Supplementary Material for *"Neural Auto-Curricula in Two-player Zero-sum Games"*

Table of Contents

A Meta-solver Architecture

In this section, we recap the meta-solver properties that we need and illustrate how we designed models to achieve them. There exist two properties the model should have.

- The model should handle a variable-length matrix input.
- The model should be subject to row-permutation equivariance and column-permutation invariance.

Three different techniques can be utilised to achieve the first property, which also corresponds to the three different models we propose: MLP based, Conv1d based and GRU based model. If not specifically mentioned, we utilise ReLU as the activation function for all MLP used in our meta-solver.

A.1 MLP-based Meta-Solver

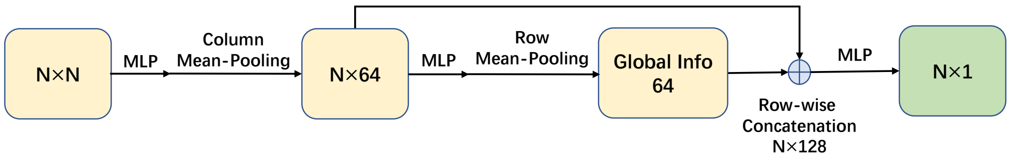

Figure 6: MLP based Meta-Solver

The first model is based on MLP. Inspired by PointNet[37], we utilise MLP + pooling operation + row-wise operation to handle variable-length matrix inputs and permutation invariance/equivariance. The first MLP + Column Mean-Pooling operation generates row-wise features: $N \times N \rightarrow N \times N \times 64 \rightarrow N \times 64$. Then the model transforms it to global matrix information by MLP + Row Mean-Pooling operation: $N \times 64 \rightarrow N \times 64 \rightarrow 64$. Finally, the model conducts row-wise concatenation between row-wise features and the global matrix information, and transforms it with the last MLP for the final output: $N \times (64 + 64) \rightarrow N \times 128 \rightarrow N \times 1$.

The MLP-based model successfully satisfies the properties we need. However, empirically we find that it does not always perform well within our training framework. We empirically find out even if the model violates the second property, it can still work well. We believe this is because there exists some particular patterns of meta distribution in the PSRO iterations, so even if the model is not a generally proper meta-strategy solver, it can still work well under PSRO. Next, we will detail Conv1d-based and GRU-based models which are not completely subject to the second property.

A.2 Conv1D-based Meta-Solver

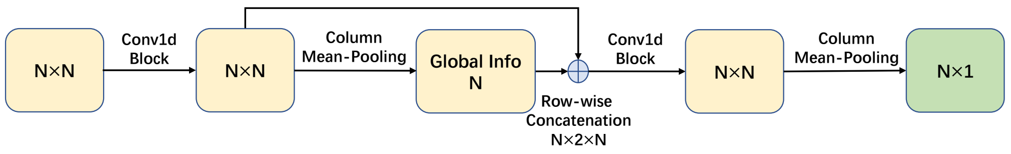

Figure 7: Conv1D based Meta-Solver

Our second model is Conv1d based. To satisfy variable-length matrix inputs, we make use of a Fully Convolutional Neural Network [27] with Conv1d on the row vectors $\mathbf{M}_i$ which, by construction, with particular kernel and padding size will not decrease the feature size in the forward pass. The procedure is shown as follows. Firstly, the model has multiple Conv1d-LeakyReLU layers (as a Conv1d block) for generating row-wise features: $N \times N \rightarrow N \times N$. Then similar column Mean-Pooling and row-wise concatenation are utilised to achieve global matrix information: $N \times N \rightarrow N; N \times (1+1) \times N \rightarrow N \times 2 \times N$. The final Conv1d block + Column Mean-Pooling operation gives the final prediction result: $N \times 2 \times N \rightarrow N \times N \rightarrow N \times 1$.

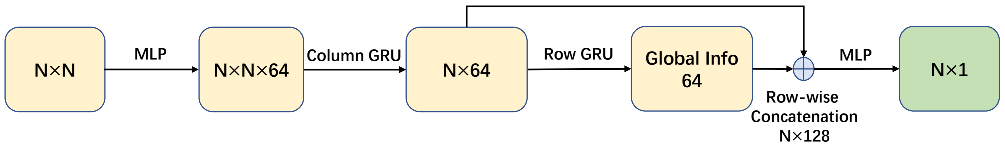

Figure 8: GRU based Meta-solver

Note that Conv1d-based model follows property 1 and row permutation equivariance. However, it violates the column permutation invariance property since we conduct Conv1d operation on column vectors.

A.3 GRU-based Meta-Solver

The final model is based on GRU, which can take in variable-length input sequences. To achieve a variable-length matrix input, we utilise GRU on both the column vectors and row vectors. The procedure is shown as follows. Firstly, the model utilises MLP + column GRU to aggregate the column vector for row-wise features: $N \times N \rightarrow N \times N \times 64 \rightarrow N \times 64$. With similar row GRU + row-wise concatenation + MLP, the model gets the final result: $N \times 64 \rightarrow 64$; $N \times (64+64) \rightarrow N \times 1$

Note that the GRU-based model only follows property 1. With GRU plugged in, it cannot hold both row permutation equivariance and column permutation invariance.

B Proof of Remark 3.1

Remark 3.1. *For a given distribution of game $p(\boldsymbol{G})$, by denoting exploitability at the final iteration $\mathfrak{M}\left(\phi^{BR}_{T+1}, \langle \boldsymbol{\pi}_T, \boldsymbol{\Phi}_T \rangle\right)$ as $\mathfrak{M}_{T+1}$, the meta-gradient for $\boldsymbol{\theta}$ (see also Fig. 1) is*

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{G}}\left[\frac{\partial \mathfrak{M}_{T+1}}{\partial \phi^{BR}_{T+1}} \frac{\partial \phi^{BR}_{T+1}}{\partial \boldsymbol{\theta}} + \frac{\partial \mathfrak{M}_{T+1}}{\partial \boldsymbol{\pi}_T} \frac{\partial \boldsymbol{\pi}_T}{\partial \boldsymbol{\theta}} + \frac{\partial \mathfrak{M}_{T+1}}{\partial \boldsymbol{\Phi}_T} \frac{\partial \boldsymbol{\Phi}_T}{\partial \boldsymbol{\theta}}\right], \text{where} \tag{6}$$

$$\frac{\partial \boldsymbol{\pi}_T}{\partial \boldsymbol{\theta}} = \frac{\partial f_{\boldsymbol{\theta}}(\mathbf{M}_T)}{\partial \boldsymbol{\theta}} + \frac{\partial f_{\boldsymbol{\theta}}(\mathbf{M}_T)}{\partial \mathbf{M}_T} \frac{\partial \mathbf{M}_T}{\partial \boldsymbol{\Phi}_T} \frac{\partial \boldsymbol{\Phi}_T}{\partial \boldsymbol{\theta}}, \quad \frac{\partial \phi^{BR}_{T+1}}{\partial \boldsymbol{\theta}} = \frac{\partial \phi^{BR}_{T+1}}{\partial \boldsymbol{\pi}_T} \frac{\partial \boldsymbol{\pi}_T}{\partial \boldsymbol{\theta}} + \frac{\partial \phi^{BR}_{T+1}}{\partial \boldsymbol{\Phi}_T} \frac{\partial \boldsymbol{\Phi}_T}{\partial \boldsymbol{\theta}}, \tag{7}$$

$$\frac{\partial \boldsymbol{\Phi}_T}{\partial \boldsymbol{\theta}} = \left\{ \frac{\partial \boldsymbol{\Phi}_{T-1}}{\partial \boldsymbol{\theta}}, \frac{\partial \phi^{BR}_{T}}{\partial \boldsymbol{\theta}} \right\}, \tag{8}$$

and Eq. (8) can be further decomposed by iteratively applying Eq. (7) from iteration $T-1$ to 0.

Proof of Remark 3.1. Here we will only consider the *single-population* case, which is the same as in the main paper for notation convenience. Note that the whole framework can be easily extended to the *multi-population* case. Firstly, we illustrate how the forward process works.

Assume we have t policies in the policy pool $\boldsymbol{\Phi}_t$ at the beginning of iteration $t+1$.

$$\mathbf{M}_t = \{\mathfrak{M}(\phi_i, \phi_j)\} \; \forall \phi_i, \phi_j \in \boldsymbol{\Phi}_t \tag{15}$$

We generate the curricula (meta distribution) by meta-solver $f_{\boldsymbol{\theta}}$.

$$\boldsymbol{\pi}_t = f_{\boldsymbol{\theta}}(\mathbf{M}_t) \tag{16}$$

Then we utilise best-response optimisation w.r.t the mixed meta-policy for the new policy which is added into the policy pool.

$$\mathfrak{M}(\phi, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi}_t \rangle) := \sum_{k=1}^{t} \boldsymbol{\pi}_t^k \mathfrak{M}(\phi, \phi_k) \tag{17}$$

$$\phi^{\text{BR}}_{\boldsymbol{t+1}} = \arg\max_{\phi} \mathfrak{M}\left(\phi, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi}_t \rangle\right) \tag{18}$$

$$\boldsymbol{\Phi}_{t+1} = \boldsymbol{\Phi}_t \cup \phi^{\text{BR}}_{\boldsymbol{t+1}} \tag{19}$$

After the final iteration T, we get the policy pool $\boldsymbol{\Phi}_T$ and calculate the exploitability of the final meta-policy:

$$\mathbf{M}_T = \{\mathfrak{M}(\boldsymbol{\phi}_i, \boldsymbol{\phi}_j)\} \; \forall \boldsymbol{\phi}_i, \boldsymbol{\phi}_j \in \boldsymbol{\Phi}_T \tag{20}$$

$$\boldsymbol{\pi}_T = f_{\boldsymbol{\theta}}(\mathbf{M}_T) \tag{21}$$

$$\boldsymbol{\phi}^{\text{BR}}_{\boldsymbol{T+1}} = \arg\max_{\boldsymbol{\phi}} \mathfrak{M}\left(\boldsymbol{\phi}, \langle \boldsymbol{\pi}_T, \boldsymbol{\Phi}_T \rangle\right) \tag{22}$$

$$\mathfrak{Exp} = \mathfrak{M}\left(\boldsymbol{\phi}^{\text{BR}}_{\boldsymbol{T+1}}, \langle \boldsymbol{\pi}_T, \boldsymbol{\Phi}_T \rangle\right) \tag{23}$$

Given a distribution $P(\boldsymbol{G})$ over game $\boldsymbol{G}$, the meta-gradient for $\boldsymbol{\theta}$ can be derived by applying the chain rule:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{G}}\Big[\frac{\partial \mathfrak{M}_{T+1}}{\partial \boldsymbol{\phi}^{\text{BR}}_{T+1}} \frac{\partial \boldsymbol{\phi}^{\text{BR}}_{T+1}}{\partial \boldsymbol{\theta}} + \frac{\partial \mathfrak{M}_{T+1}}{\partial \boldsymbol{\pi}_T} \frac{\partial \boldsymbol{\pi}_T}{\partial \boldsymbol{\theta}} + \frac{\partial \mathfrak{M}_{T+1}}{\partial \boldsymbol{\Phi}_T} \frac{\partial \boldsymbol{\Phi}_T}{\partial \boldsymbol{\theta}}\Big], \text{where} \tag{24}$$

$$\frac{\partial \boldsymbol{\pi}_T}{\partial \boldsymbol{\theta}} = \frac{\partial f_{\boldsymbol{\theta}}(\mathbf{M}_T)}{\partial \boldsymbol{\theta}} + \frac{\partial f_{\boldsymbol{\theta}}(\mathbf{M}_T)}{\partial \mathbf{M}_T} \frac{\partial \mathbf{M}_T}{\partial \boldsymbol{\Phi}_T} \frac{\partial \boldsymbol{\Phi}_{\boldsymbol{T}}}{\partial \boldsymbol{\theta}}, \tag{25}$$

$$\frac{\partial \boldsymbol{\phi}^{\text{BR}}_{T+1}}{\partial \boldsymbol{\theta}} = \frac{\partial \boldsymbol{\phi}^{\text{BR}}_{T+1}}{\partial \boldsymbol{\pi}_T} \frac{\partial \boldsymbol{\pi}_T}{\partial \boldsymbol{\theta}} + \frac{\partial \boldsymbol{\phi}^{\text{BR}}_{T+1}}{\partial \boldsymbol{\Phi}_T} \frac{\partial \boldsymbol{\Phi}_T}{\partial \boldsymbol{\theta}}, \tag{26}$$

$$\frac{\partial \boldsymbol{\Phi}_{\boldsymbol{T}}}{\partial \boldsymbol{\theta}} = \left\{ \frac{\partial \boldsymbol{\Phi}_{\boldsymbol{T-1}}}{\partial \boldsymbol{\theta}}, \frac{\partial \boldsymbol{\phi}^{\text{BR}}_T}{\partial \boldsymbol{\theta}} \right\}. \tag{27}$$

□

Note that Eq. (27) can be further decomposed by iteratively applying Eq. (25) and Eq. (26), which means the gradients will backpropagate through multiple iterations. The whole process is similar to the backpropagation through time (BPTT) process in RNN.

In the following section, we detail how the gradient is calculated with two different best-response oracles - a Gradient-Descent oracle and a Reinforcement Learning oracle, in particular showing how we take meta-gradients for both.

B.1 Gradient-Descent Best-Response with direct Meta-gradient

For a GD based best-response oracle, the payoff function of the game is differentiable, so we can directly obtain gradients $\frac{\partial \mathfrak{M}_{T+1}}{\partial \boldsymbol{\phi}^{\text{BR}}_{T+1}}, \frac{\partial \mathfrak{M}_{T+1}}{\partial \boldsymbol{\pi}_T}, \frac{\partial \mathfrak{M}_{T+1}}{\partial \boldsymbol{\Phi}_T}$ by automatic differentiation.

An example for a GD oracle with one gradient-descent step:

$$\boldsymbol{\phi}^{\text{BR}}_{t+1} = \boldsymbol{\phi}_0 + \alpha \frac{\partial \mathfrak{M}\left(\boldsymbol{\phi}_0, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi}_t \rangle\right)}{\partial \boldsymbol{\phi}_0}, \tag{28}$$

where $\boldsymbol{\phi}_0$ and α denote the initial parameters and learning rate respectively. The backward gradients of one-step GD share similarities with MAML [12], which can be written as:

$$\frac{\partial \boldsymbol{\phi}^{\text{BR}}_{t+1}}{\partial \boldsymbol{\pi}_t} = \alpha \frac{\partial^2 \mathfrak{M}\left(\boldsymbol{\phi}_0, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi}_t \rangle\right)}{\partial \boldsymbol{\phi}_0 \partial \boldsymbol{\pi}_t}, \quad \frac{\partial \boldsymbol{\phi}^{\text{BR}}_{t+1}}{\partial \boldsymbol{\Phi}_t} = \alpha \frac{\partial^2 \mathfrak{M}\left(\boldsymbol{\phi}_0, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi}_t \rangle\right)}{\partial \boldsymbol{\phi}_0 \partial \boldsymbol{\Phi}_t}. \tag{29}$$

Eq. (28) and (29) can be easily extended to situations where we take a few gradient steps. Eq. (27) can be calculated iteratively by calling for the previous gradient terms,

$$\begin{aligned} \frac{\partial \boldsymbol{\Phi}_{\boldsymbol{T}}}{\partial \boldsymbol{\theta}} &= \left\{ \frac{\partial \boldsymbol{\phi}^{\text{BR}}_1}{\partial \boldsymbol{\theta}}, \frac{\partial \boldsymbol{\phi}^{\text{BR}}_2}{\partial \boldsymbol{\theta}}, \ldots, \frac{\partial \boldsymbol{\phi}^{\text{BR}}_T}{\partial \boldsymbol{\theta}} \right\} \\ &= \left\{ \alpha \frac{\partial^2 \mathfrak{M}\left(\boldsymbol{\phi}_0, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi}_t \rangle\right)}{\partial \boldsymbol{\phi}_0 \partial \boldsymbol{\pi}_t} \frac{\partial \boldsymbol{\pi}_t}{\partial \boldsymbol{\theta}} + \alpha \frac{\partial^2 \mathfrak{M}\left(\boldsymbol{\phi}_0, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi}_t \rangle\right)}{\partial \boldsymbol{\phi}_0 \partial \boldsymbol{\Phi}_t} \frac{\partial \boldsymbol{\Phi}_t}{\partial \boldsymbol{\theta}} \right\}_{t \in \{1,2,\ldots,T\}} \end{aligned} \tag{30}$$

.

B.2 Gradient-Descent Best-Response with Implicit Gradient based Meta-gradient

The direct meta-gradient formulation above becomes easily intractable when the computational graph including hundreds of gradient updates. Thus, here we offer another alternative based on implicit gradients for efficient meta-gradient backpropagation. The main issue here is to solve the gradient terms $\frac{\partial \boldsymbol{\phi}_{T+1}^{\mathrm{BR}}}{\partial \boldsymbol{\pi}_T}, \frac{\partial \boldsymbol{\phi}_{T+1}^{\mathrm{BR}}}{\partial \boldsymbol{\Phi}_T}$.

Firstly, we can get an exact best-response by hundreds of gradient steps to achieve:

$$\boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}} = \arg\max_{\boldsymbol{\phi}} \mathfrak{M}\left(\boldsymbol{\phi}, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi}_t \rangle\right) \tag{31}$$

Since $\boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}}$ is a minimiser of the inner loop optimisation, we can derive the stationary point condition by implicit function theorem:

$$\begin{aligned}
&\frac{\partial \mathfrak{M}\left(\boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}}, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi}_t \rangle\right)}{\partial \boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}}} = 0 \\
\rightarrow\ &\frac{\partial}{\partial \boldsymbol{\pi}_t} \frac{\partial \mathfrak{M}\left(\boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}}, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi}_t \rangle\right)}{\partial \boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}}} = 0 \\
\rightarrow\ &\frac{\partial^2 \mathfrak{M}\left(\boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}}, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi}_t \rangle\right)}{\partial \boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}} \partial \boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}\ T}} \frac{\partial \boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}}}{\partial \boldsymbol{\pi}_t} + \frac{\partial^2 \mathfrak{M}\left(\boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}}, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi}_t \rangle\right)}{\partial \boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}} \partial \boldsymbol{\pi}_t} = 0 \\
\rightarrow\ &\frac{\partial \boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}}}{\partial \boldsymbol{\pi}_t} = -\left[\frac{\partial^2 \mathfrak{M}\left(\boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}}, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi}_t \rangle\right)}{\partial \boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}} \partial \boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}\ T}}\right]^{-1} \frac{\partial^2 \mathfrak{M}\left(\boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}}, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi}_t \rangle\right)}{\partial \boldsymbol{\phi}_{\boldsymbol{t+1}}^{\mathrm{BR}} \partial \boldsymbol{\pi}_t}
\end{aligned} \tag{32}$$

Note that this implicit gradient requires the Hessian matrix $\frac{\partial^2 \mathfrak{M}(\boldsymbol{\phi}_{t+1}^{\mathrm{BR}}, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi_t} \rangle)}{\partial \boldsymbol{\phi}_{t+1}^{\mathrm{BR}} \partial \boldsymbol{\phi}_{t+1}^{\mathrm{BR}\ T}}$ to be invertible, so it may not hold in some situations (like normal form games). Following the same reasoning, we can get:

$$\frac{\partial \boldsymbol{\phi}_{t+1}^{\mathrm{BR}}}{\partial \boldsymbol{\Phi}_t} = -\left[\frac{\partial^2 \mathfrak{M}(\boldsymbol{\phi}_{t+1}^{\mathrm{BR}}, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi_t} \rangle)}{\partial \boldsymbol{\phi}_{t+1}^{\mathrm{BR}} \partial \boldsymbol{\phi}_{t+1}^{\mathrm{BR}\ T}}\right]^{-1} \frac{\partial^2 \mathfrak{M}(\boldsymbol{\phi}_{t+1}^{\mathrm{BR}}, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi_t} \rangle)}{\partial \boldsymbol{\phi}_{t+1}^{\mathrm{BR}} \partial \boldsymbol{\Phi_t}}. \tag{33}$$

B.3 Reinforcement Learning Best-response Oracle with Direct Meta-gradient

For a Reinforcement Learning based best-response oracle, the only difference is that we need to replace gradient terms with policy gradient estimation. We utilise first-order policy gradients for estimating $\frac{\partial \mathfrak{M}_{T+1}}{\partial \boldsymbol{\phi}_{T+1}^{\mathrm{BR}}}, \frac{\partial \mathfrak{M}_{T+1}}{\partial \boldsymbol{\pi}_T}, \frac{\partial \mathfrak{M}_{T+1}}{\partial \boldsymbol{\Phi}_T}$. For the best-response process, a one-step RL example is to replace Eq. (28) with:

$$\begin{aligned}
\boldsymbol{\phi}_{t+1}^{\mathrm{BR}} &= \boldsymbol{\phi}_0 + \alpha \frac{\partial \mathfrak{M}\left(\boldsymbol{\phi}_0, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi}_t \rangle\right)}{\partial \boldsymbol{\phi}_0} \\
&= \boldsymbol{\phi}_0 + \alpha \sum_{k=1}^{t} \boldsymbol{\pi}_t^k \nabla_{\boldsymbol{\phi}_0} E_{\boldsymbol{\tau} \sim P(\boldsymbol{\tau} | \boldsymbol{\phi}_0, \boldsymbol{\phi}_k^{\mathrm{BR}})} \left[R(\tau)\right],
\end{aligned} \tag{34}$$

where α and $\boldsymbol{\tau}$ refer to the learning rate and the joint trajectories for two agents respectively. Reward R represents the trajectory return for the first agent. The backward meta-gradient for the best-response process can be computed as:

$$\begin{aligned}
\frac{\partial \boldsymbol{\phi}_{t+1}^{\mathrm{BR}}}{\partial \boldsymbol{\pi}_t} &= \alpha \frac{\partial^2 \mathfrak{M}\left(\boldsymbol{\phi}_0, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi}_t \rangle\right)}{\partial \boldsymbol{\phi}_0 \partial \boldsymbol{\pi}_t} \\
&= \alpha \left(\nabla_{\boldsymbol{\phi}_0} E_{\boldsymbol{\tau} \sim P(\boldsymbol{\tau} | \boldsymbol{\phi}_0, \boldsymbol{\phi}_k^{\mathrm{BR}})} \left[R(\tau)\right]\right)_{\{k=1,2,\ldots,t\}},
\end{aligned} \tag{35}$$

$$\begin{aligned}
\frac{\partial \boldsymbol{\phi}_{t+1}^{\mathrm{BR}}}{\partial \boldsymbol{\Phi}_t} &= \alpha \frac{\partial^2 \mathfrak{M}\left(\boldsymbol{\phi}_0, \langle \boldsymbol{\pi}_t, \boldsymbol{\Phi}_t \rangle\right)}{\partial \boldsymbol{\phi}_0 \partial \boldsymbol{\Phi}_t} \\
&= \alpha \left(\nabla_{\boldsymbol{\phi}_0} \nabla_{\boldsymbol{\phi}_k^{\mathrm{BR}}} E_{\boldsymbol{\tau} \sim P(\boldsymbol{\tau} | \boldsymbol{\phi}_0, \boldsymbol{\phi}_k^{\mathrm{BR}})} \left[R(\tau)\right]\right)_{\{k=1,2,\ldots,t\}}.
\end{aligned} \tag{36}$$

Eq. (27) for a Reinforcement Learning based oracle can be handled following a similar manner by replacing gradients with policy gradient estimation.

So the main issue is: how can we estimate the second-order policy gradient $\nabla_{\boldsymbol{\phi_1}} \nabla_{\boldsymbol{\phi_2}} E_{\tau \sim p(\tau|\boldsymbol{\phi_1},\boldsymbol{\phi_2})}[R(\boldsymbol{\tau})]$, where $\boldsymbol{\phi_1}$, $\boldsymbol{\phi_2}$ denotes policy for two agents. There are several higher order gradient estimators like DICE [13], LVC [40] that can help us. In our case, we utilise DICE which is entirely compatible with automatic differentiation toolbox. In the following part, we follow similar analysis way like [40] to show how second-order policy gradient is like and how we can estimate unbiased first-order and second-order policy gradient with DICE. In the following part, $P(\boldsymbol{\tau} \mid \boldsymbol{\phi_1}, \boldsymbol{\phi_2})$ and $P_{\boldsymbol{\phi_1},\boldsymbol{\phi_2}}(\boldsymbol{\tau})$ represent the probability of the joint trajectory.

$$\begin{aligned}
&\nabla_{\boldsymbol{\phi_1}} \nabla_{\boldsymbol{\phi_2}} \mathbb{E}_{\boldsymbol{\tau} \sim P(\boldsymbol{\tau}|\boldsymbol{\phi_1},\boldsymbol{\phi_2})}[R(\boldsymbol{\tau})] \\
&= \nabla_{\boldsymbol{\phi_1}} \mathbb{E}_{\boldsymbol{\tau} \sim P(\boldsymbol{\tau}|\boldsymbol{\phi_1},\boldsymbol{\phi_2})} \left[\nabla_{\boldsymbol{\phi_2}} \log P_{\boldsymbol{\phi_1},\boldsymbol{\phi_2}}(\boldsymbol{\tau}) R(\boldsymbol{\tau})\right] \\
&= \nabla_{\boldsymbol{\phi_1}} \int P(\boldsymbol{\tau} \mid \boldsymbol{\phi_1}, \boldsymbol{\phi_2}) \nabla_{\boldsymbol{\phi_2}} \log P_{\boldsymbol{\phi_1},\boldsymbol{\phi_2}}(\boldsymbol{\tau}) R(\boldsymbol{\tau}) d\boldsymbol{\tau} \\
&= \int \Big[P(\boldsymbol{\tau} \mid \boldsymbol{\phi_1}, \boldsymbol{\phi_2}) \nabla_{\boldsymbol{\phi_1}} \log P_{\boldsymbol{\phi_1},\boldsymbol{\phi_2}}(\boldsymbol{\tau}) \nabla_{\boldsymbol{\phi_2}} \log P_{\boldsymbol{\phi_1},\boldsymbol{\phi_2}}(\boldsymbol{\tau})^\top R(\boldsymbol{\tau}) \\
&\qquad\qquad + P(\boldsymbol{\tau} \mid \boldsymbol{\phi_1}, \boldsymbol{\phi_2}) \nabla_{\boldsymbol{\phi_1}} \nabla_{\boldsymbol{\phi_2}} \log P_{\boldsymbol{\phi_1},\boldsymbol{\phi_2}}(\boldsymbol{\tau}) R(\boldsymbol{\tau})\Big] d\boldsymbol{\tau} \\
&= \mathbb{E}_{\boldsymbol{\tau} \sim P(\boldsymbol{\tau}|\boldsymbol{\phi_1},\boldsymbol{\phi_2})} \left[R(\boldsymbol{\tau}) \left(\nabla_{\boldsymbol{\phi_1}} \nabla_{\boldsymbol{\phi_2}} \log P_{\boldsymbol{\phi_1},\boldsymbol{\phi_2}}(\boldsymbol{\tau}) + \nabla_{\boldsymbol{\phi_1}} \log P_{\boldsymbol{\phi_1},\boldsymbol{\phi_2}}(\boldsymbol{\tau}) \nabla_{\boldsymbol{\phi_2}} \log P_{\boldsymbol{\phi_1},\boldsymbol{\phi_2}}(\boldsymbol{\tau})^\top\right)\right].
\end{aligned} \tag{37}$$

In fact, we can show that $\nabla_{\boldsymbol{\phi_1}} \nabla_{\boldsymbol{\phi_2}} \log P_{\boldsymbol{\phi_1},\boldsymbol{\phi_2}}(\boldsymbol{\tau}) = 0$.

$$\begin{aligned}
\nabla_{\boldsymbol{\phi_1}} \nabla_{\boldsymbol{\phi_2}} \log P_{\boldsymbol{\phi_1},\phi_2}(\tau) &= \nabla_{\boldsymbol{\phi_1}} \nabla_{\boldsymbol{\phi_2}} \log \prod_{i=0}^{n} P_{\boldsymbol{\phi_1},\boldsymbol{\phi_2}}(a_i^1, a_i^2 | s_i^1, s_i^2) \\
&= \nabla_{\boldsymbol{\phi_1}} \nabla_{\boldsymbol{\phi_2}} \log \prod_{i=0}^{n} \pi_{\boldsymbol{\phi_1}}(a_i^1|s_i^1) \pi_{\boldsymbol{\phi_2}}(a_i^2|s_i^2) \\
&= \nabla_{\phi_1} \nabla_{\boldsymbol{\phi_2}} \sum_{i=0}^{n} \left(\log(\pi_{\boldsymbol{\phi_1}}(a_i^1|s_i^1)) + \log(\pi_{\boldsymbol{\phi_2}}(a_i^2|s_i^2))\right) \\
&= 0
\end{aligned} \tag{38}$$

where n denotes the length of the RL trajectory, $\boldsymbol{\pi_{\phi_1}}$ and $\pi_{\boldsymbol{\phi_2}}$ represent stochastic policies for two agents respectively. Note that $P_{\boldsymbol{\phi_1},\boldsymbol{\phi_2}}(a_i^1, a_i^2|s_i^1, s_i^2) = \pi_{\boldsymbol{\phi_1}}(a_i^1|s_i^1)\pi_{\boldsymbol{\phi_2}}(a_i^2|s_i^2)$ because the agent only relies on its own state. Following [13]'s formulation, we have:

$$\begin{aligned}
J^{\text{DICE}} &= \sum_{t=0}^{H-1} \square\left(\left\{a_{j\in\{1,2\}}^{t'\le t}\right\}\right) r_t \\
&= \sum_{t=0}^{H-1} \exp\left(\sum_{t'=0}^{t} \log \pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right) \log \pi_{\boldsymbol{\phi_2}}\left(a_{t'}^2 \mid s_{t'}^2\right) - \perp\left(\log \pi_{\boldsymbol{\phi_1}}(a_{t'}^1 \mid s_{t'}^1) \log \pi_{\boldsymbol{\phi_1}}(a_{t'}^2 \mid s_t^2)\right)\right) r_t
\end{aligned} \tag{39}$$

We denote $\perp$ as the "stop gradient" operator and $\rightarrow$ as the "evaluates to" operator. "Evaluates to" operator $\rightarrow$ is in contrast with =, which also brings the equality of gradients. So the "stop gradient" operator here means that $\perp(f_{\boldsymbol{\theta}}(x)) \rightarrow f_{\boldsymbol{\theta}}(x)$ but $\nabla_{\boldsymbol{\theta}} \perp (f_{\boldsymbol{\theta}}(x)) \rightarrow 0$.

To make the DICE loss concise, we reformulate it as follows:

$$J^{\text{DICE}} = \sum_{t=0}^{H-1} \left(\prod_{t'=0}^{t} \frac{\pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right) \pi_{\boldsymbol{\phi_2}}(a_{t'}^2 \mid s_{t'}^2)}{\perp\left(\pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right) \pi_{\boldsymbol{\phi_2}}\left(a_{t'}^2 \mid s_{t'}^2\right)\right)}\right) r_t \tag{40}$$

where r_t refers to the reward agent 1 gets at timestep t.

$$\begin{aligned}
\nabla_{\boldsymbol{\phi_1}} J^{\text{DICE}} &= \sum_{t=0}^{H-1} \nabla_{\boldsymbol{\phi_1}} \left(\prod_{t'=0}^{t} \frac{\pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right) \pi_{\boldsymbol{\phi_2}}(a_{t'}^2 \mid s_{t'}^2)}{\perp\left(\pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right) \pi_{\boldsymbol{\phi_2}}\left(a_{t'}^2 \mid s_{t'}^2\right)\right)}\right) r_t \\
&= \sum_{t=0}^{H-1} \left(\prod_{t'=0}^{t} \frac{\pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right) \pi_{\boldsymbol{\phi_2}}(a_{t'}^2 \mid s_{t'}^2)}{\perp\left(\pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right) \pi_{\boldsymbol{\phi_2}}\left(a_{t'}^2 \mid s_{t'}^2\right)\right)}\right) \left(\sum_{t'=0}^{t} \nabla_{\boldsymbol{\phi_1}} \log \pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right)\right) r_t \\
&\rightarrow \sum_{t=0}^{H-1} \left(\sum_{t'=0}^{t} \nabla_{\boldsymbol{\phi_1}} \log \pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right)\right) r_t
\end{aligned} \tag{41}$$

$$\begin{aligned}
\mathbb{E}_{\boldsymbol{\tau}\sim P(\boldsymbol{\tau}|\boldsymbol{\phi_1},\boldsymbol{\phi_2})}\left[\nabla_{\boldsymbol{\phi_1}} J^{\mathrm{DICE}}\right] &= \mathbb{E}_{\boldsymbol{\tau}\sim P(\boldsymbol{\tau}|\boldsymbol{\phi_1},\boldsymbol{\phi_2})}\left[\sum_{t=0}^{H-1}\left(\sum_{t'=0}^{t}\nabla_{\boldsymbol{\phi_1}}\log\pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right)\right) r_t\right] \\
&= \nabla_{\boldsymbol{\phi_1}}\mathbb{E}_{\boldsymbol{\tau}\sim P(\boldsymbol{\tau}|\boldsymbol{\phi_1},\boldsymbol{\phi_2})}\left[R(\boldsymbol{\tau})\right] \qquad (42)
\end{aligned}$$

which corresponds to standard policy gradients for single-agent in a multi-agent environment (agents will consider other agents as part of the environment). And the hessian for the DICE loss is:

$$\begin{aligned}
&\nabla_{\boldsymbol{\phi_2}}\nabla_{\boldsymbol{\phi_1}} J^{\mathrm{DICE}} \\
&= \sum_{t=0}^{H-1}\nabla_{\boldsymbol{\phi_2}}\left(\prod_{t'=0}^{t}\frac{\pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right)\pi_{\boldsymbol{\phi_2}}\left(a_{t'}^2 \mid s_{t'}^2\right)}{\perp\left(\pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right)\pi_{\boldsymbol{\phi_2}}\left(a_{t'}^2 \mid s_{t'}^2\right)\right)}\right)\left(\sum_{t'=0}^{t}\nabla_{\boldsymbol{\phi_1}}\log\pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right)\right) r_t \\
&= \sum_{t=0}^{H-1}\left(\prod_{t'=0}^{t}\frac{\pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right)\pi_{\boldsymbol{\phi_2}}\left(a_{t'}^2 \mid s_{t'}^2\right)}{\perp\left(\pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right)\pi_{\boldsymbol{\phi_2}}\left(a_{t'}^2 \mid s_{t'}^2\right)\right)}\right)\cdot \\
&\qquad\qquad \left(\sum_{t'=0}^{t}\nabla_{\boldsymbol{\phi_1}}\log\pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right)\right)\left(\sum_{t'=0}^{t}\nabla_{\boldsymbol{\phi_2}}\log\pi_{\boldsymbol{\phi_2}}\left(a_{t'}^2 \mid s_{t'}^2\right)\right)^{\top} r_t
\end{aligned}$$

which can be evaluated via the following:

$$\rightarrow \sum_{t=0}^{H-1}\left(\sum_{t'=0}^{t}\nabla_{\boldsymbol{\phi_1}}\log\pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right)\right)\left(\sum_{t'=0}^{t}\nabla_{\boldsymbol{\phi_2}}\log\pi_{\boldsymbol{\phi_2}}\left(a_{t'}^2 \mid s_{t'}^2\right)\right)^{\top} r_t \qquad (43)$$

So finally we have:

$$\begin{aligned}
&\mathbb{E}_{\boldsymbol{\tau}\sim P_{\mathcal{T}}(\boldsymbol{\tau}|\boldsymbol{\phi_1},\boldsymbol{\phi_2})}\left[\nabla_{\boldsymbol{\phi_1}}\nabla_{\boldsymbol{\phi_2}} J^{\mathrm{DICE}}\right] \\
&= \mathbb{E}_{\boldsymbol{\tau}\sim P(\boldsymbol{\tau}|\boldsymbol{\phi_1},\boldsymbol{\phi_2})}\left[\sum_{t=0}^{H-1}\left(\sum_{t'=0}^{t}\nabla_{\boldsymbol{\phi_1}}\log\pi_{\boldsymbol{\phi_1}}\left(a_{t'}^1 \mid s_{t'}^1\right)\right)\left(\sum_{t'=0}^{t}\nabla_{\boldsymbol{\phi_2}}\log\pi_{\boldsymbol{\phi_2}}\left(a_{t'}^2 \mid s_{t'}^2\right)\right)^{\top} r_t\right] \\
&= \mathbb{E}_{\boldsymbol{\tau}\sim P(\boldsymbol{\tau}|\boldsymbol{\phi_1},\boldsymbol{\phi_2})}\left[R(\boldsymbol{\tau})\nabla_{\boldsymbol{\phi_1}}\log P_{\boldsymbol{\phi_1},\boldsymbol{\phi_2}}(\boldsymbol{\tau})\nabla_{\boldsymbol{\phi_2}}\log P_{\boldsymbol{\phi_1},\boldsymbol{\phi_2}}(\boldsymbol{\tau})^{\top}\right] \\
&= \nabla_{\boldsymbol{\phi_1}}\nabla_{\boldsymbol{\phi_2}}\mathbb{E}_{\boldsymbol{\tau}\sim P(\boldsymbol{\tau}|\boldsymbol{\phi_1},\boldsymbol{\phi_2})}[R(\boldsymbol{\tau})] \qquad (44)
\end{aligned}$$

In all, we have shown that by plugging DICE into the computation graph, we can obtain unbiased first-order and second-order policy gradient estimation and also the overall meta-gradient estimation.

C Pseudo-Codes of the Proposed Algorithms

C.1 Gradient-Descent Best-Response Oracles

Non-Implicit Version

Here we provide details of NAC where few-step gradient descent is used as the best-response oracle in Alg. (2), and therefore we are in the non-implicit setting.

Algorithm 2 NAC with Gradient-Descent Best-Response Oracles

```
Require: Game distribution $p(\boldsymbol{G})$, inner learning rate $\alpha$, outer learning rate $\beta$, time window $T$.
 1: Randomly initialise policy pool $\boldsymbol{\phi_0}$, Initialise parameters $\boldsymbol{\theta}$ of the meta solver $f_{\boldsymbol{\theta}}$.
 2: for each training iteration do
 3:     Sample games $\{G_k\}_{k=1,\dots,K}$ from $p(\boldsymbol{G})$.
 4:     for each game $G_k$ do
 5:         for each iteration $t$ do
 6:             Compute the meta-policy $\boldsymbol{\pi}_{t-1} = f(\mathbf{M}_{t-1})$.
 7:             Initialise random best-response policy $\boldsymbol{\phi}_0$.
 8:             for gradient updates $n$ do
 9:                 Compute $\boldsymbol{\phi}_{n+1} = \boldsymbol{\phi}_n + \alpha \frac{\partial \mathfrak{M}(\boldsymbol{\phi}_n, \langle \boldsymbol{\pi}_{t-1}, \boldsymbol{\Phi}_{t-1} \rangle)}{\partial \boldsymbol{\phi}_n}$ via Eq. (9)
10:             end for
11:             Expand the population $\boldsymbol{\Phi_t} = \boldsymbol{\Phi_{t-1}} \cup \{\boldsymbol{\phi}_t^{\text{BR}}\}$
12:         end for
13:         Compute the meta-policy $\boldsymbol{\pi}_T = f(\mathbf{M}_T)$.
14:         Compute $\mathfrak{Exp}_i(\boldsymbol{\pi}_T, \boldsymbol{\Phi}_T)$ by Eq. (3)
15:     end for
16:     Compute the meta-gradient $\mathbf{g}_k$ via Eq. (6) with br meta-gradient following Eq. (10)
17:     Update meta-solver's parameters $\boldsymbol{\theta}' = \boldsymbol{\theta} - \beta \frac{1}{K} \sum_k \mathbf{g}_k$.
18: end for
```

Implicit Version

Here we provide details of NAC where many-step gradient descent is used as the best-response oracle in Alg. (3), and therefore we are in the implicit setting.

Algorithm 3 NAC with Gradient-Descent Best-Response Oracles-Implicit

```
Require: Game distribution $p(\boldsymbol{G})$, inner learning rate $\alpha$, outer learning rate $\beta$, time window $T$.
 1: Randomly initialise policy pool $\boldsymbol{\phi_0}$, Initialise parameters $\boldsymbol{\theta}$ of the meta solver $f_{\boldsymbol{\theta}}$.
 2: for each training iteration do
 3:     Sample games $\{G_k\}_{k=1,\dots,K}$ from $p(\boldsymbol{G})$.
 4:     for each game $G_k$ do
 5:         for each iteration $t$ do
 6:             Compute the meta-policy $\boldsymbol{\pi}_{t-1} = f(\mathbf{M}_{t-1})$.
 7:             Initialise random best-response policy $\boldsymbol{\phi}_0$.
 8:             for gradient updates $n$ (Large $n$) do
 9:                 Compute $\boldsymbol{\phi}_{n+1}^{\text{BR}} = \boldsymbol{\phi}_n + \alpha \frac{\partial \mathfrak{M}(\boldsymbol{\phi}_n, \langle \boldsymbol{\pi}_{t-1}, \boldsymbol{\Phi}_{t-1} \rangle)}{\partial \boldsymbol{\phi}_n}$ via Eq. (9)
10:             end for
11:             Expand the population $\boldsymbol{\Phi_t} = \boldsymbol{\Phi_{t-1}} \cup \{\boldsymbol{\phi}_t^{\text{BR}}\}$
12:         end for
13:         Compute the meta-policy $\boldsymbol{\pi}_T = f(\mathbf{M}_T)$.
14:         Compute $\mathfrak{Exp}_i(\boldsymbol{\pi}_T, \boldsymbol{\Phi}_T)$ by Eq. (3)
15:     end for
16:     Compute the meta-gradient $\mathbf{g}_k$ via Eq. (6) with br meta-gradient following Eq. (11)
17:     Update meta-solver's parameters $\boldsymbol{\theta}' = \boldsymbol{\theta} - \beta \frac{1}{K} \sum_k \mathbf{g}_k$.
18: end for
```

C.2 Reinforcement Learning Best-Response Oracles

Here we provide details of NAC where reinforcement learning is used as the best-response oracle in Alg. (4), where we apply DICE for unbiased meta-gradient estimation.

Algorithm 4 NAC with Reinforcement Learning Best-Response Oracles

```
Require: Game distribution $p(\boldsymbol{G})$, inner learning rate $\alpha$, outer learning rate $\beta$, time window $T$.
 1: Randomly initialise policy pool $\boldsymbol{\phi_0}$, Initialise parameters $\boldsymbol{\theta}$ of the meta solver $f_{\boldsymbol{\theta}}$.
 2: for each training iteration do
 3:     Sample games $\{G_k\}_{k=1,\dots,K}$ from $p(\boldsymbol{G})$.
 4:     for each game $G_k$ do
 5:         for each iteration $t$ do
 6:             Compute the meta-policy $\boldsymbol{\pi}_{t-1} = f(\mathbf{M}_{t-1})$.
 7:             Initialise random best-response policy $\boldsymbol{\phi}_0$.
 8:             for gradient updates $n$ do
 9:                 Compute $\boldsymbol{\phi}_{n+1} = \boldsymbol{\phi}_n + \alpha \frac{\partial \mathfrak{M}(\boldsymbol{\phi}_n, \langle \boldsymbol{\pi}_{t-1}, \boldsymbol{\Phi}_{t-1} \rangle)}{\partial \boldsymbol{\phi}_n}$ with DICE in Eq. (12)
10:             end for
11:             Expand the population $\boldsymbol{\Phi_t} = \boldsymbol{\Phi_{t-1}} \cup \{\boldsymbol{\phi}_t^{\mathrm{BR}}\}$
12:         end for
13:         Compute the meta-policy $\boldsymbol{\pi}_T = f(\mathbf{M}_T)$.
14:         Compute $\mathfrak{Exp}_i(\boldsymbol{\pi}_T, \boldsymbol{\Phi}_T)$ by Eq. (3)
15:     end for
16:     Compute the meta-gradient $\mathbf{g}_k$ via Eq. (6) obtained by differentiating DICE in Eq. (12)
17:     Update meta-solver's parameters $\boldsymbol{\theta}' = \boldsymbol{\theta} - \beta \frac{1}{K} \sum_k \mathbf{g}_k$.
18: end for
```

C.3 Optimising the Meta-Solver through Evolution Strategies

ES-NAC with Approximate Tabular Best-response V1

Here we provide details of NAC-ES where we use Tabular Approximate Best-Response V1 as the best-response oracle in Alg. (5).

Algorithm 5 ES-NAC with Approximate Tabular Best-response V1

```
Require: Game distribution $p(\boldsymbol{G})$, outer learning rate $\alpha$, time window $T$, perturbations $n$, precision $\sigma$.
1: Randomly initialise policy pool $\boldsymbol{\phi_0}$, Initialise parameters $\boldsymbol{\theta}$ of the meta solver $f_{\boldsymbol{\theta}}$.
2: for each training iteration do
3:     Sample games $\{G_k\}_{k=1,\dots,K}$ from $p(\boldsymbol{G})$.
4:     Sample $\boldsymbol{\epsilon}_1, \dots, \boldsymbol{\epsilon}_n \sim \mathcal{N}(0, I)$ and store $n$ models $f_{(\boldsymbol{\theta}+\boldsymbol{\epsilon}_i)}$.
5:     for each stored model $f$ do
6:         for each game $G_k$ do
7:             for each iteration $t$ do
8:                 Compute the meta-policy $\boldsymbol{\pi}_{t-1} = f(\mathbf{M}_{t-1})$.
9:                 Initialise tabular best-response policy $\boldsymbol{\phi}_t^{\text{BR}}$.
10:                for each state $s$ do
11:                    for each action $a$ do
12:                        Get exp. val. of $a$ against $\boldsymbol{\pi}_{t-1}$, $\mathbb{E}_{\boldsymbol{\pi}_{t-1}}(v(a)|s)$, by traversing game-tree.
13:                    end for
14:                    Compute $a' = \arg\max_a \mathbb{E}_{\boldsymbol{\pi}_{t-1}}(v(a)|s)$
15:                    Set $\boldsymbol{\phi}_t^{\text{BR}}(a'|s) = 0.75$ and $\boldsymbol{\phi}_t^{\text{BR}}(a^{\neg\prime}|s) = 0.25$
16:                end for
17:                Expand the population $\boldsymbol{\Phi_t} = \boldsymbol{\Phi_{t-1}} \cup \{\boldsymbol{\phi}_t^{\text{BR}}\}$
18:            end for
19:            Compute the meta-policy $\boldsymbol{\pi}_T = f(\mathbf{M}_T)$.
20:            Compute $\mathfrak{Exp}_i(\boldsymbol{\pi}_T, \boldsymbol{\Phi}_T)$ by Eq. (3)
21:        end for
22:    end for
23:    Compute the meta-gradient $\mathbf{g}_k$ via Eq. (13)
24:    Update meta-solver's parameters $\boldsymbol{\theta}' = \boldsymbol{\theta} - \alpha\frac{1}{K}\sum_k \mathbf{g}_k$.
25: end for
```

ES-NAC with Approximate Tabular Best-response V2

Here we provide details of NAC-ES where we use Tabular Approximate Best-Response V2 as the best-response oracle in Alg. (6).

Algorithm 6 ES-NAC with Approximate Tabular Best-response V2

```
Require: Game distribution $p(\boldsymbol{G})$, outer learning rate $\alpha$, time window $T$, perturbations $n$, precision
    $\sigma$.
 1: Randomly initialise policy pool $\boldsymbol{\phi_0}$, Initialise parameters $\boldsymbol{\theta}$ of the meta solver $f_{\boldsymbol{\theta}}$.
 2: for each training iteration do
 3:     Sample games $\{G_k\}_{k=1,...,K}$ from $p(\boldsymbol{G})$.
 4:     Sample $\boldsymbol{\epsilon}_1, ..., \boldsymbol{\epsilon}_n \sim \mathcal{N}(0, I)$ and store $n$ models $f_{(\boldsymbol{\theta}+\boldsymbol{\epsilon}_i)}$.
 5:     for each stored model $f$ do
 6:         for each game $G_k$ do
 7:             for each iteration $t$ do
 8:                 Compute the meta-policy $\boldsymbol{\pi}_{t-1} = f(\mathbf{M}_{t-1})$.
 9:                 Initialise tabular best-response policy $\boldsymbol{\phi}_t^{\text{BR}}$.
10:                 for each state $s$ do
11:                     for each action $a$ do
12:                         Get exp. val. of $a$ against $\boldsymbol{\pi}_t$, $\mathbb{E}_{\boldsymbol{\pi}_{t-1}}(v(a)|s)$, by traversing game-tree.
13:                     end for
14:                     Compute $a' = \arg\max_a \mathbb{E}_{\boldsymbol{\pi}_{t-1}}(v(a)|s)$
15:                     Sample $\boldsymbol{\eta}_1, \boldsymbol{\eta}_2 \sim \mathcal{N}(0, 1)$
16:                     Set $\boldsymbol{\phi}_t^{\text{BR}}(a'|s) = 1 + \boldsymbol{\eta}_1$ and $\boldsymbol{\phi}_t^{\text{BR}}(a^{\neg\prime}|s) = \boldsymbol{\eta}_2$
17:                     Normalise $\boldsymbol{\phi}_t^{\text{BR}}(s)$
18:                 end for
19:                 Expand the population $\boldsymbol{\Phi_t} = \boldsymbol{\Phi_{t-1}} \cup \{\boldsymbol{\phi}_t^{\text{BR}}\}$
20:             end for
21:             Compute the meta-policy $\boldsymbol{\pi}_T = f(\mathbf{M}_T)$.
22:             Compute $\mathfrak{Exp}_i(\boldsymbol{\pi}_T, \boldsymbol{\Phi}_T)$ by Eq. (3)
23:         end for
24:     end for
25:     Compute the meta-gradient $\mathbf{g}_k$ via Eq. (13)
26:     Update meta-solver's parameters $\boldsymbol{\theta}' = \boldsymbol{\theta} - \alpha\frac{1}{K}\sum_k \mathbf{g}_k$.
27: end for
```

D Additional Experimental Results

The Pseudo code is in Appendix C. In this section we offer additional experimental results for better illustration of NAC.

D.1 Kuhn Poker Experiments

We provide the in-task training results for the Kuhn Poker exact tabular best-response method in Fig. (9), which was used to generate the exact best-response generalisation results from Kuhn Poker to Leduc Poker in Fig. (5). Notably, whilst our model is slightly outperformed by PSRO, both achieve an exploitability of very close to 0 and therefore have both converged to an ϵ-Nash equilibrium.

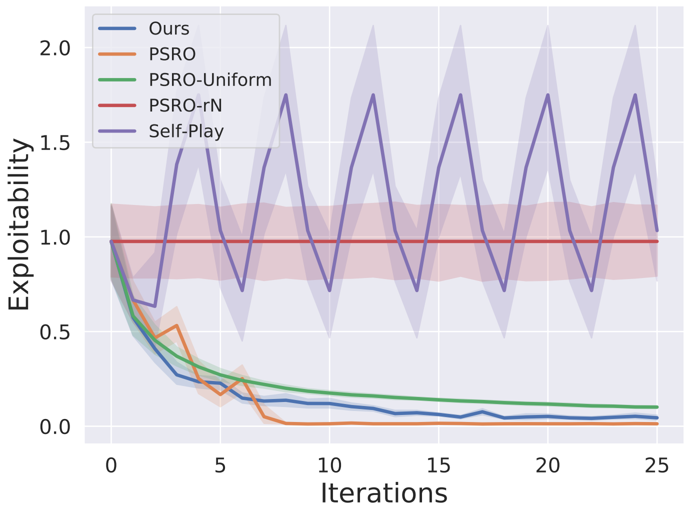

Figure 9: In-task training performance on Kuhn Poker using an exact tabular best-response oracle.

D.2 Visualisation of the Learned Curricula

In Fig. (D.2) we offer a truncated view into the auto-curricula generated by PSRO and NAC. Here, we extend the visualisation to the full iterative process for PSRO and NAC in Fig. (10) and Fig. (11) respectively. Due to the the approximate best-response setting, PSRO converges at iteration 7 and fail to reach all 7 Gaussian distributions. We suspect this is because fictitious play can only generate a difficult to beat auto-curricula without considering whether the best-response process is capable of learning a strong enough policy. In contrast, NAC generates a more smooth and appropriate auto-curricula, in which even an approximate best-response is able to learn a useful policy to explore each distribution one by one. Finally NAC essentially explores all 7 Gaussian distributions and achieves lower exploitability. Another interesting point is that the meta-solver only offers higher probability over the points near the Gaussian centres, which validates its ability to accurately evaluate the policies in a population.

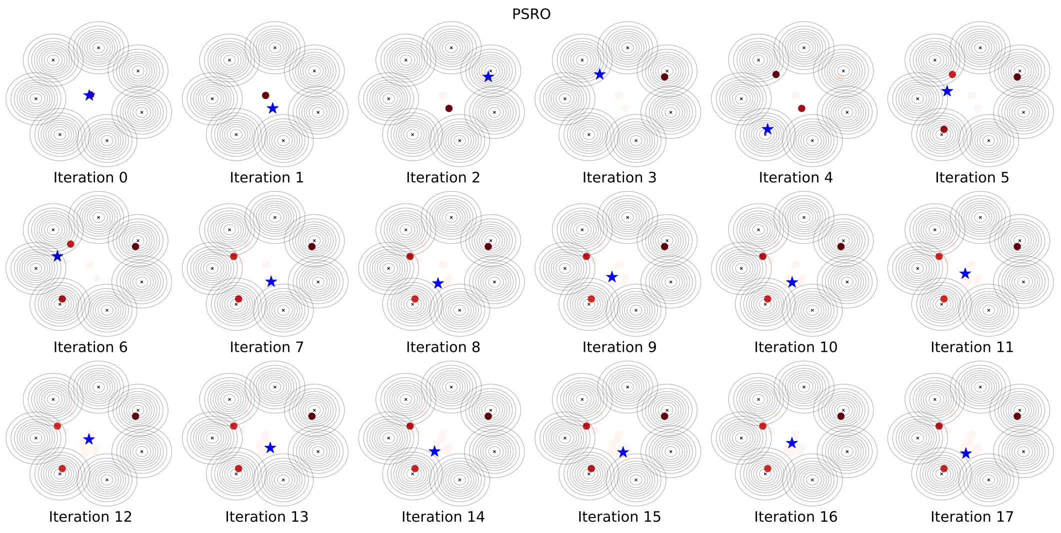

Figure 10: Visualisation of the full curricula on 2D-RPS using PSRO. Red points denote the meta-solver output, and darker refers to higher probability in π. The blue star is the latest best-response.

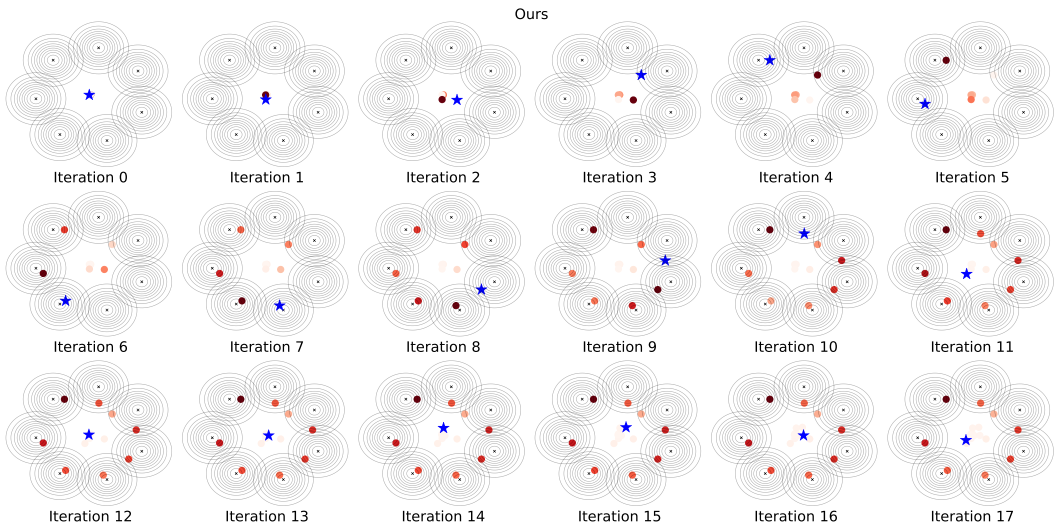

Figure 11: Visualisation of the full curricula on 2D-RPS using NAC. Red points denote the meta-solver output, and darker refers to higher probability in π. The blue star is the latest best-response.

In addition, we offer a similar visualisation of 12 iterations' worth of policy distributions produced by NAC on Kuhn Poker. Due to the approximate best-response limitation, it's difficult for NAC to get the exact Nash equilibrium policy. However, the final policy distribution of NAC still achieves a great approximation to the exact one.

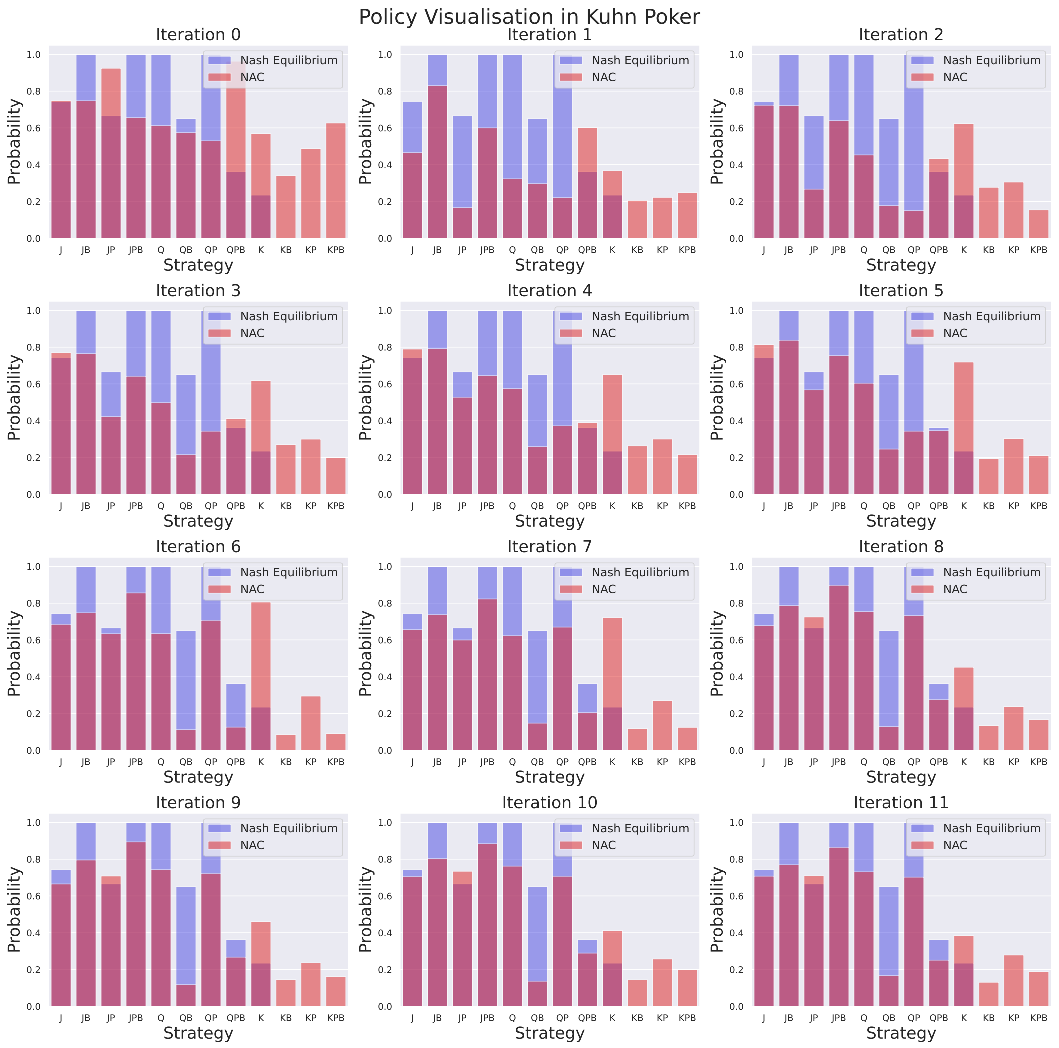

Figure 12: Visualisations of the whole 12 iterations policy distribution on Kuhn Poker for NAC and exact Nash Equilibrium. The Orange Red bar and Lighr blue bar refer to policy distribution for NAC and exact Nash Equilibrium respectively. The pink red bar represents the overlapping policy distribution.

D.3 Meta-Game Generalisation Results

[8] introduced the concept of Games of Skill where certain real-world games share a similar structure in terms of their respective meta-games, and this work additionally released a collection of meta-games sharing this structure. As we utilise Randomly generated Games of Skill as our training game for our NFG meta-solver, we additionally test the ability of our learned meta-solver to generalise to unseen Games of Skill in the collection of meta-games from [8].

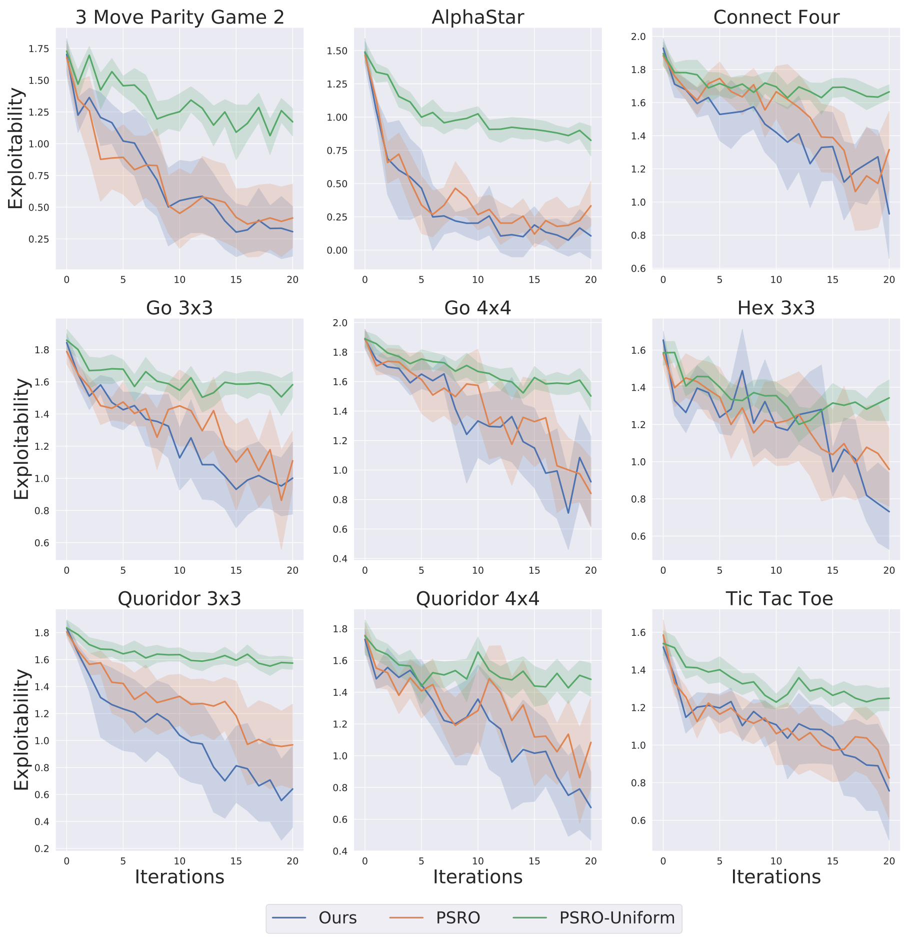

Figure 13: Exploitability results on the meta-games introduced in [8]. NAC performs at least as well as the best baseline in all settings, and often outperforms the PSRO baselines.

D.4 Meta-solver trained with Reinforcement Learning

In our paper, we also train the meta-solver with Reinforcement Learning. Note that RL here refers to the technique for training the meta-solver rather than the best-response oracle. In particular, we can treat the whole PSRO iteration as an environment, the curricula generated by the meta-solver as action, and the negative exploitability as the reward for the meta-solver RL agent. In other words, we formulate the PSRO process as an independent MDP, similar to [11]. Following this idea, we reformulate the training of the meta-solver as an RL problem with a continuous action space and solve such an MDP with Deep Deterministic Policy Gradient (DDPG).

We conduct experiments to train the meta-solver with DDPG on 2D-RPS. Empirically we find that the trained meta-solver can achieve better performance compared with PSRO-Uniform. However, it cannot beat PSRO, unlike our meta-gradient based meta-solver. We believe that this might be because the dynamics of the PSRO environment is complicated which makes it rather challenging for DDPG to learn a good policy (i.e., the meta-solver).

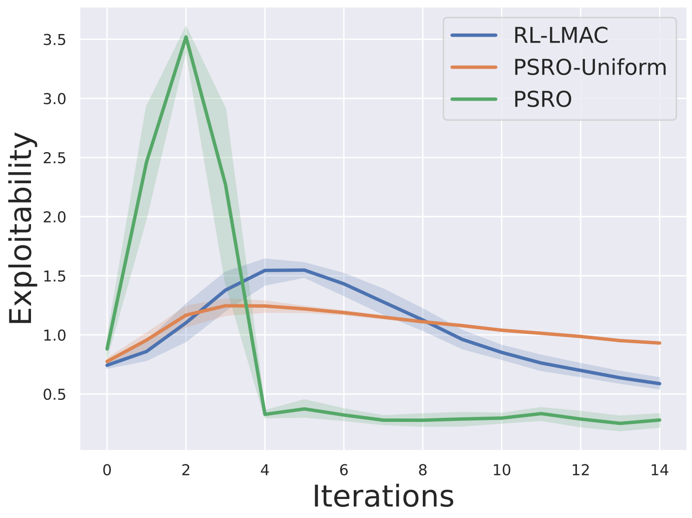

Figure 14: NAC trained with DDPG for environment on 2D-RPS.

E Additional Implementation Details

We report any relevant additional implementation details in this section.

The code can be found in: https://github.com/waterhorse1/NAC

E.1 Environment Description

Random Games of Skill [8] are normal-form games designed to consist of both a transitive and non-transitive element. The payoff function is shown as: $G_{i,j} := \frac{1}{2}(W_{i,j} - W_{j,i}) + S_i - S_j$ with $W_{i,j}, S_i \overset{i.i.d}{\sim} \mathcal{N}(0, \sigma_W^2 \text{ or } \sigma_S^2)$. The intuition behind random games of skill is to model the transitive strength of a strategy via S and the non-transitive cycles by W. [8] shows that many real worlds game exhibits the geometry property of games of skill.

In our experimental setting, we increase the presence of non-transitive cycles by substituting $\frac{1}{2}(W_{i,j} - W_{j,i})$ with $(W_{i,j} - W_{j,i})$. Note that random games of skill naturally provides us with a distribution $P(G)$ over games. We set the meta training distribution on 200*200 games of skill matrix and utilise gradient descent for best-response policy update and exploitability calculation in PSRO. This is a symmetric game so we only need to construct one policy pool for PSRO. Note that the best-response for exploitability calculation in GOS is gradient descent rather than direct maximisation over all pure actions, so it is actually an approximate exploitability. It might bring in negative exploitability but it is still a fair comparison because we use the same way to calculate the exploitability for all algorithms.

Differentiable Lotto. Differentiable lotto is a game inspired by [16] and is introduced in [3]. This game is defined over a fixed set of customers $c_i \in \mathbb{R}^2, i \in \{1, ..., n\}$ where each customer represents a fixed point on a 2D plane. In this game, each agent determines $\{(p_1, \mathbf{v}_1), \ldots, (p_k, \mathbf{v}_k)\}$, where v_i and p_i respectively denote the position and the units of resources of server i. Given two agent $(\mathbf{p}, \mathbf{v})$ and $(\mathbf{q}, \mathbf{w})$, the customers c are softly assigned to servers based on the distance between customer and server. The payoff function is then given as: $\phi((\mathbf{p}, \mathbf{v}), (\mathbf{q}, \mathbf{w})) := \sum_{i,j=1}^{c,k} (p_j v_{ij} - q_j w_{ij})$, where $(v_{i1}, \ldots, w_{ik}) := \text{softmax}(-\|\mathbf{c}_i - \mathbf{v}_1\|^2, \ldots, -\|\mathbf{c}_i - \mathbf{w}_k\|^2)$. This game is a relatively more transitive game compared with Random Games of Skill and Non-transitive Mixture Model game. And since there exists infinite points over 2d plane, Differentiable Lotto is an open-ended game.

In our experiments, we use 9 customers and 500 servers. We set meta-training distribution by randomizing the customers positions and the initial positions of servers according to $\mathcal{N}(0, 1)$. Gradient descent is utilized for best-response policy update and exploitability calculation in PSRO. Note that this is a symmetric game so we only need to construct one policy pool for PSRO.

Non-Transitive Mixture Model. Non-Transitive Mixture Model is also an open-ended game with both transitivity and non-transitivity. To achieve Nash policy, the player needs to not only climb up to the Gaussian distribution to maximize transitive payoff, but also explore each Gaussian distribution to remain un-exploitable.

In our experiment, we set $n = 7$, and randomize the center of Gaussian distribution and initial position of strategies for meta-training distribution. Gradient descent is utilized for best-response policy update and exploitability calculation in PSRO. Note that this is a symmetric game so we only need to construct one policy pool for PSRO.

Iterated Matching Pennies (IMP).

Table 1: Matching pennies

	Head	Tail
Head	(+a, -a)	(-a, +a)
Tail	(-b, +b)	(+b, -b)

We follow the works of [14] and [20] in using IMP [15], a zero-sum game in which the row player wants to have matching pennies whilst the column player wants to have clashing pennies. The original matching pennies game is shown in Table (1) as $a = b = 1$. We extend it to the iterated form where agents can condition their actions on past history. We follow [14] to model it as a memory-1 two-agent MRP and agent's action at timestep t will condition on the joint action at timestep $t-1$. As mentioned in Section (3.5), our training framework, alongside most meta-learning frameworks, cannot tolerate a large amount of inner-loop gradient steps. Fortunately, IMP is fairly simple and does not need to take many policy gradient steps to reach an approximate best-response. We show Table (1) as the stage-game of the iterated game played in IMP. We set $a, b \sim U(0.5, 2)$ as the meta-training distribution over the game. Policy gradient is utilized for best-response policy update and exploitability calculation in PSRO. The iteration length is 50.

In IMP, we follow the setting in [14] where each agent's policy is fully specified by 5 probabilities. For agent a in IMP, they are the probability of head at game start $\pi^a(H|S_0)$, and the head probabilities in the four memories: $\pi^a(H|HH)$, $\pi^a(H|HT)$, $\pi^a(H|TH)$ and $\pi^a(H|TT)$. Note that this is a non-symmetric game so we need to construct two policy pools for PSRO.

Kuhn Poker was introduced by [22] as a two-player, sequential-move, imperfect information poker game which has a total of 6 information states for each player, 12 overall. A round of Kuhn Poker is as follows: Both players start with 2 chips and both put in 1 chip in order to play. The deck is only 3 cards, and each player is dealt one card. At this point, both players have the choice of betting or passing - if both players take the same action then the player with the higher card wins, otherwise the player who made a bet wins. Kuhn Poker is a simplified version of poker which can be easily integrated with game theoretic analysis and is therefore well-aligned with the use of PSRO. Kuhn Poker has a large strategy space consisting 2^{12} pure strategies in total.

In Kuhn Poker it is easy to find an exact best-response to a meta-strategy by traversing the game-tree and selecting the the action at each state with the highest expected value. The results of using this exact approach are shown in Appendix D.1. However, the central interest of our work is when using approximate best-responses, so we also suggest two different manners in which we can specify an approximate best-response when traversing the game-tree.

In Pseudo-code 5 we illustrate our first method, where the action with the highest expected value at each state will be played the majority of the time, but the policy may also take the action with the lower expected value with a lower probability. Notably, these action probability values are fixed at 0.75 and 0.25. We note that, whilst this setting performs well on the Kuhn Poker game, it is not able to generalise effectively to Leduc Poker.

In Pseudo-code 6 we illustrate our second method, where again the action with the highest expected value at each state will be played the majority of the time, but we introduce more randomness into the process. Strictly, we sample two perturbations $\boldsymbol{\eta}_1, \boldsymbol{\eta}_2 \sim \mathcal{N}(0, 1)$ and the action with the highest expected value will be played with probability $1 + \eta_1$ and the other action will be played with probability η_2. We believe this to be a fundamentally more applicable measure as, because in Kuhn Poker it is difficult to define a distribution over games (as there is only one Kuhn Poker game), we instead have this setting defined over the distribution of best-responses allowing us to maintain a distribution setting. We believe this distribution over best-responses is what allows this method to generalise well to Leduc Poker, as it is able to explore more dynamics of the game-type.

E.2 Implementation Details

In this section we will list any specific implementation details that we used for each meta-solver training experiment.

NAC With Gradient-Descent Best-Response Oracles

- In order to control for any instances of gradient explosion, we apply gradient clip normalisation on the meta-gradient with the clip parameter being reported in Appendix F.
- In order to speed up the training process, we distributed each game in a batch across multiple training nodes.

NAC With Gradient-Descent Best-Response Oracles - Implicit

- In order to control for any instances of gradient explosion, we apply gradient clip normalisation on the meta-gradient with the clip parameter being reported in Appendix F.
- In order to meet the stationary point condition for implicit gradient, we take enough inner-loop gradient steps until the gradient norm is below the threshold.

- In order to speed up the training process, we distributed each game in a batch across multiple training nodes.

NAC With Reinforcement Learning Best-Response Oracles

- In order to control for any instances of gradient explosion, we apply a special trick - layer-wise gradient normalisation on the meta-gradient with the clip parameter being reported in Appendix F.
- In order to speed up the training process, we distributed each game in a batch across multiple training nodes.
- We apply linear baseline method in the inner-loop rl based best-response for variance reduction. This is a commonly used strategy for reinforcement learning based meta learning[12].

ES-NAC

- In order to speed up the training process, we distributed each perturbation of the meta-solver across multiple training nodes.

E.3 Meta-testing

There exist some differences between the baseline algorithms and NAC when we conduct meta-testing. Since the baseline algorithms need no further training, in the testing phase, we evaluate the baseline algorithms on multiple tasks sampled from the task distribution so the confidence interval for baseline algorithms refers to the randomness brought by different tasks. However, since we need to conduct the training on multiple seeds for NAC (so there exist two random variables - task and random seed), we follow previous Meta-RL evaluation way [39] to have each trained model tested on multiple tasks and calculate the mean exploitability over tasks. It can reduce the randomness brought by task distribution. So the confidence interval in the plot for NAC refers to the standard deviation brought by different random seeds.

E.4 Computing Infrastructure

We used two internal compute servers both consisting of 4x Nvidia GeForce 1080-Ti cards, however each model is trained on at most 1 card. Additionally we made use of High Performance Computing Cluster for ES experiments.

F Hyperparameter Details

We report our hyperparameter settings we use for experiments in this section.

F.1 Games of Skill - Alg. 2

Table 2: Hyper-parameter settings for Random Games of Skill Training.

SETTINGS	VALUE	DESCRIPTION
ORACLE METHOD	GRADIENT DESCENT	SUBROUTINE OF GETTING ORACLES
OUTER LEARNING RATE	0.01	LEARNING RATE FOR META-SOLVER UPDATES
META TRAINING STEPS	100	NUMBER OF META-SOLVER UPDATE STEPS
META BATCH SIZE	5	NUMBER OF GAMES TRAINED ON PER ITERATION
MODEL TYPE	GRU	TYPE OF META-SOLVER
GRADIENT CLIP VALUE	1.0	META-GRADIENT CLIP VALUE
PSRO ITERATIONS	20	NUMBER OF PSRO ITERATIONS
WINDOW SIZE	5	NUMBER OF WINDOW SIZE
INNER LEARNING RATE	25.0	LEARNING RATE FOR BEST-RESPONSE UPDATES
INNER GD STEPS	5	NUMBER OF BEST-RESPONSE UPDATE STEPS
EXPLOITABILITY LEARNING RATE	10.0	LEARNING RATE FOR EXPLOITABILITY CALCULATION
INNER EXPLOITABILITY STEPS	20	NUMBER OF EXPLOITABILITY UPDATE STEPS

F.2 Differentiable Blotto - Alg. 2

Table 3: Hyper-parameter settings for Differentiable Lotto Training.

Settings	Value	Description
Oracle method	Gradient Descent	subroutine of getting oracles
Outer Learning rate	0.001	Learning rate for meta-solver updates
Meta training Steps	100	Number of meta-solver update steps
Meta Batch Size	5	Number of games trained on per iteration
Model Type	GRU	Type of meta-solver
Gradient Clip Value	1.0	Meta-gradient clip value
PSRO Iterations	20	Number of PSRO Iterations
Window Size	5	Number of window size
Inner Learning rate	20.0	Learning rate for best-response updates
Inner GD Steps	20	Number of best-response update steps
Exploitability Learning rate	20.0	Learning rate for exploitability calculation
Inner Exploitability Steps	30	Number of exploitability update steps

F.3 Non-transitive Mixture Model

F.3.1 Best response by Non-implicit Gradient Descent - Alg. 2

Table 4: Hyper-parameter settings for non-implicit 2D-RPS Training.

Settings	Value	Description
Oracle method	Gradient Descent	subroutine of getting oracles
Outer Learning rate	0.007	Learning rate for meta-solver updates
Meta training steps	400	Number of meta-solver update steps
Meta Batch Size	8	Number of games trained on per iteration
Model Type	Conv1D	Type of meta-solver
LR Schedule Step	100	Outer LR Scheduler step iteration
LR Schedule Gamma	0.3	Outer LR Scheduler multiplicative value
Gradient Clip Value	2.0	Meta-gradient clip value
PSRO Iterations	15	Number of PSRO Iterations
Window Size	9	Number of window size
Inner Learning rate	2.0	Learning rate for best-response updates
Inner GD Steps	5	Number of best-response update steps
Exploitability Learning rate	2.0	Learning rate for exploitability calculation
Inner Exploitability Steps	20	Number of exploitability update steps

F.3.2 Best response by Implicit Gradient Descent - Alg. 3

Table 5: Hyper-parameter settings for implicit 2D-RPS Training.

Settings	Value	Description
Oracle method	Gradient Descent	subroutine of getting oracles
Outer Learning rate	0.005	Learning rate for meta-solver updates
Meta training steps	600	Number of meta-solver update steps
Meta Batch Size	10	Number of games trained on per iteration
Model Type	Conv1D	Type of meta-solver
Gradient Clip Values	0.002	Value above which meta-gradient is clipped
PSRO Iterations	10	Number of PSRO Iterations
Window Size	10	Number of window size
Inner Learning rate	0.75	Learning rate for best-response updates
Inner GD Steps	100	Number of best-response update steps
Exploitability Learning rate	0.75	Learning rate for exploitability calculation
Inner Exploitability Steps	200	Number of exploitability update steps
Inner-loop Gradient Norm Break Value	0.001	Value at which inner-loop gradient update is stopped.

F.4 Iterated Matching Pennies - Alg. 4

Table 6: Hyper-parameter settings for Iterated Matching Pennies Training.

Settings	Value	Description
Oracle method	Reinforce	subroutine of getting oracles
Outer Learning rate	0.004	Learning rate for meta-solver updates
Meta Training Steps	50	Number of meta-solver update steps
Meta Batch Size	8	Number of games trained on per iteration
Model Type	GRU	Type of meta-solver
Layer-wise gradient normalisation threshold	0.002	Value above which layer-wise meta-gradient is clipped
PSRO Iterations	9	Number of PSRO Iterations
Window Size	3	Number of window size
Inner Learning rate	10.0	Learning rate for best-response updates
Inner GD Steps	10	Number of best-response update steps
Exploitability Learning rate	10.0	Learning rate for exploitability calculation
Exploitability Steps	20	Number of exploitability update steps
Trajectories sampled each update	32	Number of trajectories sampled each reinforce update

F.5 Kuhn Poker

F.5.1 Best response by Approximate Tabular V1 - Alg. 5

Table 7: Hyper-parameter settings for Kuhn Poker Tabular V1 Training.

Settings	Value	Description
Oracle method	Approximate Tabular V1	subroutine of getting oracles
Outer Learning rate	0.1	Learning rate for meta-solver updates
Meta Training Steps	100	Number of meta-solver update steps
Meta Batch Size	5	Number of games trained on per iteration
odel Type	Conv1D	Type of meta-solver
LR Schedule Step	50	Outer LR Scheduler step iteration
LR Schedule Gamma	0.5	Outer LR Scheduler multiplicative value
PSRO Iterations	15	Number of PSRO Iterations
ES Perturbations	30	Number of model perturbations via ES

F.5.2 Best response by Approximate Tabular V2 - Alg. 6

Table 8: Hyper-parameter settings for Kuhn Poker Tabular V2 Training.

Settings	Value	Description
Oracle method	Approximate Tabular V2	subroutine of getting oracles
Outer Learning rate	0.1	Learning rate for meta-solver updates
Meta Training Steps	100	Number of meta-solver update steps
Meta Batch Size	5	Number of games trained on per iteration
Model Type	Conv1D	Type of meta-solver
LR Schedule Step	20	Outer LR Scheduler step iteration
LR Schedule Gamma	0.5	Outer LR Scheduler multiplicative value
PSRO Iterations	15	Number of PSRO Iterations
ES Perturbations	30	Number of model perturbations via ES

F.5.3 Best response by PPO

Table 9: Hyper-parameter settings for Kuhn Poker PPO Training.

SETTINGS	VALUE	DESCRIPTION
ORACLE METHOD	PPO	SUBROUTINE OF GETTING ORACLES
OUTER LEARNING RATE	0.2	LEARNING RATE FOR META-SOLVER UPDATES
META TRAINING STEPS	100	NUMBER OF META-SOLVER UPDATE STEPS
META BATCH SIZE	3	NUMBER OF GAMES TRAINED ON PER ITERATION
MODEL TYPE	CONV1D	TYPE OF META-SOLVER
LR SCHEDULE STEP	20	OUTER LR SCHEDULER STEP ITERATION
LR SCHEDULE GAMMA	0.5	OUTER LR SCHEDULER MULTIPLICATIVE VALUE
ES PERTURBATIONS	30	NUMBER OF MODEL PERTURBATIONS VIA ES
PSRO ITERATIONS	10	NUMBER OF PSRO ITERATIONS
PPO CLIP RATIO	0.8	CLIP RATIO OF PPO TRAINER
PI LR	0.003	LR FOR POLICY OPTIMISER
VF LR	0.001	LR FOR VALUE FUNCTION OPTIMISER
PI TRAIN ITERS	100	NUMBER OF POLICY OPTIMISER TRAINING ITERS.
VF TRAIN ITERS	100	NUMBER OF VF OPTIMISER TRAINING ITERS.
TARGET KL	0.5	EARLY STOPPING CRITERIA

G Author Contributions

We summarise the main contributions from each of the authors as follows:

Xidong Feng: Idea proposing, algorithm design, code implementation and experiments running (on 2D-RPS, 2D-RPS-Implicit and IMP), and paper writing.

Oliver Slumbers: Algorithm design, code implementation and experiments running (on Gos, Blotto, Kuhn-Poker), and paper writing.

Ziyu Wan: Code implementation and experiments running for RL based NAC in Appendix D.4.

Bo Liu: Experiments running for Kuhn-Poker.

Stephen McAleer: Project discussion and paper writing.

Ying Wen: Project discussion.

Jun Wang: Project discussion and overall project supervision.

Yaodong Yang: Project lead, idea proposing, experiment supervision, and paper writing.