**Q-Learning: Principi, Caratteristiche ed Applicazioni**

Sommario

[**Introduzione al Q-Learning** 1](#_Toc184717090)

[**Obiettivo del Q-Learning** 1](#_Toc184717091)

[**Proprietà del Q-Learning** 2](#_Toc184717092)

[**Vantaggi del Q-Learning** 2](#_Toc184717093)

[**Svantaggi del Q-Learning** 3](#_Toc184717094)

[**Problema nel nostro caso del 2048** 4](#_Toc184717095)

[**Struttura del Problema 2048** 5](#_Toc184717096)

[**Dimensione della Q-Table** 6](#_Toc184717097)

[**Considerazioni e confronto con DQL** 6](#_Toc184717098)

[**Vantaggi del Deep Q-Learning** 8](#_Toc184717099)

[**Svantaggi** 8](#_Toc184717100)

[**2048 con DQL** 8](#_Toc184717101)

[**Ambiente Gym** 9](#_Toc184717102)

[**Markov Decision Process** 10](#_Toc184717103)

[**Proprietà Fondamentali** 11](#_Toc184717104)

[**Risoluzione degli MDP** 11](#_Toc184717105)

[**Applicazioni** 12](#_Toc184717106)

[**Studio in Letteratura** 12](#_Toc184717107)

[**1.** **Playing 2048 With Reinforcement Learning** 12](#_Toc184717108)

[**2.** **Q-learning for 2048** 13](#_Toc184717109)

[**ROADMAP** 15](#_Toc184717110)

# **Introduzione al Q-Learning**

Il Q-Learning è un algoritmo di Reinforcement Learning che consente a un agente di apprendere una politica ottimale massimizzando le ricompense cumulative attese. Una caratteristica fondamentale del Q-Learning è che non richiede un modello dell’ambiente. L’agente impara interagendo direttamente con esso, raccogliendo ricompense e aggiornando la propria valutazione delle azioni disponibili.

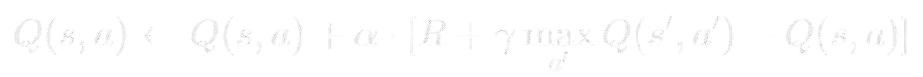
## **Obiettivo del Q-Learning**

L’obiettivo del Q-Learning è approssimare la funzione di valore-azione ottimale, indicata come **q∗(s,a).** Questa funzione esprime il massimo valore atteso a lungo termine eseguendo un’azione **a** nello stato **s**, e poi seguendo la politica ottimale successivamente.

**Funzione Q**

* **Funzione Q (Action-Value Function):** **Q(s,a)≈q∗(s,a)**

All’inizio, **Q(s,a)** è solo una stima. Interagendo con l’ambiente, l’agente la aggiorna iterativamente per avvicinarsi a **q∗(s,a)**.

**Formula di Aggiornamento**  
L’aggiornamento dei valori **Q** avviene secondo la seguente equazione:

Dove:

* Q(s,a): Valore corrente per lo stato s e l’azione a.
* α: Tasso di apprendimento (learning rate).
* R: Ricompensa immediata ottenuta dopo aver eseguito l’azione a.
* γ: Fattore di sconto (0 ≤ γ < 1), che determina l’importanza delle ricompense future.
* Immagine che contiene Carattere, Elementi grafici, tipografia, calligrafia

  Descrizione generata automaticamente: Massimo valore d’azione nello stato successivo s′.

## **Proprietà del Q-Learning**

1. **Off-Policy:**  
   Il Q-Learning apprende la politica ottimale **π** indipendentemente dalla politica utilizzata per generare i dati (μ). Ciò significa che l’agente può esplorare con una strategia μ (ad esempio epsilon-greedy) e, allo stesso tempo, convergere verso la politica ottimale π che è greedy rispetto a Q(s,a).
2. **Convergenza:**  
   Il Q-Learning converge a **q∗(s,a)** se:
   * Ogni stato e azione vengono esplorati un numero sufficiente di volte.
   * Il tasso di apprendimento α\alphaα diminuisce nel tempo ma rimane sempre positivo.

**Algoritmo Generale del Q-Learning**

1. **Inizializzazione:**
   * Imposta Q(s,a) a valori iniziali (spesso zero) per tutte le coppie (stato, azione).
2. **Ciclo di Apprendimento (per ogni episodio):**
   * Osserva lo stato corrente s.
   * Scegli un’azione a con una politica esplorativa (ad es. epsilon-greedy).
   * Esegui l’azione a, osserva la ricompensa R e il nuovo stato s′.
   * Aggiorna Q(s,a) con la formula di aggiornamento.
   * Imposta s=s′ e ripeti fino alla fine dell’episodio.
3. **Ripeti per un numero sufficiente di episodi** fino a quando il valore di Q(s,a) non converge o fino al raggiungimento di un criterio di arresto.

## **Vantaggi del Q-Learning**

* **Semplicità:** Facile da implementare.
* **Flessibilità:** Non richiede un modello esplicito dell’ambiente.
* **Teoria solida:** Garanzie di convergenza all’action-value function ottimale, date le condizioni di esplorazione adeguate.

## **Svantaggi del Q-Learning**

* **Scalabilità:** In spazi di stato/azione molto grandi, mantenere una tabella Q diventa impraticabile. La quantità di memoria necessaria e il tempo di addestramento crescono enormemente.
* **Equilibrio exploration/exploitation:** La strategia di esplorazione deve essere scelta con cura per garantire copertura adeguata dello spazio di stato-azione.

APPROFONDIMENTO: Nel RF, c'è un equilibrio fondamentale che un agente deve gestire e che all’aumentare di uno corrisponde una diminuzione dell’altro, stiamo parlando del rapporto tra:

* + **Exploration (Esplorazione)**:
    - Provare azioni nuove o meno frequenti per scoprire informazioni sull'ambiente. L'obiettivo è raccogliere dati su stati e ricompense che potrebbero migliorare il comportamento futuro dell'agente. Nel nostro caso, spostare i blocchi in una direzione non ottimale per capire come influenzerà la griglia.
  + **Exploitation (Sfruttamento)**:
    - Scegliere le azioni che, in base alle conoscenze attuali, hanno il valore Q(s,a) più alto. L'obiettivo in questo caso è massimizzare le ricompense a breve termine usando ciò che è stato appreso finora. Per noi sarebbe scegliere sempre la direzione che fonde i blocchi con il punteggio più alto.

**Confronto con SARSA**

* **Q-Learning:**  
  È più “aggressivo” e mira alla politica ottimale, anche se durante l’esplorazione può incappare in situazioni sfavorevoli.
* **SARSA:**  
  Utilizza la reale azione scelta nello stato successivo, Q(s′,a′)con a′ generato dalla politica attuale. È più “conservativo” e tende a politiche più sicure in ambienti rischiosi.

**Esempio: Cliff Walking**  
Nel problema Cliff Walking (una griglia con un dirupo), l’agente deve raggiungere un obiettivo evitando di cadere.

* **Q-Learning:** Trova il percorso ottimale che massimizza la ricompensa, ma durante l’esplorazione può cadere spesso.
* **SARSA:** Evita di avvicinarsi troppo al dirupo, preferendo percorsi meno remunerativi ma più sicuri.

**Condizioni di Applicabilità**

* **Adatto a:** Problemi con stati e azioni discreti e non troppo vasti, e quando si può interagire con l’ambiente a lungo.
* **Non adatto a:** Ambienti con spazi di stato/azione enormi o continui (a meno di ricorrere a metodi di approssimazione come il Deep Q-Learning).

# **Problema nel nostro caso del 2048**

1. **Spazio di Stato Molto Ampio**:  
   Il gioco 2048 si svolge su una griglia 4x4, ciascuna cella può assumere valori potenzialmente molto elevati (potenze di 2, fino a 2048 e oltre). Anche se il numero di combinazioni possibili non è infinito, rimane comunque enorme, abbiamo infatti i numeri 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048 per 16 celle totali. Il Q-Learning tabellare presuppone di avere una voce per ogni coppia (stato, azione). Dato l’altissimo numero di stati diversi, memorizzare una tabella Q completa è impraticabile dal punto di vista dello spazio di memoria.
2. **Generalizzazione Limitata**:  
   Il Q-Learning base considera gli stati come entità discrete e non generalizza tra stati simili. In 2048, molti stati sono strutturalmente simili (ad esempio, due griglie che differiscono per la posizione di due sole tessere), ma Q-Learning classico non sfrutta questa somiglianza. Ogni stato è visto come unico e non vi è un meccanismo per trasferire conoscenza acquisita in uno stato ad un altro simile.
3. **Nessuna Funzione di Approssimazione**:  
   Senza utilizzare funzioni di approssimazione (come reti neurali o funzioni lineari basate su feature), il Q-Learning tabellare non può trarre vantaggio da una rappresentazione compatta dello stato né può apprendere in modo efficiente. Questo rappresenta un grande limite in un gioco come 2048, dove la struttura della griglia e la presenza di pattern ricorrenti potrebbe essere sfruttata.
4. **Tempo di Convergenza**:  
   A causa delle dimensioni dello spazio degli stati e del fatto che molti di essi non saranno mai visitati, Q-Learning classico impiega molto tempo (o non riesce affatto) a convergere verso una buona politica. Nel frattempo, è possibile che l’agente continui a prendere decisioni subottimali perché non ha ancora consolidato un valore Q accurato per la maggior parte degli stati.
5. **Gestione delle Ricompense a Lungo Termine**:  
   Sebbene Q-Learning di per sé includa un fattore di sconto γ per tener conto delle ricompense future, la vastità dello spazio degli stati di 2048 rende difficile per l’agente imparare sequenze di mosse a lungo termine in tempi ragionevoli, rendendo la policy finale meno efficace.

## **Struttura del Problema 2048**

* Il gioco 2048 si svolge su una griglia **4x4**, quindi **16 celle** in totale.
* Ogni cella può essere:
  + Vuota
  + Occupata da una tessera con valore pari a una potenza di 2: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048...

Le tessere possibili per ogni cella, includendo l’opzione “vuota” e tutte le potenze di 2 fino a 2048, sono:

* Vuota
* 2 = 2¹
* 4 = 2²
* 8 = 2³
* 16 = 2⁴
* 32 = 2⁵
* 64 = 2⁶
* 128 = 2⁷
* 256 = 2⁸
* 512 = 2⁹
* 1024 = 2¹⁰
* 2048 = 2¹¹

In totale: **1 (vuota) + 11 (tessere) = 12 possibili valori** per cella.

**Spazio degli Stati**

Lo stato del gioco è l’insieme dei valori di tutte le 16 celle. Se ogni cella può assumere 12 valori diversi, allora il numero totale di stati è:

∣S∣=12^16

Stimiamo la grandezza:

* log⁡10(12)≈1.079
* 1.079×16=17.264 circa.
* Quindi 12^16≈10^17≈1.8×10^17.

In altre parole, ci sono dell’ordine di centinaia di migliaia di miliardi di miliardi di possibili stati. Questo è già un numero astronomico.

**Azioni**

In 2048, l’agente può compiere 4 azioni: muovere **su**, **giù**, **sinistra**, **destra**.

Quindi, per ogni stato, abbiamo 4 possibili azioni.

## **Dimensione della Q-Table**

La Q-table mappa ogni coppia (stato, azione) a un valore Q. Dato che per ciascuno stato abbiamo 4 azioni:

∣Q∣=∣S∣×∣A∣=12^16×4

Usando la stima di 12^16≈1.8×10^17

∣Q∣≈4×1.8×10^17=7.2×10^17

Circa 7×10^17 entries, cioè 700.000.000.000.000.000 (settecento milioni di miliardi) di valori Q da memorizzare ed aggiornare.

# **Considerazioni e confronto con DQL**

Il Q-Learning è facile da implementare e comprende un solido framework teorico di convergenza. Tuttavia, le sue prestazioni si degradano in problemi di grandi dimensioni, è doveroso quindi prendere in considerazioni il Deep Q-Learning, che permette di affrontare spazi di stato molto ampi o continui. Ma in cosa differisce il DQL?

Il **Deep Q-Learning (DQL)** rappresenta un’estensione del Q-Learning tradizionale, concepita per affrontare problemi con spazi di stato molto ampi o continui. Mentre il Q-Learning classico utilizza una tabella per immagazzinare i valori Q(s,a), il DQL fa ricorso a reti neurali profonde (Deep Neural Networks, DNN) per approssimare la funzione Q, superando così i limiti di memoria e di scalabilità imposti dall’uso di tabelle.

**Motivazione**

* **Limiti del Q-Learning Tabellare:** In molti problemi reali, lo spazio degli stati può essere enorme o addirittura continuo (es. immagini, input sensoriali complessi, ambienti 3D). Memorizzare un valore Q per ogni coppia stato-azione diventa impraticabile.
* **Soluzione con DQL:** Utilizzando una rete neurale in grado di approssimare Q(s,a), è possibile generalizzare su stati mai visti e ridurre drasticamente le risorse necessarie. Questo approccio è stato introdotto con successo da DeepMind per affrontare diversi videogiochi, dimostrando prestazioni superiori alle tecniche classiche.

**Funzionamento del Deep Q-Learning**

1. **Approssimazione con Rete Neurale:** Una rete neurale prende in input lo stato s (ad esempio un’immagine di un videogioco) e produce in output i valori Q(s,a) per ogni azione a.  
   In questo modo, non esiste più una tabella, ma una funzione Q(s,a;θ) parametrizzata dai pesi θ della rete.
2. **Aggiornamento dei Pesi:** L’obiettivo è minimizzare la differenza tra i Q-value predetti e i target attesi.
3. **Stabilizzare l’Apprendimento:**

* **Target Network:**  
  Si utilizza una “target network” separata per generare i target y. Questa rete è una copia periodicamente aggiornata dei pesi principali. Ciò riduce l’instabilità e l’oscillazione durante il training.
* **Experience Replay:** Invece di aggiornare i pesi dopo ogni transizione (s, a, r, s'), si utilizza un buffer (Replay Memory) dove sono immagazzinate esperienze passate. A ogni step di aggiornamento, si campiona casualmente un mini-batch di esperienze dal buffer per addestrare la rete.

L’Experience Replay riduce la correlazione tra le transizioni consecutive e rende l’apprendimento più stabile ed efficiente.

1. **Strategia di Esplorazione:** Come nel Q-Learning classico, anche in DQL si utilizza spesso una politica epsilon-greedy per bilanciare esplorazione e sfruttamento. Inizialmente, **ϵ** è alta e viene ridotta progressivamente man mano che l’agente impara.

## **Vantaggi del Deep Q-Learning**

* **Scalabilità:** Può gestire spazi di stato estremamente grandi (come input visivi).
* **Generalizzazione:** La rete neurale apprende una rappresentazione interna degli stati, generalizzando a situazioni mai viste in precedenza.
* **Risultati notevoli:** L’agente DQL di DeepMind ha ottenuto risultati super-umani in molti giochi Atari, partendo solo dai pixel dello schermo come input.

## **Svantaggi**

* **Instabilità:** Senza tecniche di stabilizzazione (target network, experience replay), l’apprendimento può risultare molto instabile.
* **Iperparametri Sensibili:** La scelta di α\alphaα, dimensione dei mini-batch, dimensione del replay buffer, frequenza di aggiornamento della rete target e altre impostazioni influenzano fortemente i risultati.
* **Complessità Computazionale:** L’allenamento di reti neurali su molti episodi richiede risorse di calcolo elevate.

**Varianti**

* **Double DQN:** Affronta il problema della sovrastima dei Q-value.
* **Dueling DQN:** Separa il valore dello stato dal vantaggio delle azioni, migliorando l’efficienza.
* **Prioritized Replay:** Sceglie le transizioni da ripetere più spesso se sono più “interessanti” (con errore di TD più elevato).

## **2048 con DQL**

Il numero di stati possibili rimane invariato, quindi sempre 1.8x10^17. Tuttavia, con il Deep Q-Learning, non esiste una Q-table esplicita. Invece di avere una voce Q(s,a) per ogni coppia (stato, azione), utilizziamo una **rete neurale** per approssimare la funzione Q. Mentre la Q-table deve enumerare ogni stato, la rete neurale impara pattern e regole generali che consentono di approssimare il valore Q anche di stati mai visti prima o molto simili a stati già incontrati.  
In altre parole, addestrando la rete su un sottoinsieme di stati, si spera di ottenere buone stime di Q anche per altri stati correlati, evitando di dover “visitare” ogni configurazione.

# **Ambiente Gym**

Un ambiente Gym è un'interfaccia standardizzata, proposta originariamente dalla libreria *OpenAI Gym*, che permette di interagire con una grande varietà di problemi di apprendimento per rinforzo in modo uniforme. In sostanza, un "ambiente Gym" è un contesto (ad esempio un gioco, un problema di controllo robotico, un puzzle, ecc.) che mette a disposizione dell'agente:

1. **Osservazioni (stati)**: una descrizione di ciò che l'agente "vede" in un determinato momento. Nel caso del gioco 2048, ad esempio, lo stato è l'intera griglia con le tessere.
2. **Azioni**: le mosse che l'agente può compiere. Nel 2048 le azioni sono "su", "giù", "sinistra", "destra".
3. **Ricompense**: un punteggio numerico ricevuto dall'agente dopo aver intrapreso un'azione, che misura quanto quella mossa è stata vantaggiosa in vista dell'obiettivo finale. Nel 2048, tipicamente la ricompensa può essere il punteggio guadagnato fondendo tessere.
4. **Segnale di Fine Episodio (done)**: un flag booleano che indica se l'ambiente ha raggiunto uno stato terminale (ad esempio, se non si possono più fare mosse oppure se si è raggiunta una determinata condizione di vittoria).

Attraverso queste quattro componenti (stato, azione, ricompensa, done), un ambiente Gym fornisce un'interfaccia standard e coerente, in modo che un algoritmo di apprendimento per rinforzo possa essere applicato a diversi problemi senza dover cambiare il proprio codice se non per specificare quale ambiente utilizzare.

In pratica, usare un ambiente Gym significa poter chiamare:

* **env.reset()** per inizializzare l'ambiente e ottenere lo stato iniziale.
* **env.step(azione)** per eseguire una mossa e ottenere:
  + Il nuovo stato (osservazione successiva)
  + La ricompensa per l'azione intrapresa
  + Il flag done che indica se l'episodio è terminato
  + Un dizionario info con informazioni aggiuntive (opzionali)

Questa standardizzazione rende più semplice confrontare e sviluppare algoritmi di RL su diversi problemi.

# **Markov Decision Process**

Un **Markov Decision Process (MDP)** è un modello matematico ampiamente utilizzato nel campo del Reinforcement Learning e del controllo sequenziale. Esso fornisce una struttura formale per descrivere un agente che interagisce con un ambiente nel tempo, prendendo decisioni (azioni) e ricevendo ricompense, con l’obiettivo di massimizzare una misura cumulativa delle ricompense ottenute.

Un MDP è tipicamente definito dalla quintupla (S,A,P,R,γ):

1. **Stati (S):**  
   L’ambiente è descritto da un insieme di stati s∈S. Ogni stato rappresenta una configurazione completa del sistema. Ad esempio, nello scenario del gioco 2048, lo stato potrebbe essere la configurazione corrente della griglia; in un problema di controllo robotico, lo stato potrebbe includere la posizione e la velocità del robot.
2. **Azioni (A):**  
   L’agente, trovandosi in uno stato sss, può scegliere un’azione a∈A(s) (cioè un’azione tra quelle consentite in quello stato). L’insieme delle azioni disponibili può essere lo stesso in tutti gli stati (azioni globali) o variare in base allo stato.
3. **Transizioni (P):**  
   Le dinamiche dell’ambiente sono descritte da una funzione di probabilità di transizione:

P(s′∣s,a)

che indica la probabilità di finire nello stato s′s's′ eseguendo azione aaa dallo stato sss. Questo riflette la natura stocastica e incerta dell’ambiente. La proprietà di Markov implica che la probabilità del prossimo stato dipende solo dallo stato attuale e dall’azione intrapresa, non dalla storia passata.

1. **Ricompense (R):**  
   Il sistema fornisce una ricompensa immediata R(s,a,s′) quando l’agente esegue l’azione a nello stato sss e si ritrova in s′. La ricompensa è un numero (spesso reale) che misura quanto sia “buono” il risultato di quella transizione. L’obiettivo dell’agente è massimizzare la somma cumulativa di queste ricompense nel lungo periodo.
2. **Fattore di Sconto (γ\gammaγ):**  
   γ∈[0,1] è un parametro che definisce quanto l’agente preferisce ricompense immediate rispetto a quelle future. Se γ è vicino a 1, l’agente è “paziente” e valuta quasi allo stesso modo ricompense immediate e future. Se γ è basso, le ricompense future contano di meno rispetto a quelle immediate.

## **Proprietà Fondamentali**

* **Proprietà di Markov:**  
  La probabilità di transizione e la ricompensa dipendono esclusivamente dallo stato corrente e dall’azione scelta, non dalla storia passata. Questo semplifica notevolmente l’analisi, poiché non è necessario considerare l’intera sequenza di stati precedenti.
* **Politica (Policy) π:**  
  Una politica π è una funzione che specifica quale azione intraprendere in ogni stato, ad esempio π(s)=a. L’obiettivo dell’agente è trovare una politica ottimale π\* che massimizza il valore atteso cumulativo delle ricompense.
* **Funzioni di Valore:**  
  Per valutare la bontà di una politica, si definiscono delle funzioni di valore, come:
  + **Funzione Valore dello Stato Vπ(s):** Il valore atteso (scontato) delle ricompense future partendo dallo stato sss e seguendo la politica π\piπ.
  + **Funzione Valore Stato-Azione Qπ(s,a):** Il valore atteso (scontato) delle ricompense future se dall’attuale stato sss si esegue l’azione a e poi si continua a seguire π.

## **Risoluzione degli MDP**

Esistono vari metodi per risolvere (o approssimare la soluzione di) un MDP:

* **Programmazione Dinamica (DP):**  
  Se abbiamo un modello completo dell’MDP (cioè conosciamo esattamente PPP e RRR), possiamo usare metodi come l’Iterazione di Valore (Value Iteration) o l’Iterazione di Politica (Policy Iteration) per trovare una politica ottimale.
* **Reinforcement Learning (RL):**  
  Quando il modello non è noto o non è pratico da usare, l’agente può imparare dalla sola interazione con l’ambiente usando algoritmi di apprendimento per rinforzo. Ad esempio, Q-Learning o SARSA, che approssimano i valori di stato-azione a partire dall’esperienza empirica.

## **Applicazioni**

Gli MDP trovano applicazione in numerosi contesti:

* Controllo di robotica e droni (decidere le azioni di controllo in base allo stato).
* Gestione di risorse e finanza (decidere investimenti o consumi in condizioni incerte).
* Giochi (ad esempio, trovare una strategia ottimale per vincere).
* Sistemi di raccomandazione, controllo dei traffici, logistica, produzione industriale, ecc.

In breve, un MDP è un quadro matematico potente che consente di formalizzare problemi di decisione sequenziale in cui le scelte dell’agente influenzano sia gli stati futuri che le ricompense, garantendo una base rigorosa per lo sviluppo di algoritmi di pianificazione, apprendimento e controllo ottimale.

# **Studio in Letteratura**

* 1. **Playing 2048 With Reinforcement Learning**

[..\..\..\..\..\..\..\OneDrive\Desktop\Università\IA\Paper1.pdf](file:///C:\Users\rocco\OneDrive\Desktop\Università\IA\Paper1.pdf)

**Obiettivi**

* Utilizzare tecniche di Reinforcement Learning per vincere il gioco 2048.
* Valutare l'efficacia del **Deep Q-Learning** e del **Beam Search**.

**Approcci Implementati**

1. **Deep Q-Learning (DQL)**:
   * Modelli basati su reti neurali:
     + **Model 1**: Strati lineari e dropout per prevenire overfitting.
     + **Model 2**: Architettura convoluzionale con più kernel per stabilizzare l'apprendimento.
   * Politica di esplorazione **ε-greedy** (ϵ=0.3).
   * Addestramento su 1000 partite, minimizzando l'errore quadratico medio per aggiornare i valori Q.
2. **Beam Search**:
   * Algoritmo di ricerca euristica basato su profondità d=20 e massimo k=10 nodi migliori per livello.
   * Utilizza una funzione di "goodness" che considera:
     + Numero di celle vuote.
     + Valore massimo delle tessere.
     + Monotonicità e smoothness della griglia.

**Risultati**

* **Deep Q-Learning**:
  + Prestazioni inferiori rispetto al Beam Search, faticando a raggiungere la tessera 2048.
  + I modelli mostrano difficoltà di convergenza a causa dell'incertezza nel posizionamento delle tessere.
* **Beam Search**:
  + Raggiunge la tessera 2048 nel **28,5%** delle partite.
  + Strategia più vicina a quella umana grazie alle euristiche.

**Conclusioni**

* Il Beam Search, basato su strategie umane, è più performante del DQL non supervisionato.
* Il Deep Q-Learning richiede un maggiore addestramento e dati per avvicinarsi a strategie ottimali.
* Future work: incorporare simmetrie (rotazioni e riflessioni) nella rete per generare più dati di addestramento.
  1. **Q-learning for 2048**

[..\..\..\..\..\..\..\OneDrive\Desktop\Università\IA\Paper2.pdf](file:///C:\Users\rocco\OneDrive\Desktop\Università\IA\Paper2.pdf)

**Obiettivi**

* Combinare il **Q-Learning** con la ricerca ad albero (Expectimax) per migliorare le performance nel gioco 2048.
* Valutare strategie basate su RL puro e algoritmi ibridi.

**Approcci Implementati**

1. **Q-Learning Puro**:
   * Rappresentazione dello stato:
     + Magnitudo logaritmica delle tessere.
     + Differenze logaritmiche tra tessere adiacenti.
   * Politica di esplorazione **epsilon-greedy** con epsilon decrescente.
   * Problemi iniziali:
     + Overfitting e difficoltà nel generalizzare su tessere di alto valore.
     + Addestramento lento senza ottimizzazioni (ad es., esperienza ritardata).
2. **Algoritmo Ibrido (Q-Learning + Expectimax)**:
   * Uso della funzione Q per potare nodi nell'albero di ricerca Expectimax.
   * Conservazione di **2 mosse promettenti** per livello, raddoppiando la profondità esplorabile.

**Risultati**

* **Q-Learning Puro**:
  + Faticava a superare la tessera 128 a causa dell'imprevedibilità nella generazione delle tessere.
* **Algoritmo Ibrido**:
  + Strategia più organizzata rispetto all'Expectimax puro.
  + Maggior velocità rispetto all'Expectimax tradizionale, ma con punteggi finali inferiori.

**Conclusioni**

* Il Q-Learning puro è inefficiente per il gioco 2048 senza supporto da strategie di ricerca.
* L'algoritmo ibrido migliora l'efficienza e la capacità di generalizzazione, ma soffre nei casi di tessere non favorevoli.

# **ROADMAP**

**1. Sviluppare l'Ambiente di Gioco**

Definire l’ambiente di gioco:

* Gli **stati** (ad esempio, la griglia 4x4).
* Le **azioni** disponibili (su, giù, sinistra, destra).
* Le **ricompense** (ad esempio, il valore delle tessere fuse).
* La logica per controllare se il gioco è terminato.

**Passaggi**:

1. Definire l'ambiente del gioco ereditando da Gym. \*
2. Implementa le funzioni chiave:
   * reset(): Per inizializzare la griglia e aggiungere tessere.
   * step(action): Per applicare un'azione e restituire (nuovo stato, ricompensa, done).
   * render(): Per visualizzare la griglia.
3. Testa l'ambiente manualmente con azioni casuali.

**2. Implementare l'Agente RL**

Dopo l'ambiente, sviluppare un agente in grado di interagire con esso.

**Passaggi**:

1. Definire una politica epsilon-greedy, ad esempio 0.3 di partenza come nel paper che decresce nel corso del training.
2. Aggiungere un **Replay Buffer**:
   * Salvare le esperienze per addestrare il modello in batch.
3. Integra una struttura base di Deep Q-Learning.

**3. Progettare la Rete Neurale**

Progettare una rete neurale per approssimare la funzione Q(s,a).

**Passaggi**:

1. Input: La griglia 4×4 (normalizzata o trasformata in un vettore 1D).
2. Output: Un valore Q(s,a) per ogni azione (4 nodi di output).
3. Compilare il modello con un ottimizzatore, una funzione di perdita (Huber o MSE), e un learning rate.

**4. Integrare Tutto con Gym**

Mettere insieme i vari pezzi: l'ambiente, l'agente e il modello per avviare l'addestramento.