

TRESSETTE AI

Un agente AI per il gioco del
Tressette



Gioco con regole complesse che richiede decisioni strategiche, ottimo per testare il Reinforcement Learning



Obiettivi principali



- Modellare le **regole** e le **dinamiche** del gioco
- **Allenare** l'agente AI a migliorare con l'esperienza
- Metterla alla prova con **diverse strategie**
- Permettere all'utente di sfidarla

Regole e Dinamiche di gioco



- Mazzo, distribuzione carte e turni
- Punteggi
- Condizioni di vittoria
- Esempi di mani giocate

```
class Trick:  
    """Rappresenta una presa (trick), parziale o completa.  
  
    Attributi:  
        - leader: seat che ha aperto la presa  
        - plays: lista di tuple (seat, card_id) in ordine di gioco  
    """  
    leader: int  
    plays: List[Tuple[int, int]]  
  
    def lead_suit(self) -> int:  
        """Seme d'uscita della presa (quello della prima carta giocata)."""  
        if not self.plays:  
            return -1  
        return id_to_card(self.plays[0][1]).suit  
  
    def is_complete(self) -> bool:  
        """True se nella presa sono state giocate 4 carte."""  
        return len(self.plays) == 4  
  
    def winner(self) -> int:  
        """Ritorna il seat del giocatore che ha vinto la presa.  
        Vince la carta più forte del seme di uscita.  
        """  
        assert self.is_complete(), "Trick incompleto"  
        suit = self.lead_suit()  
        best_seat, best_card = self.plays[0]  
        best_strength = id_to_card(best_card).strength  
        for seat, cid in self.plays[1:]:  
            c = id_to_card(cid)  
            if c.suit == suit and c.strength > best_strength:  
                best_strength = c.strength  
                best_seat = seat  
        return best_seat  
  
    def legal_actions(hand: List[int], trick: Trick) -> List[int]:  
        """Ritorna le carte legali giocabili dalla mano.  
        - Se non c'è ancora una carta nel trick: qualsiasi carta è valida.  
        - Altrimenti, se ho almeno una carta del seme di uscita: devo seguirlo.  
        - Se non ne ho: posso giocare qualsiasi carta.  
        """  
        if not trick.plays:  
            return sorted(hand)  
        lead = trick.lead_suit()  
        same_suit = [cid for cid in hand if id_to_card(cid).suit == lead]  
        return sorted(same_suit if same_suit else hand)  
  
    def score_cards_thirds(cards: List[int]) -> int:  
        """Somma i terzi di punto delle carte (A=3, 2/3/R/C/F=1, altre=0)."""  
        return sum(id_to_card(cid).thirds for cid in cards)  
  
    def score_team_from_captures(captured_cards: List[int], took_last_trick: bool) -> int:  
        """Calcola i punti di una squadra dalle carte catturate.  
  
        Regole:  
        - ogni Asso vale 1 punto, 2/3/Re/Cavallo/Fante valgono 1/3  
        - si arrotonda per difetto ai punti interi  
        - +1 punto bonus a chi prende l'ultima presa  
        """  
        thirds = score_cards_thirds(captured_cards)  
        pts = thirds // 3 # arrotondamento per difetto  
        if took_last_trick:  
            pts += 1  
        return int(pts)
```

Architettura del sistema



Schema generale:

- 🧠 Gamestate
- 🛡 Policy
- 💰 Rewards
- 🔍 Training loop

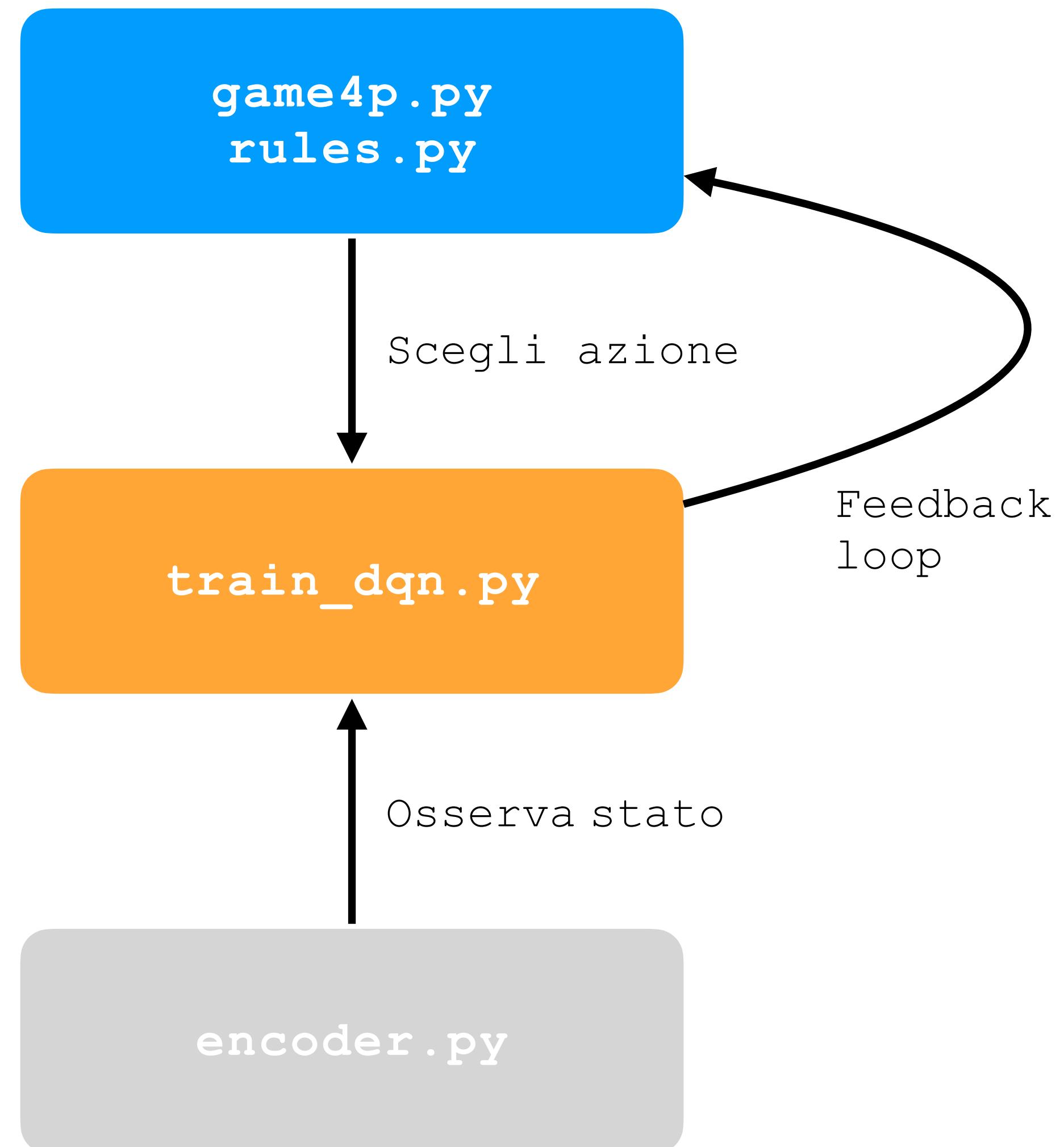
```
def step(state: GameState, card_id: int) -> Tuple[GameState, Dict[int,int], bool, Dict]:  
    """Esegue una giocata dal giocatore corrente e calcola eventuali segnali."""  
    p = state.current_player  
  
    if card_id not in state.hands[p]:  
        print("DEBUG ERRORE → seat:", p,  
              "action:", card_id,  
              "hand:", state.hands[p],  
              "plays:", state.trick.plays)  
  
    assert card_id in state.hands[p], (  
        f"[ERRORE STEP] Carta {card_id} non in mano al giocatore {p}. "  
        f"Mano attuale: {state.hands[p]}")  
    la = legal_actions(state.hands[p], state.trick)  
    assert card_id in la, "Mossa illegale: devi seguire il seme se puoi"  
  
    ns = state.clone() #Ns-> State -> Gamestate -> mano di un giocatore  
    ns.hands[p].remove(card_id)  
    ns.trick.plays.append((p, card_id))  
  
    rewards = {0: 0, 1: 0}  
    info = {}  
  
    # Se è il leader del trick → comunica il segnale  
    if len(ns.trick.plays) == 1:  
        played_card = id_to_card(card_id)  
        seme = played_card.suit  
        rest = [c for c in ns.hands[p] if id_to_card(c).suit == seme]  
  
        # logica aggiornata  
        if any(id_to_card(c).rank in ["2", "3"] for c in rest):  
            signal = "busso"  
        elif not rest:  
            signal = "volo"  
        else:  
            signal = "striscio"  
  
        ns.signals[p] = {"suit": seme, "signal": signal}  
        info["signal"] = {"seat": p, "suit": seme, "signal": signal}  
  
    # Se la presa non è ancora completa  
    if len(ns.trick.plays) < 4:  
        ns.current_player = (p + 1) % 4  
        return ns, rewards, False, info #Se non hanno giocato tutti la carte ritorna False che nel menù  
        ci darà break  
  
    # Altrimenti la presa è completa → determina vincitore  
    winner = ns.trick.winner()  
    ns.last_trick_winner = winner  
    taken = [cid for _, cid in ns.trick.plays]  
    team = TEAM_OF_SEAT[winner]  
    nscaptures_team[team].extend(taken)  
    ns.tricks_played += 1  
  
    ns.trick = Trick(winner, [])  
    ns.current_player = winner  
  
    if ns.tricks_played == 10:  
        last_team = TEAM_OF_SEAT[ns.last_trick_winner] if ns.last_trick_winner is not None else None  
        t0 = score_team_from_captures(nscaptures_team[0], last_team == 0)  
        t1 = score_team_from_captures(nscaptures_team[1], last_team == 1)  
        rewards = {0: t0, 1: t1}  
        info.update({"points": rewards, "captures": nscaptures_team, "signals": ns.signals})  
        return ns, rewards, True, info  
  
    return ns, rewards, False, info
```

Architettura del sistema



File principali

- 🎲 **game4p.py** – Logica e turni del gioco in 4 giocatori
- 📏 **rules.py** – Punteggi e regole di presa
- 🏋️ **train_dqn.py** – Addestramento agente
- 💻 **menu_cli.py** – Interfaccia per giocare
- 🧩 **encoder.py** – Codifica lo stato per la rete neurale
- 🃏 **cards.py** – Gestione delle carte
- 👀 **watch_game.py** – Visualizzazione partite



🧠 Osserva stato → 🎯 Scegli azione → ⚡ Feedback loop

Deep Q-Learning 🎮

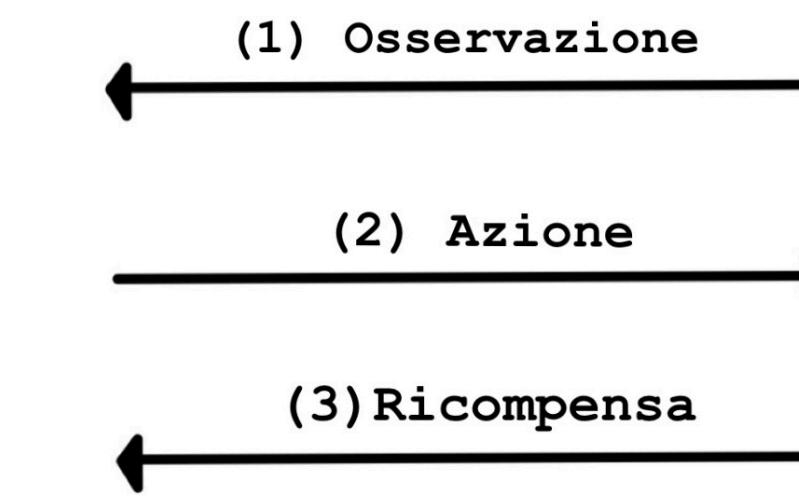
Il **Deep Q-Learning (DQN)** combina il Q-Learning con reti neurali profonde per approssimare la funzione di valore $Q(s, a)$, cioè quanto è conveniente compiere un'azione in un certo stato.

Durante l'addestramento, **l'agente esplora l'ambiente, riceve ricompense e aggiorna i pesi della rete** per migliorare le proprie decisioni.

Componenti chiave 🔑

- **Rete neurale**: stima i valori Q per tutte le azioni possibili.
- **Replay Buffer**: memorizza esperienze passate, riducendo la dipendenza temporale dei dati.
- **Target Network**: aggiornata periodicamente per evitare oscillazioni.
- **ϵ -greedy policy**: combina scelte casuali (esplorazione) e migliori azioni note (sfruttamento).
- **Reward shaping**: fornisce ricompense intermedie che accelerano l'apprendimento.

Obiettivo:
Trovare una policy ottimale che massimizzi la ricompensa cumulativa nel tempo.



Deep Q-Learning

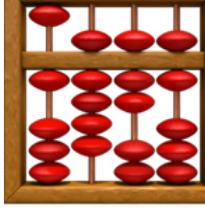
Perché?

Alternative analizzate:

- **SARSA**: aggiorna Q durante l'interazione, ma tende a sottostimare i valori e a convergere lentamente
→ poco adatto a problemi complessi.
- **Double DQN**: riduce l'overestimation dei Q-values, ma richiede due reti e un tuning più delicato
→ maggiore complessità e memoria.
- **Dueling DQN**: separa il valore dello stato dal vantaggio dell'azione, ma introduce instabilità nelle prime fasi
→ meno adatto a domini con segnali di reward deboli.
- **Actor-Critic / Policy Gradient**: ottimi per spazi continui, ma instabili e computazionalmente più costosi nei giochi a stati discreti come il Tresette.

```
if metodo == "SARSA": rifiuta("lento")
elif metodo == "Double DQN": rifiuta("troppo complesso")
elif metodo == "Dueling DQN": rifiuta("instabile")
elif metodo == "Actor-Critic": rifiuta("eccessivo per domini discreti")
else:
    metodo = "Deep Q-Learning" accetta("semplice, stabile, efficace")
```

Deep Q-Learning



Policy

- **Predizione Q-Values**

La rete valuta le azioni possibili nello stato corrente

- **Policy ϵ -greedy**

L'agente sceglie tra esplorazione casuale e sfruttamento della rete

- **Replay buffer**

Le esperienze passate vengono salvate per aggiornare la rete



```
# Forward pass: la rete predice i Q-values per tutte le azioni possibili  
q_values = policy(state, mask)
```



```
if random.random() < eps:  
    action = random.choice(legal_moves) # esplorazione  
else:  
    action = torch.argmax(q_values[legal_moves]) # sfruttamento
```



```
# Salva esperienza per il training  
replay.push(state, mask, action, reward, next_state, next_mask, done)
```

Deep Q-Learning



Q-Values

Segnali intermedi per ogni azione utile → **Q-Values** aggiornati più frequentemente.

Penalità per carte giocate male, bonus per chiusura trick e cattura carte chiave.

```
def compute_reward(state, next_state, seat, action, prev_captures):
    r = 0.0
    card = id_to_card(action)

    # penalizza carte forti giocate presto
    if card.rank == "A":
        r -= 0.2
    elif card.rank in ["2", "3"]:
        r -= 0.1
    else:
        r += 0.05

    # reward per chiusura trick
    trick_closed = next_state.tricks_played > state.tricks_played
    if trick_closed:
        my_team = TEAM_OF_SEAT[seat]

        # se ho preso il trick → +0.5, altrimenti -0.5
        if len(next_state.captures_team[my_team]) >
           len(state.captures_team[my_team]):
            r += 0.5
        else:
            r -= 0.5

    # bonus se ho catturato 2, 3 o Asso
    captured = set(next_state.captures_team[my_team]) -
               set(prev_captures[my_team])
    for cid in captured:
        c = id_to_card(cid)
        if c.rank in ["2", "3", "A"]:
            r += 0.3

    return r
```

Deep Q-Learning



Euristica

- Segui segnali del compagno
- Scegli il seme più forte
- Cambiare seme se non ricevi alcun segnale
- Assicurarsi l'ultima presa

```
def choose_action(state, legal_idx):  
    """Sceglie un'azione euristica dato lo stato e gli id legali delle carte."""  
    legal_cards = [id_to_card(cid) for cid in legal_idx]  
  
    # =====  
    # 1. Assecondare dichiarazioni del compagno  
    # =====  
    for seat_signal, sig in state.signals.items():  
        suit_signal = sig["suit"]  
        signal_type = sig["signal"]  
        if signal_type in ("busso", "striscio"):  
            for cid_signal, card_signal in zip(legal_idx, legal_cards):  
                if card_signal.suit == suit_signal:  
                    return cid_signal  
        elif signal_type == "volo":  
            continue  
  
    # =====  
    # 2. Palo più forte con gestione dell'asso  
    # =====  
    suits_in_hand = [card_suit.suit for card_suit in legal_cards]  
    strongest_suit = max(  
        set(suits_in_hand),  
        key=lambda s: sum(RANK_TO_STRENGTH[card_suit.rank]  
                           for card_suit in legal_cards if card_suit.suit == s)  
    )  
    for cid_strong, card_strong in zip(legal_idx, legal_cards):  
        if card_strong.suit != strongest_suit:  
            continue  
        hand_ranks_strong = [id_to_card(cid_hand).rank for cid_hand in state.hands[state.current_player]]  
        if card_strong.rank == 'A':  
            if '2' in hand_ranks_strong and '3' in hand_ranks_strong:  
                return cid_strong  
            else:  
                continue  
        else:  
            return cid_strong  
  
    # =====  
    # 3. Giocare 2 se hai esattamente due carte di quel seme  
    # =====  
    for cid_two, card_two in zip(legal_idx, legal_cards):  
        same_suit_cards_two = [c for c in legal_cards if c.suit == card_two.suit]  
        if card_two.rank == '2' and len(same_suit_cards_two) == 2:  
            return cid_two  
  
    # =====  
    # 4. Cambiare sempre gioco se nessun segnale  
    # =====  
    for cid_change, card_change in zip(legal_idx, legal_cards):  
        return cid_change # fallback semplice  
  
    # =====  
    # 5. Assicurarsi l'ultima presa (gioca carta più alta)  
    # =====  
    return max(legal_idx, key=lambda cid_max: RANK_TO_STRENGTH[id_to_card(cid_max).rank])
```

Training



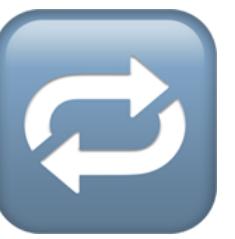
Inizializzazione e loop episodi

- Inizializza **reti policy** e target
- Crea **replay buffer** per memorizzare esperienza
- Loop su migliaia di episodi e mani
- Possibilità di riprendere da un **checkpoint**



```
def train(resume_from: str | None = None):  
    in_dim = feature_dim()  
    policy = DQNNet(in_dim).to(DEVICE)  
    target = DQNNet(in_dim).to(DEVICE)  
    target.load_state_dict(policy.state_dict())  
    target.eval()  
  
    opt = optim.Adam(policy.parameters(), lr=LR)  
    rb = Replay(REPLAY_CAP)  
    opt_steps = 0  
    start_ep = 1  
  
    reward_history = []  
    total_tricks = 0  
    total_hands = 0  
  
    for ep in range(start_ep, EPISODES + 1):  
        state = deal(leader=ep % 4)  
        void_flags = [[0]*4 for _ in range(4)]  
        reward_log = []  
        done = False  
  
        while not done:  
            seat = state.current_player  
            x, mask = encode_state(state, seat, void_flags)  
            x, mask = x.to(DEVICE), mask.to(DEVICE)  
  
            eps = epsilon(opt_steps)  
            legal_idx = torch.nonzero(mask[0]).view(-1).tolist()  
            action = legal_idx[0] if legal_idx else 0  
            ...
```

Training



Scelta azione

- **Fase casuale**
Esplorazione libera
- **Fase euristica**
Decisioni guidate da regole semplici
- **Fase DQN**
 ϵ -greedy con Q-values della rete

```
...  
if ep < 1000:  
    if random.random() < 0.7:  
        action = HeuristicAgent.choose_action(state, legal_idx)  
    else:  
        action = random.choice(legal_idx)  
elif ep < 600000:  
    action = HeuristicAgent.choose_action(state, legal_idx)  
else:  
    if random.random() < eps:  
        action = random.choice(legal_idx)  
    else:  
        with torch.no_grad():  
            q = policy(x, mask).squeeze(0)  
            q_legal = q[legal_idx]  
            best_idx = torch.argmax(q_legal).item()  
            action = legal_idx[best_idx]  
  
    if action not in legal_idx:  
        print(f"[WARNING] Azione {action} non valida per giocatore {seat}.  
Legal idx: {legal_idx}")  
        action = random.choice(legal_idx)  
  
    update_void_flags(void_flags, state, seat, action)  
    prev_captures = {0: list(state.captures_team[0]), 1:  
list(state.captures_team[1])}  
    next_state, rewards, done, _ = step(state, action)  
    r_shape = compute_reward(state, next_state, seat, action, prev_captures)  
...
```

Training 💪

Replay buffer e aggiornamento della rete

- Salva ogni esperienza nel **replay buffer**
- Campiona batch per aggiornare **Q-values**
- Calcola loss e ottimizza la rete
- Target network aggiornato periodicamente

```
...
x_next, mask_next = encode_state(next_state, seat, void_flags)
x_next, mask_next = x_next.to(DEVICE), mask_next.to(DEVICE)
rb.push(x, mask, action, r, x_next, mask_next, float(done))
state = next_state

if len(rb) >= BATCH_SIZE:
    s, m, a, r_b, s2, m2, d = rb.sample(BATCH_SIZE)
    with amp_autocast():
        q = policy(s, m).gather(1, a)
        with torch.no_grad():
            q2 = target(s2, m2).max(dim=1, keepdim=True)[0]
            y = r_b + (1.0 - d) * GAMMA * q2
            loss = F.mse_loss(q, y)

    opt.zero_grad()
    scaler.scale(loss).backward()
    scaler.unscale_(opt)
    nn.utils.clip_grad_norm_(policy.parameters(), 1.0)
    scaler.step(opt)
    scaler.update()
    opt_steps += 1

...
final_model = f"dqn_tressette_ep{EPISODES}.pt"
torch.save({
    "model": policy.state_dict(),
    "config": {"in_dim": in_dim, "hidden": 256}
}, final_model)
print(f"Training completato. Modello finale salvato: {final_model}")
```



```
#💡 Scelta dei parametri del DQN

EPISODES = 700_000          # Addestramento lungo → convergenza stabile e ampia esplorazione

GAMMA = 0.99                # Premia strategie a lungo termine, non solo prese immediate

LR = 3e-4                   # Learning rate basso → aggiornamenti graduali e stabili

BATCH_SIZE = 512            # Usa esperienze varie per gradienti più affidabili

REPLAY_CAP = 1_000_000       # Mantiene memoria ampia → evita overfitting su situazioni ripetute

TARGET_SYNC = 2000           # Aggiorna lentamente la rete target → maggiore stabilità

EPS_START = 1.0              # All'inizio esplora casualmente (fase di scoperta)

EPS_END = 0.05               # Alla fine sfrutta le mosse apprese (fase di sfruttamento)

EPS_DECAY_STEPS = 200_000    # Passaggio graduale tra esplorazione e sfruttamento

CHECKPOINT_EVERY = 15_000   # Salvataggio periodico per allenamenti lunghi su GPU
```

Simulazione match



```
def play_one_game(verbose=True, seed=None):
    if seed is None:
        seed = int(time.time() * 1e6) % (2**32 - 1)
    rng = random.Random(seed)

    state = deal(leader=rng.randint(0, 3), rng=rng)
    void_flags = [[0]*4 for _ in range(4)]
    done = False

    if verbose:
        print(f"[DEBUG] Seed partita: {seed}")

    while not done:
        seat = state.current_player
        action = choose_action(seat, state, void_flags)
        if verbose:
            print(f"Giocatore {seat} ({PLAYERS[seat]}) gioca {id_to_card(action)})")

        update_void_flags(void_flags, state, seat, action)
        state, rewards, done, _ = step(state, action)

    if verbose:
        print("== Fine partita ==")
        print(f"Team 0 (seat 0+2) → {rewards[0]} punti")
        print(f"Team 1 (seat 1+3) → {rewards[1]} punti")
        if rewards[0] > rewards[1]:
            print("Vince TEAM 0")
        elif rewards[1] > rewards[0]:
            print("Vince TEAM 1")
        else:
            print("Pareggio!")

    return rewards
```

watch_game.py



```
def play_many_games(n_matches=N_MATCHES, batch_size=BATCH_SIZE):
    wins_team0 = wins_team1 = draws = 0
    all_seeds = [int(time.time() * 1e6) % (2**32 - 1) + i for i in range(n_matches)]

    for start_idx in tqdm(range(0, n_matches, batch_size), desc="🎮 Playing"):
        batch_seeds = all_seeds[start_idx:start_idx + batch_size]
        states = [deal(leader=random.randint(0, 3), rng=random.Random(s)) for s in batch_seeds]
        void_flags_batch = [[[0]*4 for _ in range(4)] for _ in states]
        done_flags = [False] * len(states)
        rewards_batch = [None] * len(states)

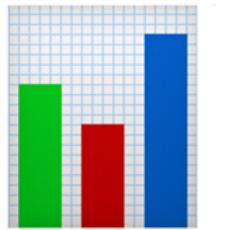
        while not all(done_flags):
            active_idx = [i for i, d in enumerate(done_flags) if not d]
            current_states = [states[i] for i in active_idx]
            current_voids = [void_flags_batch[i] for i in active_idx]
            actions = choose_action_batch(current_states, current_voids)

            for idx, act in zip(active_idx, actions):
                seat = states[idx].current_player
                update_void_flags(void_flags_batch[idx], states[idx], seat, act)
                states[idx], rewards, done, _ = step(states[idx], act)
                if done:
                    rewards_batch[idx] = rewards
                    done_flags[idx] = True

        for rewards in rewards_batch:
            if rewards is None:
                continue
            if int(rewards[0]) > int(rewards[1]):
                wins_team0 += 1
            elif int(rewards[1]) > int(rewards[0]):
                wins_team1 += 1
            else:
                draws += 1
```

watch_game_parallel.py

Risultati



Heuristic vs Random

```
[(venv) raffaele.neri2@slurmng-n4:/scratch.hpc/raffaele.neri2/Tresette_AI$ python watch_game_parallel.py
CUDA disponibile: True
Nome GPU: NVIDIA L40
Memoria allocata: 0.0 MB
Memoria riservata: 0.0 MB
✓ Modello caricato da dqn_tresette_checkpoint_ep690000.pt
GPU Device attivo: cuda
== TORNEO PARALLELO TRESSETTE ==
🎮 Playing: 100%|██████████| 512/512 [01:29<00:00, 5.74it/s]

== RISULTATI FINALI ==
Team0 (seat 0+2: heuristic + heuristic) → 8220/16384
Team1 (seat 1+3: random + random) → 8164/16384]
```

DQN vs Random

```
[(venv) raffaele.neri2@slurmng-n4:/scratch.hpc/raffaele.neri2/Tresette_AI$ python watch_game_parallel.py
CUDA disponibile: True
Nome GPU: NVIDIA L40
Memoria allocata: 0.0 MB
Memoria riservata: 0.0 MB
✓ Modello caricato da dqn_tresette_checkpoint_ep690000.pt
GPU Device attivo: cuda
== TORNEO PARALLELO TRESSETTE ==
🎮 Playing: 100%|██████████| 512/512 [01:29<00:00, 5.69it/s]

== RISULTATI FINALI ==
Team0 (seat 0+2: dqn + dqn) → 8280/16384
Team1 (seat 1+3: random + random) → 8104/16384]
```

DQN vs Heuristic

```
[(venv) raffaele.neri2@slurmng-n4:/scratch.hpc/raffaele.neri2/Tresette_AI$ python watch_game_parallel.py
CUDA disponibile: True
Nome GPU: NVIDIA L40
Memoria allocata: 0.0 MB
Memoria riservata: 0.0 MB
✓ Modello caricato da dqn_tresette_checkpoint_ep690000.pt
GPU Device attivo: cuda
== TORNEO PARALLELO TRESSETTE ==
🎮 Playing: 100%|██████████| 512/512 [01:28<00:00, 5.79it/s]

== RISULTATI FINALI ==
Team0 (seat 0+2: dqn + dqn) → 8279/16384
Team1 (seat 1+3: heuristic + heuristic) → 8105/16384]
```



Interpretazione dei Risultati



Euristica molto forte

- L'agente *Heuristic* segue regole esperte e copre gran parte delle strategie ottimali.
- In un gioco strutturato come il **Tressette**, le decisioni “buone” sono spesso già determinate dalle regole.



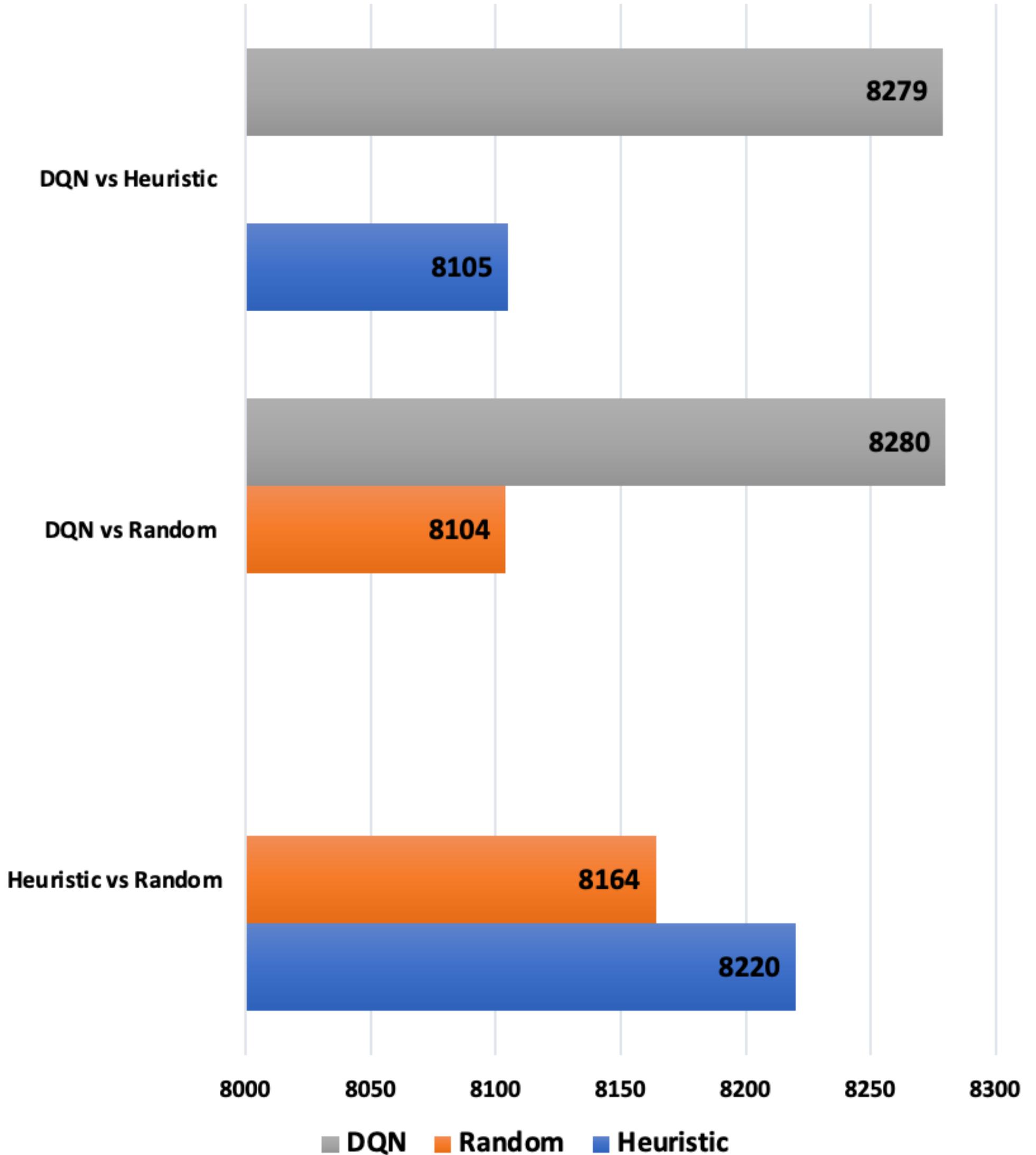
Random sorprendentemente competitivo

- Il gioco presenta **molti vincoli** (carte disponibili, turni, prese obbligate).
- Anche un agente casuale finisce spesso per compiere mosse accettabili → **ridotto margine di errore**.



DQN leggermente superiore

- Il modello apprende comportamenti coerenti e migliora le scelte, ma il vantaggio resta contenuto perché lo spazio decisionale è limitato.



GRAZIE PER L'ATTENZIONE

