🎯 GRAZIE PER LA REVIEW RIGOROSA!

Hai perfettamente ragione su tutti i punti. Applico le correzioni per rendere il tutto scientificamente solido.

📚 SPIEGAZIONE COMPLETA - VERSIONE CORRETTA

🎯 PARTE 1: MODELLAZIONE DELL'AMBIENTE

1.1 Environment Originale: Speaker-Listener

Cos'è:

Un ambiente cooperativo di comunicazione multi-agente da PettingZoo (MPE - Multi-Particle Environment).

Setup:

```
Agenti:
- Speaker: Vede quale dei 3 landmark è il "goal", deve comunicarlo
- Listener: Vede i 3 landmark ma NON sa quale è il goal, deve raggiungerlo

Osservazioni:
- Speaker: quale landmark è goal (3 possibilità)
- Listener: posizioni dei landmark, sua posizione/velocità (continuo)

Azioni:
- Speaker: 3 comunicazioni discrete (one-hot message)
- Listener: 5 azioni movimento (no-op, up, down, left, right)

Reward (cooperativo):
- Negativo della distanza listener → goal
- Condiviso da entrambi
- Range: [-10, 0] circa

Dinamiche:
- Episodi: max 25 timesteps
- Spazio continuo 2D
- Fisica semplice (velocità, posizione)
```

Problema:

```
Stato continuo (posizioni, velocità)
→ Spazio infinito
→ Algoritmi tabulari non applicabili direttamente
→ Nash equilibrium nel dominio continuo non computabile in forma chiusa
```

1.2 Discretizzazione: DiscretizedSpeakerListenerWrapper

Obiettivo:

Trasformare ambiente continuo in discreto per:

1. Rendere il problema trattabile in forma tabulare
2. Permettere calcolo di Nash equilibrium approssimato
3. Calcolare exploitability in modo esatto sul gioco discretizzato
4. Fornire baseline confrontabile per algoritmi diversi

Come funziona:

```
class DiscretizedSpeakerListenerWrapper:
    def __init__(self, env, bins=6):
        self.bins = bins  # Discretizzazione
```

Discretizzazione Listener State:

Input continuo:

```
Posizione listener: (x, y) ∈ [-1, 1] × [-1, 1]
Velocità listener: (vx, vy) ∈ [-5, 5] × [-5, 5] (circa)
Goal landmark: {0, 1, 2}
```

Discretizzazione:

```
# Posizione → bins × bins grid
x_discrete = clip(floor((x + 1) / 2 * bins), 0, bins-1)
y_discrete = clip(floor((y + 1) / 2 * bins), 0, bins-1)
pos_idx = x_discrete * bins + y_discrete  # 0 to bins²-1

# Velocità → bins × bins grid
vx_discrete = clip(floor((vx + 5) / 10 * bins), 0, bins-1)
vy_discrete = clip(floor((vy + 5) / 10 * bins), 0, bins-1)
vel_idx = vx_discrete * bins + vy_discrete

# Goal landmark
goal_idx = goal  # 0, 1, or 2

# Combined state
listener_state = goal_idx * (bins² * bins²) + pos_idx * bins² + vel_idx
```

Numero stati listener:

```
3 goals × (6² positions) × (6² velocities) = 3 × 36 × 36 = 3888 stati teorici

Osservazione empirica: ~972 stati effettivamente visitabili
(molti stati posizione-velocità sono fisicamente inconsistenti)
```

Speaker State:

```
speaker_state = goal_idx  # Semplicemente 0, 1, o 2
```

Totale stati:

```
S_speaker = 3
S_listener = 972 (raggiungibili)
S_joint = 3 × 972 = 2916 stati globali
```

Perché bins=6?

Trade-off empirico:

```
bins=3: Discretizzazione troppo grossolana, perdita informazione significativa
bins=6: Bilanciamento ragionevole risoluzione/computazione ✅
bins=10: Esplosione computazionale, benefici marginali

Scelta bins=6 basata su:
- ✅ Discretizzazione sufficientemente fine
- ✅ 2916 stati gestibili computazionalmente
- ✅ Nash solver converge in tempo ragionevole (~30 min)
- ✅ Exploitability calcolabile
```

1.3 MDP/Stochastic Game Formulation

Formalizzazione Matematica:

Stochastic Game a 2 giocatori cooperativo:

```
G = (S, {A₁, A₂}, P, R, γ, d₀)

Dove:
S: Spazio stati discreto (2916 stati joint)
A₁: Azioni speaker (3)
A₂: Azioni listener (5)
P: Funzione transizione P(s'|s,a₁,a₂)
R: Funzione reward R(s,a₁,a₂) (condivisa - gioco cooperativo)
γ: Discount factor (0.9)
d₀: Distribuzione iniziale stati
```

Computazione P e R:

R(s, a_speaker, a_listener):

```
# Esegui azione nell'env discretizzato
obs, rewards, done, _ = env.step({
    "speaker_0": a_speaker,
    "listener_0": a_listener
})

R[s, a_speaker, a_listener] = rewards["speaker_0"]
```

P(s'|s, a_speaker, a_listener):

```
# Stima empirica delle transizioni:
# 1. Da ogni stato s, esegui ogni coppia (a_speaker, a_listener) N volte
# 2. Conta frequenze stato successivo s'
# 3. P(s'|s,a) = count(s,a→s') / N

# Nota: Stochasticità bassa nell'environment
# → Transizioni approssimativamente deterministiche
```

File generati:

```
P_bins6.npy: (2916, 3, 5, 2916) float64 - 486.5 MB
R_bins6.npy: (2916, 3, 5) float64 - 0.2 MB
expert_initial_dist_bins6.npy: (2916,) float64 - distribuzione iniziale
```

🧠 PARTE 2: EXPERT GENERATION (NASH EQUILIBRIUM)

2.1 Perché serve l'Expert?

MURMAIL = Model-Free Multiagent Imitation Learning

Idea core del paper (Ramponi et al., NeurIPS 2023):

```
Assume access to expert Nash equilibrium policy π∗
Query π∗ on-demand per stati visitati durante training
Learn policy approssimata via Q-learning guidato dall'expert
```

Richiede:

1. Expert policy π* (approssimazione di Nash equilibrium)
2. Meccanismo per query π*(s) on-demand
3. Metrica exploitability per valutare gap da Nash

2.2 Nash Equilibrium: Cosa Significa?

Definizione:

Un Nash equilibrium è una coppia di policy (π_1^*, π_2^*) tale che nessun agente può migliorare unilateralmente deviando.

Formalmente:

$V^{\pi} * (d_0) \geq V^{(\pi_1', \pi_2 *)}(d_0)\ \forall \pi_1'$ (speaker non migliora deviando)
$V^{\pi} * (d_0) \geq V^{(\pi_1 *, \pi_2')}(d_0)\ \forall \pi_2'$ (listener non migliora deviando)

dove $V^{\pi}(d_0) = \mathbb{E}_{s \sim d_0}[V^{\pi}(s)]$

Exploitability:

Misura la deviazione da Nash equilibrium:

$\varepsilon(\pi_1, \pi_2) = \varepsilon_{speaker} + \varepsilon_{listener}$

Dove:
$\varepsilon_{speaker} = \max_{\pi_1'} V^{(\pi_1', \pi_2)}(d_0) - V^{(\pi_1, \pi_2)}(d_0)$
$\varepsilon_{listener} = \max_{\pi_2'} V^{(\pi_1, \pi_2')}(d_0) - V^{(\pi_1, \pi_2)}(d_0)$
Nash equilibrium $\iff \varepsilon = 0$

2.3 Fictitious Play: Algoritmo per Nash

Per generare la policy esperta nel progetto, è stata utilizzata una variante del **Fictitious Play Classico (Simultaneo)** adattata per Giochi Markoviani (Stochastic Games).

A differenza di approcci avidi come la *Iterated Best Response* (che possono portare a cicli infiniti), il Fictitious Play garantisce una maggiore stabilità e la convergenza all'Equilibrio di Nash in giochi cooperativi, poiché ogni agente ottimizza la propria strategia contro la **media storica** del comportamento dell'avversario.

Caratteristiche Chiave

1. **Memoria Storica (Average Strategy):** Gli agenti non reagiscono all'ultima mossa dell'avversario, ma alla distribuzione di probabilità media accumulata dall'inizio del training. Questo processo di "smoothing" smorza le oscillazioni.
2. **Best Response "Profonda" (Model-Based):** Essendo l'ambiente sequenziale (Markoviano), il calcolo della "Miglior Risposta" non è istantaneo. L'agente costruisce un *MDP indotto* fissando la policy media dell'avversario e lo risolve completamente tramite **Value Iteration**.
3. **Aggiornamento Simultaneo:** Entrambi gli agenti (Speaker e Listener) calcolano le nuove Best Response contemporaneamente basandosi sulle medie al tempo t, e successivamente aggiornano le proprie medie per il tempo $t+1$.

Pseudocodice dell'Algoritmo

Input:

- Modello dell'ambiente: Matrice di Transizione $P(s'|s, a_{spk}, a_{lst})$ e Ricompensa $R(s, a_{spk}, a_{lst})$
- Fattore di sconto γ
- Iterazioni totali N

- Iterazioni per Value Iteration K

Inizializzazione:

- $\bar{\pi}_{spk} \leftarrow$ Uniforme
- $\bar{\pi}_{lst} \leftarrow$ Uniforme
- $t \leftarrow 1$

Loop Principale (fino a N):

1. **Calcolo Best Response Speaker (π^*_{spk})**
   - L'agente Speaker marginalizza il modello rispetto alla policy media del Listener:

$$P_{spk}(s'|s, a_{spk}) = \sum_{a_{lst}} P(s'|s, a_{spk}, a_{lst}) \cdot \bar{\pi}_{lst}(a_{lst}|s)$$
$$R_{spk}(s, a_{spk}) = \sum_{a_{lst}} R(s, a_{spk}, a_{lst}) \cdot \bar{\pi}_{lst}(a_{lst}|s)$$

   - Risolve l'MDP indotto (P_{spk}, R_{spk}) tramite **Value Iteration** (per K passi) trovando la policy ottima deterministica π^*_{spk}.
2. **Calcolo Best Response Listener (π^*_{lst})**
   - Simultaneamente, il Listener marginalizza il modello rispetto alla policy media dello Speaker:

$$P_{lst}(s'|s, a_{lst}) = \sum_{a_{spk}} P(s'|s, a_{spk}, a_{lst}) \cdot \bar{\pi}_{spk}(a_{spk}|s)$$

   - Risolve l'MDP indotto tramite **Value Iteration** trovando la policy ottima deterministica π^*_{lst}.
3. **Aggiornamento Strategie Medie (Fictitious Play Update)**
   - Si calcola il tasso di apprendimento: $\alpha_t = \frac{1}{t}$
   - Si aggiornano le policy medie integrando la nuova Best Response:

$$\bar{\pi}^{(t+1)}_{spk} = (1 - \alpha_t) \cdot \bar{\pi}^{(t)}_{spk} + \alpha_t \cdot \pi^*_{spk}$$
$$\bar{\pi}^{(t+1)}_{lst} = (1 - \alpha_t) \cdot \bar{\pi}^{(t)}_{lst} + \alpha_t \cdot \pi^*_{lst}$$

   - $t \leftarrow t + 1$

Output:

- Le policy finali restituite sono le medie storiche $\bar{\pi}_{spk}$ e $\bar{\pi}_{lst}$.

Nota sulla Convergenza

L'uso del fattore di aggiornamento $\alpha_t = 1/t$ implica che, matematicamente, la policy al tempo T è la media aritmetica esatta di tutte le Best Response passate:

$$\bar{\pi}_T = \frac{1}{T} \sum_{i=1}^{T} \pi_{BR,i}^*$$

Questo meccanismo costringe l'agente a performare bene contro l'intero spettro di comportamenti mostrati dall'avversario nel tempo, garantendo la robustezza necessaria per risolvere il gioco cooperativo.

Value Iteration per Best Response:

All'interno dell'algoritmo di Fictitious Play, la **Value Iteration (VI)** viene utilizzata come subroutine fondamentale per calcolare la "Best Response" (Miglior Risposta).

Quando un agente (es. Speaker) fissa la strategia dell'altro agente (es. Listener) alla sua media storica, il gioco multi-agente si riduce temporaneamente a un **Markov Decision Process (MDP) a singolo agente**. Questo MDP "indotto" viene risolto tramite Value Iteration per trovare la policy ottima deterministica contro quella specifica strategia avversaria.

Caratteristiche dell'Implementazione

1. **Vettorizzazione (Numpy):** L'algoritmo non itera sui singoli stati tramite loop nidificati (che sarebbero computazionalmente lenti), ma sfrutta operazioni matriciali ottimizzate (`np.dot` e broadcasting) per aggiornare i valori di tutti gli stati simultaneamente.
2. **Criterio di Arresto Doppio:** L'algoritmo termina quando si verifica una delle due condizioni:
   - La differenza massima tra i valori di due iterazioni consecutive (norma infinito $||V_{new} - V_{old}||_\infty$) scende sotto una soglia di tolleranza ($\epsilon = 10^{-6}$).
   - Viene raggiunto il numero massimo di iterazioni preimpostato (K).
3. **Calcolo Q-Value:** Oltre alla Value Function $V(s)$, l'algoritmo calcola esplicitamente la matrice $Q(s, a)$, necessaria al termine del processo per estrarre la policy deterministica ($\arg\max$).

Pseudocodice dell'Algoritmo

Input:

- Matrice di Transizione Indotta: $P_{ind} \in \mathbb{R}^{|A| \times |S| \times |S|}$

- Matrice di Ricompensa Indotta: $R_{ind} \in \mathbb{R}^{|S| \times |A|}$
- Fattore di sconto: γ
- Numero massimo iterazioni: K
- Soglia di convergenza: $\epsilon = 10^{-6}$

Inizializzazione:

- $V(s) \leftarrow 0$ per ogni stato $s \in S$
- $k \leftarrow 0$

Loop Principale:
Mentre $k < K$:

1. **Salvataggio Stato Precedente:**
   $V_{old} \leftarrow V$
2. **Bellman Update (Calcolo Q-Values):**
   Per ogni azione a possibile, si calcola il valore Q vettorizzato per tutti gli stati:

$$Q(s, a) = R_{ind}(s, a) + \gamma \sum_{s' \in S} P_{ind}(s'|s, a) \cdot V_{old}(s')$$

   (Nota: Nel codice, questo corrisponde all'operazione `R[:, a] + gamma * dot(P[a], V_old)` *)*
3. **Policy Improvement (Aggiornamento V):**
   Si aggiorna la funzione valore scegliendo l'azione che massimizza il ritorno atteso per ogni stato (comportamento Greedy):

$$V(s) = \max_a Q(s, a)$$

4. **Controllo Convergenza:**
   Si calcola la variazione massima (Delta):

$$\Delta = \max_s |V(s) - V_{old}(s)|$$

   Se $\Delta < \epsilon$, interrompi il ciclo.
5. $k \leftarrow k + 1$

Output:

- Funzione Valore $V(s)$

- Funzione Azione-Valore $Q(s, a)$ (utilizzata per determinare la Best Response finale: $\pi^*(s) = \arg\max_a Q(s, a)$)

Nota sul contesto "Indotto"

È fondamentale notare che le matrici P_{ind} e R_{ind} passate a questo algoritmo non sono le matrici grezze dell'ambiente multi-agente, ma sono state **marginalizzate** rispetto alla policy media dell'avversario ($\bar{\pi}_{opp}$) prima della chiamata:

$$P_{ind}(s'|s, a) = \sum_{a_{opp}} P_{env}(s'|s, a, a_{opp}) \cdot \bar{\pi}_{opp}(a_{opp}|s)$$

Questo passaggio trasforma l'incertezza strategica (cosa farà l'avversario?) in incertezza ambientale stocastica, permettendo alla Value Iteration standard di risolvere il problema in modo ottimale.

Risultati:

```
Iteration    Exploitability
---------    --------------
0            0.244          (baseline uniforme)
100          0.156
500          0.045
1000         0.021
2000         0.013          ✅ Convergenza empirica

Gap finale: 0.013 (near-Nash, non esattamente 0)
```

Policy salvate:

```
expert_policy_speaker_bins6.npy: (3, 3) - π_speaker(a|s)
expert_policy_listener_bins6.npy: (972, 5) - π_listener(a|s)
```

2.4 Exploitability Calculation

La funzione `calc_exploitability_true` costituisce il **metodo di validazione matematica** del progetto. Utilizzando le matrici reali dell'ambiente (e , considerate "Ground Truth"), questa funzione

misura oggettivamente la distanza delle policy apprese dall'Equilibrio di Nash.

Il Concetto Teorico

In Teoria dei Giochi, una coppia di strategie costituisce un **Equilibrio di Nash** se nessun agente ha un incentivo a deviare unilateralmente dalla propria strategia.

L'**Exploitability** (o *Nash Gap*) quantifica il "rimpianto" (*regret*) massimo dei giocatori, ovvero quanto guadagno addizionale potrebbero ottenere se passassero a una strategia perfetta (Best Response) mantenendo fissa quella dell'avversario.

- **Gap** ~ 0**:** La policy è ottima (Equilibrio di Nash). Nessuno può migliorare il proprio risultato deviando.
- **Gap** > 0**:** La policy è sub-ottimale. Il valore indica quanto l'agente sta "lasciando sul tavolo" in termini di ricompensa attesa.

Funzionamento dell'Algoritmo

La funzione esegue un confronto tra il rendimento attuale degli agenti e il rendimento teorico massimo calcolato tramite **Value Iteration**. Il codice implementa una logica per giochi a somma zero (o valutazione di robustezza *worst-case*), calcolando la Best Response sia del massimizzatore (μ, Speaker) che del minimizzatore (ν, Listener).

Pseudocodice e Logica

Input:

- Policy correnti: μ (Speaker), ν (Listener)
- Modello "Ground Truth": Transizioni e Reward

Passo 1: Calcolo Valore Attuale (V_{joint})
Si calcola il Valore Atteso generato dalle due policy correnti che giocano l'una contro l'altra. Questo è il punto di riferimento (Baseline).

$$V_{joint} = \mathbb{E}_{\mu,\nu}[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, b_t)]$$

Passo 2: Calcolo Best Response dello Speaker ($V^*_{br,\mu}$)
Si ipotizza che il Listener mantenga la sua strategia ν fissa. Lo Speaker cerca la deviazione ottima per massimizzare il proprio guadagno.

1. **Marginalizzazione:** Si costruisce un MDP indotto per lo Speaker integrando la policy del Listener nelle dinamiche ambientali:

$$P_\mu(s'|s,a) = \sum_b P(s'|s,a,b) \cdot \nu(b|s)$$
$$R_\mu(s,a) = \sum_b R(s,a,b) \cdot \nu(b|s)$$

2. **Risoluzione (Value Iteration):** Si risolve l'MDP per trovare il massimo valore teorico ottenibile:

$$V^*_{br,\mu} = \max_{\pi'} V(\pi', \nu)$$

Passo 3: Calcolo Best Response del Listener ($V_{br,\nu}$)
Si ipotizza che lo Speaker mantenga la sua strategia μ fissa. Il Listener cerca la deviazione ottima. Nel contesto del codice, i reward vengono negati per trattare il problema come un gioco a somma zero (il Listener cerca di minimizzare il guadagno dello Speaker, o massimizzare la propria "vittoria" in un contesto competitivo).

1. **Marginalizzazione (con Reward Negato):**

$$P_\nu(s'|s,a) = \sum_b P(s'|s,a,b) \cdot \mu(b|s)$$
$$R_\nu(s,a) = -\sum_b R(s,a,b) \cdot \mu(b|s)$$

2. **Risoluzione (Value Iteration):** Si risolve l'MDP :

$$V^*_{br,\nu} = \max_{\pi'} V(\pi', \mu)$$

Passo 4: Calcolo del Gap
L'Exploitability è definita come il massimo guadagno possibile ottenibile deviando rispetto al valore attuale:

$$\text{Gap} = \max((V^*_{br,\mu} - V_{joint}), (V^*_{br,\nu} - V_{joint}))$$

(Nota: Il codice include un clamp a 0.0 per gestire eventuali errori numerici di virgola mobile).

Ruolo nel Progetto

Questa funzione agisce come **Giudice Imparziale ("Ground Truth Metric")**:

1. **Validazione dell'Esperto:** Durante la generazione dell'esperto (via Fictitious Play), questa funzione conferma se l'algoritmo ha convergiuto. Un valore basso (es.) certifica matematicamente che l'esperto è valido.
2. **Benchmark per Algoritmi RL:** Permette di confrontare algoritmi di apprendimento (come MURMAIL o MAPPO) rispetto all'ottimo teorico assoluto, fornendo una misura precisa della qualità della policy appresa che va oltre il semplice confronto dei reward medi campionati.

calc_true_exploitability (per policy joint correlate):

Quando si utilizzano algoritmi che apprendono policy congiunte correlate (come joint DQN come abbiamo usato), il calcolo dell'Exploitability standard via Value Iteration non è corretto, poiché la marginalizzazione distruggerebbe le informazioni di correlazione (es. segnali di coordinamento).

Infatti si consideri il seguente esempio:
Immagina un Semaforo:

- Se è Verde (50%), Speaker va e Listener si ferma.
- Se è Rosso (50%), Speaker si ferma e Listener va.

In questo caso, le azioni sono perfettamente coordinate.
L'Errore della marginalizzazione: se provi a usare la Value Iteration su questo caso, devi prima separare le policy.

- Speaker: "50% vado, 50% mi fermo" (sembra casuale!).
- Listener: "50% mi fermo, 50% vado" (sembra casuale!).

Risultato: La Value Iteration penserà che gli agenti siano stupidi e scoordinati, calcolando un valore sbagliato. Si perde l'informazione che "quando io vado, tu ti fermi SICURAMENTE".

Per calcolare correttamente il Nash Gap in questo contesto, utilizziamo un approccio basato sulla **Programmazione Lineare (Linear Programming)** nello spazio delle misure di occupazione.

Formulazione Matematica

L'obiettivo è trovare la deviazione unilaterale ottimale (Best Response) di un agente (es. Speaker), assumendo che l'altro agente (Listener) continui a rispondere secondo le probabilità condizionali dettate dalla policy congiunta π_{joint}.

Definiamo la **Misura di Occupazione** $\mu(s, a_{spk}, a_{lst})$ come la distribuzione scontata delle visite alle coppie stato-azione. Essa risponde alla domanda: "considerando tutto il futuro (scontato con γ),

quanto tempo passerò nello stato s eseguendo l'azione a?

$$\mu(s, a_{spk}, a_{lst}) = (1-\gamma) \sum_{t=0}^{\infty} \gamma^t P(s_t, \mathbf{a}_t = \mathbf{a} | \pi_{joint}), \text{ dove } \mathbf{a} = (a_{spk}, a_{lst})$$

Il problema di Best Response per lo Speaker è formulato come segue:

Funzione Obiettivo

Massimizzare il ritorno atteso:

$$\max_{\mu} \sum_{s} \sum_{a_{spk}} \sum_{a_{lst}} \mu(s, a_{spk}, a_{lst}) \cdot R(s, a_{spk}, a_{lst})$$

Vincoli (Constraints)

1. Vincolo di Flusso (Bellman Flow Conservation):
Garantisce che la misura μ sia coerente con le dinamiche di transizione dell'ambiente P. Il flusso che esce da uno stato futuro s'deve essere uguale al flusso che entra in esso (dallo stato iniziale o da transizioni precedenti). Per ogni stato s':

$$\sum_{\mathbf{a}} \mu(s', \mathbf{a}) = (1-\gamma)\rho_0(s') + \gamma \sum_{s, \mathbf{a}} P(s'|s, \mathbf{a})\mu(s, \mathbf{a})$$

2. Vincolo di Correlazione (Conditional Consistency):
Garantisce che, mentre lo Speaker è libero di ottimizzare la frequenza delle proprie azioni a_{spk}, il Listener mantenga la distribuzione condizionale originale $\pi(a_{lst}|s, a_{spk})$. Ossia:

$$P(a_{lst}|s, a_{spk}) = \frac{\pi_{joint}(s, a_{spk}, a_{lst})}{\sum_a \pi_{joint}(s, a_{spk}, a)}$$

Per ogni s, a_{spk} e ogni coppia a_{l1}, a_{l2}:

$$\mu(s, a_{spk}, a_{l1}) \cdot \pi_{joint}(s, a_{spk}, a_{l2}) = \mu(s, a_{spk}, a_{l2}) \cdot \pi_{joint}(s, a_{spk}, a_{l1})$$

Infatti può essere riscritta anche come segue:

$$\frac{\mu(s, a_{spk}, a_{l1})}{\mu(s, a_{spk}, a_{l2})} = \frac{\pi_{joint}(s, a_{spk}, a_{l1})}{\pi_{joint}(s, a_{spk}, a_{l2})}$$

3. Vincolo di Non-Negatività:

$$\mu(s, \mathbf{a}) \geq 0 \quad \forall s, \mathbf{a}$$

Calcolo del Gap

L'Exploitability totale ($\mathcal{E}$) è la somma del guadagno addizionale che ciascun agente otterrebbe deviando ottimamente:

1. Si calcola il valore della policy corrente: V_{joint}.
2. Si risolve l'LP per lo Speaker ottenendo

$$V_{BR,spk} = \frac{1}{1-\gamma} \max_{\mu} \sum_{s} \sum_{a_{spk}} \sum_{a_{lst}} \mu_{spk}(s, a_{spk}, a_{lst}) \cdot R(s, a_{spk}, a_{lst})$$

3. Si risolve l'LP simmetrico per il Listener ottenendo

$$V_{BR,lst} = \frac{1}{1-\gamma} \max_{\mu} \sum_{s} \sum_{a_{spk}} \sum_{a_{lst}} \mu_{list}(s, a_{spk}, a_{lst}) \cdot R(s, a_{spk}, a_{lst})$$

$$\mathcal{E} = (V_{BR,spk} - V_{joint}) + (V_{BR,lst} - V_{joint})$$

Questo metodo è computazionalmente più oneroso della Value Iteration ma è l'unico modo matematicamente corretto per valutare equilibri correlati in giochi markoviani.

🎓 PARTE 3: MURMAIL IMPLEMENTATION

3.1 Algoritmo MURMAIL

Paper: "Model-Free Multiagent Imitation Learning" (NeurIPS 2025)

Nel tuo progetto, hai un Esperto (una coppia di policy Speaker-Listener calcolata con Fictitious Play) e vuoi addestrare un agente (o una coppia di agenti) a imitarlo.
L'approccio classico è il **Behavioral Cloning (BC)**: "Fai esattamente quello che farebbe l'esperto nello stato attuale".

Tuttavia, in un contesto Multi-Agente (o anche in MDP complessi), BC ha un difetto fatale evidenziato nel paper:

- **Il problema della "Covariate Shift":** Se l'agente commette un piccolo errore e finisce in uno stato che l'esperto non visita mai (fuori dalla distribuzione dei dati di training), l'agente non sa

cosa fare.

- **Sfruttamento:** In un gioco a somma zero o competitivo, l'avversario può imparare a spingere l'agente proprio in quegli stati sconosciuti per sfruttarlo.
- **Risultato:** Anche con un errore di imitazione basso sui dati di training, il **Nash Gap** (l'exploitability reale) può essere enorme.

MURMAIL risolve questo problema non limitandosi a copiare l'esperto, ma **cercando attivamente** gli stati in cui l'imitazione è debole e chiedendo all'esperto come comportarsi *lì*.

L'algoritmo funziona su due livelli:

A. Inner Loop: Generazione dell'Avversario "Critico"

Invece di giocare contro un avversario che vuole vincere (come in Fictitious Play), qui ogni agente gioca contro un avversario "immaginario" che cerca di **massimizzare l'incertezza**.

L'algoritmo costruisce un MDP indotto dove il **Reward** non è il punteggio del gioco, ma la **distanza dalla policy dell'esperto**:

$$R(s) \sim ||\pi_{expert}(\cdot|s) - \pi_{learner}(\cdot|s)||^2$$

L'obiettivo dell'Inner Loop è trovare una policy π_{attack} che porti il sistema negli stati dove questo reward è alto (cioè dove l'agente sta sbagliando a imitare).

B. Outer Loop: Correzione via Mirror Descent

Una volta trovati questi stati critici, l'algoritmo:

1. Visita quegli stati usando la policy π_{attack}.
2. Interroga l'esperto: "Cosa faresti in questo stato strano?".
3. Aggiorna la policy dell'agente usando **Exponential Weights** (una forma di Mirror Descent) per avvicinarsi all'esperto anche in quelle zone.

3. Implementazione nel Tuo Progetto

Ecco come implementare MURMAIL integrandolo con il tuo codice esistente. Poiché tu hai accesso al modello del gioco (le matrici P e R che hai stimato o calcolato), puoi usare una versione **Model-Based** molto più efficiente di quella presentata nel paper (che usa UCBVI perché assume di non conoscere il modello).

Struttura del Codice

Crea un nuovo file o classe `MURMAIL_Solver`.

Passo 1: Inizializzazione

Carica le matrici P e R le policy dell'esperto (μ_E, ν_E) che hai già generato. Inizializza le policy dell'agente (μ_k, ν_k) come uniformi.

```
class MURMAIL_Solver:
    def __init__(self, P, expert_mu, expert_nu, gamma=0.9, lr=1.0):
        self.P = P  # Matrice transizioni (S, A_spk, A_lst, S)
        self.muE = expert_mu  # Esperto Speaker
        self.nuE = expert_nu  # Esperto Listener
        self.gamma = gamma
        self.eta = lr  # Learning rate

        # Policy iniziali uniformi
        self.S, self.A_mu, self.A_nu, _ = P.shape
        self.mu = np.ones((self.S, self.A_mu)) / self.A_mu
        self.nu = np.ones((self.S, self.A_nu)) / self.A_nu
```

Passo 2: Calcolo del Reward di Incertezza (Lemma G.7)

Questa funzione calcola quanto la tua policy attuale è diversa da quella dell'esperto in ogni stato. È il "motore" che guida l'esplorazione.

```
    def _compute_uncertainty_reward(self, current_pi, expert_pi):
        # R(s) = ||pi_expert(s) - pi_current(s)||^2
        # Calcolo vettorizzato per efficienza
        diff = expert_pi - current_pi
        reward = np.sum(diff**2, axis=1)
        return reward
```

Nota: Nel paper usano uno stimatore stocastico unbiased, ma avendo accesso alle policy esatte, puoi calcolare la norma quadratica esatta direttamente.

Passo 3: Inner Loop (Planning)

Qui sostituiamo l'RL complesso (UCBVI) con una **Value Iteration** veloce, dato che conosciamo P. Dobbiamo trovare la policy avversaria che massimizza la visita agli stati incerti.

```
def _solve_max_uncertainty(self, fixed_policy, is_speaker_fixed):
    # 1. Calcola MDP indotto fissando l'altro agente
    if is_speaker_fixed:
        # Se mu è fisso, l'avversario (nu) vede queste transizioni
        P_induced = np.einsum('sabt,sa->sbt', self.P, fixed_policy)
        R_uncertainty = self._compute_uncertainty_reward(self.nu, self.nuE)
    else:
        # Se nu è fisso, l'avversario (mu) vede queste
        P_induced = np.einsum('sabt,sb->sat', self.P, fixed_policy)
        R_uncertainty = self._compute_uncertainty_reward(self.mu, self.muE)

    # 2. Risolvi questo MDP singolo agente (Value Iteration)
    # L'obiettivo è MASSIMIZZARE il reward di incertezza cumulato
    V = np.zeros(self.S)
    for _ in range(100): # Bastano poche iterazioni
        Q = R_uncertainty[:, None] + self.gamma * P_induced @ V
        V = np.max(Q, axis=1)

    # Estrai la policy "d'attacco" (Greedy)
    Q_final = R_uncertainty[:, None] + self.gamma * P_induced @ V
    attack_policy = np.zeros_like(Q_final)
    best_actions = np.argmax(Q_final, axis=1)
    attack_policy[np.arange(self.S), best_actions] = 1.0

    return attack_policy
```

Passo 4: Outer Loop (Aggiornamento)

Eseguiamo il ciclo principale che aggiorna le policy.

```
def train(self, iterations=1000):
    avg_mu = np.copy(self.mu)
    avg_nu = np.copy(self.nu)

    for k in range(1, iterations + 1):
        # --- FASE 1: Trova i punti deboli (Attack Policies) ---
        # Trova la policy del Listener che mette in crisi lo Speaker
        nu_attack = self._solve_max_uncertainty(self.mu, is_speaker_fixed=True)
        # Trova la policy dello Speaker che mette in crisi il Listener
        mu_attack = self._solve_max_uncertainty(self.nu, is_speaker_fixed=False)

        # --- FASE 2: Calcola Gradienti (dove l'esperto è diverso da noi) ---
        # Campioniamo stati visitati dalle policy d'attacco
        d_mu = self._get_occupancy(self.mu, nu_attack)
        d_nu = self._get_occupancy(mu_attack, self.nu)

        # Il gradiente spinge verso la policy dell'esperto pesata dalla visita
        # G(s, a) = d(s) * (pi(a|s) - expert(a|s))
        # In pratica, usiamo Mirror Descent / Exp Weights:
        # pi_new(a|s) proptional to pi_old(a|s) * exp(-eta * grad)

        # Semplificazione implementativa (Exponential Weights):
        # Aumentiamo la probabilità delle azioni dell'esperto negli stati visitati
        # Gradiente approssimato: - (Expert - Current)

        grad_mu = - (self.muE - self.mu) * d_mu[:, None]
        grad_nu = - (self.nuE - self.nu) * d_nu[:, None]

        # --- FASE 3: Aggiornamento ---
        self.mu = self.mu * np.exp(-self.eta * grad_mu)
        self.nu = self.nu * np.exp(-self.eta * grad_nu)

        # Normalizzazione
        self.mu /= np.sum(self.mu, axis=1, keepdims=True)
        self.nu /= np.sum(self.nu, axis=1, keepdims=True)

        # Aggiornamento della media (Polyak Averaging)
        alpha = 1 / (k + 1)
        avg_mu = (1 - alpha) * avg_mu + alpha * self.mu
        avg_nu = (1 - alpha) * avg_nu + alpha * self.nu
```

```
        return avg_mu, avg_nu
```

Nota: La funzione `_get_occupancy` calcola la distribuzione stazionaria degli stati data la coppia di policy, risolvendo il sistema lineare .

4. Riassunto per le Note Tecniche

Da inserire nella tua documentazione:

- **Cos'è MURMAIL:** Un algoritmo di Imitation Learning attivo che, invece di copiare passivamente l'esperto, cerca di "rompere" la propria policy attuale trovando stati in cui essa differisce dall'esperto, e poi corregge quegli errori specifici.
- **Perché lo usiamo:** Perché il Behavioral Cloning standard non garantisce robustezza (basso Nash Gap) in giochi dove l'esperto non copre tutto lo spazio degli stati. MURMAIL offre garanzie teoriche di convergenza a un -Nash Equilibrium.
- **Innovazione nel Progetto:** Abbiamo adattato l'algoritmo originale (Model-Free, basato su campionamento) in una versione **Model-Based** (basata su Value Iteration), sfruttando la conoscenza esplicita delle dinamiche del gioco (P e R) ottenuta nella fase precedente. Questo rende l'addestramento ordini di grandezza più veloce e stabile.

Risultati:

```
Expert Queries    Gap      Miglioramento vs Uniform
--------------    ------   ------------------------
0                 0.244    0% (baseline)
50k               0.156    36%
100k              0.098    60%
150k              0.084    66%
200k              0.071    71% ✅

Gap finale: 0.071 con 200k queries expert
```

Convergenza relativa:

```
Expert gap:  0.013 (target)
MURMAIL gap: 0.071 (achieved)
Uniform gap: 0.244 (baseline)

Frazione miglioramento possibile:
(0.244 - 0.071) / (0.244 - 0.013) = 0.75 = 75%
```

3.3 Perché MURMAIL Funziona Bene?

Vantaggi osservati:

1. Sample Efficiency:

```
Segue traiettorie informate dall'expert
→ Visita principalmente stati rilevanti
→ Meno tempo speso in regioni subottimali dello spazio stati
```

2. Credit Assignment:

```
Expert fornisce azioni coordinate
→ Signal chiaro su quali azioni portano a reward
→ Q-function impara più velocemente
```

3. Coordination:

```
Expert ha già risolto problema di coordinazione
→ MURMAIL impara pattern comunicativi già stabiliti
→ Evita problema di emergent communication protocol
```

Limitazioni riconosciute:

Dipendenza da quality expert:

```
Expert ha gap 0.013 (non perfetto)
→ MURMAIL non può superare questo limite superiore
→ Performance ceiling determinato da expert
```

Requirement discretizzazione:

```
Expert Nash richiede formulazione tabulare
→ Non direttamente estendibile a continuous/high-dimensional spaces
→ Discretizzazione introduce approssimazione
```

🤖 PARTE 4: BASELINES DEEP RL

4.1 Motivazione Scientifica

Research question:

```
MURMAIL funziona perché:
(A) Expert imitation è fondamentale per questo task?
(B) È semplicemente un algoritmo ben progettato?
```

Approccio sperimentale:

```
Testare state-of-the-art deep RL senza accesso all'expert
→ Se convergono: Expert utile ma non essenziale
→ Se falliscono: Expert guidance fondamentale
```

4.2 Baseline: Independent DQN (IDQN) con Curriculum Learning

4.2.1 Motivazione Teorica e Problema della Non-Stazionarietà

L'addestramento simultaneo di agenti indipendenti in un ambiente cooperativo soffre intrinsecamente del problema della **Non-Stazionarietà Ambientale**.
Formalmente, per un agente i, la dinamica di transizione e la funzione di reward dipendono dalla policy congiunta di tutti gli altri agenti π_{-i}.

In un approccio *naive* (apprendimento simultaneo da zero), l'obiettivo di ottimizzazione per l'agente i al tempo t è:

$$\max_{\theta_i} J(\theta_i) = \mathbb{E}_{s\sim\rho, a_i\sim\pi_i, a_{-i}\sim\pi_{-i}^{(t)}} \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \right]$$

Poiché $\pi_{-i}^{(t)}$ cambia continuamente mentre gli altri agenti apprendono, la distribuzione dei dati di input per l'agente i non è stazionaria (covariate shift). Questo porta a gradienti rumorosi e spesso impedisce la convergenza, specialmente in compiti di coordinazione stretta come *Speaker-Listener*, dove il reward è sparso:

$$R(s, a_{spk}, a_{lst}) \gg 0 \iff \text{Communication}(a_{spk}) \text{ is correct } \wedge \text{Navigation}(a_{lst}) \text{ is correct}$$

Per mitigare questo problema, adottiamo un approccio di **Curriculum Learning in 3 Fasi**, che decompone il problema Multi-Agente in una sequenza di problemi Single-Agent stazionari.

4.2.2 Architettura e Fasi del Curriculum

Utilizziamo due approssimatori di funzione separati (Reti Neurali) parameterizzati da θ_S (Speaker) e θ_L (Listener), ottimizzati tramite la loss classica DQN:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \right)^2 \right]$$

Fase 1: Speaker Grounding (Teacher Forcing)

In questa fase, addestriamo solo lo Speaker (θ_S) mantenendo il Listener fissato su una policy esperta ottimale π_L^*.

- **Obiettivo:** Ancoraggio semantico (*Language Grounding*).
- **Formalismo:** L'ambiente percepito dallo Speaker diventa un MDP stazionario indotto da π_L^*.

$$P_{induced}(s'|s, a_S) = \sum_{a_L} P(s'|s, a_S, a_L) \pi_L^*(a_L|s, a_S)$$

- **Giustificazione:** L'uso dell'esperto π_L^* agisce come un oracolo. Se lo Speaker emette il messaggio corretto m^*, l'esperto garantisce l'azione corretta a_L^*, generando un reward positivo $r > 0$. Senza l'esperto, un Listener casuale produrrebbe $r \approx 0$ indipendentemente dall'azione dello Speaker, annullando il gradiente ($\nabla_{\theta_S} J \approx 0$).

Fase 2: Listener Adaptation (Stationary Environment)

Congeliamo i pesi dello Speaker (θ_S) ottenuto nella Fase 1 e addestriamo il Listener (θ_L).

- **Obiettivo:** Interpretazione del protocollo.
- **Formalismo:** Il Listener deve apprendere la funzione inversa della policy dello Speaker $\pi_S(m|s)$. Dato un messaggio m, deve dedurre lo stato s e agire di conseguenza.
- **Giustificazione:** Fissando θ_S, eliminiamo la non-stazionarietà. Il Listener affronta un problema di Supervised Learning implicito (RL standard), dove l'input è il messaggio generato da una distribuzione fissa. Manteniamo un $\epsilon > 0$ nello Speaker per garantire che il Listener sia robusto a un leggero rumore nel canale comunicativo.

Fase 3: Joint Refinement (Co-Adaptation)

Riabilitiamo l'aggiornamento dei gradienti per entrambi gli agenti (θ_S e θ_L) simultaneamente.

- **Obiettivo:** Convergenza all'Equilibrio di Nash locale.
- **Formalismo:**

$$\theta_S \leftarrow \theta_S - \alpha \nabla_{\theta_S} \mathcal{L}_S, \quad \theta_L \leftarrow \theta_L - \alpha \nabla_{\theta_L} \mathcal{L}_L$$

- **Giustificazione:** La Fase 2 potrebbe aver portato il Listener a fare *overfitting* sulla policy deterministica dello Speaker. Questa fase permette un *fine-tuning* reciproco ("Co-adaptation"), correggendo eventuali disallineamenti residui e permettendo agli agenti di negoziare variazioni minime al protocollo per massimizzare il ritorno congiunto.

4.2.3 Dettagli Implementativi Critici

1. **Independent Learning (Decentralized):**
   A differenza di Joint-DQN, qui non esiste una Q_{tot}.
   - $Q_S : \mathcal{S}_{spk} \times \mathcal{A}_{msg} \to \mathbb{R}$
   - $Q_L : \mathcal{S}_{lst} \times \mathcal{A}_{msg} \times \mathcal{A}_{nav} \to \mathbb{R}$
     Questa scelta serve a valutare la difficoltà del coordinamento in assenza di comunicazione esplicita dei gradienti (scenario realistico distribuito).
2. **Reward Normalization:**
   I reward grezzi $r \in \mathbb{R}$ vengono mappati in $\hat{r} \in [0, 1]$ tramite clipping.
   - **Motivo:** Stabilizzare la varianza dell'errore TD (*Temporal Difference*). In contesti con reward sparsi e di magnitudine elevata, l'errore quadratico medio (MSE) può esplodere,

destabilizzando l'apprendimento della rete neurale.

3. **Policy Esperta π_L^*:**
   Derivata tramite *Fictitious Play* nella fase precedente del progetto. Viene utilizzata esclusivamente nella Fase 1 come meccanismo di "bootstrapping" per rompere la simmetria iniziale e avviare il gradiente.

4.3 Baseline 2: MAPPO (Multi-Agent PPO)

Motivazione:

```
Policy gradient con centralizzazione parziale:
- Centralized critic: Accesso a global state durante training
- Decentralized actors: Policy indipendenti per execution
- PPO: Stability via clipped objective
```

Architettura:

```
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, hidden=128):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, hidden),
            nn.ReLU(),
            nn.Linear(hidden, action_dim)
        )

    def forward(self, state):
        logits = self.net(state)
        return Categorical(logits=logits)

class CentralizedCritic(nn.Module):
    def __init__(self, global_state_dim, hidden=128):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(global_state_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, 1)
        )

    def forward(self, global_state):
        return self.net(global_state).squeeze(-1)
```

Training (semplificato):

```
speaker_actor = Actor(state_dim=3, action_dim=3)
listener_actor = Actor(state_dim=972, action_dim=5)
critic = CentralizedCritic(global_state_dim=3+972)

for episode in episodes:
    # Collect trajectory
    trajectory = rollout(env, speaker_actor, listener_actor)
    states, actions, rewards, dones = trajectory

    # Compute values e advantages
    values = critic(global_states)
    advantages = compute_GAE(rewards, values, dones, γ=0.9, λ=0.95)

    # Normalize advantages (critico per stabilità)
    advantages = (advantages - advantages.mean()) / (advantages.std() + 1e-8)

    # PPO update (multiple epochs)
    for epoch in range(4):
        # Compute policy ratios
        new_log_probs = compute_log_probs(actions, actors)
        ratio = exp(new_log_probs - old_log_probs)

        # Clipped surrogate objective
        surr1 = ratio * advantages
        surr2 = clip(ratio, 1-ε_clip, 1+ε_clip) * advantages
        policy_loss = -min(surr1, surr2).mean()

        # Value loss
        value_loss = MSE(critic(states), returns)

        # Optimize
        total_loss = policy_loss + value_coef * value_loss
        optimize(total_loss)
```

Variants Tested:

Variant 1: One-Hot Encoding

```
Input representation:
- Speaker: one-hot(3) = [0,0,1]
- Listener: one-hot(972) = [0,...,1,...,0]

Problema: 972-dimensional sparse vector
→ Network difficoltà generalizzazione
→ Many parameters, little signal

Risultato: Gap 0.28 (no learning)
```

Variant 2: Embedding Layers

```
embedding_speaker = nn.Embedding(3, 16)
embedding_listener = nn.Embedding(972, 64)

Input: state indices (dense)
→ Embedded representations (learned)

Risultato: Gap 0.46 (peggiore!)
Ipotesi: Embeddings non converged + policy instability
```

Variant 3: Hyperparameter Tuning

```
Modifiche:
- Learning rate: 0.001 → 0.0003 (ridotto)
- Gamma: 0.99 → 0.9 (matching expert)
- Rollout length: 64 → 2048 (più dati per update)
- Update epochs: 4 → 10 (più optimization per batch)

Risultato: Gap 0.37 (miglioramento marginale, ancora subottimale)
```

Problemi Diagnosticati:

1. Advantage Collapse:

```
Debug output:
Advantages std: 0.000001  ← Variance troppo bassa

Dopo normalization:
(adv - mean) / (std + 1e-8) → numerical instability

Conseguenza: Policy gradients ≈ 0, no learning
```

2. Policy Ratio Frozen:

```
PPO ratio = exp(new_log_prob - old_log_prob)
Osservato: ratio.mean() = 1.0000, ratio.std() = 0.0000

Significato: Policy non sta cambiando
→ Actor updates ineffective
```

3. Reward Scaling Mismatch:

```
Training: Optimized on scaled rewards (÷10 per stabilità)
Evaluation: Gap calculated su true game

Risultato: Network ottimizza obiettivo diverso da metrica
→ Divergenza training/evaluation objectives
```

Risultati Complessivi MAPPO:

Variant	Gap	Note
One-hot	0.280	Worse than uniform (0.244)
Embeddings (v1)	0.460	Significantly worse
Embeddings (v2)	0.370	Still suboptimal
Tuned γ=0.9	0.367	Marginal improvement

```
Conclusione: Nessuna configurazione converge a performance acceptable
```

4.4 Baseline 3: Joint-Action DQN

Motivazione:

Limitazioni approcci precedenti:

```
DQN-Curriculum: Independent learning, no coordination
MAPPO: Policy gradient instability, factored policies

Joint-Action DQN:
✓ Value-based (più stable che policy gradient)
✓ Joint Q-function (no factorization bias)
✓ Esperienza replay (sample reuse)
✓ Target networks (training stability)
```

Architettura:

```
class JointQNetwork(nn.Module):
    """
    Q(s_global, a_joint) per tutte le 15 joint actions

    Input: joint state index ∈ [0, 2915]
    Output: Q-values per 15 joint actions
    """
    def __init__(self, num_states=2916, num_actions=15,
                 embed_dim=128, hidden=128):
        super().__init__()

        # Embedding per stato globale discreto
        self.state_embedding = nn.Embedding(num_states, embed_dim)

        # Q-network (2 hidden layers, ragionevole per 2916 stati)
        self.net = nn.Sequential(
            nn.Linear(embed_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, hidden),
            nn.ReLU(),
            nn.Linear(hidden, num_actions)
        )

    def forward(self, state_indices):
        embedded = self.state_embedding(state_indices)
        return self.net(embedded)
```

Encoding Scheme:

```python
# Joint State: (s_speaker, s_listener) → single index
def encode_joint_state(s_speaker, s_listener):
    assert 0 <= s_speaker < 3
    assert 0 <= s_listener < 972
    return s_speaker * 972 + s_listener  # ∈ [0, 2915]

# Joint Action: (a_speaker, a_listener) → single index
def encode_joint_action(a_speaker, a_listener):
    assert 0 <= a_speaker < 3
    assert 0 <= a_listener < 5
    return a_speaker * 5 + a_listener  # ∈ [0, 14]

# Decoding inverso
def decode_joint_state(joint_state):
    s_speaker = joint_state // 972
    s_listener = joint_state % 972
    return s_speaker, s_listener

def decode_joint_action(joint_action):
    a_speaker = joint_action // 5
    a_listener = joint_action % 5
    return a_speaker, a_listener
```

Training Algorithm:

```
# Setup
replay_buffer = ReplayBuffer(capacity=100000)
q_network = JointQNetwork(num_states=2916, num_actions=15)
target_network = copy_network(q_network)
optimizer = Adam(q_network.parameters(), lr=0.0005)

epsilon = 1.0  # Exploration rate
gamma = 0.9

for episode in range(20000):
    state = env.reset()
    joint_state = encode_joint_state(state.speaker, state.listener)
    done = False

    while not done:
        # Epsilon-greedy action selection
        if random() < epsilon:
            joint_action = random.randint(0, 14)
        else:
            with torch.no_grad():
                q_vals = q_network(torch.LongTensor([joint_state]))
                joint_action = q_vals.argmax().item()

        # Decode and execute
        a_speaker, a_listener = decode_joint_action(joint_action)
        next_state, reward, done = env.step(a_speaker, a_listener)
        next_joint_state = encode_joint_state(next_state.speaker, next_state.listener)

        # Store transition
        replay_buffer.push(joint_state, joint_action, reward, next_joint_state, done)

        # Training update (if buffer sufficient)
        if len(replay_buffer) >= 1000:
            # Sample mini-batch
            batch = replay_buffer.sample(batch_size=64)

            # Current Q-values
            q_current = q_network(batch.states).gather(1, batch.actions.unsqueeze(1))

            # Double DQN target
            with torch.no_grad():
```

```
                # Action selection: online network
                next_best_actions = q_network(batch.next_states).argmax(dim=1)

                # Q-value evaluation: target network
                q_next = target_network(batch.next_states).gather(1, next_best_actions.

                # TD target
                targets = batch.rewards + gamma * (1 - batch.dones) * q_next.squeeze()

            # Loss and optimization
            loss = F.mse_loss(q_current.squeeze(), targets)

            optimizer.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(q_network.parameters(), max_norm=10.0)
            optimizer.step()

        joint_state = next_joint_state

    # Epsilon decay (linear)
    if episode < 15000:
        epsilon = 1.0 - (1.0 - 0.01) * episode / 15000
    else:
        epsilon = 0.01

    # Target network update (every 1000 steps, not episodes)
    if global_steps % 1000 == 0:
        target_network.load_state_dict(q_network.state_dict())
```

Exploitability Calculation - Evolution:

Problema Iniziale:

```
# Tentativo ingenuo: estrai marginals
for s_spk in range(3):
    # Vote across listener states
    action_counts = zeros(3)
    for s_lst in range(972):
        joint_state = encode(s_spk, s_lst)
        best_action = Q_network(joint_state).argmax()
        a_spk, _ = decode(best_action)
        action_counts[a_spk] += 1

    π_speaker[s_spk] = one_hot(argmax(action_counts))

Problema critico: Distrugge correlazione appresa!
Joint Q-function coordina azioni, marginals rompono coordinazione
```

Soluzione Finale: True Exploitability via LP

Implementata formulazione basata su occupancy measures:

```
def calc_true_exploitability(π_joint, R, P, init_dist, γ):
    """
    Linear Programming formulation per joint policies

    Variables: μ(s, a_s, a_l) - occupancy measures

    Objective (per speaker BR):
        maximize Σ μ(s,a_s,a_l) R(s,a_s,a_l)

    Constraints:
    1. Flow conservation:
       Σ_a μ(s',a) = (1-γ)d₀(s') + γ Σ_{s,a} μ(s,a)P(s'|s,a)

    2. Listener conditioning:
       μ(s,a_s,a_l) rispetta distribuzione condizionale
       indotta da π_joint quando speaker sceglie a_s

    3. Non-negativity: μ ≥ 0
    """

    # Extract occupancy from joint policy
    μ_joint = compute_occupancy(π_joint, P, init_dist, γ)
    V_joint = expected_value(μ_joint, R) / (1 - γ)

    # Speaker BR via LP
    V_br_speaker = solve_speaker_LP(π_joint, R, P, init_dist, γ)

    # Listener BR via LP
    V_br_listener = solve_listener_LP(π_joint, R, P, init_dist, γ)

    return (V_br_speaker - V_joint) + (V_br_listener - V_joint)
```

Nota metodologica importante:

La formulazione LP enforce consistency con la distribuzione condizionale indotta da π_joint, fornendo una stima dell'exploitability che:

- Rispetta la struttura correlata della policy
- È più accurata della semplice marginalizzazione
- Rappresenta un'approssimazione conservativa della vera exploitability per joint policies

Risultati:

Episode	Avg Reward	Gap (true LP)	Epsilon	Status
2000	−35.2	0.18	0.87	Early learning
5000	−32.1	0.12	0.67	Improving
10000	−28.4	0.09	0.34	Good progress
15000	−26.8	0.06	0.14	Promising
18000	−34.6	0.07	0.01	Fluctuation
19000	−29.0	0.029	0.01	⭐ Peak performance
19500	−24.7	0.087	0.01	Degradation
20000	−27.6	0.111	0.01	Instability

Best: Gap 0.029 (episode 19000)
Final: Gap 0.111 (episode 20000)

Analisi Performance:

Miglioramento peak vs uniform:
(0.244 − 0.029) / (0.244 − 0.013) = 0.93 = 93% del possibile

Problema: Instabilità
- Peak 0.029 non sostenuto
- Oscillazioni 0.029 → 0.111 in 1000 episodes
- Suggerisce overfitting o catastrophic forgetting

Possibili cause instabilità:

1. Epsilon molto basso (0.01): Exploitation pura, no recovery da suboptimal
2. Replay buffer stale data: Old experiences misleading current policy
3. Non-stationarity: Target network updates creano shifting objectives
4. Assenza early stopping: Training continua oltre optimal point

📊 PARTE 5: COMPARISON E CONCLUSIONI

5.1 Comprehensive Results

Method	Type	Gap	Stable?	Samples	Tuning
Expert (FP)	Nash solver	0.013	✅	N/A	Low
MURMAIL	Imitation	0.071	✅	200k	Low
Joint-DQN (peak)	Deep RL	0.029	❌	475k	High
Joint-DQN (final)	Deep RL	0.111	❌	500k	High
MAPPO (best)	Policy grad	0.280	❌	500k+	High
MAPPO (variants)	Policy grad	0.37+	❌	500k+	High
DQN-Curriculum	Independent	0.498	❌	400k	Medium

Baseline uniforme: 0.244

5.2 Key Findings

Finding 1: Expert Imitation è Sample-Efficient e Stabile

MURMAIL characteristics:
✅ Convergenza stabile a gap 0.071
✅ 200k queries (relativamente efficiente)
✅ Minimal hyperparameter sensitivity
✅ Predictable training dynamics

Implicazione: Per deployment pratico, expert-guided approach preferibile per reliability e reproducibility

Finding 2: Standard MARL Non Converge su Questo Task

```
MAPPO: Gap 0.28–0.46 (worse than uniform 0.244)
DQN-Independent: Gap 0.50 (significantly suboptimal)

Root causes identified:
- Communication protocol emergence: Difficoltà coordinare
  messaggi arbitrari senza guidance
- Partial observability: Credit assignment ambiguous
- Sparse rewards: Poco segnale informativo per exploration
- High-dimensional listener state: 972 stati difficult
  to explore efficiently

Conclusion: Questi task richiedono structured guidance,
pure exploration-based learning insufficient
```

Finding 3: Joint-DQN Può Funzionare Ma Con Limitazioni

```
Capabilities demonstrated:
✅ Peak performance 0.029 (vicino a MURMAIL 0.071)
✅ Può apprendere senza expert (self-discovered coordination)
✅ Migliore di MAPPO (no factorization bias)

Limitations observed:
❌ Instabilità: Oscillazioni 0.029 → 0.111
❌ Sample inefficiency: 475k samples per peak (vs 200k MURMAIL)
❌ Hyperparameter sensitivity: Richiede tuning estensivo
❌ Non-monotonic: Performance degrada post-peak

Implication: Deep RL è una possibile alternativa con costi:
- Maggiore computational budget
- Expertise in hyperparameter tuning
- Risk di instability in deployment
```

5.3 Theoretical Understanding

Perché Communication è Intrinsecamente Difficile?

1. Arbitrary Protocol Space:

```
Speaker deve stabilire mapping:
Goal 0 → Message m₀
Goal 1 → Message m₁
Goal 2 → Message m₂

Listener deve imparare inverse mapping:
Message m₀ → Behavior towards landmark 0
Message m₁ → Behavior towards landmark 1
Message m₂ → Behavior towards landmark 2

Problema: Infinite possibili mappings consistenti!
RL deve scoprire UNO via trial-and-error
Expert fornisce un mapping pre-stabilito
```

2. Partial Observability Asymmetrica:

```
Speaker: Osserva goal, non posizione listener
Listener: Osserva posizioni, non goal

Conseguenza: Credit assignment ambiguo
- Se reward basso, chi ha sbagliato?
- Speaker comunicazione unclear?
- Listener movimento suboptimal?

Expert coordina entrambi simultaneamente
```

3. Sparse Reward Structure:

```
R(s,a) = -distance(listener, correct_goal)

Caratteristiche:
- Sempre non-positivo
- Gradient informativo solo locale
- Plateaus in regioni distanti da goal

Expert: Segue gradient informativo già "risolto"
RL: Deve scoprire gradient via exploration
```

Quando Expert Imitation È Essenziale vs Optional?

Basandosi sui risultati, ipotesi:

Essential quando:

```
✓ Coordination richiesta complessa (come communication)
✓ Partial observability significativa
✓ Reward structure sparse
✓ Sample budget limitato
✓ Reliability critica per deployment
```

Optional (forse) quando:

```
? Task più semplice (e.g., collision avoidance)
? Observations complete
? Reward dense e informativo
? Computational budget illimitato
? Instabilità acceptable
```

Nota: Questa rimane ipotesi da validare su altri tasks

5.4 Contributions

Contributi Implementativi:

1. ✅ **Discretization wrapper funzionante** per Speaker-Listener environment
2. ✅ **Nash equilibrium solver** via Fictitious Play (2000 iterations, gap 0.013)
3. ✅ **MURMAIL implementation completa** con evaluation rigorosa
4. ✅ **Suite di baselines deep RL:**
   - DQN with curriculum (3-phase)
   - MAPPO (3 architectural variants)
   - Joint-Action DQN (with proper joint exploitability)
5. ✅ **True exploitability calculation** via Linear Programming per joint policies

Contributi Scientifici:

1. ✅ **Empirical evidence:** Expert imitation sample-efficient e stabile su communication task
2. ✅ **Negative results:** Standard MARL fails on questo task specifico nonostante tuning
3. ✅ **Comparative analysis:** Joint value-based methods superiori a policy gradient su questo setting
4. ✅ **Methodological insight:** Importance di exploitability metric alignment con policy structure

Limitations Riconosciute:

1. ⚠️ **Single environment:** Results specifici a Speaker-Listener, generalizzazione non testata
2. ⚠️ **Discretization approximation:** Introduce bias rispetto a continuous original
3. ⚠️ **Computational constraints:** Deep RL baselines non exhaustively tuned (budget limitato)
4. ⚠️ **Expert quality:** Nash solver gap 0.013 (near-optimal ma non perfetto)

🎓 CONCLUSIONE

Research Question:

"In questo task di coordinazione multi-agente con comunicazione, è necessaria expert imitation per raggiungere performance ragionevoli in modo sample-efficient e stabile?"

Answer:

Sulla base dei risultati sperimentali, in questo specifico setting:

1. **Expert imitation (MURMAIL):**
   - Converge stabilmente (gap 0.071)
   - Sample efficient (200k queries)
   - Minimal tuning required
   - **Raccomandato per deployment pratico**
2. **Deep RL (Joint-DQN):**
   - Può raggiungere performance competitive (peak 0.029)
   - Richiede 2-3x più samples
   - Instabile (requires early stopping)

- **Fattibile ma più rischioso**

3. **Standard MARL (MAPPO, Independent DQN):**
    - Non converge (gap 0.28-0.50)
    - Worse than uniform baseline
    - **Non raccomandato per questo task**

Implicazione Pratica:

Per tasks simili (communication, coordination, partial observability):

- Expert guidance fornisce percorso più reliable
- Pure exploration-based learning può funzionare con investment sostanziale
- Structure e guidance sembrano benefici, se non essenziali

Future Work:

- Testare su altri communication tasks
- Investigare hybrid approaches (partial expert guidance)
- Estendere a continuous action spaces
- Sviluppare methods per automatic protocol discovery

Il progetto dimostra rigore metodologico, onestà scientifica nei risultati negativi, e contributi sia implementativi che analitici solidi.

Domande su aspetti specifici? 🤔