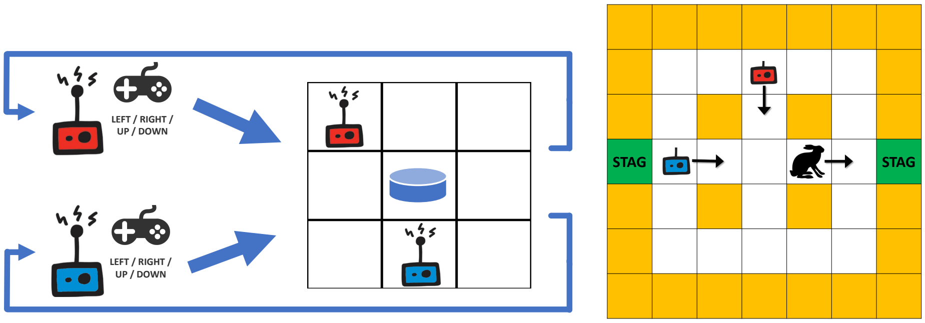
# *Appendix for*
# STATUS-QUO POLICY GRADIENT IN MULTI-AGENT REINFORCEMENT LEARNING

## A DESCRIPTION OF ENVIRONMENTS USED FOR DYNAMIC SOCIAL DILEMMAS



(a) (b)

Figure 5: Illustration of two agents (Red and Blue) playing the dynamic games: (a) Coin Game and the (b) Stag-Hunt Game

### A.1 COIN GAME

Figure 5a illustrates the agents playing the Coin Game. The agents, along with a Blue or Red coin, appear at random positions in a $3 \times 3$ grid. An agent observes the complete $3 \times 3$ grid as input and can move either left, right, up, or down. When an agent moves into a cell with a coin, it picks the coin, and a new instance of the game begins where the agent remains at their current positions, but a Red/Blue coin randomly appears in one of the empty cells. If the Red agent picks the Red coin, it gets a reward of $+1$, and the Blue agent gets no reward. If the Red agent picks the Blue coin, it gets a reward of $+1$, and the Blue agent gets a reward of $-2$. The Blue agent's reward structure is symmetric to that of the Red agent.

### A.2 STAG-HUNT

Figure 5b shows the illustration of two agents (Red and Blue) playing the visual Stag Hunt game. The STAG represents the maximum reward the agents can achieve with HARE in the center of the figure. An agent observes the full $7 \times 7$ grid as input and can freely move across the grid in only the empty cells, denoted by white (yellow cells denote walls that restrict the movement). Each agent can either pick the STAG individually to obtain a reward of $+4$, or coordinate with the other agent to capture the HARE and obtain a better reward of $+25$.

## B $GameDistill$: ARCHITECTURE AND PSEUDO-CODE

### B.1 $GameDistill$: ARCHITECTURE DETAILS

$GameDistill$ consists of two components.

**The first component is the state sequence encoder** that takes as input a sequence of states (input size is $4 \times 4 \times 3 \times 3$, where $4 \times 3 \times 3$ is the dimension of the game state, and the first index in the state input represents the data channel where each channel encodes data from both all the different colored agents and coins) and outputs a fixed dimension feature representation. We encode each state in the sequence using a common trunk of 3 convolution layers with *relu* activations and kernel-size $3 \times 3$, followed by a fully-connected layer with $100$ neurons to obtain a finite-dimensional feature representation. This unified feature vector, called the trajectory embedding, is then given as input to the different prediction branches of the network. We also experiment with different dimensions of this embedding and provide results in Figure 6.

The two branches, which predict the self-reward and the opponent-reward (as shown in Figure 1), independently use this trajectory embedding as input to compute appropriate output. These branches take as input the trajectory embedding and use a dense hidden layer (with 100 neurons) with linear activation to predict the output. We use the *mean-squared error (MSE)* loss for the regression tasks in the prediction branches. Linear activation allows us to cluster the trajectory embeddings using a linear clustering algorithm, such as Agglomerative Clustering (Friedman et al., 2001). In general, we can choose the number of clusters based on our desired level of granularity in differentiating outcomes. In the games considered in this paper, agents broadly have two types of policies. Therefore, we fix the number of clusters to two.

We use the *Adam* (Kingma & Ba, 2014) optimizer with learning-rate of $3e-3$. We also experiment with K-Means clustering in addition to Agglomerative Clustering, and it also gives similar results. We provide additional results of the clusters obtained using $GameDistill$ in Appendix D.

**The second component is the oracle network** that outputs an action given a state. For each oracle network, we encode the input state using 3 convolution layers with kernel-size $2 \times 2$ and *relu* activation. To predict the action, we use 3 fully-connected layers with *relu* activation and the cross-entropy loss. We use *L2* regularization, and *Gradient Descent* with the *Adam* optimizer (learning rate $1e-3$) for all our experiments.

### B.2 $GameDistill$: PSEUDO-CODE

**Algorithm 1:** Pseduo-code for $GameDistill$

```
1  Collect list of episodes with $(r1, r2) > 0$ from random game play;
2  for agents do
3      Create dataset: $\{listEpisodes, myRewards, opponentRewards\} \leftarrow \{[\,], [\,], [\,]\}$;
4      for episode in episodes do
5          for (s,a,r,s') in episode do
6              if $r > 0$ then
7                  add sequence of last three states leading up to $s'$ to listEpisodes ;
8                  add respective rewards to myRewards and opponentRewards
9              end
10         end
11     end
12     Train Sequence Encoding Network;
13     Train with NetLoss;
14     Cluster embeddings using Agglomerative Clustering;
15     Map episode to clusters from Step 14;
16     Train oracle for each cluster.
17 end
```

## C $SQLoss$: EVOLUTION OF COOPERATION

Equation 6 (Section 2.3.2) describes the gradient for standard policy gradient. It has two terms. The $log\pi^1(u^1_t|s_t)$ term maximises the likelihood of reproducing the training trajectories $[(s_{t-1}, u_{t-1}, r_{t-1}), (s_t, u_t, r_t), (s_{t+1}, u_{t+1}, r_{t+1}), \dots]$. The return term pulls down trajectories that have poor return. The overall effect is to reproduce trajectories that have high returns. We refer to this standard loss as $Loss$ for the following discussion.

**Lemma 1.** For agents trained with random exploration in the IPD, $Q_\pi(D|s_t) > Q_\pi(C|s_t)$ for all $s_t$.

Let $Q_\pi(a_t|s_t)$ denote the expected return of taking $a_t$ in $s_t$. Let $V_\pi(s_t)$ denote the expected return from state $s_t$.

$$\begin{aligned}
Q_\pi(C|CC) &= 0.5[(-1) + V_\pi(CC)] + 0.5[(-3) + V_\pi(CD)] \\
Q_\pi(C|CC) &= -2 + 0.5[V_\pi(CC) + V_\pi(CD)] \\
Q_\pi(D|CC) &= -1 + 0.5[V_\pi(DC) + V_\pi(DD)] \\
Q_\pi(C|CD) &= -2 + 0.5[V_\pi(CC) + V_\pi(CD)] \\
Q_\pi(D|CD) &= -1 + 0.5[V_\pi(DC) + V_\pi(DD)] \\
Q_\pi(C|DC) &= -2 + 0.5[V_\pi(CC) + V_\pi(CD)] \\
Q_\pi(D|DC) &= -1 + 0.5[V_\pi(DC) + V_\pi(DD)] \\
Q_\pi(C|DD) &= -2 + 0.5[V_\pi(CC) + V_\pi(CD)] \\
Q_\pi(D|DD) &= -1 + 0.5[V_\pi(DC) + V_\pi(DD)]
\end{aligned} \tag{9}$$

Since $V_\pi(CC) = V_\pi(CD) = V_\pi(DC) = V_\pi(DD)$ for randomly playing agents, $Q_\pi(D|s_t) > Q_\pi(C|s_t)$ for all $s_t$.

**Lemma 2.** Agents trained to only maximize the expected reward in IPD will converge to mutual defection.

This lemma follows from Lemma 1. Agents initially collect trajectories from random exploration. They use these trajectories to learn a policy that optimizes for a long-term return. These learned policies always play $D$ as described in Lemma 1.

Equation 7 describes the gradient for $SQLoss$. The $log\pi^1(u^1_{t-1}|s_t)$ term maximises the likelihood of taking $u_{t-1}$ in $s_t$. The imagined episode return term pulls down trajectories that have poor imagined return.

**Lemma 3.** Agents trained on random trajectories using only $SQLoss$ oscillate between $CC$ and $DD$.

For IPD, $s_t = (u^1_{t-1}, u^2_{t-1})$. The $SQLoss$ maximises the likelihood of taking $u_{t-1}$ in $s_t$ when the return of the imagined trajectory $\hat{R}_t(\hat{\tau}_1)$ is high.

Consider state $CC$, with $u^1_{t-1} = C$. $\pi^1(D|CC)$ is randomly initialised. The $SQLoss$ term reduces the likelihood of $\pi^1(C|CC)$ because $\hat{R}_t(\hat{\tau}_1) < 0$. Therefore, $\pi^1(D|CC) > \pi^1(C|CC)$.

Similarly, for $CD$, the $SQLoss$ term reduces the likelihood of $\pi^1(C|CD)$. Therefore, $\pi^1(D|CD) > \pi^1(C|CD)$. For $DC$, $\hat{R}_t(\hat{\tau}_1) = 0$, therefore $\pi^1(D|DC) > \pi^1(C|DC)$. Interestingly, for $DD$, the $SQLoss$ term reduces the likelihood of $\pi^1(D|DD)$ and therefore $\pi^1(C|DD) > \pi^1(D|DD)$.

Now, if $s_t$ is $CC$ or $DD$, then $s_{t+1}$ is $DD$ or $CC$ and these states oscillate. If $s_t$ is $CD$ or $DC$, then $s_{t+1}$ is $DD$, $s_{t+2}$ is $CC$ and again $CC$ and $DD$ oscillate. This oscillation is key to the emergence of cooperation as explained in section 2.3.1.

**Lemma 4.** For agents trained using both standard loss and $SQLoss$, $\pi(C|CC) > \pi^1(D|CC)$.

For $CD$, $DC$, both the standard loss and $SQLoss$ push the policy towards $D$. For $DD$, with sufficiently high $\kappa$, the $SQLoss$ term overcomes the standard loss and pushes the agent towards $C$. For $CC$, initially, both the standard loss and $SQLoss$ push the policy towards $D$. However, as training progresses, the incidence of $CD$ and $DC$ diminish because of $SQLoss$ as described in Lemma 3. Therefore, $V_\pi(CD) \approx V_\pi(DC)$ since agents immediately move from both states to $DD$. Intuitively, agents lose the opportunity to exploit the other agent. In equation 9, with $V_\pi(CD) \approx V_\pi(DC)$, $Q_\pi(C|CC) > Q_\pi(D|CC)$ and the standard loss pushes the policy so that $\pi(C|CC) > \pi(D|CC)$. This depends on the value of $\kappa$. For very low values, the standard loss overcomes $SQLoss$ and agents defect. For very high values, $SQLoss$ overcomes standard loss, and agents oscillate between cooperation and defection. For moderate values of $\kappa$ (as shown in our experiments), the two loss terms work together so that $\pi(C|CC) > \pi(D|CC)$.

# D EXPERIMENTAL DETAILS AND ADDITIONAL RESULTS

## D.1 INFRASTRUCTURE FOR EXPERIMENTS

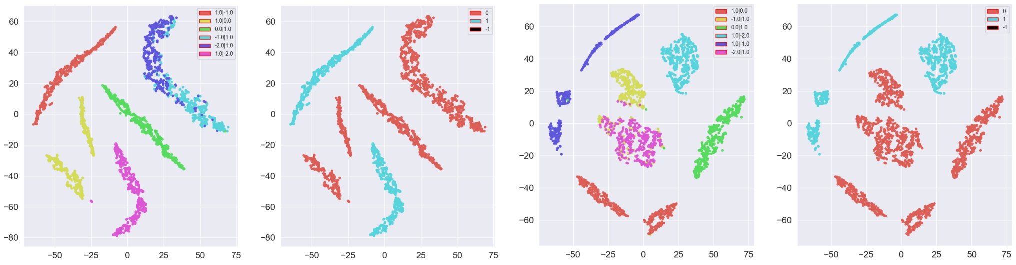
We performed all our experiments on an AWS instance with the following specifications. We use a 64-bit machine with Intel(R) Xeon(R) Platinum 8275CL CPU @ 3.00GHz installed with Ubuntu 16.04LTS operating system. It had a RAM of 189GB and 96 CPU cores with two threads per core. We use the TensorFlow framework for our implementation.

## D.2 SQLOSS

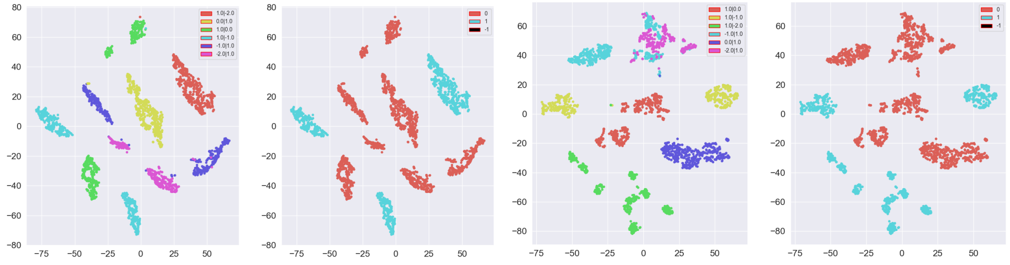
For our experiments with the Selfish and Status-Quo Aware Learner ($SQLearner$), we use policy gradient-based learning to train an agent with the Actor-Critic method (Sutton & Barto, 2011). Each agent is parameterized with a policy actor and critic for variance reduction in policy updates. During training, we use $\alpha = 1.0$ for the REINFORCE and $\beta = 0.5$ for the imaginative game-play. We use gradient descent with step size, $\delta = 0.005$ for the actor and $\delta = 1$ for the critic. We use a batch size of $4000$ for Lola-PG (Foerster et al., 2018) and use the results from the original paper. We use a batch size of 200 for $SQLearner$ for roll-outs and an episode length of 200 for all iterated matrix games. We use a discount rate ($\gamma$) of $0.96$ for the Iterated Prisoners' Dilemma, Iterated Stag Hunt, and Coin Game. For the Iterated Matching Pennies, we use $\gamma = 0.9$ to be consistent with earlier works. The high value of $\gamma$ allows for long time horizons, thereby incentivizing long-term rewards. Each agent randomly samples $\kappa$ from $\mathbb{U} \in (1, z)$ ($z = 10$, discussed in Appendix D.6) at each step.

## D.3 $GameDistill$ CLUSTERING

Figures 6 and 7 show the clusters obtained for the state sequence embedding for the Coin Game and the dynamic variant of Stag Hunt respectively. In the figures, each point is a t-SNE projection of the feature vector (in different dimensions) output by the $GameDistill$ network for an input sequence



(a) 3-dimensional (b) 4-dimensional



(c) 10-dimensional (d) 100-dimensional

Figure 6: Representation of the clusters learned by $GameDistill$ for Coin Game. Each point is a t-SNE projection of the feature vector (in different dimensions) output by the $GameDistill$ network for an input sequence of states. For each of the sub-figures, the figure on the left is colored based on actual rewards obtained by each agent ($r_1|r_2$). The figure on the right is colored based on clusters as learned by $GameDistill$. $GameDistill$ correctly identifies two types of trajectories, one for cooperation and the other for defection.

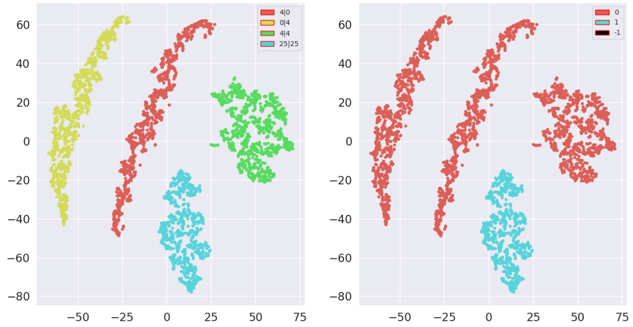


Figure 7: t-SNE plot for the trajectory embeddings obtained from the Stag Hunt game along with the identified cooperation and defection clusters.

of states. For each of the sub-figures, the figure on the left is colored based on actual rewards obtained by each agent ($r_1|r_2$). The figure on the right is colored based on clusters, as learned by $GameDistill$. $GameDistill$ correctly identifies two types of trajectories, one for cooperation and the other for defection for both the games Coin Game and Stag-Hunt.

Figure 6 also shows the clustering results for different dimensions of the state sequence embedding for the Coin Game. We observe that changing the size of the embedding does not have any effect on the results.

### D.4 ILLUSTRATIONS OF TRAINED ORACLE NETWORKS FOR THE COIN GAME

Figure 8 shows the predictions of the oracle networks learned by the Red agent using $GameDistill$ in the Coin Game. We see that the cooperation oracle suggests an action that avoids picking the coin of the other agent (the Blue coin). Analogously, the defection oracle suggests a selfish action that picks the coin of the other agent.

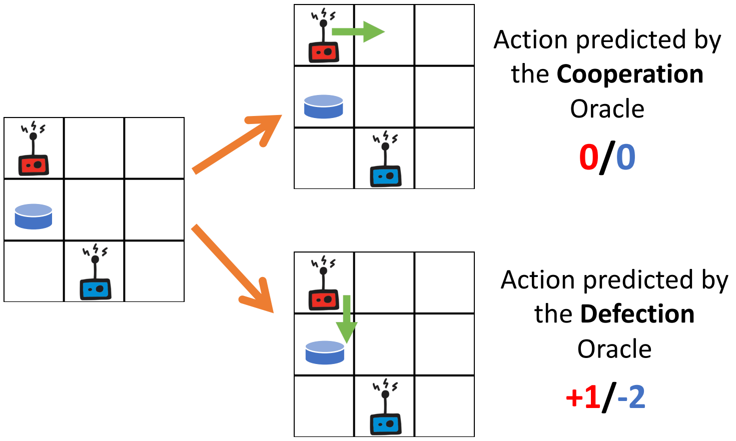


Figure 8: Illustrative predictions of the oracle networks learned by the Red agent using $GameDistill$ in the Coin Game. The numbers in red/blue show the rewards obtained by the Red and the Blue agent respectively. The cooperation oracle suggests an action that avoids picking the coin of the other agent while the defection oracle suggests an action that picks the coin of the other agent

### D.5 RESULTS FOR THE ITERATED STAG HUNT (ISH) USING SQLOSS

We provide the results of training two $SQLearner$ agents on the Iterated Stag Hunt game in Figure 9. In this game also, $SQLearner$ agents coordinate successfully to obtain a near-optimal NDR value (0) for this game.

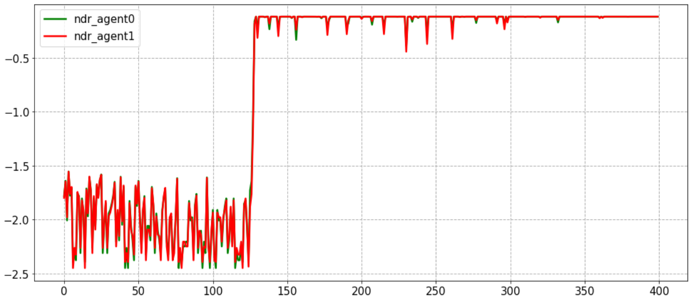


Figure 9: NDR values for $SQLearner$ agents in the ISH game. $SQLearner$ agents coordinate successfully to obtain a near optimal NDR value (0) for this game.

### D.6 $SQLoss$: EFFECT OF $z$ ON CONVERGENCE TO COOPERATION

We explore the effect of the hyper-parameter $z$ (Section 2) on convergence to cooperation, we also experiment with varying values of $z$. In the experiment, to imagine the consequences of maintaining the status quo, each agent samples $\kappa_t$ from the Discrete Uniform distribution $\mathbb{U}\{1, z\}$. A larger value of $z$ thus implies a larger value of $\kappa_t$ and longer imaginary episodes. We find that larger $z$ (and hence $\kappa$) leads to faster cooperation between agents in the IPD and Coin Game. This effect plateaus for $z > 10$. However varying and changing $\kappa_t$ across time also increases the variance in the gradients and thus affects the learning. We thus use $\kappa = 10$ for all our experiments.

### D.7 SQLEARNER: EXPLOITABILITY AND ADAPTABILITY

Given that an agent does not have any prior information about the other agent, it must evolve its strategy based on its opponent's strategy. To evaluate an $SQLearner$ agent's ability to avoid exploitation by a selfish agent, we train one $SQLearner$ agent against an agent that always defects in the Coin Game. We find that the $SQLearner$ agent also learns to always defect. This persistent defection is important since given that the other agent is selfish, the $SQLearner$ agent can do no better than also be selfish. To evaluate an $SQLearner$ agent's ability to exploit a cooperative agent, we train one $SQLearner$ agent with an agent that always cooperates in the Coin Game. In this case, we find that the $SQLearner$ agent learns to always defect. This persistent defection is important since given that the other agent is cooperative, the $SQLearner$ agent obtains maximum reward by behaving selfishly. Hence, the $SQLearner$ agent is both resistant to exploitation and able to exploit, depending on the other agent's strategy.

## E REPRODUCIBILITY CHECKLIST

We follow the reproducibility checklist from (Pineau, 2019) and include further details here. For all the models and algorithms we have included details that we think would be useful for reproducing the results of this work.

- For all **models** and **algorithms** presented, check if you include:
  1. *A clear description of the mathematical setting, algorithm, and/or model*: **Yes**. The algorithm is described in detail in Section 2, with all the loss functions used for training being clearly defined. The details of the architecture, hyperparameters used and other algorithm details are given in Section 3. Environment details are explained in the sections that they are introduced in.

2. *An analysis of the complexity (time, space, sample size) of any algorithm*: **No**. We do not include a formal complexity analysis of our algorithm. However, we do highlight the additional computational steps (in terms of losses and parameter updates) in Section 2 over standard multi-agent independently learning RL algorithms that would be needed in our approach.
3. *A link to a downloadable source code, with specification of all dependencies, including external libraries.*: **Yes**. We have made the source code available at Code (2019).

- For any **theoretical claim**, check if you include:
  1. *A statement of the result*: **NA**. Our paper is primarily empirical and we do not have any major theoretical claims. Hence this is Not Applicable.
  2. *A clear explanation of any assumptions*: **NA**.
  3. *A complete proof of the claim*: **NA**.
- For all **figures** and **tables** that present empirical results, check if you include:
  1. *A complete description of the data collection process, including sample size*: **NA**. We did not collect any data for our work.
  2. *A link to a downloadable version of the dataset or simulation environment*: **Yes**. We have made the source code available at Code (2019).
  3. *An explanation of any data that were excluded, description of any pre-processing step*: **NA**. We did not perform any pre-processing step.
  4. *An explanation of how samples were allocated for training / validation / testing*: **Yes**. For $GameDistill$ the details regarding data used for training is given in Section 2.4. The number of iterations used for learning (training) by $SQLearner$ is given in Figures 3a, 3b and 4a. The details of the number of runs and the batch sizes used for various experiments are given in Section 3.
  5. *The range of hyper-parameters considered, method to select the best hyper-parameter configuration, and specification of all hyper-parameters used to generate results*: **Yes**. We did not do any hyper-parameter tuning as part of this work. All the hyper-parameters that we used are specified in Section 3.
  6. *The exact number of evaluation runs*: **Yes**. For all our environments, we repeat the experiment 20 times. For evaluation of performance, we use an average of 200 Monte Carlo estimates. We state this in Section 3. We do not need to fix any seed. In order to test reproducibility use the seed in the code. The details of the number of runs and the batch sizes used for various experiments are also given here.
  7. *A description of how experiments were run*: **Yes**. The README with instructions on how to run the experiments along with the source code is provided at Code (2019).
  8. *A clear definition of the specific measure or statistics used to report results*: **Yes**. We plot the mean and the one standard deviation region over the mean for all our numerical experiments. This is stated in Section 3.
  9. *Clearly defined error bars*: **Yes**. We plot the mean and the one standard deviation region over the mean for all our numerical experiments. This is stated in Section 3.
  10. *A description of results with central tendency (e.g. mean) & variation (e.g. stddev)*: **Yes**. We plot the mean and the one standard deviation region over the mean for all our numerical experiments. This is stated in Section 3.
  11. *A description of the computing infrastructure used*: **Yes**. We have provided this detail in the Supplementary material in Section D.1.