Relazione Tecnica: Agente AI per il Gioco Forza 4 con Apprendimento per Rinforzo

Analisi generata da Gemini

7 ottobre 2025

Indice

1 Introduzione e Obiettivi del Progetto

Il progetto si pone l'obiettivo di sviluppare un'intelligenza artificiale in grado di apprendere a giocare al gioco da tavolo **Forza 4**. L'approccio scelto non è basato su algoritmi di ricerca classici come Minimax, ma su tecniche di **Apprendimento per Rinforzo (Reinforcement Learning - RL)**. Questo permette all'agente di imparare una strategia di gioco ottimale non attraverso la conoscenza pregressa delle regole, ma tramite l'esperienza diretta, giocando un gran numero di partite e ricevendo feedback (ricompense) in base ai risultati.

La metodologia specifica implementata è l'algoritmo **SARSA(λ) con Approssimazione Lineare di Funzione**. Questa scelta è motivata dalla necessità di gestire l'enorme numero di configurazioni possibili del tabellone di Forza 4, che rende impraticabile l'uso di approcci tabellari semplici.

Questa relazione analizzerà nel dettaglio:

- Il **framework teorico** alla base dell'algoritmo SARSA(λ).
- L'**architettura del software** e il ruolo di ogni singolo file `.m`.
- La **strategia di addestramento** a più fasi implementata.
- Una **guida all'utilizzo** e una discussione sui possibili miglioramenti futuri.

2 Il Framework Teorico: Apprendimento per Rinforzo con SARSA(λ)

L'apprendimento per rinforzo è un paradigma del machine learning in cui un **agente** impara a interagire con un **ambiente** per massimizzare una nozione di ricompensa cumulativa.

2.1 Il Problema della Complessità e l'Approssimazione di Funzione

Un gioco come Forza 4 ha uno spazio degli stati (le possibili configurazioni del tabellone) estremamente vasto (dell'ordine di 10^{13}). Un approccio RL classico, come il Q-Learning tabellare, richiederebbe una tabella per memorizzare il valore di ogni possibile azione in ogni possibile stato, un'impresa computazionalmente impossibile.

La soluzione adottata in questo progetto è l'**Approssimazione di Funzione**. Invece di memorizzare il valore Q esatto per la coppia stato-azione (s, a), lo si approssima tramite una funzione parametrizzata. In questo caso, si usa un'approssimazione lineare:

$$Q(s, a) \approx \mathbf{w}_a^T \cdot \mathbf{F}(s) \tag{1}$$

- $\mathbf{F}(s)$ è un **vettore di features**, ovvero un vettore numerico che descrive le caratteristiche salienti dello stato s (es. numero di tris, controllo delle colonne).
- $\mathbf{w}_a$ è un **vettore di pesi** associato all'azione a. L'intero processo di apprendimento consiste nell'aggiustare questi pesi in modo che la stima del valore Q diventi il più accurata possibile.

2.2 L'Algoritmo SARSA(λ)

SARSA è un algoritmo di apprendimento per rinforzo di tipo *on-policy* e *Temporal Difference (TD)*. Il suo nome deriva dalla sequenza di eventi che compongono l'aggiornamento: **S**tate, **A**ction, **R**eward, **S**tate (successivo), **A**ction (successiva).

L'aggiornamento dei pesi si basa sull'**errore di predizione temporale (TD error)**, denotato con δ:

$$\delta = r + \gamma \cdot Q(s', a') - Q(s, a) \tag{2}$$

dove:

- r è la ricompensa ricevuta dopo aver eseguito l'azione a nello stato s.
- γ è il **fattore di sconto**, che bilancia l'importanza delle ricompense immediate rispetto a quelle future.
- $Q(s', a')$ è il valore stimato dell'azione *effettivamente scelta* (a') nel nuovo stato s'.

La variante implementata è **SARSA(λ)**, che utilizza le **Tracce di Eligibilità (Eligibility Traces)**. Le tracce, rappresentate dal vettore $\mathbf{z}$, sono un meccanismo che permette di "ricordare" le coppie stato-azione visitate di recente. Quando si verifica un errore di predizione δ, l'aggiornamento non viene applicato solo all'ultima coppia, ma viene propagato a ritroso a tutte quelle recenti, pesato dal parametro λ.

L'aggiornamento finale dei pesi segue questa regola:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \cdot \delta \cdot \mathbf{z} \tag{3}$$

dove α è il **tasso di apprendimento (learning rate)**. Questo meccanismo accelera notevolmente la convergenza, poiché il risultato di una partita (vittoria o sconfitta) influenza l'intera sequenza di mosse che ha portato a quel risultato.

3 Analisi dei Componenti del Progetto

Il progetto è composto da una serie di file `.m` che collaborano per implementare l'algoritmo, gestire il gioco e visualizzare i risultati. Di seguito, un'analisi dettagliata per gruppo funzionale.

3.1 Lo Script Principale (`MC_f4.m`)

Questo file è il **centro di controllo** dell'intero progetto. Orchestra tutte le fasi, dalla definizione dei parametri all'addestramento, fino alla valutazione e al gioco interattivo. La sua struttura è la seguente:

1. **Inizializzazione dei Parametri**: Vengono definiti i parametri fondamentali dell'algoritmo:
   - `A = 7`: Numero di azioni.
   - `numEpisodes = 10000`: Numero di partite per ogni fase di addestramento.
   - `epsilon = 0.3`: Probabilità di compiere un'azione casuale (esplorazione).
   - `gamma = 0.9`: Fattore di sconto.

- `lambda = 0.2`: Parametro per il decadimento delle tracce di eligibilità.
- `d = 332`: **Dimensione del vettore di features**. Questo valore è cruciale e indica che il vettore di feature $F(s)$ ha 332 elementi. *Nota: questo valore è hardcodato e sembra non corrispondere al calcolo derivante dall'analisi del file `Features.m` (che ne produrrebbe 244). È probabile che la versione di `Features.m` utilizzata per generare questo script fosse diversa. Ai fini pratici, `d=332` è il valore operativo del modello.*

2. **Fase 1: Addestramento contro Avversario Casuale**: Viene addestrato prima il **Giocatore 1** (`w`) e poi il **Giocatore 2** (`w3`) facendoli giocare `numEpisodes` partite contro un avversario che esegue mosse puramente casuali (usando rispettivamente `grid1.m` e `grid2.m`).

3. **Fase 2: Auto-Apprendimento (Self-Play)**: Questa è la fase più importante. Viene eseguito un ciclo di 3 iterazioni in cui:
   - `AutoLearn2` allena l'agente 2 (`w3`) contro la policy corrente dell'agente 1 (`w`).
   - `AutoLearn1` allena l'agente 1 (`w`) contro la policy appena migliorata dell'agente 2 (`w3`).
   - Entrambi gli agenti vengono ulteriormente affinati giocando contro un avversario casuale.

4. **Visualizzazione e Analisi dei Pesi**: Vengono generati grafici `surf` per visualizzare la "superficie" dei pesi `w` e `w3` prima e dopo l'auto-apprendimento.

5. **Valutazione Finale e Gioco Interattivo**: Viene testata la policy finale e una sezione `GAME VS ME` permette a un utente umano di giocare contro l'agente AI.

3.2 L'Ingegneria delle Features (`Features.m` e funzioni correlate)

Il cuore della "intelligenza" dell'agente risiede in come percepisce il mondo. Questo è gestito dal file `Features.m`, che aggrega le informazioni da diverse funzioni:

- `extract_column_features.m`: Conta le pedine per colonna per ogni giocatore, aiutando a valutare il controllo verticale.
- `count_threes.m`: Utilizza `count_consecutive.m` per contare le sequenze di 3 pedine, fondamentale per riconoscere minacce immediate.
- `get_diagonal_features.m`: Estrae features relative al numero di pedine presenti sulle diagonali.
- `d=(reshape(dec2bin(s(:),4),1,[])-'0')'`: Questa linea converte l'intera matrice di stato in un lungo vettore binario, fornendo una rappresentazione grezza ma completa del tabellone.

3.3 Gli Algoritmi di Apprendimento (`Learning_random.m`, `AutoLearn*.m`)

Questi file contengono l'implementazione del ciclo di apprendimento SARSA(λ).

- `Learning_random.m` e `Learning_random1.m`: Implementano l'addestramento contro un avversario casuale (usando `grid1.m` o `grid2.m`).
- `AutoLearn1.m` e `AutoLearn2.m`: Gestiscono l'addestramento in modalità self-play, dove l'avversario è l'altro agente AI.

3.4 L'Ambiente di Gioco (`grid*.m` files)

Questi file simulano una mossa completa nel gioco.

- `grid1.m` & `grid2.m`: L'agente gioca contro un avversario **casuale**.
- `gridAuto.m` & `gridAuto2.m`: Un agente gioca contro un altro **agente AI** che usa i propri pesi.
- `grid3.m`: L'agente gioca contro un **giocatore umano** che inserisce la mossa da tastiera.

3.5 La Logica di Gioco e le Utilità

- `checker.m`: Funzione fondamentale che controlla se una mossa ha prodotto una vittoria.
- `possibleaction.m`: Utility che restituisce le colonne disponibili per la mossa successiva.
- `Board.m`: Funzione ausiliaria per aggiornare la matrice di stato.

3.6 La Visualizzazione

- `DrawBoard.m, DrawX.m, DrawO.m`: Funzioni grafiche di base per disegnare la griglia e le pedine.
- `MakeBoard.m`: Popola la griglia grafica in base alla matrice di stato.
- `in_board.m`: Funzione principale che gestisce la finestra del grafico.
- `swapRows.m`: Inverte verticalmente le righe della matrice per una visualizzazione corretta, poiché in MATLAB la riga 1 è in alto, mentre in Forza 4 le pedine cadono in basso.

3.7 I Dati Addestrati (`MC_f3.mat`, `MC_f4.mat`)

Questi file binari contengono le variabili salvate al termine di una sessione di addestramento, in particolare i vettori dei pesi `w` e `w3`, che rappresentano la "conoscenza" acquisita dagli agenti.

4 Guida all'Utilizzo e Possibili Miglioramenti

4.1 Come Avviare il Progetto

1. **Setup**: Assicurarsi che tutti i file `.m` siano presenti nella stessa cartella o nel path di MATLAB.

2. **Addestramento Completo**: Eseguire lo script `MC_f4.m`. Questo avvierà l'intero processo di addestramento, che richiederà una notevole quantità di tempo.

3. **Giocare contro l'AI**: Dopo l'addestramento, la sezione `GAME VS ME` si avvierà automaticamente. In alternativa, si può caricare un file `.mat` pre-addestrato e eseguire solo la sezione di gioco interattivo.

4.2 Analisi Critica e Miglioramenti

Il progetto è una solida implementazione, ma può essere migliorato.

- **Ingegneria delle Features (Impatto Alto)**: Aggiungere features più sofisticate per catturare concetti come "trappole" (forks), dove una mossa crea due minacce simultanee.

- **Ottimizzazione dell'Algoritmo (Impatto Medio)**: Implementare un **epsilon decrescente** per bilanciare meglio esplorazione e sfruttamento. Introdurre il **reward shaping** con piccole ricompense intermedie potrebbe accelerare l'apprendimento.

- **Architettura del Software (Impatto a Lungo Termine)**: Riorganizzare il codice utilizzando classi (es. `Agente`, `Ambiente`) migliorerebbe la leggibilità. Per un salto di qualità, si potrebbe passare a una **Deep Q-Network (DQN)**, eliminando la necessità di feature engineering manuale.

5 Conclusione

Il progetto rappresenta un eccellente esempio di applicazione di tecniche avanzate di RL a un gioco da tavolo. L'uso di SARSA(λ) con approssimazione di funzione e una strategia di addestramento a più fasi dimostra una profonda comprensione dei principi dell'IA. Sebbene la sua efficacia sia intrinsecamente legata alla qualità delle features definite manualmente, il sistema è completo, funzionante e costituisce una base formidabile per ulteriori sperimentazioni nel campo dell'intelligenza artificiale applicata al gioco.