

Appendix for Status-Quo Policy Gradient in Multi-agent Reinforcement Learning

A Description of Environments Used for Dynamic Social Dilemmas

The three matrix games tested in the paper are canonical games that appear in the sequential social dilemma literature. Hence, we selected these to demonstrate the effectiveness of SQ policy gradient approach. We have not tested our approach beyond the social dilemma setting and hence limit our claims to the same.

A.1 Coin Game

Figure 7 illustrates the agents playing the Coin Game. The agents, along with a Blue or Red coin, appear at random positions in a 3×3 grid. An agent observes the complete 3×3 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.

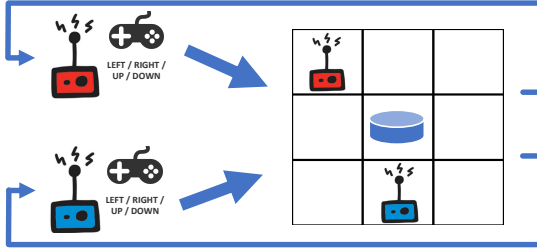


Figure 7: Illustration of two agents (Red and Blue) playing the dynamic game Coin Game

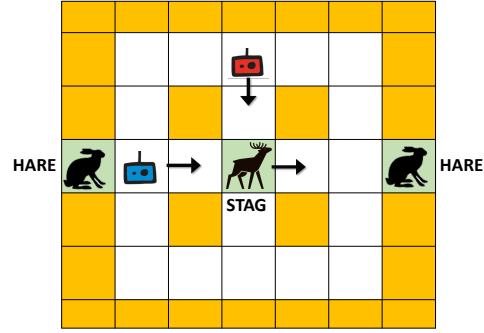


Figure 8: Illustration of two agents (Red and Blue) playing the dynamic game Stag-Hunt Game

A.2 Stag-Hunt

Figure 8 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 7×7 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*: Oracles, Network Architecture and pseudo-code

B.1 Oracles from *GameDistill*

The oracles are important in games with visual input. The agent uses these oracles to reduce the games to their matrix equivalents. While we call the two oracles learned from *GameDistill* as Cooperation and Defection oracles, we do not need the notion of cooperation or defection, nor do we need to explicitly label these clusters as ‘cooperation’ or ‘defection’ to learn these oracles. These oracles are learned by clustering the outcomes of random play into two distinct clusters. In social dilemmas, these two distinct clusters represent cooperation and defection outcomes. Hence, we use the names ‘Cooperation’ and ‘Defection’ oracles for the oracles learned from these clusters. It is

important to mention that *SQLoss* (without oracles) achieves high-degree of cooperation in matrix games.

Algorithm 1 Pseduo-code for *GameDistill*

```

1: Input: Game Environment env, Agents agents, Clustering Technique AggClustering
2: for agent in agents do
3:   t_data = collect_data(env, agent)                                // Collect trajectory data
4:   rewardPredNet = createNetwork()
5:   train_network(rewardPredNet, t_data)
6:   feats = get_features(rewardPredNet, t_data)                    // Extract shared features & cluster
7:   clus_ids = AggClustering(n = 2).fit(feats)
8:   oracle_nets = []                                              // Train oracles corresponding to each cluster
9:   for k in range(2) do
10:    index = np.where(clus_ids == k)
11:    cluster_data = t_data[index]
12:    oracle_nets[k] = create_oracle_net(env)
13:    train_oracle(oracle_nets[k], cluster_data)
14:   end for
15: end for
16: Output: Trained oracle networks oracle_nets

```

In visual-input games with complex actions (such as up, down, left, right, eat-coin, etc.), it is not clear which action or sequence of actions constitute cooperation or defection. In such games, the role of the cooperation oracle is to recommend, at each step, which action (out of up down, left, right, eat-coin, etc.) constitutes *cooperative* behavior. Similarly, the role of the defection oracle is to recommend, at each step, which action constitutes *defection* behavior. Algorithm 1 describes (at a high level) how agents train and use these oracles in the game-play life cycle. Algorithm 6 (in Appendix B) describes how the oracles are trained.

We also provide details about the oracle network architecture from in Appendix B.1 in the supplementary material.

B.2 *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 3×3 , 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 9.

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 and 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 E. **The second**

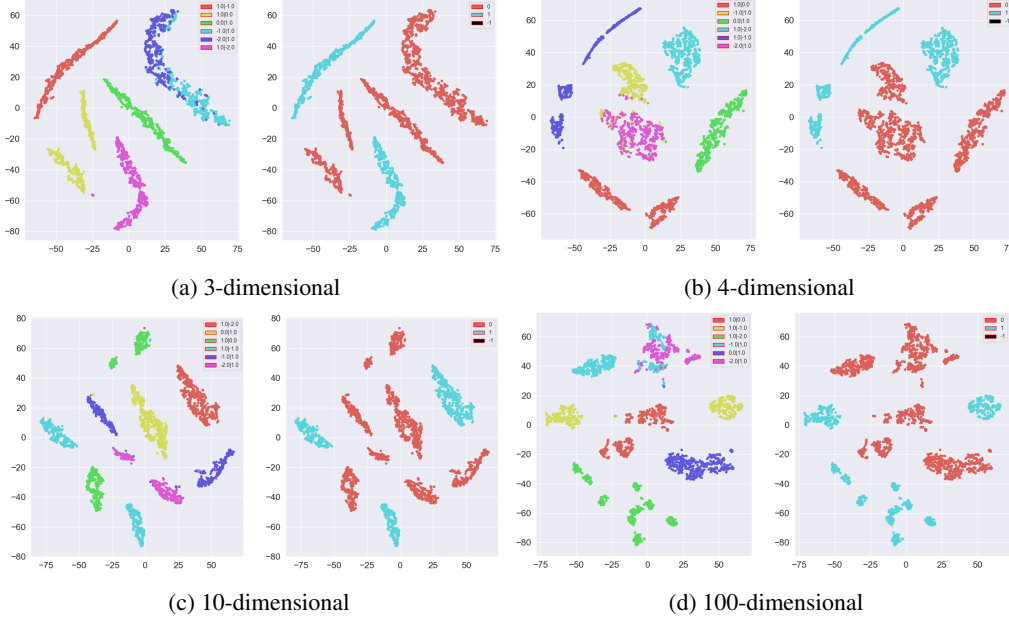


Figure 9: Representation of the clusters learned by *GameDistill* for Coin Game. Each point is a t-SNE projection of the feature vector (in different dimensions) output by the *GameDistill* network for an input sequence of states. For each of the sub-figures, the figure on the left is colored based on 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.

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 2×2 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.3 *GameDistill*: Pseudo-Code

Algorithm 3 Pseduo-code for *create_network*

```

1: net = conv(states_placeholder, kernel = 3, num_outputs = 64, activation = relu)
2: net = conv(net, kernel = 3, num_outputs = 64, activation = relu)
3: net = conv(net, kernel = 3, num_outputs = 64, activation = relu)
4: feat = flatten(net) // the trajectory embedding
5:
6: self_ft, opp_ft = FC(feat, num_outputs = 100), FC(feat, num_outputs = 100)
7:
8: // Predict the opponent and the self rewards
9: s_reward_pred = FC(self_ft, num_outputs = 1)
10: o_reward_pred = FC(opp_ft, num_outputs = 1)
11: return s_reward_pred, o_reward_pred

```

Algorithm 2 Pseduo-code for *collect_data*

```
1: Input: Game Environment Env, Minimum Samples min_samples = 2000, Batch Size  
   batch = 100, look_back = 5  
2: env = Env.spawn(batch)  
3: state_q = Queue(batch, look_back)  
4: reward_seq_dict = dict()  
5: keep_running = True  
6: while keep_running do  
7:   actions = random(env.actions, size = (batch, env.n_agents))  
8:   rewards, moves, states = env.step(actions)  
9:   check = False  
10:  for b in range(batch) do  
11:    if state_q[b].full() then  
12:      state_q[b].pop()  
13:    end if  
14:    state_q[b].put(states[b])  
15:  
16:    // for any non zero reward tuple  
17:    if abs(rewards).sum() > 0 then  
18:      reward_tpl = tuple(rewards)  
19:      if reward_tpl not in reward_seq_dict then  
20:        reward_seq_dict[reward_tpl] = []  
21:      end if  
22:      obs = state_q[b].pop_all()  
23:      reward_seq_dict[reward_tpl].append(obs)  
24:      check = True  
25:    end if  
26:  end for  
27:  if check then  
28:    keys = env.get_all_possible_reward_tuples()  
29:    count_stop = 0  
30:    for k in keys do  
31:      if len(reward_seq_dict[k]) > min_samples then  
32:        count_stop += 1  
33:      end if  
34:    end for  
35:    if count_stop >= len(keys) then  
36:      keep_running = False  
37:    end if  
38:  end if  
39: end while  
40:  
41: train_data = [] // Collect the final training data  
42: for rewards_tup in reward_seq_dict.keys() do  
43:   for traj in t_data[rewards_tup][: min_samples] do  
44:     // trajectory has shape [look_back, h, w, c] and "rewards_tup" is tuple of rewards of agents  
45:     traj = traj.permute(1, 2, 3, 0).reshape(h, w, -1)  
46:     train_data.append((traj, rewards_tup))  
47:   end for  
48: end for  
49: return shuffle(train_data)
```

Algorithm 4 Pseduo-code for *train_network*

```
1: Input: Reward Prediction Network as net, data train_data, loss term weights  $\mathcal{A}$  and  $\mathcal{B}$ 
2: optimizer = Adam(lr = 0.003)
3: while convergence do
4:   state, reward = sample(train_data)
5:   my_reward, opp_reward = net.forward(state)
6:   loss =  $\mathcal{A} * l2\_loss(my\_reward, reward(0)) + \mathcal{B} * l2\_loss(opp\_reward, reward(1))$ 
7:   loss.backward(), optimizer.step()
8: end while
```

Algorithm 5 Pseduo-code for *create_oracle_net*

```
1: Input: Game Environment env
2: net = conv(state_placeholder, kernel = 2, num_outputs = 128, activation = relu)
3: net = conv(net, kernel = 2, num_outputs = 128, activation = relu)
4: net = conv(net, kernel = 2, num_outputs = 64, activation = relu)
5: net = flatten(net)
6: net = fc(net, num_outputs = 128, activation = relu) // Encode the environment state
7:
8: // Predict the probability of taking an action
9: logits = fc(net, num_outputs = env.num_actions)
10: action_predict = softmax(logits)
11: return Action predictions action_predict
```

Algorithm 6 Pseduo-code for *train_oracle*

```
1: Input: Oracle Network net, Clustered trajectory data train_data
2: actions_data = []
3: for data in train_data do
4:   states = data[0]
5:   for i in range(1, len(states)) do
6:     action = deduce_move(states[i - 1], states[i])
7:     actions_data.append((states[i - 1], action))
8:   end for
9: end for
10: optimizer = SGD(lr = 0.01)
11: while not convergence do
12:   state, action = sample(actions_data)
13:   my_action = net.forward(state)
14:   loss = cross_entropy(my_action, action)
15:   loss.backward(), optimizer.step()
16: end while
```

C *SQLoss*: Emergence of Cooperation

Equation 6 (Section 2.3.2) describes the gradient for standard policy gradient. It has two terms. The $\log \pi^1(u_t^1 | s_t)$ 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 $\log \pi^1(u_{t-1}^1|s_t)$ 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, $s_t = (u_{t-1}^1, u_{t-1}^2)$. 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 $u_{t-1}^1 = C$. $\pi^1(D|CC)$ 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 $SQLoss$, $\pi(C|CC) > \pi(D|CC)$.*

For CD , DC , both the standard loss and $SQLoss$ push the policy towards D . For DD , with sufficiently high κ , 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 κ . 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 κ (as shown in our experiments), the two loss terms work together so that $\pi(C|CC) > \pi(D|CC)$.

D Games with more than 2 players

Our formulation of $SQLoss$ has the distinct advantage of being fully ego-centric, that is, the agent that is learning does not require any information regarding its opponents. This feature enables a straightforward extension of $SQLoss$ beyond the two agent setting, without any change in each agent’s learning algorithm. In order to test this extension of $SQLoss$ beyond 2-players, we consider as an example, the problem described in the popular Braess’ paradox, which is a well-known extension of the Prisoner’s Dilemma problem to more than 2 agents. We construct a simplified environment to

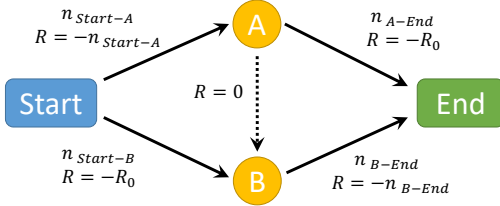
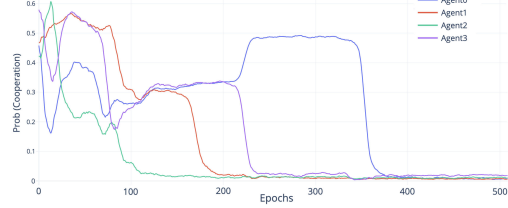
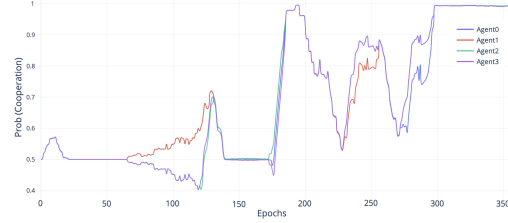


Figure 10: Braess’ Paradox



(a) Results for 4 NL agents in Braess’ Environment



(b) Results for 4 SQLearner agents in Braess’ Environment

Figure 11

interpret the policies in the problem given in Braess’ paradox as a sequential social dilemma problem. This problem, which we henceforth refer to as the Braess’ problem, is concerned with traffic flow from a source to a destination with two alternative paths, through two intermediate locations. This is illustrated in Fig. 10.

In this, each agent has to travel from the source (Start) to the destination (End) by choosing one of these paths. Each path has two segments separated by the respective intermediate location and the cost of traversing each segment (proportional to the time of travel) is either fixed or is determined based on the number of agents using that segment. This cost function is shown in Eq. 10. In the original setup, the two intermediate locations are not connected and each agent has to only choose between one of these paths. The equilibrium solution in this case is for half the agents to choose one path and the remaining half the other. The modification that entails is the paradox is connecting the two intermediate locations with a cost free directed bridge, thus leading a third possible path for all agents. This modification shifts the equilibrium in such a way that all agents now prefer to choose the new path that uses this directed bridge, although this results in a per-agent cost that is higher than each agent’s original cost in the absence of this new path. This problem has been well studied and the paradox highlights the fact that in strategic settings, increase in the number of available options for all agents, may also lead to decrease in utility for all agents individually and collectively.

We model this problem as a sequential social dilemma, where we define the initial strategy of the agents to choose one of the two original paths as *Cooperation* and the strategy that results in an agent choosing the new path with the directed bridge as *Defection*. For simplicity of exposition, we assign integer IDs to agents and define Cooperation for agents with odd-IDs as choosing the path Start-A-End in Fig. 10 and Cooperation for agents with even-IDs as choosing the path Start-B-End in Fig. 10. In this notation, the Defection can be defined as an agent choosing the path Start-A-B-End. In the Fig. 10 we provide reward numbers, such that the paradox is realized.

As described earlier, in this game, Cooperate and Defect have the following interpretation:

1. For an odd numbered vehicle – Cooperation implies taking the Start-A-End route.

2. For an even numbered vehicle – Cooperation implies taking the Start-B-End route (This and above point follow from the assumption that before addition of the new edge A-B, the vehicles were in this equilibrium).
3. For any vehicle – Defection implies taking the Start-A-B-End route.

Let n_{X-Y} denote the number of agents using the edge $X - Y$. Then the reward structure for the odd-numbered vehicles (denoted by R_{odd}) and the even-numbered vehicles (denoted by R_{even}) is computed as shown below.

$$\begin{aligned}
N_0 &= \text{Total no. of agents} \\
R_0 &= \text{base reward for the agents} = (2.5 * N_0)/2 \\
R_{odd} &= \begin{cases} -(n_{Start-A} + n_{B-End}) & \text{if Defection} \\ -(n_{Start-A} + R_0), & \text{if Cooperation} \end{cases} \\
R_{even} &= \begin{cases} -(n_{Start-A} + n_{B-End}) & \text{if Defection} \\ -(R_0 + n_{B-End}), & \text{if Cooperation} \end{cases}
\end{aligned} \tag{10}$$

We performed two separate set of experiments in this game with 4 and 6 agents. For each setup, we simulated the result when all agents are selfish learners (the *SL* agent(s)) and also when all agents use *SQLoss* (the *SQLearner*(s)). For both these setups, as expected, we observe that when using selfish learners, all agents converge to Defection and when using *SQLoss*, all agents converge to Cooperation. We present the results for the Braess' Paradox experiment in Figure 11.

The Figure 11a shows the results for 4 SL-agents playing in the environment. From the figure it is clear that if 4 SL agents play in the environment, then they eventually end up defecting i.e. $\lim_{E \rightarrow \infty} \mathbb{P}(Cooperation) \rightarrow 0$, where E denotes the epoch. Figure 11b illustrates the behaviour of the *SQLearner* agents. The *SQLearner* agents eventually learn to cooperate and thus the $\lim_{E \rightarrow \infty} \mathbb{P}(Cooperation) \rightarrow 1$. To the best of our knowledge, this is the first demonstration of selfish agents learning to cooperate in a sequential social dilemma with more than 2 agents.

E Experimental Details

E.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.

E.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 and 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 (γ) 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 γ allows for long time horizons, thereby incentivizing long-term rewards. Each agent randomly samples κ from $\mathbb{U} \in (1, z)$ ($z = 10$, discussed in Appendix K) at each step.

F Visualizing clusters obtained from *GameDistill*

Figures 9 and 12 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 of states. For each of the sub-figures, the figure

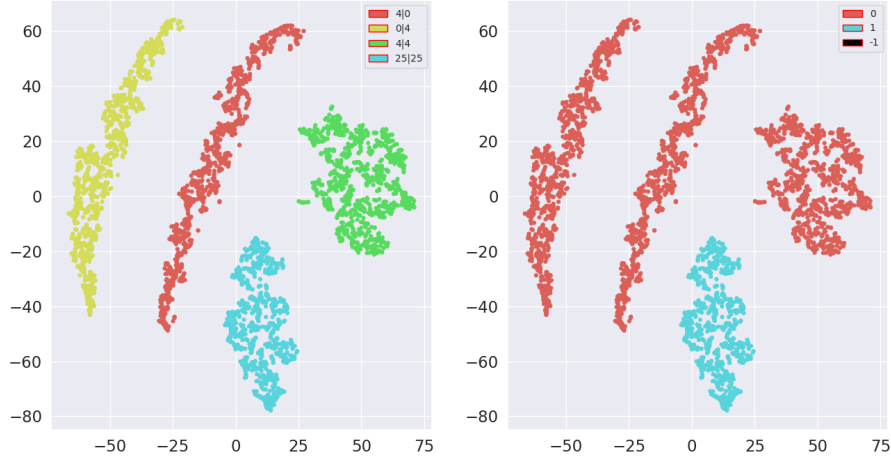


Figure 12: t-SNE plot for the trajectory embeddings obtained for the Stag Hunt game using *GameDistill* along with the identified cooperation and defection clusters. Details on how to read the figures is provide with Figure 9

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 9 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.

G Results for Visual Stag-Hunt game with *GameDistill* and *SQLoss*

Figure 14 shows the results for the *SQLearner* agents using the trained oracles obtained from *GameDistill* along with *SQLoss* in the visual StagHunt environment.

H Results for the Iterated Chicken Game (ICG) using *SQLoss*

Figure 2 presents the payoff matrix for the Chicken Game. In Figure 13 we compare the results of a *SQLearner* agent on the ICG Game with a LOLA agent. The payoff matrix for the game is shown in the Table 2. From the payoff, it is clear that the agents may defect out of greed. In this game also,

	<i>C</i>	<i>D</i>
<i>C</i>	(-1, -1)	(-3, 0)
<i>D</i>	(0, -3)	(-4, -4)

Table 2: Chicken Game

SQLearner agents coordinate successfully to obtain a near-optimal NDR value (0) for this game.

I *SQLoss* vs *GameDistill*

In this section we present the results for the additional experiments we do to disentangle the effect of *GameDistill* and *SQLoss* for the performance in Figure 5 by training a variant of LOLA that uses the oracles learned by *GameDistill*.

To discuss the observation regarding how much of the performance shown in Figure 5 is attributable to *GameDistill*, we further discuss Figures 3 and 5. Figure 3 shows the different methods applied

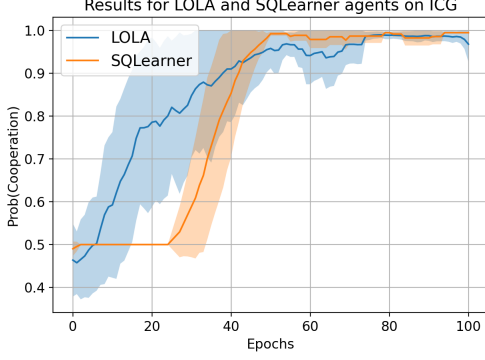


Figure 13: $P(\text{Cooperation})$ for LOLA and *SQLea*ner agents in ICG. Both agents eventually obtain a near optimal probability of 1.0

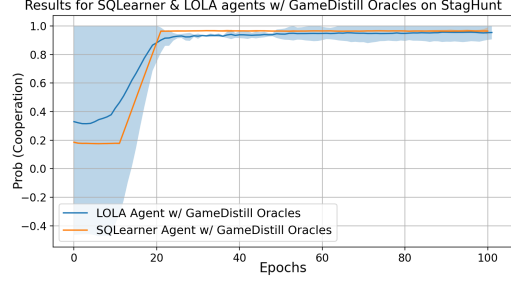


Figure 14: $P(\text{Cooperation})$ for *SQLea*ner agents in visual StagHunt with *GameDistill* oracles. Both agents eventually learn to capture Stag roughly 95% of the time which is close to the optimal for the game.

to the matrix formulation of the Iterated Prisoner’s Dilemma (IPD) game. In this formulation, each method (*SQLea*ner, LOLA, etc.) uses the same reduced action space consisting of cooperation and defection. Figure 3 shows how agents trained using *SQLoss* outperform those trained using LOLA in this setting. For our experiments on the Coin Game (Figure 5), we use (*GameDistill* + *SQLoss*) against LOLA.

From the results above, it is clear that the performance of the *SQLea*ner agent is due to the *SQLoss* which promotes cooperation (as shown in multiple matrix games), the *GameDistill* algorithm further expands this gain by reducing the dimensionality of the games with visual input. The other baselines, e.g. Lola-PG and *SL* do not use *GameDistill*. To further test this hypothesis, we use the oracles developed using *GameDistill* with the *SL* agents, and observe that the *SL* agents with *GameDistill* do converge faster to *DD* than traditional *SL* agents, motivating us to use *GameDistill*.

J Illustrations of Trained Oracle Networks

J.1 Coin Game

Figure 15 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. Empirically, we train the Cooperation and Defection oracles and obtain a probability of picking self-colored coin (or $P(\text{Cooperation})$) close to 0.916 and 0.006 respectively.

J.2 StagHunt

GameDistill produces two oracles when trained on the StagHunt environment. One of the trained oracles results in both the agents capturing the Stag 99% of the times i.e. $\text{Probability}(\text{Capturing a Stag} | \text{Stag, Hare}) = 0.99$, while the other oracle forces the agents to eat the Hare with similar probability of 0.99. Both the oracles learn to accurately capture the desired item in the environment.

K *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 κ_t from the Discrete Uniform distribution $\mathbb{U}\{1, z\}$. A larger value of z thus implies a larger value of κ_t and longer imaginary episodes. We find that larger z (and hence κ) leads to faster cooperation between agents in the IPD and Coin Game. This effect plateaus for

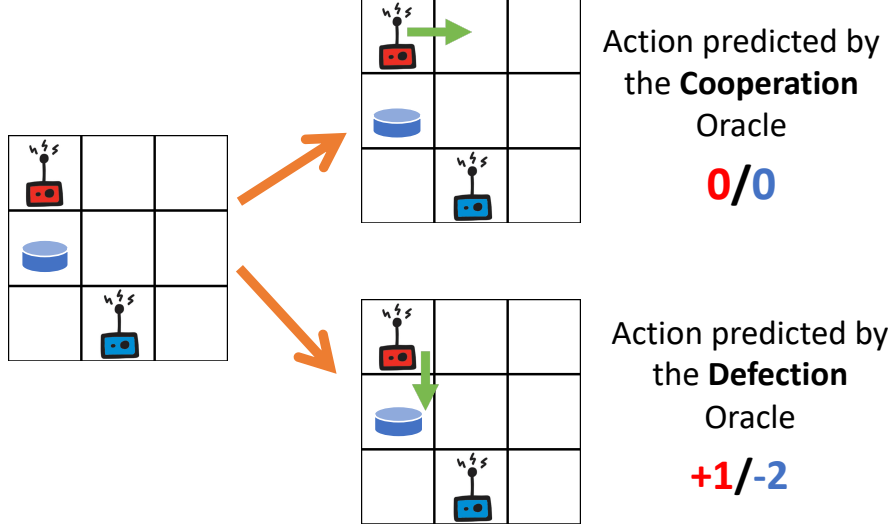


Figure 15: Illustrative predictions of the oracle networks learned by the Red agent using *GameDistill* in the Coin Game. The numbers in **red/blue** show the rewards obtained by the Red and the Blue agent respectively. The cooperation oracle suggests an action that avoids picking the coin of the other agent while the defection oracle suggests an action that picks the coin of the other agent

$z > 10$. However varying and changing κ_t across time also increases the variance in the gradients and thus affects the learning. We thus use $\kappa = 10$ for all our experiments.

L *SQ*Learner: Exploitability and Adaptability

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