Sviluppo di un Agente Intelligente per il Gioco 2048

Mario Lezzi[a], Daniele Dello Russo[a], Rocco Fortunato[a]

[a] *University of Salerno, Salerno, Italy*

1. Introduzione

Negli ultimi anni, gli algoritmi di Reinforcement Learning hanno dimostrato notevoli capacità nell'affrontare problemi complessi di decisione sequenziale. Il gioco 2048 rappresenta un banco di prova interessante per questi algoritmi. Questo lavoro si propone di sviluppare un agente intelligente in grado di giocare autonomamente a 2048, ottimizzando le sue prestazioni attraverso tecniche di Reinforcement Learning. In particolare, l'obiettivo è progettare e addestrare un agente che apprenda strategie efficaci per combinare i blocchi, massimizzando il punteggio e minimizzando le penalità derivanti da mosse subottimali. A tal fine, si è adottato l'algoritmo Q-learning, una tecnica classica di RL basata sull'aggiornamento iterativo dei valori di stato-azione per migliorare le decisioni dell'agente. La metodologia implementata prevede la creazione di un ambiente simulato del gioco 2048, la definizione degli stati e delle azioni disponibili, e l'adozione di un sistema di reward e penalità per guidare l'apprendimento. Attraverso un processo di training iterativo, l'agente viene ottimizzato per ottenere prestazioni sempre migliori nel raggiungimento di punteggi elevati. I risultati di questo studio contribuiscono a dimostrare l'efficacia del RL nel contesto del gioco 2048 e forniscono spunti utili per future applicazioni in problemi decisionali simili.

1.1. Introduzione al Q-learning

Per approfondire il Q-learning, un punto di partenza per il lavoro svolto è stato l'articolo di *Samina Amin*, *Deep Q-Learning (DQN)*[4], che spiega come il deep Q-learning estenda il Q-learning tradizionale utilizzando reti neurali profonde per approssimare la funzione Q. L'articolo fornisce una panoramica dei concetti chiave del reinforcement learning, tra cui la funzione Q, l'aggiornamento dei pesi tramite l'equazione di Bellman, e introduce tecniche come l'*experience replay* per migliorare la stabilità dell'apprendimento e il ruolo della *target network* per ridurre le fluttuazioni nei pesi. Un'altra fonte di ispirazione è stato il lavoro di *Shruti Dhumne*[5] , che approfondisce l'implementazione di un DQN, con particolare attenzione all'architettura delle reti neurali, alla gestione degli stati e delle azioni, e alla minimizzazione della funzione di errore di Bellman. L'articolo sottolinea anche l'importanza della scelta di una funzione di ricompensa adeguata e dei parametri di esplorazione/esploitazione per ottimizzare le prestazioni dell'agente nel lungo periodo.

2. Stato dell'arte

Sono stati presi in considerazione i progetti elencati di seguito.

2.1. DQN-2048 di Sergio Iommi

Il progetto *DQN-2048* di *Sergio Lommi* [3] , mira a creare un agente intelligente in grado di giocare al gioco 2048 utilizzando tecniche di deep reinforcement learning. In particolare, è stato implementato un deep Q-network (DQN) che impiega reti neurali profonde e reti neurali convoluzionali per apprendere le politiche di controllo direttamente dallo stato dell'ambiente, rappresentato dalla griglia del gioco 2048. Durante il training, l'agente ha mostrato miglioramenti progressivi, raggiungendo tessere con valori sempre più alti (256, 512, 1024 e alcune volte 2048) e incrementando il punteggio medio nel tempo. Nel progetto *DQN-2048*, i numeri della griglia di gioco vengono rappresentati attraverso una codifica multi-canale basata su one-hot encoding. Ogni cella della matrice 4×4 viene trasformata in una rappresentazione binaria su 16 canali, ognuno dei quali corrisponde a una potenza di 2. Ad esempio, se una cella contiene il valore 0, solo il primo canale avrà valore 1, mentre gli altri saranno 0; se contiene 2, sarà attivo solo il secondo canale, e così via. Questa rappresentazione evita problemi di scala, migliora la separabilità dei dati e facilita l'apprendimento delle regole di combinazione delle tessere. La trasformazione della griglia avviene convertendo ogni numero in una serie di canali binari corrispondenti alle potenze di 2.

2.2. DQN-2048 di Aju22

Nel progetto di *Aju22*[1], è realizzato un modello basato su TensorFlow, utilizzando una rete neurale personalizzata con layer convoluzionali. Questo approccio permette di catturare meglio le strutture spaziali della griglia di gioco, migliorando la capacità dell'agente di apprendere schemi strategici.

2.3. 2048rl di Lok Hin Chan

Nel lavoro di *Lock Hin Chan*[2], l'input della rete neurale è modellato con una forma ($16 \times 4 \times 4$), dove 16 rappresenta il numero di canali, ciascuno corrispondente a una potenza di 2 nella griglia di gioco 4×4. Questo approccio è fondamentale perché consente di mantenere un'architettura con 16 mappe di attivazione separate, permettendo alla rete di apprendere più rapidamente le relazioni tra le celle. Questo lavoro ha fornito indicazioni cruciali nell'accelerare il processo di training, migliorando l'efficienza nell'ottimizzazione del modello.

3. Esperimenti

In questa sezione vengono presentati i quattro esperimenti condotti per valutare l'efficacia dell'algoritmo Deep Q-Network (DQN) applicato al gioco 2048. Ad ogni esperimento sono state apportate delle modifiche all'architettura della rete, al meccanismo di aggiornamento e/o alla rappresentazione dello stato. Queste varianti mirano a migliorare la convergenza e le performance dell'agente.

3.1. DQN con Rete Fully-Connected

Nell' esperimento è stata utilizzata una rete neurale fully-connected. La rete prevede un layer Flatten per appiattire lo stato, una serie di layer Dense con unità decrescenti (1024, 512, 256, 128, 64, 32, 16, 8) e funzione di attivazione ReLU e infine un layer di output lineare avente un numero di neuroni pari allo spazio delle azioni. L'ottimizzatore scelto è Adam con learning rate pari a 0.001 e la funzione di perdita è l'Huber loss.

3.2. DQN con Prioritized Experience Replay

In questo esperimento si integra il meccanismo di Prioritized Experience Replay per dare maggior peso alle esperienze più rilevanti. Le modifiche principali includono: l'implementazione di una classe PrioritizedSequentialMemory che estende la memoria sequenziale standard, modifica la funzione di campionamento per pesare le transizioni in base alla loro priorità. Infine, viene utilizzato l'ottimizzatore SGD con un learning rate ridotto (5^{-5}) per una maggiore stabilità.

3.3. DQN con Rete Convoluzionale

Questo esperimento prevede l'utilizzo di una rete convoluzionale per migliorare la capacità di estrazione delle caratteristiche dallo stato del gioco. Sono stati utilizzati blocchi convolutivi con kernel di dimensioni multiple (1, 2, 3, 4) per catturare diverse scale spaziali, viene inserito un layer Flatten per passare ai livelli fully-connected; si è realizzato un layer Dense seguito da Dropout (50%) per ridurre l'overfitting. L'ottimizzazione utilizzato è Adam con learning rate pari a 1^{-5} e funzione di perdita MSE.

3.4. DQN con Rappresentazione Binaria dello Stato e Aggiornamento Epsilon Adattivo

In questo esperimento l'attenzione è posta sulla rappresentazione dello stato: lo stato 4x4 viene convertito in una rappresentazione binaria a 16 canali (4x4x16), in cui ogni canale rappresenta la presenza (o assenza) di una potenza di 2. In seguito è stato effettuato l'aggiornamento dinamico dei parametri ϵ e β, per modulare rispettivamente la strategia ϵ-greedy e il campionamento della memoria prioritaria.

4. Implementazione

La seguente sezione descrive le scelte progettuali adottate nell'implementazione dell'agente per il gioco 2048. Sono stati sviluppati due algoritmi: il primo è il Q-Learning base, mentre il secondo illustra il passaggio al Deep Q-Learning, in cui i valori Q vengono approssimati mediante una rete neurale.

4.1. Q-Learning base

Per poter utilizzare la Q-table, lo stato del gioco (una griglia 4x4) viene convertito in una forma hashable che permetta di usarlo come chiave nel dizionario. Questo consente di associare a ciascuno stato un vettore di valori inizialmente impostato a zero, dove ogni componente corrisponde a un'azione. L'agente adotta una strategia ϵ-greedy per bilanciare esplorazione e sfruttamento. Se un numero casuale è inferiore al parametro ϵ, viene scelta un'azione casuale; altrimenti, viene selezionata l'azione con il valore Q massimo per lo stato corrente. L'aggiornamento avviene secondo la seguente regola:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right],$$

dove α rappresenta il learning rate e γ il fattore di sconto. Questo schema permette all'agente di aggiornare iterativamente il valore associato alla coppia stato-azione in base alla reward ricevuta e al valore stimato dello stato successivo. Il tasso di esplorazione ϵ viene ridotto progressivamente durante l'allenamento. Per rendere il decadimento più efficace e adattabile, sono state definite diverse fasi (inizialmente un decadimento lento, seguito da una fase di decadimento più rapido e infine un rallentamento nel decremento) in modo da garantire una transizione graduale dall'esplorazione allo sfruttamento.

Questi elementi sono alla base dell'implementazione classica, che si focalizza sul semplice aggiornamento iterativo dei valori della Q-table. Sebbene efficace in ambienti con spazi di stato contenuti, questo approccio presenta limiti nel caso di ambienti complessi come 2048, dove lo spazio degli stati è molto ampio.

4.2. Deep Q-Learning

L'approccio Deep Q-Learning implementato nel progetto sostituisce la tradizionale Q-table con una rete neurale che approssima la funzione Q, consentendo di gestire spazi di stato di dimensioni molto maggiori e di estrarre automaticamente le caratteristiche più rilevanti dallo stato del gioco. Per migliorare l'efficienza dell'apprendimento, è stato integrato un meccanismo di *PER*. Sono presenti funzioni per accedere agli ultimi elementi della memoria e per pulire il buffer, rimuovendo i game con score particolarmente basso. Queste operazioni aiutano a mantenere una memoria di esperienze rilevanti e ad evitare che dati di bassa qualità influenzino l'allenamento. La rete neurale utilizzata per approssimare la funzione Q è strutturata in modo da estrarre efficacemente le caratteristiche dallo stato del gioco. L'input della rete è di forma $(16, 4, 4)$, ottenuto tramite una codifica one-hot a 16 canali. Sono impiegati blocchi convolutivi che utilizzano kernel di diverse dimensioni (1, 2, 3, 4). I risultati delle convoluzioni vengono concatenati e passati attraverso un'operazione ReLU per introdurre non linearità. Dopo i blocchi convolutivi, la rete effettua un'operazione di Flatten per convertire i dati in un vettore. L'output è un vettore lineare con un numero di neuroni pari al numero di azioni possibili, il quale rappresenta i Q-value stimati per ciascuna azione. La rete viene compilata utilizzando l'ottimizzatore Adam con un learning rate 5×10^{-5} e la loss adottata è il Mean Squared Error (MSE). Viene implementato un meccanismo dinamico che

riduce il learning rate in presenza di particolari condizioni (ad esempio, quando viene raggiunta una certa soglia di punteggio), contribuendo a stabilizzare l'apprendimento nelle fasi avanzate. Per stabilizzare il training, viene mantenuto una *target network* che viene periodicamente aggiornato con i pesi della rete principale. Questo riduce la varianza dei target utilizzati durante l'ottimizzazione.

Il Deep Q-Learning viene applicato all'ambiente 2048, definito in una classe che gestisce la logica del gioco (ad esempio, il movimento della griglia, l'aggiunta di nuovi numeri e la verifica dello stato terminale). Durante il training:

1. L'agente riceve lo stato corrente e lo codifica.
2. Viene selezionata un'azione sulla base dei Q-value stimati e della politica ϵ-greedy.
3. L'azione viene eseguita sull'ambiente, che restituisce il nuovo stato, la ricompensa e l'indicazione di termine episodio.
4. La transizione viene memorizzata nel `PrioritizedSequentialMemory`.
5. Al termine di ogni game, l'agente esegue il training aggiornando i pesi della rete e le priorità delle transizioni.

5. Risultati

In questa sezione vengono mostrati i risultati ottenuti utilizzando due diversi algoritmi di Reinforcement Learning: Q-Learning e Deep Q-Learning (DQN). Il grafico [1] evidenzia come l'agente abbia migliorato progressivamente la capacità di combinare blocchi per raggiungere valori sempre più elevati sulla griglia. Nei primi episodi, i valori massimi raggiunti erano limitati a 256 o 512, con poche eccezioni di 1024. Tuttavia, con l'avanzare del training, il raggiungimento del valore 2048 è diventato più frequente.

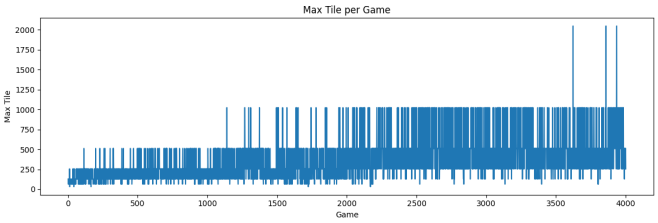

Figure 1: Grafico andamento tessera massima

Come mostrato nel grafico del punteggio per game [2], la curva in rosso indica una chiara tendenza al rialzo del punteggio medio. Questa progressione evidenzia come l'agente sia stato in grado di combinare blocchi in modo sempre più efficiente e strategico con l'aumentare dei game.

Dal grafico delle reward [3], si nota che, nei primi game, l'agente otteneva principalmente penalità, con valori di reward ridotti. Con l'avanzare del training, le reward positive sono diventate più frequenti, dimostrando un miglioramento nell'efficacia delle strategie adottate.

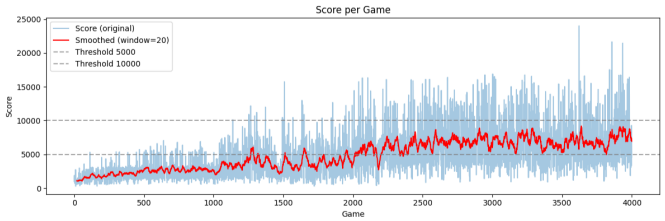

Figure 2: Grafico andamento punteggio per game

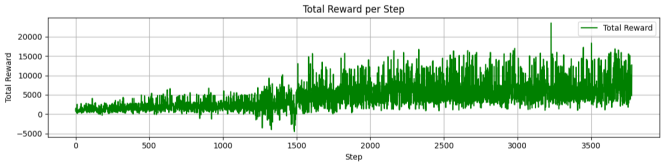

Figure 3: Grafico andamento reward per game

5.1. Confronto tra Q-Learning e Deep Q-Learning

Di seguito un'analisi comparativa dei due algoritmi utilizzati con l'obiettivo di valutare le prestazioni degli agenti addestrati in termini di capacità di raggiungere il max tile nella griglia di gioco del 2048 e di ottimizzare il punteggio complessivo. Il Q-Learning è stato implementato con quattro diverse tecniche di calcolo della reward e altri accorgimenti. Verrà mostrato solo l'approccio migliore. I risultati sono riassunti nel seguente grafico:

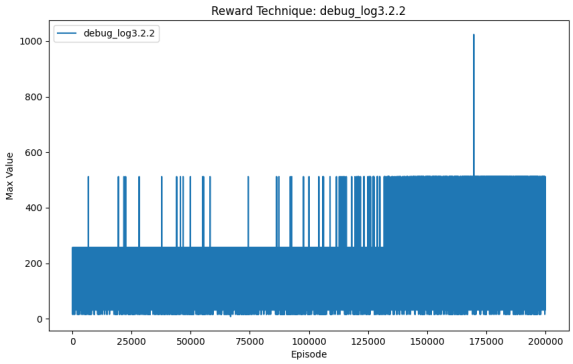

Figure 4: Andamento delle tessere massime raggiunte per episodio con il Q-Learning.

Il grafico delle max tile per game [4] mostra un incremento sporadico dei valori massimi raggiunti durante il training. Due dei quattro approcci hanno raggiunto la max tile pari a 1024, ma solo verso le fasi finali del training, che si componeva di 200.000 game. Il Deep Q-Learning (DQN) si distingue per l'uso di una rete neurale per approssimare la funzione Q. Il grafico della max tile [5] mostra i seguenti risultati.

Il DQN ha raggiunto il valore massimo 2048, dimostrando una capacità superiore nell'ottimizzazione delle mosse e delle strategie. Inoltre, si nota come la max tile trovata più frequentemente nell'approccio con DQN è stata 512, poi il 256 e poi il 1024. Per quanto riguarda la max tile, il Q-Learning si è fer-

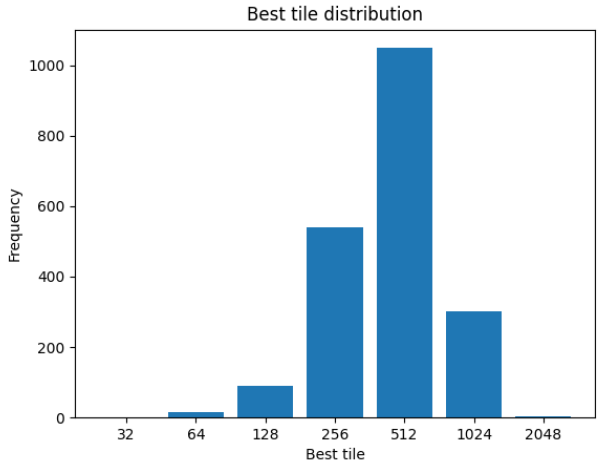

Figure 5: Andamento del massimo valore di tessera con il Deep Q-Learning.

mato a un massimo di 1024, mentre il DQN ha raggiunto il valore di 2048, vincendo difatti il gioco. Il DQN ha mostrato un miglioramento costante nello score per game, dimostrando una migliore capacità di ottimizzare le combinazioni dei blocchi rispetto al Q-Learning.

6. Sfide affrontate

Durante l'implementazione del progetto, sono emerse diverse sfide legate alla complessità del problema e al training del modello. Di seguito sono riportate le principali difficoltà e le soluzioni adottate. Una delle sfide principali è stata sviluppare una rete neurale capace di apprendere strategie spaziali efficaci. Utilizzando un'architettura convolutiva (con l'uso di kernel con dimensioni diverse) ha migliorato la capacità della rete di catturare le relazioni locali e globali sulla griglia. Inizialmente, le reward focalizzate solo sul punteggio limitavano le strategie, quindi è stata implementata una reward basata su combinazioni, numero di celle libere e penalità per mosse non valide; viene assenta una reward aggiuntiva al raggiungimento di tessere alte (es. 1024, 2048). Trovare valori ottimali per il Learning Rate e bilanciare esplorazione/sfruttamento (Epsilon/-Greedy) è stato critico. È stato necessario introdurre un adaptive decay, learning rate ed epsilon sono stati ridotti progressivamente per garantire stabilità e convergenza. Per prevenire la convergenza a strategie subottimali è stato utilizzato un Prioritized Experience Replay, che enfatizza esperienze rilevanti, calcolando la priorità come la differenza tra il valore stimato e il valore target aggiornato:

$$\delta_t = r_t + \gamma \cdot Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

Campioni casuali sono stati introdotti per diversificare il training, andando a selezionare il 50% dei valori con alta priorità e il 50% dei valori randomicamente. Il training richiedeva risorse significative, difatti l'uso dei checkpoint è stato fondamentale per salvare periodicamente il modello, il buffer e tutto lo stato del training. Per contrastare eventuali andamenti degradanti sono stati introdotti intra-checkpoint regolari per salvare i progressi del modello. Un meccanismo di rollback è stato utilizzato per interrompere il training e riportarlo a uno stato precedente nel caso in cui la media della max tile e della seconda max tile differivano di un valore che cresceva con la progressione del training.

7. Bibliografia

References

[1] AJU22. *DQN-2048*. 2022. URL: https://github.com/aju22/DQN-2048.git.

[2] Lok Hin Chan. *2048$_{rl}$*. 2022. URL: https://github.com/qwert12500/2048_rl.git.

[3] Sergio Lommi. *Codice 2048-2048*. 2018. URL: https://github.com/SergioIommi/DQN-2048.git.

[4] Medium. *Apprendimento Q profondo (DQN)*. 2024. URL: https://medium.com/@samina.amin/deep-q-learning-dqn-71c109586bae.

[5] Medium. *Rete Q profonda DQN*. 2023. URL: https://medium.com/@shruti.dhumne/deep-q-network-dqn-90e1a8799871.