

APPENDIX

A COMPARISON OF NASH SOLVERS FOR NORMAL-FORM GAME

In this section we show the Nash equilibrium solvers (*i.e.*, Nash solving subroutine NASH) for zero-sum normal-form games, which is an important subroutine for the proposed algorithms. Since NE for zero-sum normal-form games can be solved by linear programming, some packages involving linear programming functions like ECOS, PuLP and Scipy can be leveraged. We also implement a solver based on an iterative algorithm—multiplicative weights update (MWU), which is detailed in Appendix Sec. A.1.

A.1 MULTIPLICATIVE WEIGHTS UPDATE

The MWU algorithm (Bailey & Piliouras, 2018; Daskalakis & Panageas, 2018) is no-regret in online learning setting, which can be used for solving NE in two-player zero-sum normal-form games. Given a payoff matrix A (from the max-player’s perspective), the NE strategies will be solved by iteratively applying MWU. Specifically, for n -th iteration, state s and actions (a_i, b_j) for the max-player μ and min-player ν respectively, i, j are the entry indices of discrete action space, then the update rule of the action probabilities for two players are:

$$\mu^{(n+1)}(a_i|s) = \mu^{(n)}(a_i|s) \frac{e^{\eta(A\nu^{(n)\tau(s)})_i}}{\sum_{i'} \mu^{(n)}(a_{i'}|s) e^{\eta(A\nu^{(n)\tau(s)})_{i'}}} \quad (15)$$

$$\nu^{(n+1)}(b_j|s) = \nu^{(n)}(b_j|s) \frac{e^{-\eta(A\mu^{(n)\tau(s)})_j}}{\sum_{j'} \nu^{(n)}(b_{j'}|s) e^{-\eta(A\mu^{(n)\tau(s)})_{j'}}} \quad (16)$$

where η is the learning rate. By iteratively updating each action entry of the strategies with respect to the payoff matrix, MWU is provably converging to NE.

A.2 COMPARISON

Table 3: Comparison of different Nash solvers for zero-sum matrix game (6×6 random matrices).

Solver	Time per Sample (s)	Solvability
Nashpy (equilibria) ¹	0.751	all
Nashpy (equilibrium)	0.0016	not for some degenerate matrices
ECOS ²	0.0015	all
MWU (single)	0.008	all (but less accurate)
MWU (parallel)	fast, depends on batch size ³	all (but less accurate)
CVXPY	0.009	all
PuLP	0.020	all
Scipy	—	not for some
Gurobipy	0.01	not for some

¹ Nashpy (<https://github.com/drvinceknight/Nashpy>) can be adopted to achieve two versions of NE solvers: one returns a single NE, and another returns all Nash equilibria for given payoff matrices.

² ECOS (<https://github.com/embotech/ecos>) is a package for solving convex second-order cone programs.

³ MWU solver is self-implemented for solving either a single payoff matrix or solving a batch of matrices in parallel.

⁴ CVXPY (<https://github.com/cvxpy/cvxpy>) is a Python package for convex optimization.

⁵ PuLP (<https://github.com/coin-or/pulp>) is a linear programming package with Python.

⁶ Scipy ([scipy.optimize.linprog\(\)](https://docs.scipy.org/doc/scipy/reference/optimize.linprog.html)) is a general package for numerical operations.

⁷ Gurobipy (<https://www.gurobi.com/>) is a package for linear and quadratic optimization.

We conduct experiments to compare different solvers for NE subroutine, including Nashpy (for single Nash equilibrium or all Nash equilibria), ECOS, MWU (single or parallel), CVXPY, PuLP, Scipy, Gurobipy. The test is conducted on a Dell XPS 15 laptop with only CPU computation. The code will be released after the review process (anonymous during review process). Experiments are evaluated on 6×6 random matrices and averaged over 1000 samples. The zero-sum property of the generated Markov games is guaranteed by generating each random matrix as one player’s payoff and take the negative values as its opponent’s payoff.

As in Table. 3, the solvability indicates whether the solver can solve all possible randomly generated matrices (zero-sum). Nashpy for solving all equilibria cannot handle some degenerate matrices. Scipy and Gurobipy also cannot solve for some payoff matrices. Other solvers can solve all random payoff matrices in our tests but with different solving speed and accuracy. The solvability is essential for the program since the values within the payoff matrix can be arbitrary as a result of applying function approximation. The results support our choice of using the ECOS-based solver as the default Nash solving subroutine for the proposed algorithms, due to its speed and robustness for solving random matrices. ECOS is originally built for solving convex second-order cone programs, which covers linear programming (LP) problem. It tries to transform the input matrices to be Scipy sparse matrices and speeds up the solving procedure. By formulating the NE solving as LP on normal-form game, we can plug in the ECOS solver to get the solution. Some constraints like positiveness and constant sum need to be handled carefully. Other solvers like Nashpy (equilibrium) and MWU (parallel) can achieve a similar level of computational time, but less preferred due to either not being able to solve some matrices or less accurate results. Specifically, for the case with a large batch size and a small number of inner-loop iterations for MWU, MWU can be faster than ECOS. However, the accuracy of MWU depends on the number of iterations for update (Bailey & Piliouras, 2018; Daskalakis & Panageas, 2018). More iterations lead to more accurate approximation but also longer computational time for MWU method. Empirically, we find that the accuracy of the subroutine solver is critical for our proposed algorithms with function approximation, especially in video games with long horizons. Moreover, although we have already selected the best solver through comparisons, the computational time of the solvers used in each inference or update step still account for a considerable portion. This leaves some space for improvement of running-time efficiency.

B ALGORITHMS ON TABULAR MARKOV GAMES

In this section, we provide further details on connections of our algorithms to tabular algorithms NASH_VI and NASH_VI_EXPLOITER in Sec.B.1, and prove the theoretical guarantees of the latter two algorithms in Sec.B.2. We then provide all the pseudo-codes for subroutines and algorithms used in this paper. In particular, we introduce the pseudo-codes for important subroutines in Sec. B.3, and then the pseudo-codes for several algorithms: self-play (SP, B.5), fictitious self-play (FSP, B.6), double oracle (DO, B.7), Nash value iteration (NASH_VI, B.8) and Nash value iteration with exploiter (NASH_VI_EXPLOITER, B.9). SP, FSP and DO are the baseline methods in experimental comparisons, while NASH_VI and NASH_VI_EXPLOITER are the tabular version of our proposed algorithms NASH_DQN and NASH_DQN_EXPLOITER, without function approximation.

B.1 CONNECTIONS OF NASH_DQN, NASH_DQN_EXPLOITER TO TABULAR ALGORITHMS

We first note that the ϵ -greedy version of NASH_VI and NASH_VI_EXPLOITER algorithms (as shown in Section B.8, B.9), are simply the optimistic Nash-VI algorithm in Liu et al. (2021) and GOLF_WITH_EXPLOITER algorithm in Jin et al. (2021b) when applied to the tabular setting, with optimistic exploration replaced by ϵ -greedy exploitation.

Comparing our algorithms NASH_DQN (algorithm 1) and NASH_DQN_EXPLOITER (algorithm 11) with the ϵ -greedy version of NASH_VI (Algorithm 9) and NASH_VI_EXPLOITER (Algorithm 10), we notice that, besides the minor difference between episodic setting versus infinite horizon discounted setting, the latter two algorithms are special cases of the former two algorithms when

1. specialize the neural network structure to represent a table of values for each state-action pairs (i.e. specialize both algorithms to the tabular setting);
2. let the minibatch \mathcal{M} to contain all previous data (i.e., use the full batch \mathcal{D});
3. let the number of gradient step m to be sufficiently large so that GD finds the minimizer of the objective function;
4. let $N = 1$, that is update the target network at every iterations.

We remark that the use of small minibatch size, and small gradient steps are to speed up the training in practice beyond tabular settings. The delay update of the target networks is used to stabilize the training process.

B.2 PROOF OF THEOREM 1

The result of optimistic Nash-VI algorithm in Liu et al. (2021), and the result of GOLF_WITH_EXPLOITER algorithm in Jin et al. (2021b) (when specialized to the tabular set-

ting) already prove that both optimistic versions of NASH_VI and NASH_VI_EXPLOITER can find ϵ -approximate Nash equilibria for **episodic** Markov games in $\text{poly}(S, A, B, H, \epsilon^{-1}, \log(1/\delta))$ steps with probability at least $1 - \delta$. Here H is the horizon length of the episodic Markov games.

To convert the episodic results to the infinite-horizon discounted setting in this paper, we can simply truncate the infinite-horizon games up to $H = \frac{1}{1-\gamma} \ln \frac{2}{(1-\gamma)\epsilon}$ steps so that the remaining cumulative reward is at most

$$\sum_{h=H}^{\infty} \gamma^h = \frac{\gamma^H}{1-\gamma} \leq \frac{e^{-(1-\gamma)H}}{1-\gamma} \leq \frac{\epsilon}{2}$$

which is smaller than the target accuracy. To further address the non-stationarity of the value and policy in the the episodic setting (which requires both value and policy to depends on not only the state, but also the steps), we can augment the state space s to (s, h) to include step information (up to the truncation point H) in the state space. Now, we are ready to apply the episodic results to the infinite horizon discounted setting, which shows that both optimistic versions of NASH_VI and NASH_VI_EXPLOITER can find ϵ -approximate Nash equilibria for **infinite-horizon discounted** Markov games in $\text{poly}(S, A, B, (1-\gamma)^{-1}, \epsilon^{-1}, \log(1/\delta))$ steps with probability at least $1 - \delta$. Here γ is the discount coefficient.

B.3 SUBROUTINES

Algorithm 2 META_NASH: Meta-Nash Equilibrium Solving Subroutine

- 1: **Input** two strategy sets μ, ν ; evaluation iterations N
 - 2: Initialize payoff matrix: $M_{i,j} = 0, i \in [|\mu|], j \in [|\nu|]$
 - 3: **for** $\mu_i \in \mu$ **do**
 - 4: **for** $\nu_j \in \nu$ **do**
 - 5: **for** episodes $k = 1, \dots, N$ **do**
 - 6: Rollout policies μ_i, ν_j to get episodic reward r_k
 - 7: $M_{i,j} = \frac{1}{N} \sum_{k=1}^N r_k$
 - 8: $(\rho_\mu, \rho_\nu) = \text{NASH}(M)$
 - 9: **Return** ρ_μ or ρ_ν
-

Before introducing the pseudo-code for each algorithm, we summarize several subroutines – NASH, META_NASH, BEST_RESPONSE and BEST_RESPONSE_VALUE – applied in the algorithms. These subroutines are marked in magenta color in the this and the following sections.

NASH: As a NE solving subroutine for normal-form games, it returns the NE strategy given the payoff matrix as the input. Specifically it uses the solvers introduced in Appendix Sec. A, and ECOS is the default choice in our experiments.

META_NASH: As a meta-Nash solving subroutine (Algorithm 2), it returns the one-side meta NE strategy given two strategy sets: $\mu = \{\mu_1, \dots, \mu_i, \dots\}, \nu = \{\nu_1, \dots, \nu_i, \dots\}$. A one-by-one matching for each pair of polices $(\mu_i, \nu_j), i \in [|\mu|], j \in [|\nu|]$ is evaluated in the game to get an estimated payoff matrix, with the average episodic return as the estimated payoff values of two players for each entry in the payoff matrix. The NASH subroutine is called to solve the meta-Nash strategies. It is applied in DO algorithm, which is detailed in Sec. B.7.

BEST_RESPONSE: As a best response subroutine, it returns the best response strategy of the given strategy, which satisfies Eq. (4). To be noticed, the best response we discuss here is the best response of a meta-distribution ρ_μ over a strategy set $\{\mu^0, \mu^1, \dots, \mu^n\}$, which covers the case of best response against a single strategy by just making the distribution one-hot. We use this setting for the convenience of being applied in SP, FSP, DO algorithms. Here we discuss two types of best response subroutine that are used at different positions in the algorithms: (1) **BEST_RESPONSE I** (as Algorithm 3) is a best response subroutine with oracle transition and reward function of the game, which is used for evaluating the exploitability of the model after training; (2) **BEST_RESPONSE II** (as Algorithm 4) is a best response subroutine with Q -learning agent for approximating the best response, without knowing the true transition and reward function of the game. It is used in the procedure of methods based on iterative best response, like SP, FSP, DO. We claim here for the following sections, by default, **BEST_RESPONSE** will use **BEST_RESPONSE II**, and **BEST_RESPONSE_VALUE** will use **BEST_RESPONSE I**.

BEST_RESPONSE_VALUE: It has the same procedure as **BEST_RESPONSE** as a best response subroutine, but returns the average value of the initial states as $V_1^{\mu, \dagger}(s_1)$ in Eq. (7) with the given

strategy $\hat{\mu}$. Since the best response value estimation is used in evaluating the exploitability of a certain strategy, it by default adopts **BEST_RESPONSE I** (Algorithm 3) as an oracle process, which returns the ground-truth best response values because of knowing the transition and reward functions.

Algorithm 3 BEST_RESPONSE I: Best Response Subroutine in Markov Game (known transition, reward functions)

- 1: **Input** mixture policy ρ_μ as a distribution over $\{\mu^0, \mu^1, \dots, \mu^n\}$
- 2: Initialize non-Markovian policies $\hat{\mu} = \{\hat{\mu}_h\}, \hat{\nu} = \{\hat{\nu}_h\}, h \in [H], \mu_h : (\mathcal{S} \times \mathcal{A} \times \mathcal{B})^{(h-1)} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1], \nu_h : (\mathcal{S} \times \mathcal{A} \times \mathcal{B})^{(h-1)} \times \mathcal{S} \times \mathcal{B} \rightarrow [0, 1]$
- 3: Initialize Q table for non-Markovian policies $\hat{\mu}, \hat{\nu}, Q = \{Q_h\}, h \in [H], Q_h : (\mathcal{S} \times \mathcal{A} \times \mathcal{B})^h \rightarrow [0, 1]$
- 4: Initialize V table for non-Markovian policies $\hat{\mu}, \hat{\nu}, V = \{V_h\}, h \in [H], V_h : (\mathcal{S} \times \mathcal{A} \times \mathcal{B})^{(h-1)} \times \mathcal{S} \rightarrow [0, 1]$
- 5: **for** $h = 1, \dots, H$ **do**
- 6: **For all** τ_{h-1} :

$$Q_h^{\hat{\mu}, \dagger}(\tau_{h-1}, s_h, a_h, b_h) = \sum_{s' \in \mathcal{S}} \mathbb{P}_h(s_{h+1} | s_h, a_h, b_h) [r_h(s_h, a_h, b_h) + V_{h+1}^{\hat{\mu}, \dagger}(\tau_h, s_{h+1})] \quad (17)$$

$$V_h^{\hat{\mu}, \dagger}(\tau_{h-1}, s_h) = \min_{\nu_h} \hat{\mu}_h(\cdot | \tau_{h-1}, s_h) Q_h^{\hat{\mu}, \dagger}(\tau_{h-1}, s_h, \cdot, \cdot) \nu_h^\top(\cdot | \tau_{h-1}, s_h) \quad (18)$$

$$\hat{\nu}_h(\tau_{h-1}, s_h) = \arg \min_{\nu_h} \hat{\mu}_h(\cdot | \tau_{h-1}, s_h) Q_h^{\hat{\mu}, \dagger}(\tau_{h-1}, s_h, \cdot, \cdot) \nu_h^\top(\cdot | \tau_{h-1}, s_h) \quad (19)$$

where

$$\hat{\mu}_h(a_h | \tau_{h-1}, s_h) := \frac{\sum_i \mu_h^i(a_h | s_h) \rho(i) \prod_{t'=1}^{h-1} \mu_{t'}^i(a_{t'} | s_{t'})}{\sum_j \rho(j) \prod_{t'=1}^{h-1} \mu_{t'}^j(a_{t'} | s_{t'})}$$

- 7: **Return** $\hat{\nu}$ or $V_1^{\hat{\mu}, \dagger}(s_1)$
% $\hat{\mu}$ is the posterior policy of non-Markovian mixture μ , $\hat{\nu}$ is the best response of it
-

Algorithm 4 BEST_RESPONSE II: Best Response Subroutine in Markov Game (Q -learning based, unknown transition, reward functions)

- 1: **Input** mixture policy ρ_μ as a distribution over $\{\mu^0, \mu^1, \dots, \mu^n\}$; best response Q -learning iterations N ; soft update coefficient α
 - 2: Initialize the $Q = \{Q_h | h \in [H]\} \in \mathbb{R}^{|\mathcal{S}| \times |\mathcal{B}|}$ table for the best response player,
 - 3: **for** episodes $k = 1, \dots, N$ **do**
 - 4: Sample policy $\mu_k \sim \rho_\mu$
 - 5: **for** $t = 1, \dots, H$ **do**
 - 6: *% collect data*
 - 7: Sample greedy action $a_t \sim \mu_k(\cdot | s_t)$
 - 8: With ϵ probability, sample random action b_t ;
 - 9: Otherwise, sample greedy action $b_t \sim \nu(\cdot | s_t)$ according to Q
 - 10: Rollout environment to get sample $(s_t, a_t, b_t, r_t, \text{done}, s_{t+1})$ (r_t is for the learning player)
 - 11: *% update best response Q-value*
 - 12: **if not done then**
 - 13: $Q_t^{\text{target}}(s_t, b_t) = r_t + V_{t+1}(s_{t+1})$
 - 14: where $V_{t+1}(s_{t+1}) = \max_{b'} Q_{t+1}(s_{t+1}, b')$
 - 15: **else**
 - 16: $Q_t^{\text{target}}(s_t, b_t) = r_t$
 - 17: $Q_t(s_t, b_t) \leftarrow \alpha \cdot Q_t^{\text{target}}(s_t, b_t) + (1 - \alpha) \cdot Q_t(s_t, b_t)$
 - 18: **if done then**
 - 19: break
 - 20: Represent Q as a greedy policy $\hat{\nu}$
 - 21: **Return** $\hat{\nu}$
-

B.4 NASH Q-LEARNING

The pseudo-code for Nash Q-Learning is shown in Algorithm.5 below.

Algorithm 5 Nash Q-Learning

```

Initialize  $Q : \mathcal{S} \times \mathcal{A} \times \mathcal{B} \rightarrow \mathbb{R}$ , given  $\epsilon, \gamma, \alpha$ .
for  $k = 1, \dots, K$  do
  for  $t = 1, \dots, H$  do
    % collect data
    With  $\epsilon$  probability, sample random actions  $a_t, b_t$ ;
    Otherwise,  $a_t \sim \mu(\cdot|s_t), b_t \sim \nu(\cdot|s_t), (\mu(\cdot|s_t), \nu(\cdot|s_t)) = \text{NASH}(Q(s_t, \cdot, \cdot))$ 
    Rollout environment to get sample  $(s_t, a_t, b_t, r_t, \text{done}, s_{t+1})$ 
    % update Q-value
    if not done then
      Compute  $(\hat{\mu}, \hat{\nu}) = \text{NASH}(Q(s_{t+1}, \cdot, \cdot))$ 
      Set  $Q^{\text{target}}(s_t, a_t, b_t) = r_t + \gamma \hat{\mu}^\top Q(s_{t+1}, \cdot, \cdot) \hat{\nu}$ .
    else
      Set  $Q^{\text{target}}(s_t, a_t, b_t) = r_t$ 
       $Q(s_t, a_t, b_t) \leftarrow \alpha \cdot Q^{\text{target}}(s_t, a_t, b_t) + (1 - \alpha) \cdot Q(s_t, a_t, b_t)$ 
    if done then
      break
  
```

B.5 SELF-PLAY

The pseudo-code for self-play is shown in Algorithm 6.

Algorithm 6 Self-play for Markov Game

```

1: Initialize policies  $\mu^0 = \{\mu_h\}, \nu^0 = \{\nu_h\}, h \in [H]$ 
2: Initialize policy sets:  $\mu = \{\mu^0\}, \nu = \{\nu^0\}$ 
3: Initialize meta-strategies:  $\rho_\mu = [1.], \rho_\nu = [1.]$ 
4: for  $t = 1, \dots, T$  do
5:   if  $t \% 2 == 0$  then
6:      $\nu^t = \text{BEST\_RESPONSE}(\rho_\mu, \mu)$ 
7:      $\nu = \nu \cup \{\nu^t\}$ 
8:      $\rho_\nu = (0, \dots, 1)$  as a one-hot vector with only 1 for the last entry
9:   else
10:     $\mu^t = \text{BEST\_RESPONSE}(\rho_\nu, \nu)$ 
11:     $\mu = \mu \cup \{\mu^t\}$ 
12:     $\rho_\mu = (0, \dots, 1)$  as a one-hot vector with only 1 for the last entry
13:    exploitability =  $\text{BEST\_RESPONSE\_VALUE}(\rho_\mu, \mu) + \text{BEST\_RESPONSE\_VALUE}(\rho_\nu, \nu)$ 
14: Return  $\mu, \nu$ 
  
```

B.6 FICTITIOUS SELF-PLAY

The pseudo-code for fictitious self-play is shown in Algorithm.7. We use $\text{uniform}(\cdot)$ to denote a uniform distribution over the policy set.

B.7 DOUBLE ORACLE

The pseudo-code for double oracle is shown in Algorithm.8.

B.8 NASH_VI

The pseudo-code for Nash value iteration (NASH_VI) is shown in Algorithm 9. Different from NASH_DQN (as Algorithm 1), for tabular Markov games, the Q network is changed to be the Q table and updated in a tabular manner (as Algorithm 9 line 13), given the estimated transition function $\tilde{\mathbb{P}}$ and reward function \tilde{r} . The target Q is not used. Since NASH_VI is applied for tabular Markov games, here we write the pseudo-code in an episodic setting without the reward discount factor, which is slightly different from Sec. 3.1.

Algorithm 7 Fictitious Self-play for Markov Game

```

1: Initialize policies  $\mu^0 = \{\mu_h\}, \nu^0 = \{\nu_h\}, h \in [H]$ 
2: Initialize policy sets:  $\mu = \{\mu^0\}, \nu = \{\nu^0\}$ 
3: Initialize meta-strategies:  $\rho_\mu = [1.], \rho_\nu = [1.]$ 
4: for  $t = 1, \dots, T$  do
5:   if  $t \% 2 == 0$  then
6:      $\nu^t = \text{BEST\_RESPONSE}(\rho_\mu, \mu)$ 
7:      $\nu = \nu \cup \{\nu^t\}$ 
8:      $\rho_\nu = \text{Uniform}(\nu)$ 
9:   else
10:     $\mu^t = \text{BEST\_RESPONSE}(\rho_\nu, \nu)$ 
11:     $\mu = \mu \cup \{\mu^t\}$ 
12:     $\rho_\mu = \text{Uniform}(\mu)$ 
13:    exploitability =  $\text{BEST\_RESPONSE\_VALUE}(\rho_\mu, \mu) + \text{BEST\_RESPONSE\_VALUE}(\rho_\nu, \nu)$ 
14: Return  $\mu, \rho_\mu, \nu, \rho_\nu$ 

```

Algorithm 8 Double Oracle for Markov Game

```

1: Initialize policies  $\mu^0 = \{\mu_h\}, \nu^0 = \{\nu_h\}, h \in [H]$ 
2: Initialize policy sets:  $\mu = \{\mu^0\}, \nu = \{\nu^0\}$ 
3: Initialize meta-strategies:  $\rho_\mu = [1.], \rho_\nu = [1.]$ 
4: for  $t = 1, \dots, T$  do
5:   if  $t \% 2 == 0$  then
6:      $\nu^t = \text{BEST\_RESPONSE}(\rho_\mu, \mu)$ 
7:      $\nu = \nu \cup \{\nu^t\}$ 
8:      $\rho_\nu = \text{META\_NASH}(\nu, \mu)$ 
9:   else
10:     $\mu^t = \text{BEST\_RESPONSE}(\rho_\nu, \nu)$ 
11:     $\mu = \mu \cup \{\mu^t\}$ 
12:     $\rho_\mu = \text{META\_NASH}(\nu, \mu)$ 
13:    exploitability =  $\text{BEST\_RESPONSE\_VALUE}(\rho_\mu, \mu) + \text{BEST\_RESPONSE\_VALUE}(\rho_\nu, \nu)$ 
14: Return  $\mu, \rho_\mu, \nu, \rho_\nu$ 

```

Algorithm 9 Nash Value Iteration (NASH_VI, ϵ -greedy sample version)

```

1: Initialize  $Q = \{Q_h\}, h \in [H], Q_h : \mathcal{S}_h \times \mathcal{A}_h \times \mathcal{B}_h \rightarrow \mathbb{R}$ , buffer  $\mathcal{D} = \phi$ , given  $\epsilon$ , update interval  $p$ .
2: for  $k = 1, \dots, K$  do
3:   for  $t = 1, \dots, H$  do
4:     % collect data
5:     With  $\epsilon$  probability, sample random actions  $a_t, b_t$ ;
6:     Otherwise,  $a_t \sim \mu_t(\cdot | s_t), b_t \sim \nu_t(\cdot | s_t), (\mu_t(\cdot | s_t), \nu_t(\cdot | s_t)) = \text{NASH}(Q_t(s_t, \cdot, \cdot))$ .
7:     Rollout environment to get sample  $(s_t, a_t, b_t, r_t, \text{done}, s_{t+1})$  and store in  $\mathcal{D}$ .
8:     % update Q-value
9:     if  $|\mathcal{D}| \% p = 0$  then
10:      for  $\forall (s, a, b, s') \in \mathcal{S}_h \times \mathcal{A}_h \times \mathcal{B}_h \times \mathcal{S}_{h+1}, h \in [H]$  do
11:        Estimate  $\tilde{\mathbb{P}}_h(s_{h+1} = s' | s_h = s, a_h = a, b_h = b) = \frac{1}{n} \sum_{i=1}^n \mathbb{1}(s_{h+1} = s'_i), (s, a, b, s'_i) \in \mathcal{D}$ .
12:        Estimate  $\tilde{r}_h(s_h = s, a_h = a, b_h = b) = \frac{1}{m} \sum_{i=1}^m r_i(s, a, b), (s, a, b, r_i) \in \mathcal{D}$ .
13:         $Q_h(s, a, b) = \tilde{r}_h(s, a, b) + (\tilde{\mathbb{P}}_h V_{h+1}^{\hat{\mu}_{h+1}, \hat{\nu}_{h+1}})(s, a, b) \cdot \mathbb{I}[s' \text{ is non-terminal}]$ ,
14:        where  $(\hat{\mu}_{h+1}, \hat{\nu}_{h+1}) = \text{NASH}(Q_{h+1})$ .
15:      if done then
16:        break

```

B.9 NASH_VI_EXPLOITER

The pseudo-code for Nash value iteration with Exploiter (NASH_VI_EXPLOITER) is shown in Algorithm.10. Different from NASH_DQN_EXPLOITER (as Algorithm 11), for tabular Markov games, the Q network and exploiter \tilde{Q} network are changed to be Q tables and updated in a tabular manner (as Algorithm 10 line 14 and line 16), given the estimated transition function $\tilde{\mathbb{P}}$ and reward function \tilde{r} . The target Q and target \tilde{Q} are not used. Since NASH_VI_EXPLOITER is applied for tabular Markov games, here we write the pseudo-code in an episodic setting without the reward discount factor, which is slightly different from Sec. 3.2.

Algorithm 10 Nash Value Iteration with Exploiter (NASH_VI_EXPLOITER, ϵ -greedy sample version)

```

1: Initialize  $Q = \{Q_h\}, \tilde{Q} = \{\tilde{Q}_h\}, h \in [H], \tilde{Q}_h, Q_h : \mathcal{S}_h \times \mathcal{A}_h \times \mathcal{B}_h \rightarrow \mathbb{R}$ , buffer  $\mathcal{D} = \phi$ , given
    $\epsilon$ , update interval  $p$ .
2: for  $k = 1, \dots, K$  do
3:   for  $t = 1, \dots, H$  do
4:     % collect data
5:     With  $\epsilon$  probability, sample random actions  $a_t, b_t$ ;
6:     Otherwise,  $a_t \sim \mu_t(\cdot|s_t), b_t \sim \tilde{\nu}_t(\cdot|s_t)$ ,
7:      $(\mu_t(\cdot|s_t), \nu_t(\cdot|s_t)) = \text{NASH}(Q(s_t, \cdot, \cdot), \tilde{\nu}_t(\cdot|s_t) = \arg \min_{\nu} \mu_t^\top(\cdot|s_t) \tilde{Q}_t(s_t, \cdot, \cdot) \nu$ .
8:     Rollout environment to get sample  $(s_t, a_t, b_t, r_t, \text{done}, s_{t+1})$  and store in  $\mathcal{D}$ .
9:     % update Q-value
10:    if  $|\mathcal{D}| \% p = 0$  then
11:      for  $\forall (s, a, b, s') \in \mathcal{S}_h \times \mathcal{A}_h \times \mathcal{B}_h \times \mathcal{S}_{h+1}, h \in [H]$  do
12:        Estimate  $\tilde{\mathbb{P}}_h(s_{h+1} = s' | s_h = s, a_h = a, b_h = b) = \frac{1}{n} \sum_{i=1}^n \mathbb{1}(s_{h+1} = s'_i), (s, a, b, s'_i) \in \mathcal{D}$ .
13:        Estimate  $\tilde{r}_h(s_h = s, a_h = a, b_h = b) = \frac{1}{m} \sum_{i=1}^m r_i(s, a, b), (s, a, b, r_i) \in \mathcal{D}$ .
14:         $Q_h(s, a, b) = \tilde{r}_h(s, a, b) + (\tilde{\mathbb{P}}_h V_{h+1}^{\hat{\mu}_{h+1}, \hat{\nu}_{h+1}})(s, a, b) \cdot \mathbb{I}[s' \text{ is non-terminal}]$ ,
15:        where  $(\hat{\mu}_{h+1}, \hat{\nu}_{h+1}) = \text{NASH}(Q_{h+1})$ .
16:         $\tilde{Q}_h(s, a, b) = \tilde{r}_h(s, a, b) + (\tilde{\mathbb{P}}_h V_{h+1}^{\text{Exploit}})(s, a, b)$ ,
17:        where  $V_{h+1}^{\text{Exploit}}(s') = \begin{cases} \min_{b' \in \mathcal{B}_{h+1}} \hat{\mu}_{h+1}(s')^\top \tilde{Q}_{h+1}(s', \cdot, b') & \text{for non-terminal } s' \\ 0 & \text{for terminal } s' \end{cases}$ .
18:      if done then
19:        break

```

B.10 NASH_DQN_EXPLOITER

The pseudo-code for double oracle is shown in Algorithm.11.

B.11 COMPARISONS OF NASH_VI, NASH Q-LEARNING, GOLF_WITH_EXPLOITER AND NASH_DQN

We will detail the essential similarities and differences of the four algorithms NASH_VI, Nash Q-Learning, GOLF_WITH_EXPLOITER and NASH_DQN from four aspects: model-based/model-free, update manner, replay buffer, and exploration method.

- NASH_VI: model-based; update using full batch, no soft update; there is a buffer containing all samples so far; ϵ -greedy exploration.
- Nash Q-Learning: model-free; update using stochastic gradient for each sample, using soft update $Q \leftarrow (1 - \alpha)Q + \alpha Q_{\text{target}}$; no replay buffer; ϵ -greedy exploration.
- GOLF_WITH_EXPLOITER: model-based; using an optimistic way of updating policy and exploiter within a confidence set; there is a buffer containing all samples so far; a different behavior policy for exploration compared with ϵ -greedy exploration.
- NASH_DQN: model-free; minibatch stochastic gradient update, using Mean Squared Error(Q, Q_{target}) for gradient-based update; there is a buffer containing all samples so far; ϵ -greedy exploration.

Algorithm 11 Nash Deep Q-Network with Exploiter (NASH_DQN_EXPLOITER)

```

1: Initialize replay buffer  $\mathcal{D} = \emptyset$ , counter  $i = 0$ , Q-network  $Q_\phi$ , exploiter network  $\tilde{Q}_\psi$ .
2: Initialize target network parameters:  $\phi^{\text{target}} \leftarrow \phi$ ,  $\psi^{\text{target}} \leftarrow \psi$ .
3: for episode  $k = 1, \dots, K$  do
4:   reset the environment and observe  $s_1$ .
5:   for  $t = 1, \dots, H$  do
6:     % collect data
7:     sample actions  $(a_t, b_t)$  from  $\begin{cases} \text{Uniform}(\mathcal{A} \times \mathcal{B}) & \text{with probability } \epsilon \\ (\mu_t, \nu_t) \text{ computed according to (12)} & \text{otherwise.} \end{cases}$ 
8:     execute actions  $(a_t, b_t)$ , observe reward  $r_t$ , next state  $s_{t+1}$ .
9:     store data sample  $(s_t, a_t, b_t, r_t, s_{t+1})$  into  $\mathcal{D}$ .
10:    % update Q-network and exploiter network
11:    randomly sample minibatch  $\mathcal{M} \subset \{1, \dots, |\mathcal{D}|\}$ .
12:    for all  $j \in \mathcal{M}$  do
13:      compute  $(\hat{\mu}, \hat{\nu}) = \text{NASH}(Q_{\phi^{\text{target}}}(s_{j+1}, \cdot, \cdot))$ 
14:      set  $y_j = r_j + \gamma \hat{\mu}^\top Q_{\phi^{\text{target}}}(s_{j+1}, \cdot, \cdot) \hat{\nu}$ .
15:      set  $\tilde{y}_j = r_j + \gamma \min_{b \in \mathcal{B}} \hat{\mu}^\top \tilde{Q}_{\psi^{\text{target}}}(s_{j+1}, \cdot, b)$ 
16:      Perform  $m_1$  steps of GD on loss  $\sum_{j \in \mathcal{M}} (y_j - Q_\phi(s_j, a_j, b_j))^2$  to update  $\phi$ .
17:      Perform  $m_2$  steps of GD on loss  $\sum_{j \in \mathcal{M}} (\tilde{y}_j - \tilde{Q}_\psi(s_j, a_j, b_j))^2$  to update  $\psi$ .
18:    % update target network
19:     $i = i + 1$ ; if  $i \% N = 0$ :  $\phi^{\text{target}} \leftarrow \phi$ ,  $\psi^{\text{target}} \leftarrow \psi$ .

```

From these similarities and differences, we can see that three theoretical algorithms Nash-VI, Nash Q-learning and GOLF-with-exploiter have slight differences in details, Nash-DQN can be viewed as practical approximation of both Nash-VI and Nash Q-learning.

C HYPERPARAMETERS FOR TABULAR MARKOV GAME.

Table 4: Hyperparameters in Tabular Markov Game.

	Hyperparameter	Values
Common	Learning rate	1×10^{-4}
	Optimizer	Adam
	Batch size	640
	Replay Buffer Size	10^5
	Episodes	50000
	Episode Length	3 for I / 6 for II
	Hidden Dimension	128
	Hidden Activation	ReLU
	Hidden Layers	3
	Target Update Interval	1000
	γ	1.0
	ϵ Exponential Decay	$\epsilon_0 = 1.0, \epsilon_1 = 0.0, p = 8000$
SP	δ	1.5 for I / 2.0 for II
FSP	δ	1.5 for I / 2.0 for II
NFSP	η	0.1
PSRO	δ	1.5 for I / 2.0 for II
Nash DQN Exploiter	Exploiter Update Ratio m_2/m_1	1

This section provides detailed hyperparameters of methods **with** function approximation on the tabular Markov games, as shown in Table 4. Methods such as SP, FSP, NFSP, PSRO all use DQN as the basic RL agent, and the ‘‘Common’’ hyperparameters are applied on the DQN algorithm. For NASH_DQN and NASH_DQN_EXPLOITER, since the algorithms follows a similar routine as DQN in general (value-based, off-policy), they also adopt the same hyperparameters.

In the common hyperparameters, the basic agent applies a network with 3 hidden layers and 128 as hidden dimension. The target update interval is the delayed update of the target network, and it updates once per n times of standard network updates. n is specified by the target update interval value, and γ is the reward discounting coefficient, and it’s set to be one in tabular Markov games since the episode length (3 for I or 6 for II) is small in the experiments. ϵ is the factor in ϵ -greedy exploration and it follows an exponential decay schedule. Specifically, its value follows $\epsilon(t) = \epsilon_1 + (\epsilon_0 - \epsilon_1)e^{-t/p}$ in our experiments, where t is the timestep in update.

Since SP, FSP, PSRO algorithms follow an iterative best response procedure in the learning process, the margin for determining whether the current updating side achieves an approximate best response is set according to the average episodic reward. Once a learning agent wins over its opponent by a average reward threshold δ (depending on games), it saves the approximate best-response strategy and transfers the role to its opponent. The values of δ are different for the two tabular environments I and II, since the ranges of the return are different for the two environments. The longer horizon indicates a potentially larger range of return.

For NFSP, since actions can be sampled from either ϵ -greedy policy or the average policy, η is a hyperparameter representing the ratio of choosing actions from ϵ -greedy policy.

For NASH_DQN_EXPLOITER, the exploiter update ratio is the times of GD for the exploiter over the times GD for updating the Nash Q network, which is m_2/m_1 as in Algorithm 11.

The exploiter for exploitation test after model training is a DQN agent with exactly the same “Common” hyperparameters for the tabular Markov game test.

D RESULTS FOR TABULAR MARKOV GAMES

Table 5 shows the exploiter rewards in the exploitation test on two tabular Markov games. The exploiter reward is an approximation of $-V^{\hat{\mu}, \dagger}(s_1)$ by Eq. (7). The last column “Nash V” is the true value of $V^{\mu^*, \nu^*}(s_1)$. $V^{\mu^*, \nu^*}(s_1) - V^{\hat{\mu}, \dagger}(s_1)$ gives the true exploitability for $\hat{\mu}$. Due to the randomly generated payoff matrix, it is asymmetric for the two players: $V^{\mu^*, \nu^*}(s_1) \approx -0.296$ for environment I and $V^{\mu^*, \nu^*}(s_1) \approx -0.131$ for environment II. For example, in Table 5 the mean of approximate exploitability of Oracle Nash is $0.269 - 0.296 = -0.027$, and the theoretical value should be zero since it is the ground truth Nash equilibrium strategy. Also, since the transition and reward is assumed to be unknown and the exploitability is approximated with a DQN agent, the stochasticity of the results is larger than the tabular method test. Table 6 shows the exploitation results at different training stages for environment I. At step 0, some methods have common results since the initialized models are the same for them.

Table 5: Exploiter rewards in tabular case: the reward means and standard deviations are derived over 1000 episodes in exploitation test.

Env / Method	SP	FSP	NFSP	PSRO	Nash DQN	Nash DQN Exploiter	Oracle Nash	Nash V
Tabular Env I	0.744 ± 0.711	0.675 ± 0.811	0.613 ± 0.858	0.430 ± 1.110	0.392 ± 1.044	0.316 ± 0.998	0.269 ± 1.029	-0.296
Tabular Env II	1.370 ± 1.323	0.825 ± 1.343	0.510 ± 1.408	0.700 ± 1.337	0.148 ± 1.341	0.202 ± 1.425	0.049 ± 1.365	-0.131

Table 6: Exploiter rewards in tabular Markov game I after training for 0, 10k, 20k, 30k, 40k, 50k episodes and exploitation for 30k episodes. The reward means and standard deviations are derived over 1000 episodes in exploitation test.

Method	Exploiter Reward					
	0	10k	20k	30k	40k	50k
SP	1.114 ± 0.776	1.133 ± 0.764	1.002 ± 0.786	0.749 ± 0.907	0.885 ± 0.837	0.744 ± 0.711
FSP		0.724 ± 0.890	0.698 ± 0.808	0.690 ± 0.783	0.709 ± 0.801	0.675 ± 0.811
NFSP		0.607 ± 0.982	0.604 ± 0.887	0.610 ± 0.951	0.500 ± 0.933	0.613 ± 0.858
PSRO		0.669 ± 0.873	0.650 ± 0.829	0.714 ± 0.790	0.712 ± 0.913	0.430 ± 1.110
Nash DQN	1.214 ± 0.860	0.405 ± 1.003	0.392 ± 1.040	0.387 ± 1.003	0.411 ± 1.040	0.392 ± 1.044
Nash DQN Exploiter		0.301 ± 0.978	0.297 ± 1.056	0.273 ± 1.034	0.291 ± 0.945	0.316 ± 0.998
Oracle Nash						0.269 ± 1.029

E HYPERPARAMETERS FOR TWO-PLAYER VIDEO GAMES.

This section provides detailed hyperparameters of methods **with** function approximation on the tabular Markov games, as shown in Table 7 for *SlimeVolley* environment and Table 8 for two-player

Table 7: Hyperparameters in SlimeVolley.

	Hyperparameter	Values
Common	Learning rate	1×10^{-4}
	Optimizer	Adam
	Batch size	128
	Replay Buffer Size	10^5
	Episodes	50000
	Episode Length	≤ 300
	Hidden Dimension	128
	Hidden Layers	3
	Hidden Activation	ReLU
	Target Update Interval	1000
	γ	0.99
	ϵ Exponential Decay	$\epsilon_0 = 1.0, \epsilon_1 = 1 \times 10^{-3}, p = 1 \times 10^5$
SP	δ	3.0
FSP	δ	3.0
NFSP	η	0.1
PSRO	δ	3.0
Nash DQN	ϵ Exponential Decay	$\epsilon_0 = 1.0, \epsilon_1 = 1 \times 10^{-3}, p = 5 \times 10^6$
Nash DQN Exploiter	ϵ Exponential Decay	$\epsilon_0 = 1.0, \epsilon_1 = 1 \times 10^{-3}, p = 5 \times 10^6$
	Exploiter Update Ratio m_2/m_1	1

Table 8: Hyperparameters in two-player Atari games.

	Hyperparameter	Values
Common	Learning rate	1×10^{-4}
	Optimizer	Adam
	Batch size	128
	Replay Buffer Size	10^5
	Episodes	50000
	Episode Length	≤ 300
	Hidden Dimension	128
	Hidden Layers	4
	Hidden Activation	ReLU
	Target Update Interval	1000
	γ	0.99
	ϵ Exponential Decay	$\epsilon_0 = 1.0, \epsilon_1 = 1 \times 10^{-3}, p = 1 \times 10^5$
SP	δ	80/15/10/7/3
FSP	δ	80/15/10/7/3
NFSP	η	0.1
PSRO	δ	80/15/10/7/3
Nash DQN	ϵ Exponential Decay	$\epsilon_0 = 1.0, \epsilon_1 = 1 \times 10^{-3}, p = 5 \times 10^6$
Nash DQN Exploiter	ϵ Exponential Decay	$\epsilon_0 = 1.0, \epsilon_1 = 1 \times 10^{-3}, p = 5 \times 10^6$
	Exploiter Update Ratio m_2/m_1	3

Atari games. For Table 7 and 8, the meaning of each hyperparameter is the same as in Appendix Sec. C. For Table 8, the threshold value δ for iterative best response procedure has multiple values, which correspond in order with environments *Boxing-v1*, *Double Dunk-v2*, *Pong-v2*, *Tennis-v2*, *Surround-v1*, respectively.

It can be noticed that NASH_DQN and NASH_DQN_EXPLOITER use a different ϵ decay schedule from other methods including SP, FSP, NFSP and PSRO. Since the Nash-based methods follow a single model update routine, the slow decaying ϵ (larger p) is more proper to be applied. Other baseline methods follow the iterative best response procedure, which learns new models in each period of update iteratively. Each period within the overall training process is much shorter, therefore the model with a faster decaying ϵ will learn better. In between two periods, the ϵ is re-initialized as the starting value ϵ_0 during the whole training process.

The exploiter for exploitation test after model training is a DQN agent with exactly the same common hyperparameters for the two-player video game test.

F COMPLETE RESULTS FOR VIDEO GAMES

Table 9 shows the means and standard deviations of exploitation results over different runs and exploiters. The experiments are the same as in Table 2. Notice that the mean values of Table 9 are different from the values in Table 2. This is because in Table 2 the models with best unexploitable performance are reported, while in Table 9 it's averaged over all runs and exploitation tests.

Table 9: Approximate exploitability (lower is better) for six two-player video games: mean \pm std.

Env \ Method	SP	FSP	NFSP	PSRO	Nash DQN	Nash DQN Exploiter
SlimeVolley	0.155 \pm 0.269	0.620 \pm 0.190	0.380 \pm 0.386	3.490 \pm 4.612	0.514 \pm 0.810	-0.149 \pm 0.587
Boxing	46.930 \pm 47.502	88.069 \pm 9.300	34.107 \pm 29.232	81.078 \pm 17.983	-67.411 \pm 25.124	19.790 \pm 22.412
Double Dunk	7.537 \pm 1.125	6.314 \pm 0.801	3.874 \pm 0.878	8.087 \pm 1.363	0.025 \pm 0.888	1.679 \pm 0.888
Pong	4.877 \pm 1.211	5.019 \pm 0.770	4.342 \pm 1.119	5.171 \pm 0.241	-2.016 \pm 1.890	-1.758 \pm 1.891
Tennis	3.773 \pm 1.142	2.950 \pm 1.065	3.580 \pm 0.913	2.793 \pm 1.823	-0.418 \pm 0.393	2.622 \pm 2.168
Surround	1.567 \pm 0.407	1.642 \pm 0.244	1.721 \pm 0.247	1.665 \pm 0.177	1.028 \pm 0.307	1.300 \pm 0.258

All results for three runs and each with three exploitation tests are shown in Fig. 6 and 7, which corresponding to the exploitation of the first and second player in games respectively. The NASH_DQN_EXPLOITER method is asymmetric and the second player side is not the NE strategy, so in Fig. 7 there is no exploitation results for NASH_DQN_EXPLOITER. All experiments are conducted on a 8-GPU (Nvidia Quadro RTX A6000 48GB) server with 192 CPU cores. The exploitation test is evaluated with non-greedy DQN agent for one episode every 20 training episodes during the whole training period. All curves are smoothed with a window size of 100.

Notice that in above experiments the exploitability value can be negative sometimes, however Eq. 7 tells the exploitability for symmetric games should always be non-negative. This is because Eq. 7 shows the theoretical best response, which is not tractable for large-scale games like Atari. The data we reported tables are exploitability results approximated with single-agent RL (DQN). We expect these exploiters to be weaker than the theoretically optimal ones, therefore negative values could appear, which also indicates that the learned Nash-DQN agents are extremely difficult to be exploited.

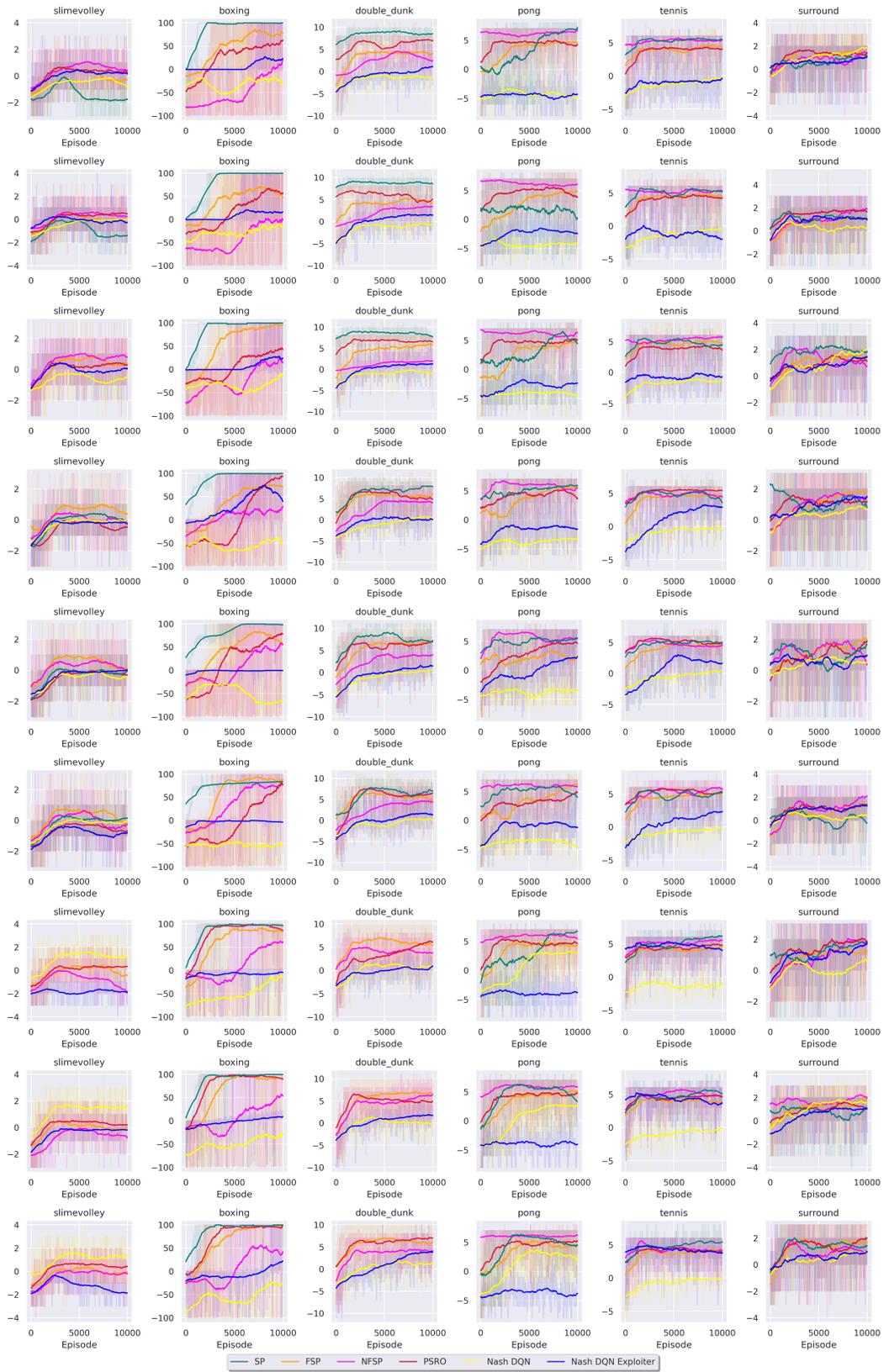


Figure 6: The exploitation tests on the first-player side on all six two-player zero-sum video games, for three runs and three exploitation for each run. The vertical axis is the episodic exploiter reward.

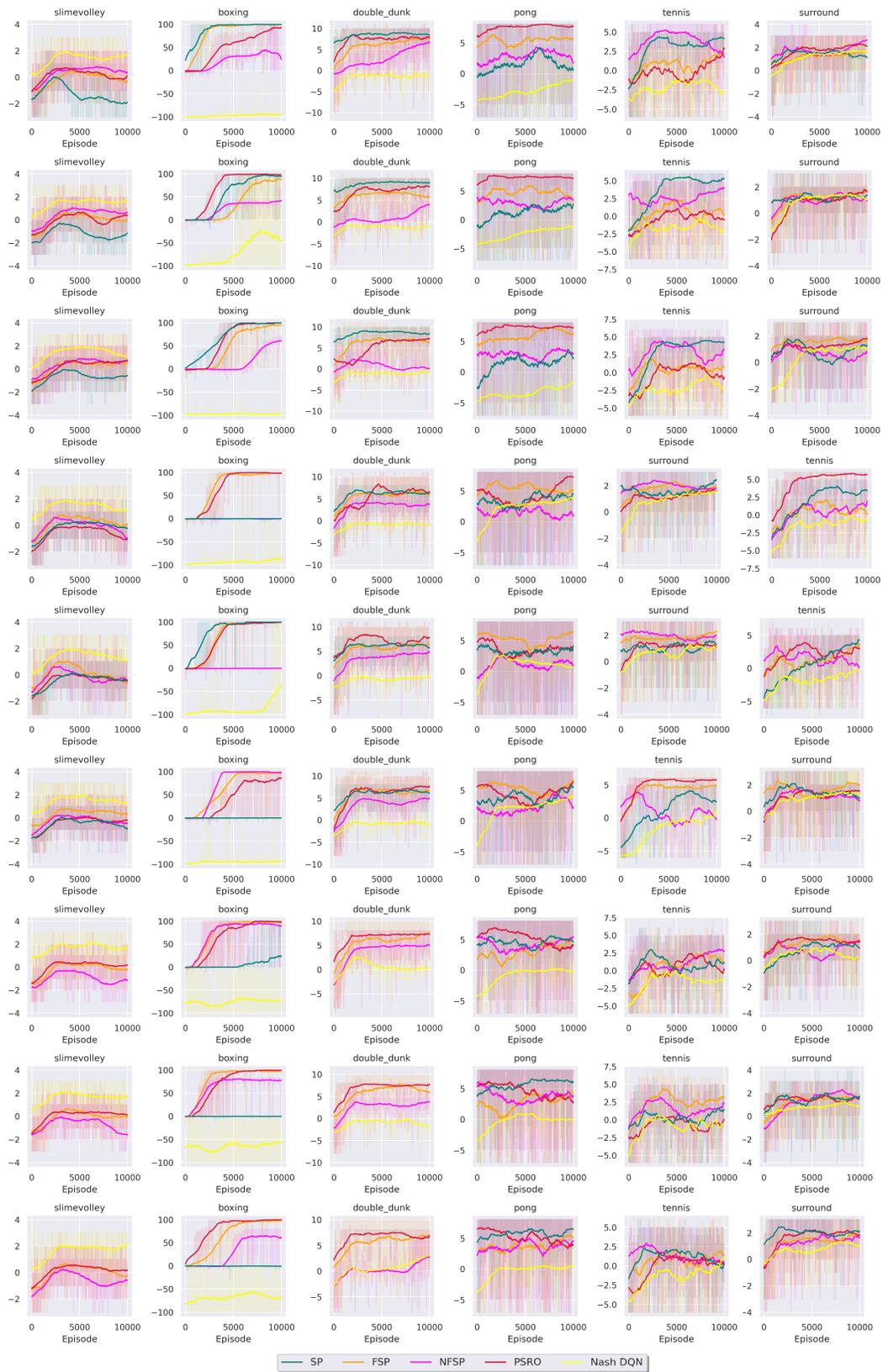


Figure 7: The exploitation tests on the first-player side on all six two-player zero-sum video games, for three runs and three exploitation for each run. The vertical axis is the episodic exploiter reward.

G EXPERIMENTS FOR FULL LENGTH ENVIRONMENTS

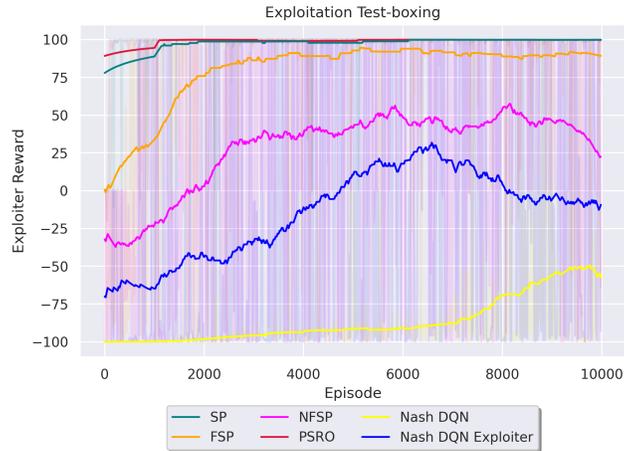


Figure 8: Comparison of the exploiter learning curves for a full-length setting on *Boxing-v1*.

In Fig.8, we show the exploitability experimentation with the full-length environment *Boxing-v1* (no truncation to 300 steps), and also the experiments with the same settings as in Sec. 4.3 except for the episode length. As shown in the figure, our methods (yellow for NASH_DQN and blue for NASH_DQN_EXPLOITER) show the best exploitability performance, which are less exploited in the exploitation tests. Specifically, even in this full length experimentation, the best performance method NASH_DQN achieves non-exploitable strategies.