APPENDIX

A METHODOLOGY

In this section we provide a more rigorous description of PSRO and our two proposed algorithms. We will begin by explaining the details of PSRO, and then explain where each of the proposed algorithms deviate from PSRO's implementation. Finally, we will discuss how Q-Mixing is leveraged as both a Transfer-Oracle and Opponent-Oracle.

A.1 POLICY-SPACE RESPONSE ORACLES

PSRO starts from an initial set of policies for each player Π^0. Commonly, the initial strategy-sets contain only a single random policy for each player. An empirical game is initialized $\tilde{\Gamma}^0$ that contains the initial strategy sets with an undefined payoff function. The algorithm then proceeds by iteratively performing a two step process: (1) complete and solve the empirical game, and (2) grow the strategy-sets.

First, the algorithm checks its progress towards a solution via solving the empirical game. All policy-profiles with missing entries, an undefined payoff, are simulated for a fixed number of episodes specified by a hyperparameter.[4] The average return experienced by each player in this profile is then filled into the appropriate game-cell. This is what is captured by the psuedocode:

Algorithm 4: Simulate and Expand the ENFG Pseudocode

Simulate missing entries in $\tilde{U}^{\Pi^e}$ from Π^e.

Enumerating and checking each profile can become computationally expensive as a result of the combinatorial explosion of the profile space. However, each player only adds one new policy to their strategy-set each epoch. Therefore, we know that we need only check profiles containing at least one of the player's new policies. Our more efficient method for completing the ENFG is as follows:

Algorithm 5: Efficient Expansion of the ENFG (ExpandENFG)

Input: $\tilde{U}^{\Pi^e}$, Π^e, e
for *player* $i \in [[n]]$ **do**
  for *opponent_profile in* Π^e_{-i} **do**
    # Skip profiles that have already been simulated.
    $profile \leftarrow (\pi^e_i$,*opponent*$\_profile)$
    if *profile* $\in \tilde{U}$ **then**
      skip

    # Build an estimate of the payoff by simulating the profile.
    $episodes \leftarrow []$
    for *many episodes* **do**
      $\tau \sim$ profile
      $episodes$.append(τ)
    $\tilde{U}(profile) =$ mean_return$(episodes)$
Output: $\tilde{U}^{\Pi^e}$

For example, if we were on epoch two of a two-player game then the ExpandENFG call would simulate the cells highlighted in Table 1.

Now with a complete payoff function $\tilde{U}^{\Pi^e}$, the empirical-game is solved using a MSS to produce a solution $\sigma^{*,e}$. Recall that the MSS is a function that solves the ENFG for a game-theoretic solution concept (e.g., Nash equilibrium). Therefore, when choosing a MSS it is important to decide what your solution concept will be and the computational limitations surrounding the MSS implementation.[5]

[4] In our work this is typically set to 30.

[5] In our work we utilized linear complementarity to solve for Nash equilibrium.

	π^0_{-i}	π^1_{-i}	π^2_{-i}
π^0_i			
π^1_i			
π^2_i			

Table 1: The cells highlighted represent the profiles simulated for a two-player game on epoch two.

The second portion of the epoch concerns growing each player's strategy set. The strategy-exploration method employed by PSRO is to add one policy to each player's strategy-set that is the best-response to the opponent's current solution profile. In psuedo-code this best-response training is denoted:

Algorithm 6: Best-Response to the ENFG solution

for *many episodes* **do**
  $\pi_{-i} \sim \sigma^{*,e-1}_{-i}$
  Train π^e_i over $\tau \sim (\pi^e_i, \pi_{-i})$

In this code-snippet we are training a best-response policy for player i. This policy is trained until a maximum training budget is hit – in our experiments this limit is specified by a cumulative number of simulator steps, and could trained using any RL algorithm. The authors of PSRO have chosen to write the RL training section of the pseudo-code this way to highlight an important implementation detail. At the beginning of each episode, the opponents' policies are drawn from their mixed strategies $\pi_{-i} \sim \sigma_{-i}$. The opponent policies are then kept fixed for an entire episode $\tau \sim (\pi^e_i, \pi_{-i})$. The policy that is being trained may be updated at any point through an episode according to the RL algorithm's specification. Once training has completed for the new policy, it is then added to the player's strategy set $\Pi^e_i = \Pi^{e-1}_i \cup \{\pi^e_i\}$.

These two steps repeat for either a fixed number of iterations, or until a convergence condition is met (e.g., stable solution, or zero-regret).

Our more expanded version of PSRO is as follows:

Algorithm 7: Policy-Space Response Oracles (Lanctot et al., 2017)

Input: Initial policy sets for all players Π^0, MSS

 # Initialize the ENFG.
$\tilde{U}^{\Pi^0} \leftarrow$ ExpandENFG$(\tilde{U}^{\Pi^0}, \Pi^0, 0)$
Simulate utilities $\tilde{U}^{\Pi^0}$ for each joint $\pi^0 \in \Pi^0$
Initialize solution $\sigma^{*,0}_i =$ Uniform(Π^0_i)

while *epoch e in* $\{1,2,...\}$ **do**
  # Strategy-set expansion.
  for *player* $i \in [[n]]$ **do**
    # Generate new policies – best-responses to the current solution.
    for *many episodes* **do**
      $\pi_{-i} \sim \sigma^{*,e-1}_{-i}$
      Train π^e_i over $\tau \sim (\pi^e_i, \pi_{-i})$
    $\Pi^e_i = \Pi^{e-1}_i \cup \{\pi^e_i\}$

  # Simulate missing entries in $\tilde{U}^{\Pi^e}$ *from* Π^e.
  $\tilde{U}^{\Pi^e} \leftarrow$ ExpandENFG$(\tilde{U}^{\Pi^e}, \Pi^e, e)$

  # Compute a solution to the current ENFG.
  $\tilde{\Gamma}^e \leftarrow (\tilde{U}^{\Pi^e}, \Pi^e, ||\Pi^e||)$
  $\sigma^{*,e} \leftarrow$ MSS$(\tilde{\Gamma}^e)$
Output: Current solution $\sigma^{*,e}_i$ for player i

A.2 MIXED-ORACLES

Mixed-Oracles differs from PSRO in the way it generates new policies for each player.

Instead of training directly against the opponent solution $\sigma_{-i}^{*,e}$ during each epoch, we will instead train a best-response to only the new opponent policy π_{-i}^{e}. The RL training proceeds in the same manner as in PSRO; however, the opponent does not need to sample a new policy at each episode due to their pure-strategy. We denote this best-response policy as λ_i^e, and each agent collects a set of these pure-strategy best-responses in the set $\Lambda_i^e = \left\{\lambda_i^j\right\}_{j=0}^{e}$. This set is meant to hold the best-response to each of the opponent's policies.

However, Mixed-Oracles, much like PSRO, still expands each player's strategy-set with the best-response to the opponent's solution $\sigma_{-i}^{*,e-1}$. To accomplish this we employ a new function entitled TransferOracle that will take in a set of policies and a distribution, and return a policy that represents an aggregation of the input policies weighted by the distribution. This function is provided as an additional input to the Mixed-Oracles algorithm. Finally, we can construct our best-response to the opponent's solution $\sigma_{-i}^{*,e-1}$ by calling TransferOracle with our agent's best-responses to the opponent's policies Λ_i and the opponent's mixture $\sigma_{-i}^{*,e-1}$. The TransferOracle can then aggregate our best-response behaviour based on how likely the opponent is to play their respective policies (that we are best-responding to). The policy is then added to the player's strategy-set and the algorithm carries forward much like PSRO.

Algorithm 8: Expanded Mixed-Oracles

Input: Initial policy sets for all players Π^0, TransferOracle function, MSS

 # Initialize the ENFG.
$\tilde{U}^{\Pi^0} \leftarrow$ ExpandENFG$(\tilde{U}^{\Pi^0}, \Pi^0, 0)$
Simulate utilities $\tilde{U}^{\Pi^0}$ for each joint $\pi^0 \in \Pi^0$
Initialize solution $\sigma_i^{*,0} =$ Uniform(Π_i^0)

 # Each player maintains a set of opponent pure-strategy (policy) best responses.
Initialize pure-strategy best-responses $\Lambda_i^0 = \emptyset$

while *epoch e in* $\{1,2,...\}$ **do**
  # Train best-responses to each new opponent policy.
  for *player* $i \in [[n]]$ **do**
    for *many episodes* **do**
      Train λ_i^e over $\tau \sim (\lambda_i^e, \pi_{-i}^{e-1})$
    $\Lambda_i^e = \Lambda_i^{e-1} \cup \{\lambda_i^e\}$

  # Generate new policies – best-responses to the current solution.
  for *player* $i \in [[n]]$ **do**
    $\pi_i^e \leftarrow$ TransferOracle$(\Lambda_i, \sigma_{-i}^{*,e-1})$
    $\Pi_i^e = \Pi_i^{e-1} \cup \{\pi_i^e\}$

  # Simulate missing entries in $\tilde{U}^{\Pi^e}$ *from* Π^e.
  $\tilde{U}^{\Pi^e} \leftarrow$ ExpandENFG$(\tilde{U}^{\Pi^e}, \Pi^e, e)$

  # Compute a solution to the current ENFG.
  $\tilde{\Gamma}^e \leftarrow (\tilde{U}^{\Pi^e}, \Pi^e, ||\Pi^e||)$
  $\sigma^{*,e} \leftarrow$ MSS$(\tilde{\Gamma}^e)$
Output: Current solution strategy $\sigma_i^{*,e}$ for player i.

A.3 MIXED-OPPONENTS

Mixed-Opponents operates the same as PSRO, except that instead of training against $\sigma_{-i}^{*,e-1}$ we will create a new policy π_{-i} representing an aggregate of the mixture. To construct the new policy we leverage an OpponentOracle that takes in a set of policies and a distribution and aggregates them into a single representative policy. Here, we aggregate all of the opponent's policies Π_{-i}^{e-1} by their solution $\sigma_{-i}^{*,e-1}$. This removes the need to sample a new policy during each episode, and eliminates any variance encountered during learning that was a result of playing against an unobserved distribution of opponents. The new best-response also serves as a best-response to the objective specified by the MSS, so the policy is simply added to the player's strategy-set and the algorithm continues similar to PSRO.

Algorithm 9: Expanded Mixed-Opponents

Input: Initial policies sets for all players Π^0, OpponentOracle function, MSS

 # Initialize the ENFG.
$\tilde{U}^{\Pi^0} \leftarrow$ ExpandENFG($\tilde{U}^{\Pi^0}$,Π^0,0)
Simulate utilities $\tilde{U}^{\Pi^0}$ for each joint $\pi^0 \in \Pi^0$
Initialize solution $\sigma_i^{*,0} =$ Uniform(Π_i^0)

while *epoch e in* $\{1,2,...\}$ **do**
  for *player* $i \in [[n]]$ **do**
    # Aggregate the opponent mixture into a single representative policy.
    $\pi_{-i} \leftarrow$ OpponentOracle(Π_{-i}^{e-1},$\sigma_{-i}^{*,e-1}$)

    # Generate new policies – best-responses to the current solution.
    for *many episodes* **do**
      Train π_i^e over $\tau \sim (\pi_i^e, \pi_{-i})$
    $\Pi_i^e = \Pi_i^{e-1} \cup \{\pi_i^e\}$

  # Simulate missing entries in $\tilde{U}^{\Pi^e}$ *from* Π^e.
  $\tilde{U}^{\Pi^e} \leftarrow$ ExpandENFG($\tilde{U}^{\Pi^e}$,Π^e,e)

  # Compute a solution to the current ENFG.
  $\tilde{\Gamma}^e \leftarrow (\tilde{U}^{\Pi^e}, \Pi^e, ||\Pi^e||)$
  $\sigma^{*,e} \leftarrow$ MSS($\tilde{\Gamma}^e$)
Output: Current solution $\sigma_i^{*,e}$ for player i.

A.4 Q-MIXING AS A TRANSFERORACLE AND OPPONENTORACLE

Mixed-Oracles and Mixed-Opponents require an additional input of a TransferOracle and OpponentOracle, respectively. Both of these functions have the same interface:

$$F : \vec{\pi}, \Delta(\vec{\pi}) \rightarrow \pi.$$

In this study we leverage the Q-Mixing (Smith et al., 2020) technology as the input for both of these functions. Q-Mixing is a general method for aggregating value-based policies, following a distribution, into a single policy. The general form of Q-Mixing is as follows:

$$Q_i(o_i, a_i \,|\, \sigma_{-i}) = \sum_{\pi_{-i}} \psi_i(\pi_{-i} \,|\, o_i, \sigma_{-i}) Q_i(o_i, a_i \,|\, \pi_{-i}),$$

where ψ is a function that determines the relative likelihood of playing an opponent $\psi_i : \mathcal{O} \rightarrow \Delta(\Pi_{-i})$. While this form allows for much more expressive methods of fusing policies, we utilize a simple version of this method. In particular, we the formulation Smith et al. (2020) refers to as *Q-Mixing: Prior*. In this version, the opponent weighting does not consider historical observations nor future uncertainty, but

instead weights each policy directly by the distribution:

$$Q_i(o_i,a_i \,|\, \sigma_{-i}) = \sum_{\pi_{-i}} \sigma_{-i}(\pi_{-i}) Q_i(o_i,a_i \,|\, \pi_{-i}).$$

We can then define the TransferOracle or OpponentOracle that uses Q-Mixing as follows (we assume that all policies are value-based and substitute the policies for value-functions):

$$F : \vec{Q},\, (\sigma = \Delta(\vec{Q})) \rightarrow Q_{\text{Q-Mixing}}(o_i,a_i \,|\, \sigma) = \sum_{Q_i \in \vec{Q}} \sigma(Q_i) Q_i(o_i,a_i),$$

where notationally $\sigma(Q_i)$ represents the probability associated with Q_i in σ.

B HYPERPARAMETER SELECTION

In PSRO, we utilize Deep Reinforcment Learning (Deep RL) to compute an approximate best-resposne policy for each agent during each iteration. Both the convergence and performance of Deep RL algorithms is highly dependent on the selection of the settings of the algorithms, also known as the hyperparameters (to distinguish from the parameters being optimized). This presents us with two major issues (1) we need the hyperparameters for the Deep RL algorithm prior to running PSRO, and (2) any particular hyperparameter configuration may not necessarily work well against a different opponent's strategy.

To address both of these issues, we take a two-step approach. First, in order to choose Deep RL parameters prior to running PSRO, we select the hyperparameters that work best for the Deep RL algorithm when playing a random opponent. Then we run PSRO with these hyperparameters to generate a static set of policies for each player.

We now turn to our second issue, we cannot select hyperparameters that work best against each opponent strategy. However, with the diverse set of policies generated by our initial PSRO run, we have a representative set of possible opponent strategies. From the resulting PSRO run, we sub-sample a set of k opponent policies from the solution's support. We utilize these selected policies to evaluate the quality of a hyperparameter setting.

In this study, we consider two hyperpameter versions: parameters for training against pure-strategy opponents (hereafter, *pure-hparams*), and parameters for training against mixed-strategy opponents (hereafter, *mix-hparams*). This distinction is drawn, because the more stochastic, or mixed, the opponent's strategy, should require more training to cope with the larger variability in experiences. The pure-hparams are the set of hyperparameters that had the highest average final return when training best-responses independently against each of the k opponent policies. The mix-hparams are the hyperparameters that had the highest final return when training a best-response against the uniform-mixed-strategy of the k opponent policies.

All hyperparameter searches are conducted by sampling 300 combinations of hyperparameters. Each hyperparameter is then either trained against the random-opponent task, or both the mixed-opponent and pure-opponent tasks. For both the mixed-opponent and pure-opponent tasks we use five opponent policies.

We consider the following hyperparameters:

Batch Size: The number of experiences to utilize in one training step.

Replay Capacity: The maximum number of experiences for an agent to maintain (First-In First-Out ordering).

Min Replay Size: The minimum number of experiences required in the replay buffer before training can begin.

Learning Rate: The learning rate of the optimizer (this is Adam in all experiments).

N Exploration Timesteps: The number of of timesteps where the agent is following an exploration policy. In this study we utilize an ϵ-greedy exploration policy with a linear decay from 1.0 to 0.03.

N Total Timesteps: The total number of timesteps allowed before training is terminated.

C Evaluation Methods

C.1 Regret

The regret of player i in profile σ is given by $\overline{\Pi}_i \subseteq \Pi_i$:

$$\text{Regret}_i(\sigma, \overline{\Pi}_i) = \max_{\pi_i \in \overline{\Pi}_i} u_i(\pi_i, \sigma_{-i}) - u_i(\sigma_i, \sigma_{-i}). \tag{2}$$

C.2 Empirical Proxy Regret

To measure the exact regret for a player with respect to a strategy profile equation 2, we would need the ability to compute the actual best response against the other players. Since this is generally infeasible and expensive to approximate in an empirical-game setting, we instead evaluate regret with respect to a static set of policies. Let $\bar{\Pi}_i \subseteq \Pi_i$ be the static set for player i, and $\tilde{\Pi}_i \subseteq \Pi_i$ by the set of policies generated by PSRO. The proxy regret for player i in profile σ is given by:

$$\text{Regret}_i(\sigma, \tilde{\Pi}_i) = \max(0, \max_{\pi_i \in \bar{\Pi}_i \cup \tilde{\Pi}_i} u_i(\pi_i, \sigma_{-i}) - u_i(\sigma_i, \sigma_{-i})). \tag{3}$$

Note: that we clip the minimum of this regret to zero, because all policies in the static-set of policies may perform worse than the profile σ_i.

C.2.1 Evaluating Static-Set Policies

In this section we validate that our static-set of evaluation policies is diverse. Each pair of policies is compared by checking their action agreement. Due to computational limitations, we cannot check their agreement across the complete state-space. So instead, we collect a sub-set of the state-space that is representative of the strategy-space explored. We simulate each profile of policies for 30 episodes and collect the states observed by all agents. Next, we remove all duplicate states that will bias any comparisons (e.g., the behaviour at the beginning of the episode will occur more frequently and be favoured more in the comparison). Now with our filtered states, we check their average agreement of their greedy actions.

In the Gathering-Small environment, 2,160,000 states were collected and 170,966 states remained after duplicates were removed. The comparison is shown in Figure 7.

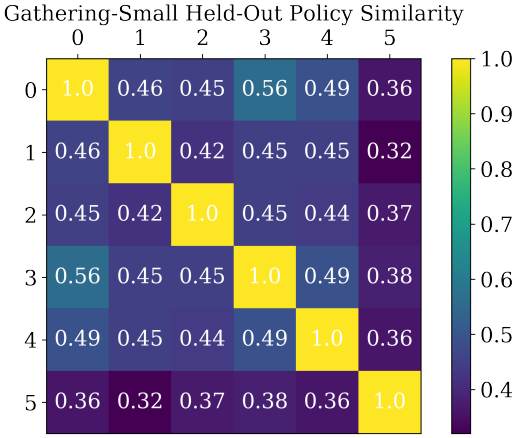


Figure 7: Gathering-Small's held-out policies similarity.

In the Gathering-Open environment, 4,860,000 states were collected and 585,963 states remained after duplicates were removed. The comparison is shown in Figure 8.

In the Leduc Poker environment, 4483 states were collected and 282 states remained after duplicates were removed. The comparison is shown in Figure 9.

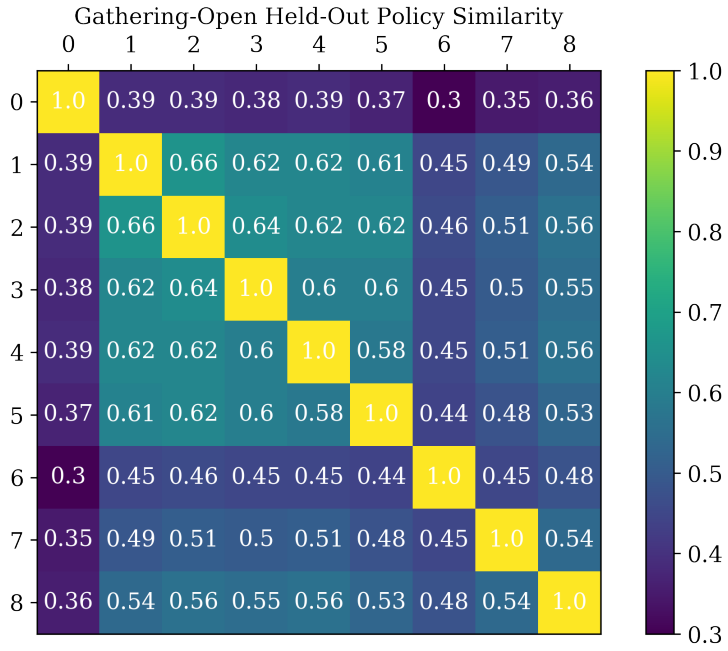


Figure 8: Gathering-Open's held-out policies similarity.

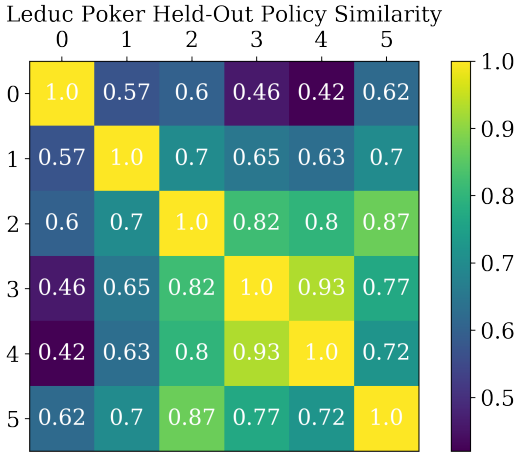


Figure 9: Leduc-Poker's held-out policies similarity.

C.3 SumRegret

$$\text{SumRegret}(\sigma,\overline{\Pi}) = \sum_{i}^{n} \text{Regret}(\sigma_i,\overline{\Pi}_i) \tag{4}$$

D Environments

D.1 Gathering

The *Gathering* environment is a commons game, where the goal of each agent is to collect "apples." The apples regrow at a rate dependent upon the configuration of the uncollected nearby apples. In this case, the more nearby apples, the higher the regrowth rate of the apples. Naturally this presents a dilemma for

the players: each want to pick as many apples as possible; however, if they over-harvest the throughput of apples diminishes – potentially falling to zero.

The original designers of this environment wanted it to be able to study similar phenomena present in human behavioral experiments. An important feature of the human experiments is they often contained the option for a participant to pay a fee to fine another participant. To this end, they endowed each agent with a "time-out beam" (hereafter, laser), which serves this purpose. The laser extends 20 tiles in front of the current agent, and has a width of 5 tiles. If an agent tags another agent with this beam, the taggee is removed from the game for 25 timesteps.

On their quest to collect apples each agent will simultaneously select one of eight possible actions:

$$\{\text{UP, RIGHT, DOWN, LEFT, ROTATE-RIGHT, ROTATE-LEFT, LASER, and NOOP}\}.$$

The first four actions represent moving in the respective cardinal directions from the perspective of the agent. The second set of two actions – ROTATE-RIGHT and ROTATE-LEFT, adjust the perspective of the agent by having them turn 90 degrees in the corresponding direction. This is an important capability of the agent because it allows the agent to aim its laser, which only fires the direction the agent is facing.

The agent's observe a rectangular window of 10 squares forward (including their position), and 10 to the left and right (including their position). The result is a [10, 20] window, where each cell contains: food, agent, opponent, wall, or nothing. This gives our agent an observation shape of [10, 20, 4] which is ravelled into a single vector of length 80. This is processed by a two-hidden layer fully-connected neural network with 50 units at each layer and ReLU activations.

Our implementation is based on the the open source implementation: `https://github.com/HumanCompatibleAI/multi-agent`. We modified this version of the environment to remove an erroneous edge-case where both agents could time-out each other in the same timestep. This led to degenerate solutions where the agents would effectively stun lock each other, so neither player could obtain any reward.

The hyperparameters are shown in Table 2 for Gathering-Small and Table 3 for Gathering-Open.

Hyperparameter	Values Considered	Pure-Hparam	Mix-Hparam
Batch Size	[32, 64]	64	32
Replay Capacity	[3e4, 5e4, 8e5, 1e5]	3e4	8e4
Min Replay Size	[1e2, 1e3, 3e3, 1e4, 3e4, 7e4, 1e5]	1e3	1e4
Learning Rate	[3e-3, 1e-4, 3e-4, 1e-5, 3e-5]	3e-4	3e-4
N Exploration Timesteps	[1e5, 2e5, 3e5, 4e5, 5e5, 7e5]	5e5	3e5
N Total Timesteps	[3e5, 5e5, 7e5, 1e6, 1.2e6, 1.5e6, 1.7e6, 2e6, 2.2e6]	1e6	2.2e6

Table 2: Gathering-Small Hyperparameters.

Hyperparameter	Values Considered	Pure-Hparam	Mix-Hparam
Batch Size	[32, 64]	64	64
Replay Capacity	[3e4, 5e4, 8e4, 1e5]	8e4	8e4
Min Replay Size	[1e2, 1e3, 3e3, 1e4, 3e4, 7e4, 1e5]	1e4	1e4
Learning Rate	[3e-3, 1e-4, 3e-4, 1e-5, 3e-5]	3e-4	3e-5
N Exploration Timesteps	[1e5, 2e5, 3e5, 4e5, 5e5, 7e5]	2e5	1e5
N Total Timesteps	[3e5, 5e5, 7e5, 1e6, 1.2e6, 1.5e6, 1.7e6, 2e6, 2.2e6]	7e5	2.2e6

Table 3: Gathering-Open Hyperparameters.

D.2 LEDUC POKER

Leduc hold'em, or *Leduc Poker*, is a reduced version of poker. In 2-Player Leduc[6] there are six cards: two suits and three ranks. The players go through two rounds of betting where each player can take one

[6]Extensions exist of Leduc to several players.

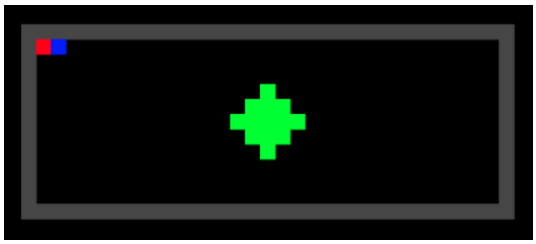
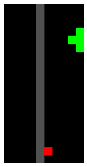


Figure 10: The Gathering-Small environment. Players randomly spawn in either the red or blue (one player per spawn).

(a) Player 0's perspective at the beginning of the episode.



(b) Player 0's perspective after turning right.

Figure 11: Example observation from the Gathering-Default environment. Note that the agents cannot distinguish between their opponents.

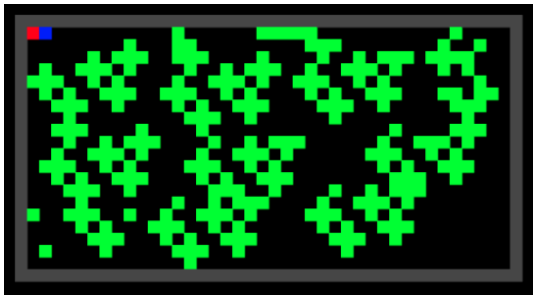


Figure 12: The Gathering-Open environment. Players randomly spawn in one of many locations throughout the map.

of three actions:

$$\{\text{FOLD, CALL, or RAISE.}\}$$

During the first round each player is dealt a single private card, and in the second round a single board card is revealed. The raise amounts are restricted to 2 or 4, and each player starts the first round with 1 already in the pot.

We utilize a two-hidden layer fully-connected neural network with 30 and 15 units each and ReLU activations. The network takes as input a vector of size 30 including:

- The player number (size 2).
- One-hot encoding of their private card (6 bits).
- One-hot encoding of the public card (6 bits).
- Sequence of actions taken in the first round (8 bits – 4 maximum actions with 2 bits to represent each action).
- Sequence of actions taken in the second round (8 bits)

The hyperparameters are shown in Table 4.

Hyperparameter	Values Considered	Pure-Hparam	Mix-Hparam
Batch Size	[32, 64]	32	64
Replay Capacity	[3e2, 1e3, 3e3, 1e4]	1e4	3e3
Min Replay Size	[1e2, 3e2, 1e3]	1e2	1e2
Learning Rate	[1e-3, 3e-3, 1e-4, 3e-4]	1e-3	1e-4
N Exploration Timesteps	[3e2, 1e3, 3e3, 1e4, 3e4, 1e5]	3e2	3e2
N Total Timesteps	[1e3, 3e3, 1e4, 3e4, 1e5, 3e5]	3e3	1e5

Table 4: Leduc Poker Hyperparameters.

E PRECISE VALUES FOR ROCK-PAPER-SCISSORS EXAMPLE

We give the exact policies and Q-values for the Rock-Paper-Scissor example in section 3.2. Player 1's strategies are $\pi_1^1 = (0,0.3,0.7)$ and $\pi_1^2 = (0.4,0.6,0)$. Player 2's BRs are R and P respectively, induced by Q-values $Q_2^1 = (0.7,0.15,0.65)$ and $Q_2^2 = (0.2,0.7,0.6)$. The resulting ENFG has payoffs for player 2 of $[[0.7,0.15],[0.2,0.7]]$. This is close to the matching-pennies game, and player 2's Nash equilibrium is $(0.52,0.48)$, so their meta-strategy plays $(0.52,0.48,0.0)$ in the original game.