazione e generazione di Immagini. Utilizzando AutoTrain Advanced, gli utenti avanzati possono controllare gli iperparametri utilizzati per l'addestramento di un qualsiasi modello, consentendo di addestrare più modelli con iperparametri diversi e confrontare i risultati.

### 4.2.1 Addestramento

Inizialmente, si era pensato di utilizzare il servizio gratuito di Google Colab o Kaggle per l'addestramento, ma a causa delle limitazioni hardware e dei problemi legati al timeout delle versioni gratuite, si è deciso di richiedere l'accesso all'infrastruttura di calcolo ad alte prestazioni (High Performance Computing) dell'Ateneo.

Per gli esperimenti, si è scelto di utilizzare una GPU NVIDIA A100 da 80GB. L'addestramento è stato effettuato per diversi tagli di epoche, nello specifico 3, 10 e 20 epoche, al fine di arginare la limitazione delle 24 ore di utilizzo di una risorsa impostata dal HPC. I modelli di linguaggio utilizzati durante l'addestramento sono: Llama2-7B, OpenHermes-7B, TinyLlama, phi-2, Mistral-7B e Orca-2-13B.

Di seguito vengono spiegati i parametri del comando utilizzato per ogni addestramento, sia per il taglio di epoche che per gli LLM utilizzati, il comando è descritto dal Codice 10.

---

#### Codice 10 Comando per l'addestramento con Autotrain Advanced

---

```
1 autotrain llm \  
2 --train \  
3 --model $LLM_FULLNAME \  
4 --project-name lunar-llm-$LLM_NAME \  
5 --data-path /MtG-json-to-ForgeScript/ \  
6 --text-column text \  
7 --lr 1e-4 \  
8 --batch-size $BATCH_SIZE \  
9 --epochs $EPOCHS \  
10 --block-size 1024 \  
11 --warmup-ratio 0.1 \  
12 --lora-r 64 \  
13 --lora-alpha 16 \  
14 --lora-dropout 0.1 \  
15 --weight-decay 0.01 \  
16 --gradient-accumulation 4 \  
17 --quantization int4 \  
18 --mixed-precision fp16 \  
19 --merge_adapter \  
20 --peft\  
21 --push-to-hub --token $HF_TOKEN \  
22 --repo-id 404NotFound/lunar-llm-$LLM_NAME
```

---

I parametri utilizzati nel comando sono:

- a. `--warmup-ratio 0.1`: Il rapporto di riscaldamento (warmup ratio) è utilizzato nell'ottimizzazione con un learning rate variabile. Durante



le prime epoche dell'addestramento, il learning rate viene aumentato gradualmente fino a raggiungere il valore massimo. Il rapporto di riscaldamento indica la frazione di epoche totali utilizzate per il riscaldamento. In questo caso, il 10% delle epoche totali viene utilizzato per il riscaldamento.

- b. `--lora-r 64`: Il parametro  $r$  di LoRA (Layer-wise Relevance Propagation) indica il numero di vettori di rango basso che vengono utilizzati per approssimare i pesi del modello. Un valore più elevato di  $r$  può migliorare l'accuratezza dell'approssimazione, ma aumenta anche la complessità computazionale.
- c. `--lora-alpha 16`: Il parametro  $\alpha$  di LoRA controlla il bilanciamento tra la sparsità e la precisione dell'approssimazione. Un valore più elevato di  $\alpha$  favorisce la sparsità a scapito della precisione.
- d. `--lora-dropout 0.1`: Il dropout di LoRA è una tecnica di regolarizzazione utilizzata per prevenire l'overfitting durante l'addestramento. Il dropout viene applicato ai vettori di rango basso in LoRA, e il parametro specifica la probabilità che un elemento venga azzerato.
- e. `--weight-decay 0.01`: Il decadimento del peso (weight decay) è un altro metodo di regolarizzazione che aggiunge una penalità L2 alla funzione obiettivo. Il parametro controlla l'intensità della penalità: valori più alti corrispondono a una regolarizzazione più forte.
- f. `--gradient-accumulation 4`: L'accumulo del gradiente è una tecnica utilizzata per ridurre la frequenza degli aggiornamenti del modello durante l'addestramento. Invece di aggiornare il modello ad ogni passo, i gradienti vengono accumulati per un certo numero di passi e poi aggiornati. Questo può migliorare la stabilità e la qualità dell'addestramento, specialmente quando si utilizzano batch di dimensioni ridotte.
- g. `--quantization int4`: La quantizzazione è una tecnica utilizzata per ridurre la dimensione e la complessità computazionale del modello. In questo caso, la quantizzazione int4 implica che i pesi del modello vengono rappresentati con interi a 4 bit, riducendo così la dimensione del modello e accelerando l'inferenza.
- h. `--mixed-precision fp16`: La precisione mista si riferisce all'utilizzo di diversi tipi di dati per rappresentare i pesi, gli attivazioni e i gradienti durante l'addestramento. In questo caso, la precisione mista fp16

indica che i calcoli vengono eseguiti in parte con numeri in virgola mobile a 16 bit (half-precision) e in parte con numeri in virgola mobile a 32 bit (single-precision). Questo può accelerare l'addestramento e ridurre l'utilizzo di memoria senza compromettere significativamente l'accuratezza del modello.

### 4.2.2 Funzione obiettivo

Autrain Advanced utilizza come funzione obiettivo la *cross Entropia*. La cross entropia è una funzione di perdita comunemente utilizzata nell'addestramento degli LLM e di altri modelli di classificazione basati sull'apprendimento profondo. La cross entropia misura la differenza tra due distribuzioni di probabilità, in questo caso, la distribuzione di probabilità prevista dal modello e la distribuzione di probabilità vera dei dati. Durante l'addestramento, l'obiettivo è minimizzare la cross entropia, il che significa che si cerca di avvicinare il più possibile la distribuzione di probabilità prevista dal modello a quella dei dati reali.

Nel contesto degli LLM, la cross entropia viene utilizzata per misurare la differenza tra le probabilità delle parole previste dal modello e le parole effettivamente osservate nel testo di addestramento. Poiché gli LLM sono modelli generativi, durante l'addestramento cercano di prevedere la parola successiva in una sequenza di testo, data una sequenza di parole precedenti. La cross entropia viene utilizzata per quantificare la "sorpresa" del modello rispetto alle parole osservate, penalizzando le previsioni errate e premiando quelle corrette.

La cross entropia  $H(p, q)$  tra due distribuzioni di probabilità  $p$  e  $q$  è definita come:

$$H(p, q) = - \sum_i p(i) \log q(i)$$

dove  $i$  indica un evento (ad esempio, una parola nel contesto degli LLM),  $p(i)$  è la probabilità dell'evento  $i$  secondo la distribuzione di probabilità vera  $p$ , e  $q(i)$  è la probabilità dell'evento  $i$  secondo la distribuzione di probabilità prevista dal modello  $q$ . Il logaritmo è tipicamente in base 2 o in base  $e$  (logaritmo naturale).

Durante l'addestramento di un LLM, la cross entropia viene calcolata su tutto il testo di addestramento e viene utilizzata per aggiornare i parametri del modello tramite tecniche di ottimizzazione, come la discesa del gradiente. Poiché la cross entropia tiene conto delle probabilità di tutte le parole nel vocabolario, essa incoraggia il modello a generare previsioni accurate non solo per le parole comuni, ma anche per quelle meno frequenti, migliorando

la capacità del modello di catturare la struttura e le sfumature del linguaggio naturale [15].

In sintesi, la cross entropia è una funzione di perdita fondamentale nell'addestramento degli LLM, poiché misura la differenza tra le probabilità previste dal modello e le probabilità vere dei dati. Minimizzare la cross entropia durante l'addestramento permette al modello di apprendere rappresentazioni linguistiche accurate e coerenti, migliorando la sua capacità di comprendere e generare il linguaggio naturale.

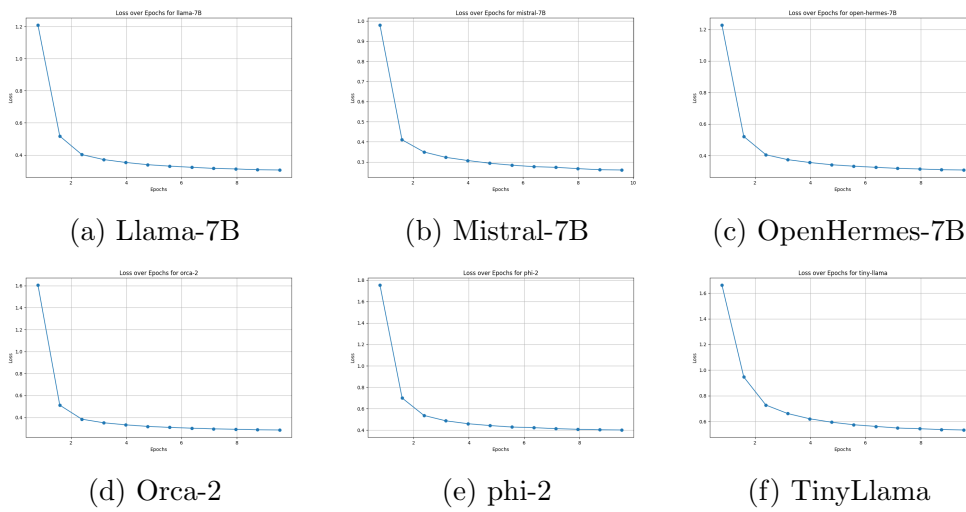


Figura 4.2: Funzione di cross-Entropia per i vari llm

### 4.2.3 Metriche di valutazione

Per prima cosa occorre identificare quali metriche utilizzate in questo esperimento:

- a. **Perplexity:** La perplexity è una misura di quanto bene un modello di linguaggio probabilistico riesce a prevedere un campione di testo. Essa corrisponde all'inverso della radice n-esima della probabilità del prodotto assegnata al testo dal modello, dove n è il numero di parole nel testo. Un valore di perplexity più basso indica che il modello ha una maggiore probabilità di prevedere correttamente il testo e, in generale, un modello con una perplexity minore è considerato migliore. La formula per calcolare la perplexity è la seguente:

$$PP(W) = \sqrt[N]{\frac{1}{P(w_1, w_2, \dots, w_N)}}$$

dove  $PP(W)$  è la perplexity,  $P(w_1, w_2, \dots, w_N)$  è la probabilità del prodotto assegnata al testo dal modello e  $N$  è il numero di parole nel testo.

- b. **Edit Distance:** L'edit distance, o distanza di Levenshtein, è una metrica che quantifica la differenza tra due stringhe di caratteri. Essa rappresenta il numero minimo di singole operazioni di modifica (inserimenti, eliminazioni o sostituzioni di caratteri) necessarie per trasformare una stringa nell'altra. Un valore di edit distance più basso indica che le due stringhe sono più simili tra loro. L'edit distance è spesso utilizzata per valutare la qualità delle traduzioni automatiche o delle correzioni ortografiche.
- c. **Accuracy:** L'accuracy è una metrica utilizzata per valutare la performance di un modello di classificazione. Essa rappresenta il rapporto tra il numero di predizioni corrette e il numero totale di predizioni effettuate. L'accuracy varia tra 0 e 1, dove un valore più vicino a 1 indica una maggiore correttezza delle predizioni del modello. La formula per calcolare l'accuracy è la seguente:

$$Accuracy = \frac{\text{Numero di predizioni corrette}}{\text{Numero totale di predizioni}}$$

Queste metriche forniscono modi quantitativi per valutare le prestazioni dei modelli di linguaggio su vari compiti. Tuttavia, è importante notare che nessuna metrica può catturare completamente tutte le sfumature delle prestazioni del modello, quindi spesso è utile utilizzare una combinazione di diverse metriche per ottenere una visione più completa delle prestazioni del modello. Nel Codice 11 è possibile vedere il codice utilizzato per la computazione delle metriche durante gli esperimenti di valutazione.

Di seguito, in Figura 4.3 e Tabella 4.1, è possibile trovare gli andamenti delle metriche per alcuni degli LLM analizzati.

**Codice 11** Script per la valutazione oggettiva di un LLM

```

1 def calculate_metrics(data, dataset_name, path_to_save, model, tokenizer, batch_size=36):
2     num_batches = int(np.ceil(len(data) / batch_size))
3
4     for batch_idx in tqdm(range(num_batches), desc=f"Processing {dataset_name}",
5         ↪ unit="batch"):
6         inputs = tokenizer(''sample_in_batch_data'', return_tensors="pt", padding=True,
7         ↪ truncation=True)
8         with torch.no_grad():
9             outputs = model(**inputs, labels=inputs["input_ids"])
10            loss = outputs.loss.mean()
11            perplexity = torch.exp(loss).item()
12
13            generated_outputs = model.generate(**inputs, max_new_tokens=300)
14            generated_texts = [tokenizer.decode(output, skip_special_tokens=True) for output in
15            ↪ generated_outputs]
16
17            for idx, (json_input, target_dsl) in enumerate(batch_data):
18                # ..
19                accuracy = calculate_token_level_accuracy([generated_tokens], [target_tokens])
20                edit_distance = wer_metric.compute(predictions=[generated_dsl],
21                ↪ references=[target_dsl])
22                # append results
23
24            # saving the metrics
25
26            return df.mean().to_dict()

```

LLM	Organizzazione	Parametri	Media Perplexity	Varianza Perplexity
Mistral	Mistral AI	7 Miliardi	460,562510	$2,910612 \cdot 10^5$
Orca 2	Microsoft	13 Miliardi	205,080949	$1,639549 \cdot 10^5$
Phi-2	Microsoft	2.7 Miliardi	208,279878	$2,069949 \cdot 10^4$
TinyLlama	TinyLlama	1.1 Miliardi	12494,401646	$8,721912 \cdot 10^8$

Tabella 4.1: Media e varianza di Perplexity per ogni LLM

## CAPITOLO 4. ESPERIMENTI E RISULTATI

---

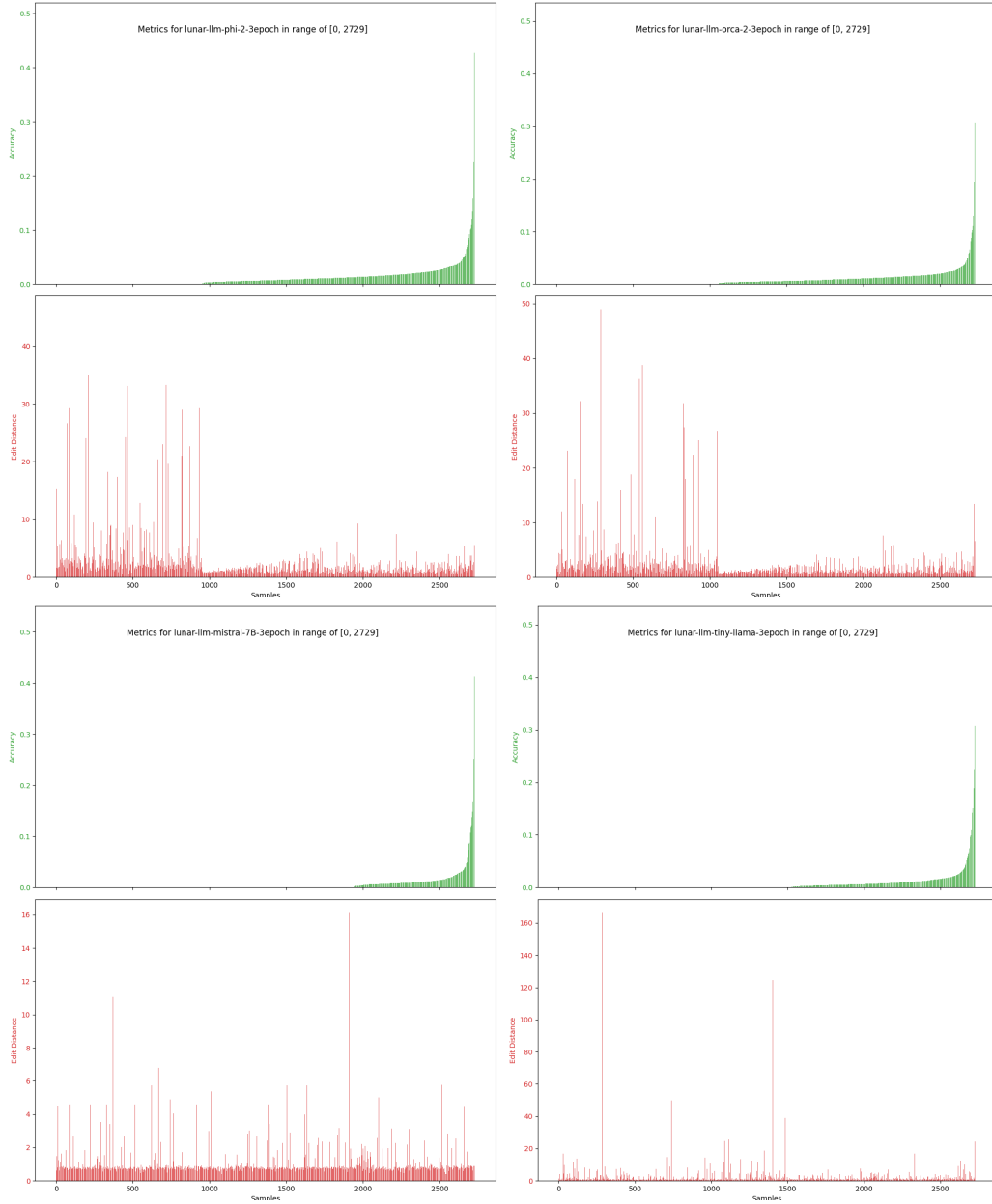


Figura 4.3: Andamenti di Edit Distance e Accuratezza per LLM

#### 4.2.4 Test qualitativo

In questa sotto sezione, viene descritto il test qualitativo condotto per valutare i modelli di linguaggio addestrati, considerando diversi aspetti rilevanti per un'azienda che desidera investire nello sviluppo del proprio GCC investendo nel campo dell'intelligenza artificiale per la generazione degli script delle carte.

Gli aspetti presi in considerazione sono:

- a. Numero di epoche necessarie per raggiungere un risultato accettabile, da cui ne deriva il relativo costo energetico per l'affitto di una GPU per l'addestramento.
- b. Dimensione del modello, tenendo conto dell'hosting dell'inferenza su una macchina virtuale nel cloud.
- c. Tempo di risposta del modello dato l'input in esame.
- d. Qualità e verosimiglianza dell'output generato rispetto al risultato atteso.

Il test è stato effettuato caricando l'inferenza del modello e richiedendo ai vari modelli di generare lo script di diverse carte dell'ultima espansione di *Magic*, quindi carte mai viste dal modello durante l'addestramento. Per confrontare qualitativamente i diversi LLM, si fa riferimento alla tabella 4.2. Il codice utilizzato per effettuare il test è mostrato nel Codice 12, mentre un esempio di output ottenuto è presentato nel Codice 13.

Analizzando i risultati ottenuti, è possibile valutare le prestazioni dei vari modelli in termini di epoche necessarie per raggiungere un risultato accettabile, costo energetico per l'affitto di una GPU, dimensione del modello e tempo di risposta. Inoltre, confrontando gli output generati, è possibile identificare il modello che produce risultati qualitativamente migliori e più verosimili rispetto a quelli attesi. Questa analisi consente di prendere decisioni informate sull'investimento nel GCC digitale e sulla scelta del modello più adatto alle esigenze dell'azienda.

LLM	Organizzazione	Parametri	Epoche	Risultato qualitativo	Tempo di risposta (hh:mm:ss)	Dimensione (GB)
OpenHermes	Teknium	7 miliardi	10	✓	2:27:59	11
Mistral	Mistral AI	7 Miliardi	10	✓	23:32	15
Orea 2	Microsoft	13 Miliardi	3	✓	14:23	27
Phi-2	Microsoft	2.7 Miliardi	3	✓	5:06	5.5
TinyLlama	TinyLlama	1.1 Miliardi	20		9:43	2.2

Tabella 4.2: LLM a confronto

È importante aggiungere che il test qualitativo è stato effettuato anche su alcuni LLM senza fine tuning, come ChatGPT 3.5 e 4. Tuttavia, il risultato del test per questi modelli è stato insoddisfacente, poiché il prompt è stato semplicemente rielaborato, senza generare lo script di una carta come richiesto. Questo evidenzia l'importanza del fine tuning per ottenere risultati migliori e più accurati in base alle specifiche esigenze del dominio applicativo.

---

### Codice 12 Script di inferenza di un LLM

---

```
1 def run_inference(model, tokenizer, prompt, max_length=300):
2     # Encode the prompt text
3     input_ids = tokenizer.encode(prompt, return_tensors='pt')
4
5     # Generate text using the model
6     output_sequences = model.generate(
7         input_ids=input_ids,
8         max_length=max_length,
9         temperature=0.5,
10        top_k=50,
11        top_p=0.95,
12        pad_token_id=tokenizer.eos_token_id,
13        do_sample=True
14    )
15
16    # Decode the generated text
17    generated_text = tokenizer.decode(output_sequences[0],
18        ↪ skip_special_tokens=True)
19
19    return generated_text
```

---



**Codice 13** Esempio di output ottenuto da microsoft/phi-2

```

1  ### Question: Create the Forge script for this magic card
2  {
3      "name": "Wrench",
4      "mana_cost": "{W}",
5      "type_line": "Artifact| Clue Equipment",
6      "oracle_text": "Equipped creature gets +1/+1 and has vigilance
    ↪ and \"{3}, {T}: Tap target creature.\"\\n{2}, Sacrifice
    ↪ CARD_NAME: Draw a card.\\nEquip {2}"
7  }
8  ### Response:
9  Name:Wrench
10 ManaCost:W
11 Types portfolios:Artifact|Clue Equipment
12 S:Mode$ Continuous | Affected$ Creature.EquippedBy | AddPower$ 1 |
    ↪ AddToughness$ 1 | AddKeyword$ Vigilance | AddAbility$ Tap |
    ↪ Description$ Equipped creature gets +1/+1 and has vigilance and
    ↪ \"{3}, {T}: Tap target creature.\"
13 SVar:Tap:AB$ Tap | Cost$ 3 T | ValidTgts$ Creature | TgtPrompt$
    ↪ Select target creature | SpellDescription$ Tap target creature.
14 A:AB$ Draw | Cost$ 2 Sac<1/CARDNAME> | NumCards$ 1 |
    ↪ SpellDescription$ Draw a card.\\nK:Equip:2
15 Oracle:Equipped creature gets +1/+1 and has vigilance and \"{3},
    ↪ {T}: Tap target creature.\"\\n{2}, Sacrifice Wrench: Draw a
    ↪ card\\nEquip {2}
16 ### Expected:
17 Name:Wrench
18 ManaCost:W
19 Types:Artifact Clue Equipment
20 S:Mode$ Continuous | Affected$ Creature.EquippedBy | AddAbility$
    ↪ ABTap | AddPower$ 1 | AddToughness$ 1 | AddKeyword$ Vigilance |
    ↪ Description$ Equipped creature gets +1/+1 and has vigilance and
    ↪ \"{3}, {T}: Tap target creature.\"
21 SVar:ABTap:AB$ Tap | Cost$ 3 T | ValidTgts$ Creature |
    ↪ SpellDescription$ Tap target creature.
22 A:AB$ Draw | Cost$ 2 Sac<1/CARDNAME> | NumCards$ 1 |
    ↪ SpellDescription$ Draw a card.
23 K:Equip:2
24 DeckHas:Ability$Sacrifice
25 Oracle:Equipped creature gets +1/+1 and has vigilance and \"{3},
    ↪ {T}: Tap target creature.\"\\n{2}, Sacrifice Wrench: Draw a
    ↪ card.\\nEquip {2}

```



# Conclusione

In questo lavoro, è stata esplorata l'applicazione degli LLM nel contesto dei giochi di carte collezionabili, in particolare *Magic: The Gathering*, un fenomeno culturale di rilevanza globale. L'obiettivo principale era migliorare la generazione automatica degli script per le carte di *Magic* utilizzando tecniche avanzate di elaborazione del linguaggio naturale e apprendimento automatico, come i trasformatori e i modelli di linguaggio.

Sono stati discussi i principi di base del gioco, il ruolo del motore di regole Forge, l'evoluzione degli algoritmi e l'architettura dei trasformatori, nonché l'importanza dei modelli di linguaggio e delle loro applicazioni. Inoltre, è stato analizzato il ruolo del linguaggio ForgeScript e della nuova proposta rappresentata da Lunar nel processo di creazione e gestione degli effetti di gioco.

Attraverso l'impiego di risorse di calcolo ad alte prestazioni e l'addestramento di diversi LLM, si è riusciti a ottenere risultati promettenti nella generazione automatica degli script per le carte di *Magic*. Dai test effettuati, si evince che i migliori LLM per questo compito sono stati Orca-2 e phi-2, ottenendo prestazioni 60 volte migliori sulla metrica Perplexity (Tabella 4.1) rispetto al meno performante TinyLlama, e di circa il 50% migliori rispetto agli LLM di taglia media come Mistral. Phi-2 è stata la scelta preferita tra i due LLM per i suoi risultati qualitativi superiori e la sua dimensione più piccola. Appartenendo alla categoria dei modelli di linguaggio più piccoli (Small Language Model), ha ottenuto ottimi risultati consumando meno risorse (calcolo, memoria, etc.), rendendolo più economico ed accessibile sia in termini di addestramento che di inferenza. Un vantaggio rispetto a Orca-2 è che, qualora si utilizzasse un servizio a consumo come AWS o Azure, si ridurrebbero i costi di inferenza poiché consumerebbe meno memoria. Infatti, utilizzando AWS come servizio di esempio, i costi mensili per i due LLM ospitati su una istanza EC2 sarebbero rispettivamente 180\$ per Orca-2 e 60\$ per phi-2, ottenendo un risparmio del 67%. Questo spiega perché Microsoft sta investendo molto su phi-2 [16].

Questa tesi costituisce il punto di partenza per la realizzazione di una serie di sviluppi, tra cui:

- a. Sviluppo di un motore di regole generico, con la possibilità di estenderlo attraverso dei wrapper per il gioco al quale si desidera giocare.
- b. Sviluppo di una grammatica ANTLR per descrivere il linguaggio Lunar nel dettaglio.
- c. Addestramento di phi-2 per generare carte di *Magic: The Gathering* utilizzando Lunar.
- d. Sviluppo di un wrapper del motore di regole per *Magic: The Gathering*.
- e. Sviluppo di un gioco di carte collezionabili originale basato sull'approccio proposto in questa tesi.

In conclusione, questo lavoro ha dimostrato l'efficacia e la flessibilità degli LLM nel generare script per giochi di carte collezionabili come *Magic: The Gathering*, fornendo una base solida per ulteriori sviluppi e applicazioni nel campo dell'intelligenza artificiale applicata ai giochi.

# Bibliografia

- [1] Wizards of the Coast. Mtg: Comprehensive rules updated at 02-06-2023. <https://media.wizards.com/2024/downloads/MagicCompRules%2020240206.pdf>, 1993.
- [2] Martin Fowler. Rules engine. <https://martinfowler.com/bliki/RulesEngine.html>, 2009.
- [3] mtgrares. Forge: An unofficial rules engine for the world’s greatest card game. <https://github.com/Card-Forge/forge>, 2007.
- [4] Wizards of the Coast. On whiteboards, naps, and living breakthrough. <https://magic.wizards.com/en/news/mtg-arena/on-whiteboards-naps-and-living-breakthrough>, 2023.
- [5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.
- [6] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. Cambridge University Press, 2023. <https://D2L.ai>.
- [7] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.
- [8] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. Peft: State-of-the-art parameter-efficient fine-tuning methods. <https://github.com/huggingface/peft>, 2022.

- [9] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.
- [10] Martin Fowler. *Domain-Specific Languages*. The Addison-Wesley signature series. Addison-Wesley, 2011.
- [11] Richard Feynman and Chapter Objectives. Ebnf: A notation to describe syntax. *Cited on*, page 10, 2016.
- [12] Forge. Card scripting api. <https://github.com/Card-Forge/forgewiki/Card-scripting-API>, 2007.
- [13] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.
- [14] Austinio. Introducing forgescribe - the ai for scripting forge cards. <https://www.slightlymagic.net/forum/viewtopic.php?f=52&t=22473&sid=d08507a55e592228d147debf94e1a5b4&start=15>, 2007.
- [15] Pieter-Tjerk de Boer, Dirk P. Kroese, Shie Mannor, and Reuven Y. Rubinstein. A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67, January 2005.
- [16] Satya Nadella @ Microsoft Ignite 2023. Microsoft loves slms. <https://youtu.be/FZhbJZEgKQ4?si=WGhjamkC1fMCEHmP&t=1291>, 2023.

# Ringraziamenti

Siamo giunti al momento dei ringraziamenti, e vorrei iniziare partendo dalla persona senza la quale non sarei tornato a finire i miei studi: Giulia (o **gigi**). Grazie per aver sempre fatto il tifo per me, per le ore passate a rileggere questo elaborato, e per avermi tirato le orecchie per riprendere il focus.

Ti amo.

Vorrei ringraziare la Prof.ssa Iotti per avermi seguito in questo percorso e per aver scommesso per su una *wild card* come me.

Ringrazio Fausto Pagani, Federico Prost, e Fabio Spataro per avermi supportato con la fase di addestramento e valutazione su HPC.

Grazie ad @austinio e @friarsol, i due sviluppatori che hanno sviluppato il linguaggio ForgeScript e che hanno condiviso molte informazioni preziose per la creazione di questa tesi.

Grazie a Lore, il mio mentore, che mi ha fatto crescere professionalmente e che mi ha insegnato a non sottostare ad ambienti di lavoro tossici.

Ringrazio la mia famiglia perchè, nonostante tutto, mi ha sempre supportato ed incitato a non mollare.

Un grazie a tutte le persone che hanno partecipato al proof of reading. Ed infine, vorrei ringraziare tutti gli studenti fuori corso come me, perchè mi hanno ricordato di festeggiare questo fallimento. Dopotutto, quale modo migliore per crescere se non attraverso un fallimento?