Università degli Studi di Salerno

Corso di Fondamenti di Intelligenza Artificiale (FIA)

Anno Accademico 2025/26

MagicSolver

MagicSolver

Report

Versione 1.0

Informazioni sul Progetto

Progetto:	MagicSolver
Documento:	Report
Versione:	1.0

Partecipanti:

Nome	**Matricola**
Russo Gerardo	0512119664
Sepe Giuseppe	0512119386
Stefanile Andrea	0512119557
Tangredi Pasquale	0512119273

Scritto da:
Russo Gerardo, Sepe Giuseppe, Stefanile Andrea, Tangredi Pasquale

LINK GITHUB: `https://github.com/a-stefanile/MagicSolver`

Indice

1 Introduzione

Il cubo di Rubik è il rompicapo più famoso della storia, icona intramontabile e simbolo degli anni 80 venne inventato da Ernő Rubik nel 1974. Si tratta di un cubo presentante 6 facce ognuna con un colore differente, composte da 9 quadratini presentati in un cubo 3×3.

Negli anni sono nate tantissime variazioni del cubo di rubik classico come il mirror cube, il pyraminx o il più simpatico 1×1 ma il più classico ed iconico rimarrà sempre il 3×3.

È un problema combinatorio di elevata complessità essendo che ogni rotazione può creare un nuovo stato del cubo, che possono essere in totale 4.3×10^{19}.

Con uno spazio degli stati così grande e un solo goal state, può un'intelligenza artificiale risolvere questo problema?

Il progetto MagicSolver si occuperà di risolvere questo dilemma.

2 Obiettivi

Lo scopo di questo progetto è quello di creare un'IA capace di "giocare" con un cubo di Rubik e di risolvere i cubi lasciati in sospeso per anni sugli scaffali.

La difficoltà è posta dal fatto che lo spazio degli stati è molto vasto e lo stato di goal è uno solo, non importa quante mosse casuali vengano fatte, è estremamente improbabile che il cubo venga risolto senza un algoritmo.

Creare un'IA capace di risolvere un problema combinatorio di questo tipo può fornirci una percezione sulla risoluzione di problemi con un ampio spazio di stati.

2.1 Specifica PEAS

- **Performance**: l'efficacia dell'agente viene valutata primariamente sulla base della lunghezza della soluzione (ovvero il numero di mosse per raggiungere lo stato obiettivo), mirando alla minimizzazione del percorso e del tempo.
- **Environment**: l'ambiente è formato da tutti i possibili stati del cubo, codificati come tensori $5 \times 5 \times 5$.
- **Actuators**: Gli attuatori sono costituiti dall'insieme delle 12 azioni di rotazione ammissibili (R, L, U, D, F, B e i rispettivi inversi).
- **Sensors**: L'utente inserirà l'input su una rappresentazione 2D del cubo, che l'agente percepirà come un tensore $5 \times 5 \times 5$.

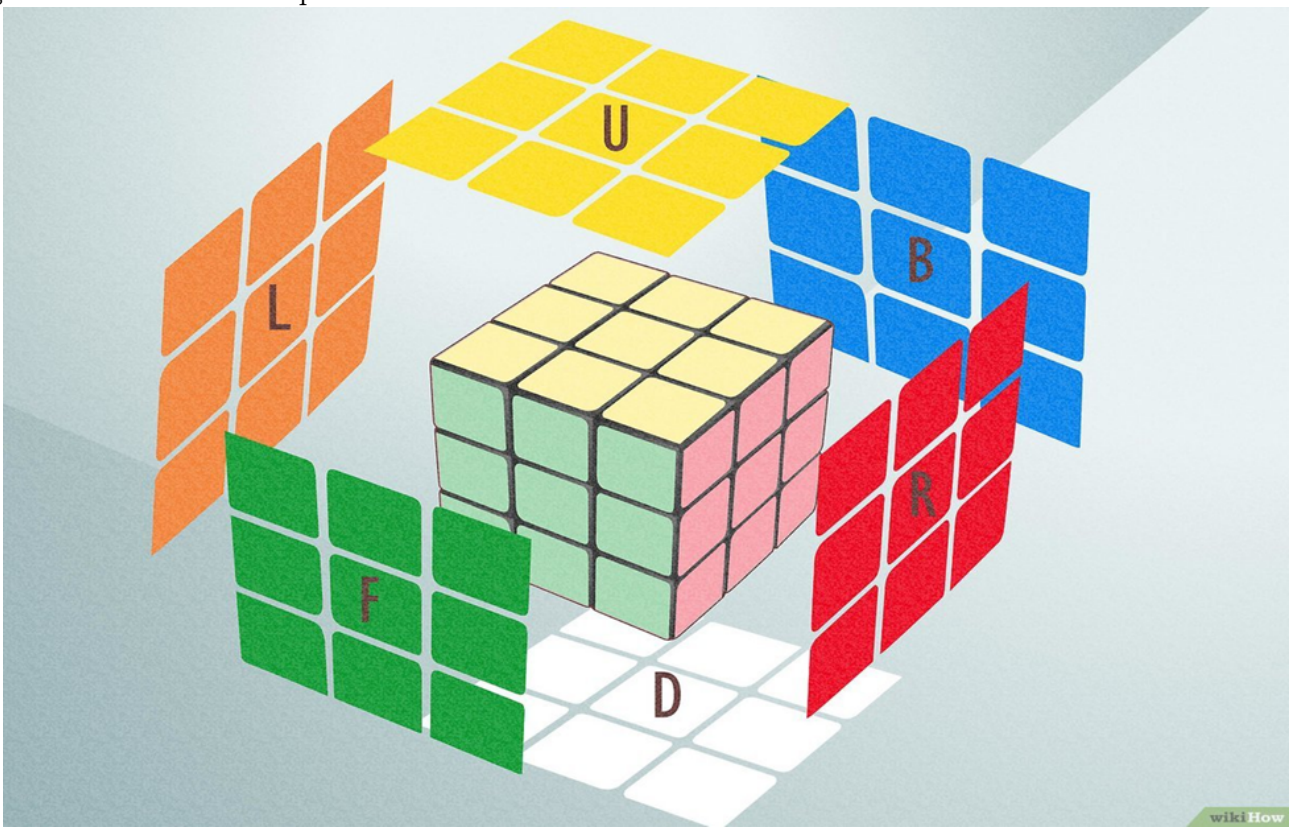
2.1.1 Caratteristiche dell'ambiente

- **Singolo agente**: sul cubo è presente un singolo agente che ha il compito di risolverlo.
- **Totalmente osservabile**: l'agente conosce in ogni istante tutti gli stati del cubo.
- **Deterministico**: ogni azione dell'agente porta a uno e un solo stato.
- **Statico**: lo stato del cubo cambia solo in risposta alle azioni dell'agente.
- **Sequenziale**: la scelta dell'azione corrente condiziona le mosse successive, la risoluzione non è composta da episodi indipendenti, ma da una catena di decisioni che porterà all'obbiettivo dell'agente.
- **Discreto**: Esiste un numero finito di stati (4.3×10^{19}) e di mosse (12 azioni).

2.2 Analisi del problema

Come già detto prima il cubo di Rubik è un problema computazionale molto complesso da risolvere considerando i vari stati che un cubo può assumere.

Per affrontarlo correttamente abbiamo quindi bisogno di formalizzare la rappresentazione degli stati del cubo e lo spazio delle azioni.

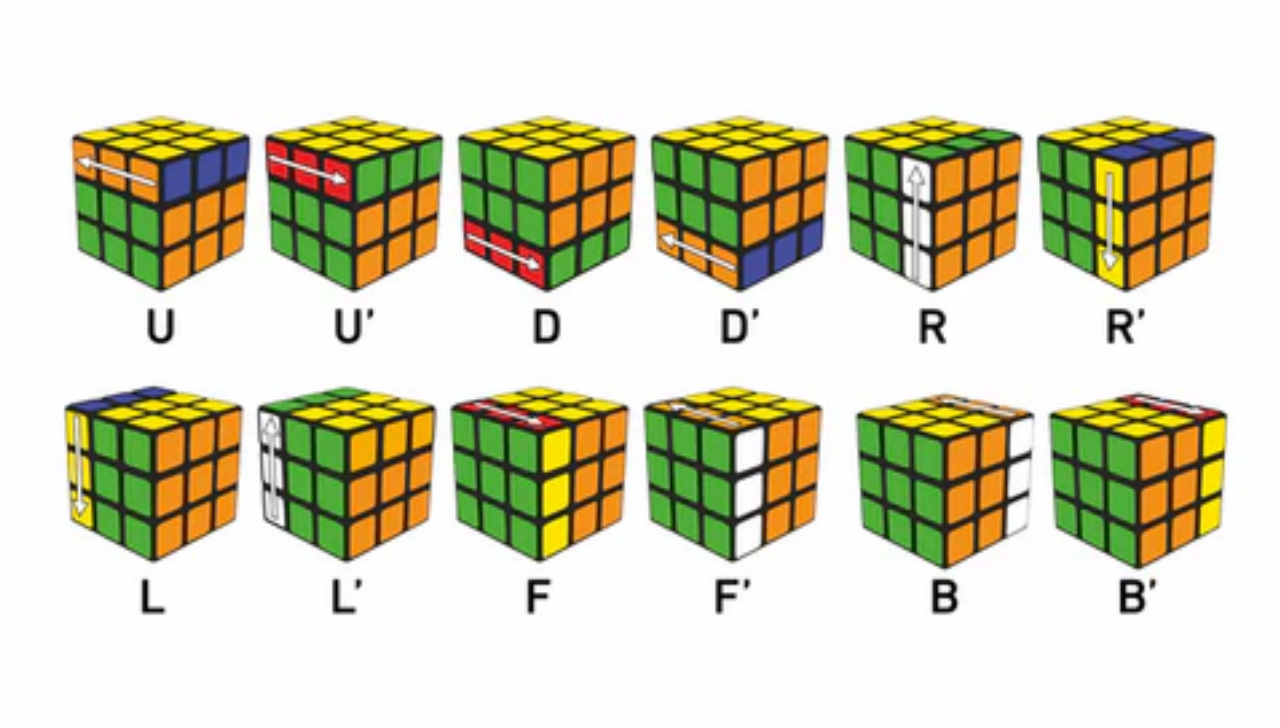


Rappresentazione dello stato e spazio delle azioni

La scelta più intuitiva per rappresentare lo stato di un cubo di Rubik è quella di una matrice $3 \times 3 \times 3$ nella quale ogni cella rappresenta un quadratino del cubo.

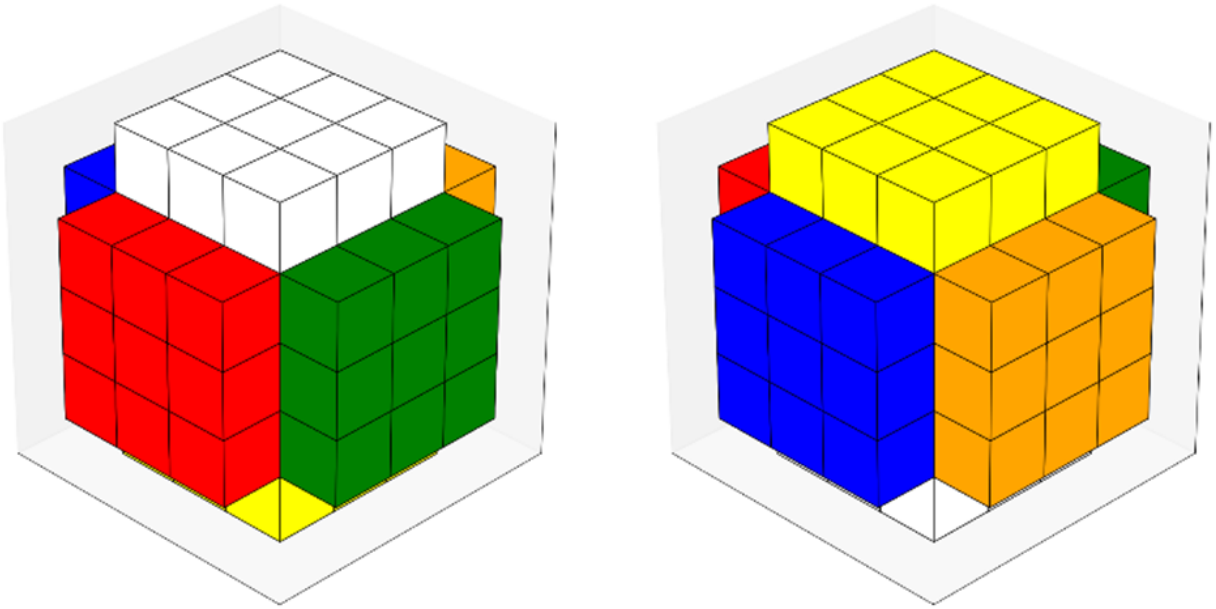
Questa soluzione è la più intuitiva ma probabilmente non la migliore, capiamo come mai:
su ogni cubo possono essere effettuate delle azioni che possiamo chiamare rotazioni.
Esse comportano la rotazione di 90 gradi della "riga" selezionata.
Le rotazioni possibili sono le seguenti:

Rot. oraria	Rot. antioraria	Cambiamento
R	R'	Il lato destro fa una rotazione di 90 gradi
L	L'	Il lato sinistro fa una rotazione di 90 gradi
U	U'	La parte superiore del cubo fa una rotazione di 90 gradi
D	D'	La parte inferiore del cubo fa una rotazione di 90 gradi
F	F'	La parte frontale del cubo fa una rotazione di 90 gradi
B	B'	Il lato posteriore del cubo fa una rotazione di 90 gradi



Con un cubo rappresentato con una matrice $3 \times 3 \times 3$ le rotazioni sono abbastanza difficili da rappresentare: si perdono informazioni importanti dei pezzi interni che non verranno viste dall'algoritmo.

La nostra rappresentazione del cubo come un tensore $5 \times 5 \times 5$, aggiungendo spazi vuoti, riesce a rendere le rotazioni più semplici ed intuitive, inoltre è una rappresentazione molto più vicina alla realtà del cubo con angoli, spigoli e centri.



Il modello di transizione è deterministico: applicando un'azione A allo stato S, si ottiene un nuovo stato S' che è una permutazione matematica delle posizioni dei colori dello stato precedente.

L'obiettivo (Goal State) è quello di arrivare a un cubo risolto con il minor numero di mosse possibili.

Puntualizziamo che un cubo di Rubik può essere considerato risolto solo e soltanto quando ogni faccia del cubo presenta un singolo colore per faccia.

Sinteticamente:

- **Stato iniziale**: cubo (tensore $5 \times 5 \times 5$) mescolato
- **Azioni**: rotazioni
- **Modello di transizione**: deterministico
- **Test obiettivo**: il cubo ha ogni faccia dello stesso colore
- **Costo del cammino**: numero di mosse

3 Strategia Risolutiva

Come già detto, la sfida principale è l'elevato spazio degli stati.

Algoritmi di ricerca non informata (come BFS ad esempio) risulterebbero intrattabili, in quanto esaurirebbero rapidamente le risorse di memoria e il tempo di calcolo per raggiungere la soluzione sarebbe insostenibile, specialmente per stati profondi (oltre le 7/8 mosse).

Per superare questo limite abbiamo adottato una strategia ibrida che combina Machine Learning e un algoritmo di ricerca informata:

- **Euristica Appresa (Machine Learning)**: abbiamo addestrato un modello Random Forest Regressor, che riceve in input lo stato del cubo e predice il numero stimato di mosse mancanti alla soluzione, valore che sarà l'euristica dell'algoritmo di ricerca.
- **Beam Search (Algoritmo di Ricerca informata)**: la predizione del modello di Machine Learning viene utilizzata come funzione euristica per guidare una Beam Search. A differenza di una ricerca esaustiva, questo algoritmo esplora solo i nodi più promettenti suggeriti dal modello.

Questa architettura ibrida permette di evitare sovraccarichi di memoria e di garantire una soluzione efficace, permettendo al sistema di risolvere configurazioni complesse che il modello da solo non saprebbe risolvere e che la ricerca classica non potrebbe raggiungere.

3.1 Dataset

Come ben sappiamo per addestrare un modello di Machine Learning è necessario un dataset, nel nostro caso abbiamo convenuto fosse più conveniente generarne uno sintetico di dimensioni significative, rispetto alla raccolta di dati esterni.

I dataset che abbiamo utilizzato in realtà sono ben due:

- Un dataset contenente lo stato del cubo e il numero di mosse di scramble per il modello che usa One Hot Encoding
- Un dataset contenente le feature ingegnerizzate per il modello basato su distanza di Manhattan

La metodologia utilizzata per generare questi dataset è quella del reverse scrambling.

L'idea è molto semplice: si parte da un cubo risolto e si mescola k volte casualmente (un numero compreso tra 1 e 20), in una tabella X sarà messo il cubo mescolato, e nell'altra tabella Y il numero di mosse che si allontanano dalla soluzione.

Questo metodo è computazionalmente meno costoso rispetto a calcolare la distanza dalla soluzione a partire da un cubo mescolato, in quanto il numero di mosse k è noto a priori per costruzione... per questo motivo abbiamo deciso di adottare questa soluzione.

Utilizzando un numero di 2 milioni di campioni nel dataset abbiamo statisticamente una media di 100mila cubi per ogni k, garantendo di avere abbastanza cubi per allenare l'intelligenza artificiale per ogni livello di profondità.

Data la mole di dati (2 milioni di campioni), l'efficienza di archiviazione è critica.

I dataset sono stati serializzati utilizzando il formato compresso .npz di NumPy, che permette di ridurre drasticamente l'occupazione su disco rispetto ai file di testo o CSV, garantendo al contempo una velocità di caricamento in memoria molto superiore.

Prima della fase di addestramento, il dataset viene suddiviso in due sottoinsiemi disgiunti per garantire una validazione corretta del modello:

- **Training Set (67%)**: Utilizzato per l'addestramento dei pesi del modello.
- **Test Set (33%)**: Utilizzato esclusivamente per la valutazione finale delle metriche (MAE), su dati mai visti durante il training per verificare la capacità di generalizzazione dell'IA.

3.2 Rappresentazione dei Dati (Feature Engineering)

Una volta generati gli stati del cubo, è fondamentale decidere come tradurre l'informazione visiva (colori e posizioni) in un formato numerico comprensibile dal modello di Machine Learning.

Abbiamo sperimentato due approcci distinti, che costituiscono le due pipeline del progetto.

3.2.1 Pipeline 1: One-Hot Encoding

Il metodo più diretto consiste nel passare al modello lo stato grezzo del cubo.

Tuttavia, rappresentare i colori con numeri interi (es. Bianco=0, Giallo=1, Rosso=2...) introdurrebbe un bias: il modello potrebbe interpretare matematicamente che "Rosso è maggiore di Bianco", creando problemi di logica evitabili.

Per evitare questo problema, abbiamo applicato il One-Hot Encoding (OHE).

Ogni singolo sticker del cubo (54 in totale) non viene rappresentata da un numero, ma da un vettore binario di lunghezza 6 (i possibili colori).

- Se lo sticker è Bianco: [1, 0, 0, 0, 0, 0]
- Se lo sticker è Giallo: [0, 1, 0, 0, 0, 0]

Questo trasforma l'input da un vettore di 54 interi a una matrice sparsa di $54 \times 6 = 324$ feature binarie.

Questo approccio lascia al modello l'onere completo di "apprendere" la geometria del cubo da zero, senza aiuti esterni.

3.2.2 Pipeline 2: Manhattan Features

Nel secondo approccio, abbiamo sfruttato la conoscenza del dominio per semplificare il lavoro dell'IA.

Invece di fornire i colori grezzi, abbiamo calcolato la distanza di Manhattan per ogni pezzo del cubo.

La distanza di Manhattan calcola il numero minimo di movimenti necessari per spostare un singolo cubetto (spigolo o angolo) dalla sua posizione attuale alla sua posizione risolta, ignorando la presenza degli altri pezzi.

L'input fornito al modello è quindi un vettore compatto contenente:

- La somma delle distanze di tutti gli 8 angoli.
- La somma delle distanze di tutti i 12 spigoli.

In questo caso, il modello non deve imparare le regole del gioco da zero, ma agisce come un correttore che "affina" una stima matematica già esistente, cercando di predire l'errore dell'euristica classica.

3.3 Modelli di Machine Learning

Prima di selezionare l'algoritmo specifico, è stato necessario definire la natura del problema di apprendimento.

L'obiettivo del nostro sistema è stimare la distanza $d(s)$ che separa lo stato corrente s dallo stato risolto.

In ambito Machine Learning, questo problema poteva essere approcciato in due modi:

- **Classificazione**: trattare ogni distanza (da 1 a 20 mosse) come una "classe" o categoria distinta.
- **Regressione**: trattare la distanza come una variabile numerica continua.

Le nostre scelte sono ricadute su quest'ultima per motivi legati alla natura dell'algoritmo di ricerca, nel particolare:

- Un classificatore non ha il concetto di "vicinanza", se sbaglia predicendo un valore di classe "13 mosse" invece che "14 mosse" per il modello diventa un errore grave quanto a dire classe "2 mosse" ... mentre un regressore, penalizza l'errore in proporzione alla distanza, spingendo il modello a imparare la struttura graduale del problema, in altri termini se non riesce a dare la risposta esatta, cerca di avvicinarcisi.
- L'algoritmo Beam Search deve ordinare migliaia di stati candidati per scegliere i migliori. Se usassimo la classificazione, l'output sarebbe un numero intero discreto (es. 8, 9, 10). Questo creerebbe molti casi di parità, rendendo difficile per l'algoritmo distinguere tra due stati entrambi classificati come "9 mosse". La regressione, restituendo un valore continuo (es. 8.42 o 8.45), fornisce una granularità fine, ciò significa che anche se le mosse reali sono intere, il valore decimale può essere interpretato come un indice di confidenza, permettendo un ordinamento più efficace dei candidati.

Una volta definito il task come problema di regressione, la scelta dell'algoritmo specifico è ricaduta sul **Random Forest Regressor**, un metodo di Ensemble Learning basato su alberi decisionali.

3.3.1 Random Forest Regressor

Come detto nel paragrafo precedente, questo è un metodo di Ensemble Learning basato su alberi decisionali, ma cosa significa?

L'algoritmo si basa su una vera e propria "foresta" di alberi decisionali che vengono eseguiti parallelamente.

La predizione finale non è altro che la media dei valori restituiti da tutti questi alberi.

Questo meccanismo rappresenta un grande vantaggio rispetto all'uso di un singolo albero, poiché ne riduce drasticamente l'instabilità e il rischio di overfitting.

La decisione di utilizzare questo specifico algoritmo è supportata dai seguenti vantaggi:

- **Gestione della Non-Linearità e delle Discontinuità**: La funzione che lega la posizione degli sticker al numero di mosse mancanti è altamente non-lineare: una singola rotazione di una faccia può cambiare drasticamente la configurazione geometrica e la distanza dalla soluzione. Mentre i modelli lineari fallirebbero nel catturare queste interazioni complicate, gli alberi decisionali operano partizionando ricorsivamente lo spazio degli stati. Questo permette al Random Forest di apprendere regole logiche complesse necessarie per decodificare la struttura del cubo.

- **Superiorità su Dati Tabellari**: Le due pipeline sviluppate (One-Hot Encoding e Manhattan) producono dati strutturati (o tabellari). La letteratura scientifica attuale dimostra che, su questa tipologia di dati, gli algoritmi basati su alberi hanno prestazioni spesso superiori o simili alle architetture Deep Learning, ma con una complessità di addestramento decisamente inferiore.

- **Efficienza in fase di Inferenza**: Questo è importante per l'integrazione con l'algoritmo di ricerca: la nostra Beam Search deve valutare migliaia di stati candidati al secondo per essere efficace. Le reti neurali richiedono operazioni matriciali pesanti che spesso necessitano di GPU per essere veloci. Al contrario, il Random Forest, essendo costituito da una serie di semplici confronti condizionali (if-else), garantisce tempi di inferenza estremamente ridotti su CPU standard.

Per configurare in modo ottimale il nostro modello è stato necessario definire iperparametri specifici, che ci ha permesso di governare la struttura e il processo di apprendimento.

La configurazione scritta in Python è la seguente:

```
model = RandomForestRegressor(
n_estimators=100,
max_depth=20,
n_jobs=-1,
verbose=1
)
```

- **Numero di alberi (n_estimators = 100)**: questo valore è un ottimo trade-off tra tempo di calcolo e stabilità della predizione.

- **Profondità massima (max_depth = 20)**: Dato il dataset di grandi dimensioni, limitare la profondità è stato fondamentale per impedire al modello di "imparare a memoria" (overfitting) e costringendolo quindi a imparare le regole del gioco.

- **Parallelismo (n_jobs = -1)**: L'addestramento usa tutti i core della CPU per processare gli alberi in parallelo.

- **Monitoraggio (verbose = 1)**: Abilita l'output dei log in tempo reale per monitorare l'avanzamento della costruzione della "foresta".

Alla fine del training i modelli hanno restituito questi risultati:

```
========================================
--- RISULTATI TEST PIPELINE 1 (OHE) ---
Errore Medio Assoluto (MAE): 1.615 mosse
Coefficiente R2 (Precisione): 75.14%
========================================
RISULTATI TEST PIPELINE 2 (MANHATTAN)
Errore Medio Assoluto (MAE): 1.6241 mosse
Coefficiente R2 (Precisione): 74.80%
========================================
```

Analizzeremo i risultati del training nel paragrafo di sperimentazione e analisi.

3.4 Algoritmo di ricerca informata

L'esecuzione delle azioni, volte alla risoluzione, è affidata all'algoritmo Beam Search. Questo algoritmo è una variante della ricerca in ampiezza, il quale mitiga il problema dell'espansione esponenziale dei nodi quando si passa da un livello all'altro.

Ciò è possibile definendo un valore k (chiamato Beam Width) che indica il limite del numero di nodi che esplora ad ogni livello e salvando solo i nodi più promettenti, questo valore può variare in base alla difficoltà del cubo.

In questo contesto ogni nodo rappresenta una configurazione del cubo, le mosse che facciamo su di esso portano il cubo in una nuova configurazione.

La scelta dei nodi da espandere viene valutata con il valore risultante dalla somma del cammino attuale (cioè il numero di mosse effettuate) e il valore predetto dal modello di machine learning (euristica) per il coefficiente di confidenza rappresentato da ϵ (che ci consente di risolvere cubi più complessi in tempi più umani)

$$f(n) = g(n) + (h(n) \cdot \epsilon)$$

La selezione dei nodi migliori può, però, portare all'esclusione di soluzioni le quali, pur non troppo promettenti nel breve periodo, potrebbero portare a soluzioni globalmente migliori rendendo in questo modo l'algoritmo incompleto.

Questo trade-off permette di ridurre drasticamente il tempo di esecuzione dell'algoritmo, poiché nel peggiore dei casi abbiamo una crescita lineare al posto di una crescita esponenziale.

Tuttavia usare il Beam Search così com'è, porta a soluzioni in cui tempi di esecuzione non sono contemplati. Per questo motivo bisogna apportare alcuni piccoli accorgimenti per portare un aumento considerevole della velocità della risoluzione.

Queste modifiche sono:

- **Introduzione di una tabella hash in cui salvare i nodi esplorati**: L'algoritmo potrebbe identificare due nodi con valori identici e di conseguenza i succesori di entrambi sarebbero equivalenti. Introdurre una tabella hash ci permette di salvare i nodi correnti ed eventualmente scartare i nodi duplicati, riducento in maniera drastica i tempi di esecuzione.

- **Riavvio casuale**: Questa tecnica interviene per risolvere situazioni di stallo, in cui l'algoritmo si trova bloccato in un ottimo locale. In tale scenario, si osserva una divergenza critica: il valore predetto dal modello di machine learning rimane costante o non mostra miglioramenti significativi, questo segnale indica che l'esplorazione non sta progredendo verso la soluzione. A questo punto selezioniamo i migliori candidati per il livello attuale e selezioniamo, attraverso una selezione pesata basata sull'euristica, i restanti nodi.

- **Suddivisione a livelli**: Come detto in precedenza, per ogni stato in cui si trova il Beam Search esso valuta solo un numero limitato di nodi, pari al valore del Beam Width. Il tempo di esecuzione è direttamente proporzionale a questo. In questo modo scegliendo un una larghezza abbastanza contenuta di nodi da espandere riusciamo a controllare in maniera molto veloce. Se non sono state trovate soluzioni l'algoritmo aumenta il Beam Width, la ϵ e la profondità e riavvia la ricerca, aumentando la probabilità di trovare una soluzione.

3.5 Interfaccia Utente

L'Interfaccia grafica (GUI) è stata sviluppata con la libreria Tkinter, permettendo visualizzazione, interazione e risoluzione di un cubo.

3.5.1 Rappresentazione Visiva e Layout

Il cubo viene visualizzato tramite una proiezione 2D "a forma di croce" dove ogni faccia è rappresentata da una classica griglia 3×3 di tasselli.

- **Mapping Logico**: Sebbene l'utente veda una rappresentazione 2D di un cubo 3x3, ogni cambiamento o rotazione effettuata viene automaticamente tradotto in un tensore $5 \times 5 \times 5$, in modo da renderlo compatibile con la nostra IA.

3.5.2 Componenti funzionali

L'interfaccia è suddivisa in quattro blocchi operativi principali:

- **Selezione colori**: L'utente può selezionare uno dei sei colori tipici del cubo e cliccando su un cubetto qualsiasi della canvas può cambiare il colore di quel cubetto, questo inserimento manuale permette all'utente di poter raffigurare sulla canvas un qualsiasi cubo che desidera risolvere.
- **La Canvas**: Il blocco in cui è rappresentato il cubo e l'ultima azione compiuta dall'utente o dall'IA, questa è la parte che aiuterà di più l'utente nel visualizzare il come risolvere il cubo step by step.
- **Controllo Manuale (Attuatori)**: In questo blocco sono presenti pulsanti raffiguranti le 12 rotazioni standard eseguibili su un cubo, ogni rotazione eseguita viene istantaneamente visualizzata sulla canvas.
- **Comandi di Sistema**: Nell'ultimo blocco sono rappresentati quattro pulsanti ognuno essenziale per il funzionamento del programma:
  - **Mescola**: Mischia il cubo per un numero definito di mosse casuali.
  - **Verifica**: Prima di tutto verifica se il cubo è un cubo che può esistere realmente, se la verifica va a buon fine invia lo stato attuale del cubo all'IA e riceve le mosse necessarie per risolvere il cubo se vengono trovate.
  - **Esegui Soluzione**: Questo tasto, premibile solo se la verifica va a buon fine ed è stata trovata una soluzione, esegue in tempo reale la soluzione trovata dall'IA, verrà effettuata una mossa alla volta (ogni 800ms) finchè il cubo non verrà risolto. Alla fine di questa riproduzione l'utente troverà una cartella contenente delle foto di ogni singolo step e stato del cubo in modo che l'utente possa riprodurle senza alcuna fretta.
  - **Reset**: Questo tasto riporta il cubo allo stato iniziale a prescindere dalle modifiche fatte.

4 Sperimentazione e Analisi

Nei paragrafi precedenti abbiamo descritto come ci siamo approcciati al problema, ma ora passiamo al momento della verità: come si comporta il tutto?

Abbiamo sviluppato vari script di benchmark direttamente sul repository del progetto, ciò ci ha consentito di avere un quadro generale su un elevato numero di cubi e di avere quindi risposte sull'efficacia e sulle performance del sistema.

4.1 Performance dei Modelli

Nel paragrafo 3.3.1 abbiamo illustrato i seguenti punteggi per le due pipeline di Machine Learning, ricordiamo che questi dati sono stati creati attraverso il Test Set del dataset di riferimento (ovvero il 33% del dataset, come specificato nel paragrafo 3.1).

Ricordiamo i risultati ottenuti:

Pipeline	**MAE**	**Coefficiente R^2 (Precisione)**
One Hot Encoding (OHE)	1.615	75.14%
Manhattan	1.6241	74.80%

Dall'analisi di questi dati emergono tre considerazioni fondamentali:

- **Significato del MAE (1.6 mosse)**: un Errore Medio Assoluto di circa 1.6 indica che, in media, la predizione del modello si allontana dalla distanza reale di meno di 2 mosse. Nel contesto di una ricerca euristica (come Beam Search), questo risultato è estremamente positivo. L'obiettivo del modello non è fornire la distanza esatta (che richiederebbe calcoli immensi), ma fornire un gradiente affidabile. Se a un cubo mancano 10 mosse e il modello ne stima 8.4 o 11.6, la "direzione" verso la soluzione rimane corretta. Questo margine di errore ridotto permette all'algoritmo di ricerca di distinguere efficacemente tra uno stato promettente e un vicolo cieco.

- **Il confronto tra le due pipeline**: ci si aspetterebbe che la Pipeline 2, che utilizza feature ingegnerizzate basate sulla geometria del cubo (Distanza di Manhattan), superi nettamente la Pipeline 1, che lavora su dati grezzi. Invece, i risultati mostrano una lievissima superiorità del modello OHE con MAE 1.615. Questo dimostra la potenza del Random Forest: l'algoritmo è stato in grado di "apprendere la geometria" del cubo e le relazioni spaziali autonomamente partendo dalla rappresentazione posizionale (bit 0 e 1), ricostruendo internamente una logica simile a quella della distanza di Manhattan senza che questa gli fosse stata insegnata esplicitamente.

- **Precisione (coefficiente R^2)**: Un coefficiente R^2 del 75% conferma che il modello ha catturato la maggior parte della varianza dei dati. Il restante 25% di varianza non spiegata è attribuibile alla natura intrinseca del cubo: esistono configurazioni che appaiono visivamente molto disordinate (alta entropia) ma sono vicine alla soluzione, e viceversa. Il modello riesce comunque a generalizzare correttamente nella stragrande maggioranza dei casi, garantendo un'euristica robusta.

4.2 Efficacia del solver

Sulla scelta dell'algoritmo del solver c'è da fare una piccola parentesi interessante.

L'algoritmo Beam Search non è stata la prima idea per il sistema; infatti, la primissima idea e sperimentazione è stata fatta su un altro noto algoritmo di ricerca: l'IDA* (Iterative Deepening A*), questo perché l'algoritmo in linea teorica era quello più consono al problema, ovvero quello di trovare il percorso più breve... Tuttavia, l'abbiamo successivamente rivalutato.

L'algoritmo era piuttosto lento, anche su configurazioni di 7 mosse l'algoritmo esplorava fin troppi nodi e non ne riusciva a venire a capo, e in termini di performance non era ottimale, dal momento che le tempistiche per risolvere cubi complessi diventavano mediamente piuttosto lunghe (anche più di 30 minuti).

Il principale ostacolo è stato la natura "sequenziale" dell'IDA*. Questo algoritmo valuta un singolo nodo alla volta, rendendo impossibile sfruttare appieno la potenza di calcolo parallelo del modello di Machine Learning (Random Forest).

Poiché il Cubo di Rubik presenta un fattore di ramificazione elevato (circa 18 mosse possibili per ogni stato), l'esplorazione esaustiva causava un'esplosione dei tempi di calcolo, portando il sistema in una condizione di stallo.

A differenza dell'IDA*, la Beam Search non esplora ogni ramo, ma mantiene ad ogni livello solo un numero limitato di candidati più promettenti (il cosiddetto beam width).

Questa scelta ha permesso di introdurre il concetto di Batch Inference: invece di interrogare il modello Random Forest per ogni singolo stato, il solver raggruppa migliaia di possibili configurazioni in un unico blocco di dati, ottenendo le predizioni in un'unica grande operazione.

Sebbene questo approccio sacrifichi la garanzia matematica dell'ottimalità assoluta (la soluzione trovata potrebbe avere 2-3 mosse in più rispetto a quella minima), ha trasformato il sistema da un modello teorico lento a uno strumento pratico capace di risolvere configurazioni complessi in tempi quantomeno decenti.

Per massimizzare l'efficacia, è stata infine implementata una strategia adattiva, che scala l'intensità della ricerca (numero di "droni" o raggio del beam) solo quando strettamente necessario, garantendo rapidità per i cubi semplici e resilienza per quelli complessi.

Abbiamo eseguito diversi benchmark su MagicSolver per verificare la percentuale di *Success Rate*, le tempistiche delle soluzioni e il numero di mosse della soluzione. Da questi benchmark possono essere fatte delle considerazioni molto interessanti.

Partiamo dalla configurazione attuale che è stata proposta. In questa configurazione decidiamo di valorizzare il tempo di risoluzione rispetto al *Success Rate* dei cubi.

Range Scramble	**N. Cubi**	**Success Rate**	**Tempo Medio**	**Mosse Medie**
1-10	200	100%	1,13 s	4,54
11-12	100	96%	18,58 s	9,25
13-14	50	80%	22,18 s	10,30
15-16	50	60%	30,20 s	11,55
17-20	40	30%	23,15 s	12,60

Tabella 1: Risultati Benchmark della Configurazione Attuale

NB: *Il tempo medio non considera i time-out. Abbiamo preferito esplicitare il tempo medio della risoluzione del cubo per evitare distorsioni statistiche con tempi pari a 3+ minuti.*

Come possiamo vedere dai dati, il modello in media riesce a trovare un numero di mosse minori in confronto alle mosse eseguite per essere mischiato. Questo ci indica che trova molto bene le scorciatoie per la soluzione migliore possibile e riesce a farlo in tempi molto ragionevoli.

La velocità però ha un costo molto elevato: si può notare che più il numero di scramble aumenta e più il *Success Rate* diminuisce. Per trovare soluzioni in tempo ragionevole siamo stati costretti a inserire dei time-out di 10, 45 e 120 secondi; inoltre abbiamo dovuto diminuire la *beam-width* e aggiungere una epsilon allo score.

Confronto con configurazioni precedenti

In una configurazione precedente abbiamo deciso di prioritizzare il numero di cubi risolti a discapito del tempo di risoluzione, questo allargando la *beam-width* e aumentando notevolmente il tempo del time-out mettendo 180 secondi. Questa configurazione effettivamente riusciva ad aumentare la percentuale di *Success Rate* dei cubi da 17 a 20 mosse portandola al 50%, ma presentava tempi di computazione intorno a 15 minuti, questo pure in caso di *failure*.

In conclusione, la nostra scelta di prioritizzare il tempo di risoluzione in confronto al *Success Rate* è dovuta ad un'ottica orientata ad un'applicazione utilizzabile dagli utenti: se un utente dovesse aspettare 15 minuti o più per la soluzione del cubo, piuttosto potrebbe imparare a risolverlo da solo.

4.3 Analisi dei Trade-Off

Nello sviluppo di un sistema di Intelligenza Artificiale complesso come un solver per il Cubo di Rubik, ogni scelta architetturale comporta un bilanciamento tra obiettivi contrastanti.

Di seguito analizziamo i principali trade-off affrontati durante il progetto:

1. **Ottimalità della Soluzione vs. Tempi di Risoluzione**: Questo è il compromesso più significativo del progetto. Utilizzare algoritmi come IDA* (Iterative Deepening A *) avrebbe garantito la soluzione ottima, ma con tempi di esecuzione esponenziali dato che l'euristica appresa non è perfettamente ammissibile. Come già detto, abbiamo optato per la Beam Search , un algoritmo "greedy" che pota lo spazio di ricerca, sacrificando la garanzia di trovare il percorso più breve in cambio della garanzia di convergere a una soluzione valida in tempi umanamente accettabili (secondi o comunque pochi minuti).

2. **Complessità del Modello vs. Latenza di Inferenza**: Avremmo potuto utilizzare reti neurali o meccanismi di Deep Learning che avrebbero di certo ridotto il MAE da 1.6 a un valore minore di 1, ma con la scelta del nostro Random Forest abbiamo accettato un'euristica leggermente meno precisa, per ottenere una velocità di inferenza drasticamente superiore su CPU. Dato che la Beam Search deve valutare migliaia di stati al secondo, un modello lento ma precisissimo avrebbe reso il solver globale paradossalmente più lento.

5 Conclusioni e Valutazioni

Il progetto ha raggiunto l'obiettivo prefissato: realizzare un solver per il Cubo di Rubik basato sull'Intelligenza Artificiale, dimostrando come tecniche di Machine Learning Supervisionato possano sostituire efficacemente le euristiche matematiche tradizionali all'interno di algoritmi di ricerca complessi.

L'architettura ibrida sviluppata, che combina un Random Forest Regressor (per l'intuizione/euristica) e una Beam Search Adattiva (per l'esplorazione), si è rivelata una strategia vincente.

Il sistema è capace di risolvere configurazioni di varia difficoltà, mantenendo tempi di esecuzione contenuti e un tasso di successo elevato.

5.1 Limitazioni Attuali

- **Soluzioni sub-ottimali su cubi complessi**: Il sistema non garantisce il raggiungimento della soluzione ottima assoluta (il cosiddetto God's Number, ≤ 20 mosse). Questo perché l'algoritmo Beam Search è di natura greedy, quindi incompleto: ad ogni livello di profondità, scarta irrevocabilmente i nodi meno promettenti per risparmiare memoria. Se la soluzione più breve si trovava in uno dei rami tagliati (perché l'euristica ne ha sottostimato temporaneamente il valore), essa viene persa per sempre.

- **Collo di bottiglia hardware (CPU bound)**: A differenza delle Reti Neurali, che possono sfruttare la potenza delle GPU per valutare batch di stati in parallelo, il Random Forest esegue una serie di controlli condizionali sequenziali sugli alberi, questo limita il numero massimo di stati che possiamo valutare al secondo (quindi il throughput). Di conseguenza, non possiamo spingere il Beam Width a valori altissimi (come 100.000 nodi ad esempio) senza rallentare drasticamente l'esecuzione, limitando la capacità di esplorazione globale del sistema.

5.2 Sviluppi futuri

Il lavoro svolto apre la strada a diverse possibilità di miglioramento ed evoluzione, ispirate anche dalla recente letteratura scientifica (abbiamo analizzato un paper scientifico, che ci ha indirizzati verso un'idea futura del sistema, in basso sarà possibile trovare un link che riporterà la fonte):

- **Deep Learning & Architetture Transformer**: L'evoluzione naturale del progetto prevede il superamento del Random Forest in favore di reti neurali profonde. Una frontiera promettente è l'uso di Transformers e meccanismi di Self-Attention. Mentre il Random Forest osserva le feature staticamente, un Transformer potrebbe analizzare il cubo come una "sequenza di relazioni" tra i cubetti, catturando dipendenze a lungo raggio (come la posizione di un angolo influenza la risolvibilità di uno spigolo opposto) in modo molto più efficace. Questo permetterebbe di abbassare il MAE sotto la soglia di 1.0.

- **Policy Network (Predizione della Mossa)**: Attualmente il nostro sistema apprende una stima della distanza. Un approccio alternativo consiste nell'addestrare una Policy Network, cioè invece di chiedere al modello "quanto manca?", gli si chiede "qual è la mossa migliore da fare adesso?". Questo trasformerebbe il problema da Regressione a Classificazione Sequenziale, permettendo al modello di generare direttamente la sequenza risolutiva senza dover esplorare migliaia di nodi con la Beam Search, riducendo drasticamente i tempi di inferenza.

- **Validazione della Rappresentazione Tensoriale**: Studi recenti confermano l'efficacia della rappresentazione del cubo come tensore sparso $5 \times 5 \times 5$ (la stessa intuizione avuta in questo progetto) rispetto alla matrice piatta $3 \times 3 \times 3$. Sviluppi futuri potrebbero sfruttare questa rappresentazione spaziale per applicare filtri convoluzionali 3D che "vedono" le rotazioni esattamente come avvengono nello spazio fisico, migliorando la capacità di generalizzazione del modello.

(Fonte: `https://towardsdatascience.com/solving-a-rubiks-cube-with-supervised-learning-intuitively-and-exhaustively-explained-4f87b72ba1e2/`)

5.3 Conclusioni

In conclusione, il progetto dimostra che non serve insegnare al computer la geometria per risolvere il cubo: con abbastanza dati, l'IA è in grado di "intuire" la distanza dalla soluzione guardando solo una sequenza di zeri e uni.

Abbiamo sacrificato la perfezione matematica della soluzione ottima per ottenere un sistema flessibile, rapido e intuitivo.