# Selectively Sharing Experiences Improves Multi-Agent Reinforcement Learning

Matthias Gerstgrasser[1], Tom Danino[2], and Sarah Keren[2]

[1]John A. Paulson School of Engineering and Applied Sciences, Harvard University
[2]The Taub Faculty of Computer Science, Technion - Israel Institute of Technology

[1]matthias@g.harvard.edu

October 30, 2022

**Abstract**

We present a novel multi-agent RL approach, *Selective Multi-Agent PER*, in which agents share with other agents a limited number of transitions they observe during training. They follow a similar heuristic as is used in (single-agent) Prioritized Experience Replay, and choose those transitions based on their td-error. The intuition behind this is that even a small number of relevant experiences from other agents could help each agent learn. Unlike many other multi-agent RL algorithms, this approach allows for largely decentralized training, requiring only a limited communication channel between agents. We show that our approach outperforms baseline no-sharing decentralized training and state-of-the art multi-agent RL algorithms. Further, sharing only a small number of experiences outperforms sharing all experiences between agents, and the performance uplift from selective experience sharing is robust across a range of hyperparameters and DQN variants.

## 1 Introduction

Multi-Agent Reinforcement Learning (RL) is often considered a hard problem: The environment dynamics and returns depend on the joint actions of all agents, leading to significant variance and non-stationarity in the experiences of each individual agent. Much recent work [24, 18] in multi-agent RL has focused on mitigating the impact of these. Our work goes in a different direction: leveraging the presence of other agents to collaboratively explore the environment more quickly.

We present a novel multi-agent RL approach that allows agents to share a small number of experiences with other agents. The intuition is that if one agent discovers something important in the environment, then sharing this with the other agents should help them learn faster. However, it is crucial that only important experiences are shared - we will see that sharing all experiences indiscriminately will not improve learning. To this end, we take inspiration from a well-established technique in single-agent RL, *prioritized experience replay* (PER) [25]. With PER an off-policy algorithm such as DQN [19] will sample experiences not uniformly, but proportionally to "how far off" the current policy's predictions are in each state, formally the *temporal difference (td) error* . We use this same metric to prioritize which experiences to share with other agents.

We dub the resulting multiagent RL approach “selective multiagent PER” or “suPER”. In this, agents independently use a DQN algorithm to learn, but with a twist: Each agent relays its highest-td-error experiences to the other agents, who insert them directly into their replay buffer, which they use for learning. This approach has several advantages:

1. It consistently leads to faster learning and higher eventual performance, across hyperparameter settings.
2. Unlike many “centralized training, decentralized execution” approaches, the suPER learning paradigm allows for (semi-) decentralized training, requiring only a limited bandwidth communication channel between agents.
3. The paradigm is agnostic to the underlying decentralized training algorithm, and can enhance many existing DQN variations.

In addition to the specific algorithm we develop, this work also introduces two key conceptual novelties.

4. We show that communication can improve multi-agent RL even during training. Most prior work consider ”learning to communicate”, also known as emergent communication, which is a difficult problem but may improve coordination at convergence. We show that ”communicate to learn” can also drastically improve performance during training.
5. Related to this, we introduce the paradigm of “decentralized training with communication”. This is a ‘middle ground between established approaches of decentralized and centralized training (including “centralized training, decentralized execution”).

In the remainder of the paper, we will discuss related literature and technical preliminaries; introduce our novel algorithm in detail; describe experimental evaluation and results; and suggest avenues for future work.

## 2 Related Work

**Multi-Agent RL** approaches in the literature can be broadly categorized according to the degree of awareness of each learning agent of the other agents in the system [32]. On one end of this spectrum is independent or decentralized learning where multiple learning agents (in the same environment) are unaware or agnostic to the presence of other agents in the system [31, 15]. From the perspective of each agent, learning regards the other agents as part of the non-stationary environment. At the opposing end of the spectrum, in centralized control a single policy controls all agents. In between these two, a number of related strands of research have emerged.

Approaches nearer the decentralized end of the spectrum often use communication between the agents [39, 6, 10]. Several forms of cooperative communication have been formulated, by which agents can communicate various types of messages, either to all agents or to specific agent groups through dedicated channels [14, 28, 21, 22]. While communication among agents could help with coordination, training emergent communication protocols also remains a challenging problem; recent empirical results underscore the difficulty of learning meaningful emergent communication protocols, even when relying on centralized training [13]. Related to this, and often used in conjunction, is modeling other agents [2, 13, 7, 37], which equips agent with some model of other agent’s behavior.

This in turn is related to a number of recent approaches using centralized training but decentralized execution. A prevailing paradigm within this line of work assumes a training stage during which a shared network (such as a critic) can be accessed by all agents to learn decentralised (locally-executable) agent policies [18, 1, 23, 7, 24]. These approaches successfully reduce variance during training, e.g. through a shared critic accounting for other agents' behavior, but rely on joint observations and actions of all agents. Within this paradigm, and perhaps closest related to our own work, [5] introduces an approach in which agents share (all of their) experiences with other agents. While this is based on an on-policy actor-critic algorithm, it uses importance sampling to incorporate the off-policy data from other agents, and the authors show that this can lead to improved performance in sparse-reward settings. Our work also relies on sharing experiences among agents, but crucially relies on selectively sharing only some experiences. [1]

**Off-Policy RL** The approach we present in this paper relies intrinsically on off-policy RL algorithms. Most notable in this class is DQN [19], which achieved human-level performance on a wide variety of Atari 2600 games. Various improvements have been made to this algorithm since then, including dueling DQN [35], Double DQN (DDQN) [34], Rainbow [11] and Ape-X [12]. Prioritized Experience Replay [25] improves the performance of many of these, and is closely related to our own work. A variant for continuous action spaces is DDPG [27]. Off-policy actor-critic algorithms [9, 17] have also been developed, in part to extend the paradigm to continuous control domains.

## 3 Preliminaries

**Reinforcement learning (RL)** deals with learning optimal policies for sequential decision making in environments for which the dynamics are not fully known [29]. *Multi-Agent Reinforcement Learning* extends RL to multi-agent settings. A common model is a *Markov game*, or *stochastic game* defined as a tuple $\langle S, \mathcal{A}, \mathcal{R}, \mathcal{T}, \gamma \rangle$ with states $S$, *joint actions* $\mathcal{A} = \{A^i\}_{i=1}^n$ as a collection of action sets $A^i$, one for each of the $n$ agents, $\mathcal{R} = \{R^i\}_{i=1}^n$ as a collection of reward functions $R^i$ defining the reward $r^i(a_t, s_t)$ that each agent receives when the joint action $a_t \in \mathcal{A}$ is performed at state $s_t$, and $\mathcal{T}$ as the probability distribution over next states when a joint action is performed. In the partially observable case, the definition also includes a joint observation function, defining the observation for each agent at each state. In this framework, at each time step an agent has an *experience* $e =< S_t, A_t, R_{t+1} S_{t+1} >$, where the agent performs action $A_t$ at state $S_t$ after which reward $R_{t+1}$ is received and the next state is $S_{t+1}$. We focus here on decentralized execution settings in which each agent follows its own individual policy $\pi_i$ and seeks to maximize its discounted accumulated return. In the special case where rewards are shared between all agents a Markov Game is also called a *decentralized POMDP* or dec-POMDP. We do not require this assumption, but we do require that the Markov game is symmetric or anonymous, meaning all agents have the same action and observation spaces and the environment reacts identically to their actions.

Among the variety of RL solution approaches [30, 29], we focus here on *value-based methods* that use state and action estimates to find optimal policies. Such methods typically use a value function $V_\pi(s)$ to represent the expected value of following policy $\pi$ from state $s$ onward, and a

[1] Interestingly, in the appendix to the arXiv version of that paper, the authors state that experience sharing in DQN did not improve performance in their experiments, in stark contrast with the result we will present in this paper. We believe this may be because their attempts shared experiences indiscriminately, compared to our selective sharing approach. We will see that sharing all experiences does not improve performance as much or as consistently as selective sharing.

*Q-function* $Q_\pi(s,a)$, to represent for a given policy $\pi$ the expected rewards for performing action $a$ in a given state $s$ and following $\pi$ thereafter.

Q-learning [36] is a *temporal difference* (td) method that considers the difference in $Q$ values between two consecutive time steps. Formally, the update rule is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t))]$$

Where, $\alpha$ is the *learning rate* or *step size* setting the amount by which values are updated during training. The learned action-value function, Q, directly approximates $q^*$, the optimal action-value function. During training, in order to guarantee convergence to an optimal policy, Q-learning typically uses an $\epsilon$-greedy selection approach, according to which the best action is chosen with probability $1-\epsilon$, and a random action, otherwise.

Given an experience $e_t$, the *td-error* represents the difference between the Q-value estimate and actual reward gained in the transition and the discounted value estimate of the next best actions. Formally,

$$\text{td}(e_t) = |R + \gamma \max_{a'} Q(S', a') - Q(S, A)| \tag{1}$$

In the vanilla version of Q-learning Q-values are stored in a table, which is impractical in many real-world problem due to large state space size. In deep Q-Learning (DQN) [20], the Q-value table is replaced by a function approximator typically modeled using a neural network such that $Q(s,a,\theta) \approx Q^*(s,a)$, where $\theta$ denotes the neural network parameters.

**Replay Buffer (RB) and Prioritized RB:** In their simplest form, RL algorithms use and discard incoming data after updating the policy. To increase efficiency an experience replay buffer (Lin, 1992) was suggested. This is a memory structure that enables online reinforcement learning algorithms to store and replay past experiences by sampling them from memory. This allows mixing experiences from different time steps in to a single policy update, allowing rare experiences to be used more than once.

Evidence shows the replay buffer to stabilize training of the value function for DQN [19, 20] and to reduce the amount of experiences required for an RL-agent to complete the learning process and achieve convergence [25].

Initial approaches that used a replay buffer, uniformly sampled experiences from the buffer. However, some transitions are more effective for the learning process of an RL agents then others. Transitions may be more or less surprising, redundant, or task-relevant. Some transitions may not be immediately useful to the agent, but might become so when the agent competence increases [26]. *Prioritized Experience Replay (PER)* [25] explores the idea that replaying and learning from some transitions, rather than others, can enhance the learning process. PER suggests replacing the standard sampling method, where transitions are replayed according to the frequency they were collected from the environment, with a td-error based method, where transitions are sampled according to the value of their td-error.

As a further extension, **stochastic prioritization** balances between strictly greedy prioritization and uniform random sampling. Hence, the probability of sampling transition $i$ is defined as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{2}$$

where $p_i > 0$ is the priority associated to transition $i$ and $\alpha$ determines the weight of the priority ($\alpha = 0$ is uniform sampling). The value of $p_i$ is determined according to the magnitude of the td-error, such that $p_i = |\delta_i| + \epsilon$, $\delta_i$ is the td-error and $\epsilon$ is a small positive constant that guarantees that transitions for which the td-error is zero have a non-zero probability of being sampled from the buffer.

## 4 suPER: Selective Multi-Agent Prioritized Experience Replay

Our approach is rooted in the same intuition as PER: that not all experiences are equally relevant. We use this insight to help agents learn by sharing between them only a (small) number of their most relevant experiences. Our approach builds on standard DQN algorithms, and adds this experience sharing mechanism between collecting experiences and performing gradient updates, in each iteration of the algorithm:

1. (DQN) Collect a rollout of experiences, and insert each agent's experiences into their own replay buffer.
2. (suPER) Each agent shares their most relevant experiences, which are inserted into all the other agents' replay buffers.
3. (DQN) Each agent samples a minibatch of experiences from their own replay buffer, and performs gradient descent on it.

Steps 1 and 3 are standard DQN; we merely add an additional step between collecting experiences and learning on them. As a corollary, this same approach works for any standard variants of steps 1 and 3, such as dueling DQN [35], DDQN [34], Rainbow [11] and other DQN improvements. Beyond DQN and its variants, suPER could in principle apply to any off-policy RL algorithm. Algorithm 1 gives a more detailed listing of this algorithm. Notice that the only interaction between the agents training algorithms is in the experience sharing step. This means that the algorithm can easily be implemented in a decentralized manner with a (limited) communications channel.

### 4.1 Experience Selection

We describe here three variants of the suPER algorithm, that differ in how they select experiences to be shared.

**Deterministic quantile-based experience selection** The learning algorithm keeps a list $l$ of the (absolute) td-errors of its last $k$ experiences ($k = 1500$ by default). For a configured bandwidth $\beta$ and a new experience $e_t$, the agent shares the experience if its absolute td-error $|\text{td}(e_t)|$ is at least as large as the $k * \beta$-largest absolute td-error in $l$. In other words, the agent aims to share the top $\beta$-quantile of its experiences, where the quantile is calculated over a sliding window of recent experiences.

$$|\text{td}(e_t)| \geq \text{quantile}_\beta(\{e_{t'}\}_{t'=t-k}^{t})$$

**Algorithm 1** suPER algorithm for DQN

```
for each training iteration do
    Collect a batch of experiences $b$                          ▷ DQN
    for each agent $i$ do                                       ▷ DQN
        Insert $b_i$ into buffer$_i$                            ▷ DQN
    end for                                                     ▷ DQN
    for each agent $i$ do                                       ▷ suPER
        Select $b_i^* \subseteq b_i$ of experiences to share[1] ▷ suPER
        for each agent $j \neq i$ do                            ▷ suPER
            Insert $b*_i$ into buffer$_j$                       ▷ suPER
        end for                                                 ▷ suPER
    end for                                                     ▷ suPER
    for each agent $i$ do                                       ▷ DQN
        Sample a train batch $b_i$ from buffer$_i$              ▷ DQN
        Learn on train batch $b_i$                              ▷ DQN
    end for                                                     ▷ DQN
end for
```

[1] See section "Experience Selection"

**Deterministic Gaussian experience selection** In this, the learning algorithm calculates the mean $\mu$ and variance $\sigma^2$ of the (absolute) td-errors of the $k$ most recent experiences ($k = 1500$ by default). It then shares an experience $e_t$ if

$$|\mathrm{td}(e_t)| \geq \mu + c \cdot \sigma^2 \tag{3}$$

where $c$ is a constant chosen such that $1 - \mathrm{cdf}_{\mathcal{N}}(c) = \beta$. In other words, we use the $c$-quantile of a normal distribution with the (sample) mean and variance of most recent experiences. We include and benchmark this variant for two reasons. One, intuitively, we might want to be more sensitive to clusters of outliers, where using a quantile of the actual data might include only part of the cluster, while a Gaussian model might lead to the entire cluster being included. Two, mean and variance could be computed iteratively without keeping a buffer of recent td-errors, and thereby reducing memory requirements. We aim to benchmark if this approximation impacts performance.

**Stochastic weighted experience selection** Finally, and most closely related to classical single-agent PER, this variant shares each experience with a probability that's proportional to its absolute td-error. In PER, given a train batch size $b$, we sample $b$ transitions from the replay buffer without replacement, weighted by each transition's td-error. In suPER, we similarly aim to sample a $\beta$ fraction of experiences, weighted by their td-errors. However, in order to be able to sample transitions online, we calculate for each experience individually a probability that approximates sampling-without-replacement in expectation. Formally, taking $p_i = |\mathrm{td}(e_i)|$, we broadcast experience $e_t$ with probability

$$p = \min\left(1, \beta \cdot \frac{p_i^\alpha}{\sum_k p_k^\alpha}\right)$$

similarly to equation 2, and taking the sum over a sliding window over recent experiences. It is easy to see that if $\beta = 1/\text{batchsize}$, this is equivalent to sampling a single experience, weighted by td-error, from the sliding window. For larger bandwidth $\beta$, this approximates sampling multiple experiences without replacement: If none of the $\beta \cdot \frac{p_i^\alpha}{\sum_k p_k^\alpha}$ terms are greater than 1, this is exact. If we have to truncate any of these terms, we slightly undershoot the desired bandwidth.

In our current experiments, we share experiences and update all quantiles, means and variances once per sample batch, for convenience and performance reasons; However, we could do both online, i.e. after every sampled transition, in real-world distributed deployments. We do not expect this to make a significant difference.

## 5 Experiments and Results

We evaluate the suPER approach on a number of multiagent benchmark domains.

### 5.1 Algorithm and control benchmarks

**Baseline: DQN** As a baseline, we use a fully independent DQN algorithm. The algorithm samples a sequence of experiences using joint actions from all the agents' policies, and then inserts each agent's observation-action-reward-observation transitions into that agent's replay buffer. Each agent's policy is periodically trained using a sample from its own replay buffer, sampled using PER. We refer to this baseline as simply "DQN" in the remainder of this section and in figures.

**suPER-DQN** We implement suPER on top of the DQN baseline. Whenever a batch of experiences is sampled, each agent shares only its most relevant experiences (according to one of the criteria described in the previous section) which are then inserted into all other agents' replay buffers. As above, we run the suPER experience sharing on whole sample batches. All DQN hyperparameters are unchanged - the only difference from the baseline is the addition of experience sharing between experience collection and learning. This allows for controlled experiments with like-for-like comparisons.

**Indiscriminate sharing-all-experiences suPER-DQN** We also compare against a DQN variant that shares *all* experiences among agents. In this variant, whenever a batch of experiences is sampled, every agent's trajectory is inserted into all the agents' replay buffers. This can be seen as both a suPER variant (when testing if experience sharing is helpful in general) or as a baseline (when testing if selectivity in experience sharing is helpful).

**Parameter-Sharing DQN** In parameter-sharing, all agents share the same policy parameters. Note that this is different from joint control, in which a single policy controls all agents simultaneously; In parameter sharing, each agent is controlled independently by a copy of the policy. It is also different from share-all suPER-DQN, in which a different policy is trained for each agent.

**DQN and Dueling DDQN** For all of the above, we consider both variants based on standard DQN, as well as variants based on dueling double DQN (DDQN). In the remainder of the text, whenever we write DDQN we mean dueling double DQN, and similarly for suPER-DDQN, parameter-sharing DDQN, etc. Whenever we write DQN we mean standard non-dueling, non-double DQN.

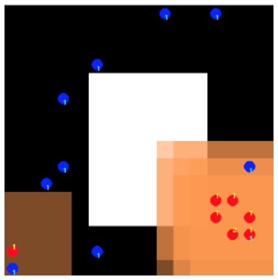

Figure 1: Pursuit Environment

**Multi-Agent Baselines** We further compare against standard multi-agent RL algorithms, specifically MADDPG [18], QMIX [24] and SEAC [5]. Like parameter-sharing, these are all considered "centralized training, decentralized execution" approaches.

## 5.2 Environments

**SISL: Pursuit** is a semi-cooperative environment, where a group of pursuers has to capture a group of evaders in a grid-world with an obstacle (see Fig. 1), The evaders (blue) move randomly, while the pursuers (red) are controlled by RL agents. If a group of two or more agents fully surround an evader, they each receive a reward, and the evader is removed from the environment. The episode ends when all evaders have been captured, or after 500 steps, whichever is earlier. Pursuers also receive a (very small) reward for being adjacent to an evader (even if the evader is not fully surrounded), and a (small) negative reward each timestep, to incentivize them to complete episodes early. We use 8 pursuers and 30 evaders.

**MAgent: Battle** is a semi-adversarial environment, where two groups of opposing teams are battling against each other. An agent is rewarded 0.2 points for attacking agents in the opposite team, and 5 points if the other agent is killed. All agents start with 10 health points (HP) and lose 2 HP in each attack received, while regaining 0.1 HP in every turn. Once killed, an agent is removed from the environment. An episode ends when all agents from one team are killed. The action space, of size 21 is identical for all agents, with (8) options to attack, (12) to move and one option to do nothing. Since no additional reward is given for collaborating with other agents in the same team, it is considered to be more challenging to form collaboration between agents in this environment. We use a map of size $18 \times 18$ and 6 agents per team.

**MAgent: Adversarial Pursuit** is a predator-prey environment, with two types of agents, prey and predator. The predators navigate through obstacles in the map with the purpose of tagging the prey. An agent in the predators team is rewarded 1 point for tagging a prey, while a prey is rewarded $-1$ when being tagged by a predator. Unlike in the Battle environment, prey agents are not removed from the game when being tagged. Note that prey agents are provided only with a negative or zero reward (when manage to avoid attacks), and their aim is thus to evade predator agents. We use 8 prey agents, and 4 predator agents.

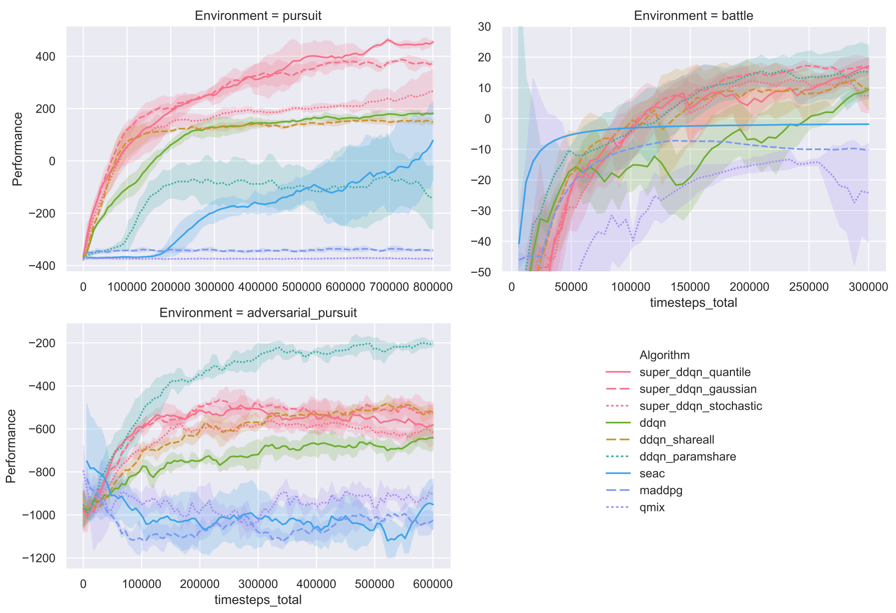

Figure 2: Performance of suPER-dueling-DDQN variants with target bandwidth 0.1 on all three domains. For Pursuit, performance is the total mean episode reward from all agents. For Battle and Adversarial-Pursuit, performance is the total mean episode reward from all agents in the sharing team (blue team in Battle, prey team in Adversarial-Pursuit). Shaded areas indicate one standard deviation.

## 5.3 Experimental Setup

In Pursuit, we train all agents concurrently using the same algorithm, for each of the algorithms listed. Note that only the pursuers are agents in this domain, whereas the evaders move randomly. In the standard variants of Battle and Adversarial-Pursuit, we first pre-train a set of agents using independent dueling DDQN, all agents on both teams being trained together and independently. We then take the pre-trained agents of the opposing team (red team in Battle, predator team in Adversarial-Pursuit), and use these to control the opposing team during main training of the blue respectively prey team. In this main training phase, only the blue / prey team agents are trained using each of the suPER and benchmark algorithms, whereas the red / predator team are controlled by the pre-trained policies with no further training. Figures 2 and 3 show learning curves from the main training phase.

In Battle and Adversarial-Pursuit we further show a variant where the opposing team are co-evolving with the blue / prey team. In this variant, all agents start from a randomly initialized policy and train concurrently, using a DDQN algorithm. However, only the blue / prey team share experiences using the suPER mechanism. We only do this for the DDQN baseline as well as discriminate and share-all suPER variants. This is in part because some of the other baseline algorithms do not support concurrently training opposing agents with a different algorithm in available implementations; and in part because we consider this variant more relevant to real-world

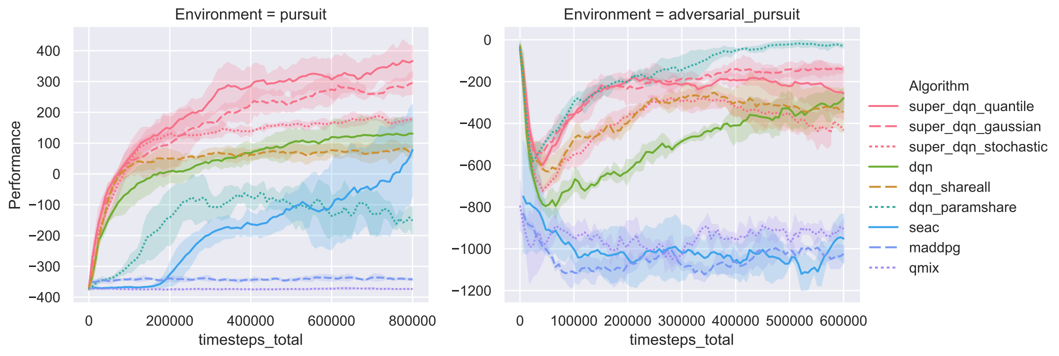

Figure 3: Performance of suPER-DQN variants with target bandwidth 0.1 on Pursuit and Adversarial-Pursuit. For Pursuit, performance is the total mean episode reward from all agents. For Adversarial-Pursuit, performance is the total mean episode reward from all agents in the prey team. Shaded areas indicate one standard deviation.

scenarios where fully centralized training may not be feasible. We aim to show here how sharing even a small number of experiences changes the learning dynamics versus to non-sharing opponents. Figure 4 shows this variant.

## 5.4 Performance Evaluation

Figure 2 shows learning curves of suPER implented on dueling DDQN ("suPER DDQN"), with stochastic, gaussian and quantile experience selection and target bandwidth 0.1, compared to standard no-sharing dueling DDQN, share-all suPER-DDQN, parameter-sharing dueling DDQN, as well as MADDPG, QMIX and SEAC. Figure 3 similarly shows suPER implemented on (non-dueling, non-double) DQN ("suPER DQN") ocmpared to non-sharing DQN, share-all suPER-DQN, parameter-sharing DQN, and the same multiagent algorithms as above (MADDPG, QMIX, SEAC; learning curves duplicated from Figure 2 for comparison). In both Figures 2 and 3 in Battle and Adversarial-Pursuit, the opposing team consists of agents pre-trained using non-sharing dueling DDQN (the same set of opposing agents are used for all the algorithms shown). In Figure 4 we show a subset of algorithms (suPER-DDQN, DDQN and share-all suPER-DDQN) training against a set of co-evolving agents. In this, the opposing team agents are also randomized at the start of training, and train together with the first team, but the opposing team agents do not utilize suPER experience sharing.

**Comparison Against Baseline DQN** We find that suPER (red curves) consistently outperforms the baseline DQN/DDQN algorithm (solid green curve), often significantly. For instance for DDQN (Fig. 2), in Pursuit suPER-DDQN achieves over twice the reward of no-sharing DDQN at convergence, increasing from 180.7 (std.dev 2.8) to 450.5 (std.dev 9.9) for quantile suPER-DDQN measured at 800k training steps. In both Battle and Adversarial-Pursuit, we enabled suPER-sharing for only one of the two teams each, and see that this significantly improved performance of the sharing team (as measured by sum of rewards of all agents in the team across each episode), especially mid-training. For instance in Battle, the blue team performance increase from -19.0 (std.dev

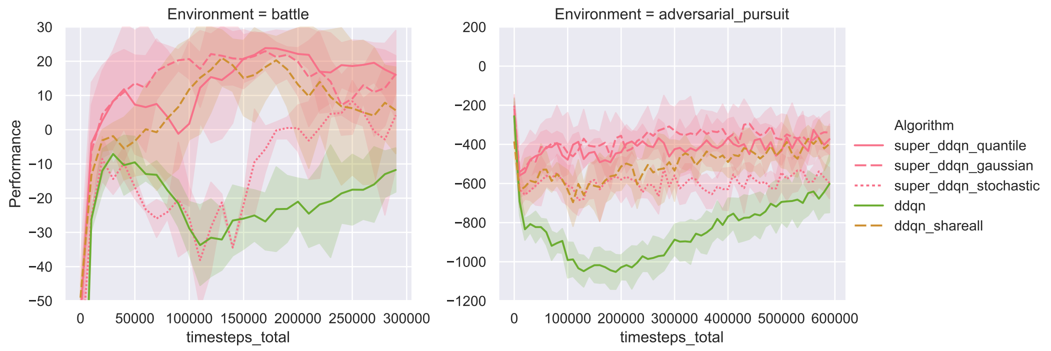

Figure 4: Performance of suPER-dueling-DDQN variants with target bandwidth 0.1 on all Battle and Adversarial=Pursuit, with co-evolving opponents. Performance is the total mean episode reward from all agents in the sharing team (blue team in Battle, prey team in Adversarial-Pursuit). Shaded areas indicate one standard deviation.

11.0) for no-sharing DDQN to 5.5 (std.dev 8.7) for quantile suPER-DDQN at 150k timesteps. In Adversarial-Pursuit, the prey performance increased from -712.3 (std.dev 55.3) to -508.0 (std.dev 34.2) at 300k timesteps. We similarly see that suPER-DQN outperforms DQN in Figure 3 in both environments, and that suPER-DDQN significantly outperforms DDQN with co-evolving opponent agents especially early in training (Fig. 4).

**Selective and Indiscriminate Sharing** When comparing suPER with bandwidth 0.1 (red lines) against suPER with indiscriminate sharing (”DDQN share-all”, dashed orange line), we see that selective experience sharing produces a much greater and more consistent performance improvement than indiscriminate sharing. This is most extreme in Pursuit, where indiscriminate sharing performs similar to, but slightly worse than, the DQN/DDQN baselines at convergence. It is also visible in Adversarial-Pursuit especially early in training, but not significant in Battle. We note that indiscriminate sharing still outperforms baselines early in training even in Pursuit.

**Comparison Against Other Multi-Agent RL Algorithms** suPER performs significantly better than SEAC (solid blue), MADDPG (dashed violet) and QMIX (dotted purple). MADDPG and QMIX performed very poorly on all domains despite extensive attempts at hyperparameter tuning. SEAC shows some learning in Pursuit, but remains well below even baseline DDQN performance, again despite extensive hyperparameter tuning. We have also run experiments with SEAC in Pursuit for significantly longer (not shown in the figure), and saw that performance eventually settles around 100-200 episode reward depending on hyperparamters and random seed. One experiment showed signs of additional learning much later on and began to surpass 200 episode reward (baseline DDQN) around 1.7M timesteps. However, we have not seen SEAC approach suPER performance in any experiment.

**Comparison Against Parameter-Sharing** suPER (red) significantly outperforms parameter-sharing DQN/DDQN (dotted turquoise line) in Pursuit and performs similar to it in Battle, whereas

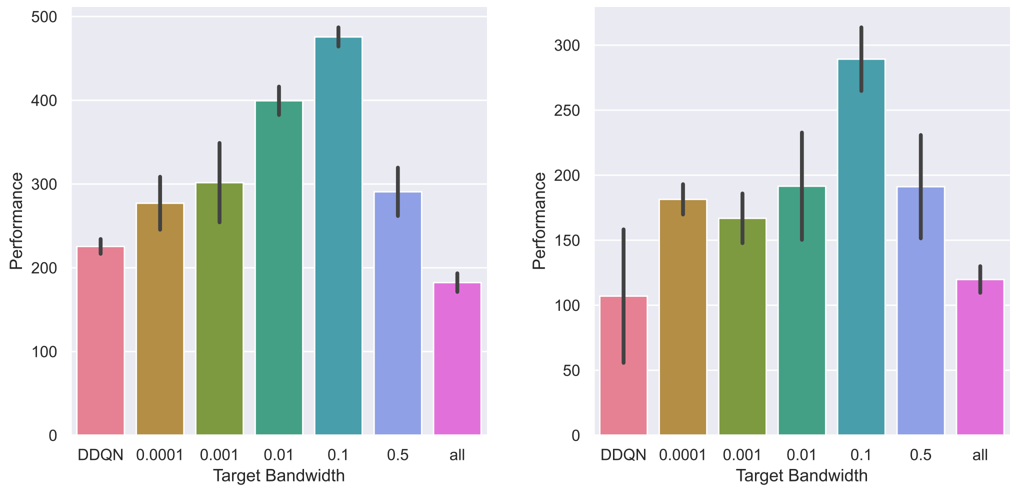

Figure 5: Performance of quantile suPER with varying bandwidth in Pursuit at 1-2M timesteps (left) and at 250k timesteps (right).

parameter-sharing performs significantly better in Adversarial-Pursuit. This holds for both the respective DQN and DDQN variants in Pursuit and Adversarial-Pursuit.

## 5.5 Bandwidth Sensitivity

In the Pursuit domain, we performed an analysis of the performance of suPER-DDQN for varying target bandwidths ranging from 0.0001 to 1 (sharing all experiences). Figure 5 (left) shows the converged performance of suPER-DDQN with quantile-based experience selection in Pursuit. A label of "DQN" indicates that no sharing is taking place, i.e. decentralized DDQN; a label of "all" indicates that all experiences are shared, without any prioritized selection. Numerical labels give different target bandwidths. Two things stand out to us: First, sharing all experiences indiscriminately does not result in increased performance. In fact, at convergence, it results in slightly lower performance than no-sharing DDQN. Second, there is a clear peak of performance around a target bandwidth of 0.01 - 0.1, which also holds for gaussian and stochastic experience selection (we refer the reader to the appendix for more details). We conclude that sharing experiences *selectively* is crucial for learning to benefit from it.

Furthermore, it is noteworthy that early in training, even small bandwidths show significantly higher performance than no-sharing DDQN. Figure 5 (right) shows performance at 250k timesteps, where even a target bandwidth of 0.0001 shows significantly improved performance over baseline. This suggests the applicability of suPER to settings where agents need to adapt rapidly to an unknown environment, even if communication is very limited. Indiscriminate sharing of all experiences again did not show a performance benefit at this stage of training. It did show increased performance (similar to selective sharing) very early in training, which we attribute to the faster replay buffer fill rate it entails. After replay buffers had been filled this performance improvement vanished.

### 5.6 Experience Selection

We see that at target bandwidth 0.1, quantile (solid red lines) and gaussian (dashed red) experience selection perform very similar across all domains. Stochastic experience selection (dotted red) performs similar or worse than both other variants, but generally still comparably or better than baseline DQN/DDQN. Gaussian experience selection performed similarly to the quantile selection we designed it to approximate. Its actual used bandwidth was however much less responsive to target bandwidth than the other two variants (e.g. in Pursuit at target bandwidth 0.001, Gaussian suPER consumed 0.018 actual bandwidth). We believe this demonstrates that in principle approximating the actual distribution of td-errors using mean and standard deviation is feasible, but that more work is needed in determining the optimal value of $c$ in equation 3.

### 5.7 Stability Across Hyperparameters

We evaluate suPER across a range of hyperparameter settings such as differing train batch sizes and exploration configurations, and find that it consistently improves performance over baseline DDQN in al settings. Appendix A shows this in more detail.

## 6 Conclusion & Discussion

**Conclusion** We present selective multiagent PER, a selective experience-sharing mechanism that can improve DQN-family algorithms in multiagent settings. Conceptually, our approach is rooted in the same intuition that Prioritized Experience Replay was based on, which is that td-error is a useful approximation of how much an agent could learn from a particular experience.

Experimental evaluation on DQN and dueling DDQN shows improved performance compared to fully decentralized training (as measured in sample efficiency and/or converged performance) across a range of hyperparameters of the underlying algorithm, and in multiple benchmark domains. We see most consistent performance improvements with suPER and a target bandwidth of 0.01-0.1 late in training, more consistent than indiscriminate experience sharing. Given that this effect appeared consistently across a wide range of hyperparameters and multiple environments, as well as on both DQN and dueling DDQN, the suPER approach may be useful as a general-purpose multi-agent RL technique. Equally noteworthy is a significantly improved performance early in training even at very low bandwidths. We consider this to be a potential advantage in future real-world applications of RL where sample efficiency and rapid adaption to new environments are crucial. suPER consistently and significantly outperforms MADDPG, QMIX and SEAC, and outperforms parameter sharing in Pursuit (but underperforms in Adversarial-Pursuit, and shows equal performance in Battle).

**Discussion** Our selective experience approach improves performance of both DQN and dueling DDQN baselines, and does so across a range of environments and hyperparameters. It outperforms state-of-the-art multi-agent RL algorithms, in particular MADDPG, QMIX and SEAC. The only pairwise comparison that suPER loses is against parameter sharing in Adversarial-Pursuit, in line with a common observation that in practice parameter sharing often outperforms sophisticated multi-agent RL algorithms. However, we note that parameter sharing is an entirely different, fully centralized training paradigm. Furthermore, parameter sharing is limited in its applicability, and does not work well if agents need to take on different roles or behavior to successfully cooperate. We see this in the Pursuit domain, where parameter sharing performs poorly, and suPER outperforms it by a large margin. The significantly higher performance than QMIX, MADDPG and SEAC is somewhat expected given that baseline non-sharing DQN algorithms often show state-of-the-art performance in practice, especially with regard to sample efficiency.

It is noteworthy that deterministic (aka "greedy") experience selection seems to perform slightly better than stochastic experience selection, while in PER the opposite is generally the case [25]. We have two hypotheses for why this is the case. One, we note that in PER, the motivation for stochastic prioritization is to avoid low-error experiences never being sampled (nor re-prioritized) in many draws from the buffer. On the other hand, in suPER we only ever consider each experience once. Thus, if in stochastic experience selection a high-error experience through random chance is not shared on this one opportunity, it will never be seen by other agents. In a sense, we may prefer deterministic experience selection in suPER for the same reason we prefer stochastic selection in PER, which is to avoid missing out on potentially valuable experiences. Two, in all our current experiments we used (stochastic) PER when sampling training batches from the replay buffer of each agent. When using stochastic suPER, each experience therefore must pass through two sampling steps before being shared and trained on by another agent. It is possible that this dilutes the probability of a high-error experience being seen too much.

Our algorithm is different from the "centralized training, decentralized execution" baselines we compare against in the sense that it does not require fully centralized training. Rather, it can be implemented in a decentralized fashion with a communications channel between agents. We see that performance improvements scale down even to very low bandwidth, making this feasible even with limited bandwidth. We think of this scheme as "decentralized training with communication" and hope this might inspire other semi-decentralized algorithms. In addition to training, we note that such a "decentralized with communication" approach could potentially be deployed during execution, if agents keep learning. While this is beyond the scope of the current paper, in future work we would like to investigate if this could help when transferring agents to new domains, and in particular with adjusting to a sim-to-real gap. Our work also shows that communication in multi-agent RL can improve training performance.

We would also like to point out a slight subtlety in our theoretical motivation for suPER: We use the sending agent's td-error as a proxy for the usefulness of an experience for the receiving agent. We believe that this is justified in symmetric settings, and our experimental results support this. However, we stress that this is merely a heuristic, and one which we do not expect to work in entirely asymmetric domains. For future work, we would be interested to explore different experience selection heuristics. As an immediate generalization to a more theoretically grounded approach, we wonder if using the td-error of each (potential) receiving agent could extend suPER to asymmetric settings, and if it could further improve performance even in symmetric settings. While this would effectively be a centralized-training approach, if it showed similar performance benefits as we have seen in symmetric settings for suPER, it could nevertheless be a promising avenue for further work. Beyond this, we would be interested to explore other heuristics for experience selection. For instance, we are curious if the sending agent could learn to approximate each receiver's td-error locally, and thus retain the decentralized-with-communication training capability of our current approach. However, given that td-error is intrinsically linked to current policy and thus highly non-stationary, we expect there would be significant practical challenges to this.

Finally, we focus on the DQN family of algorithms in this paper. In future work, we would like to explore suPER in conjunction with other off-policy RL algorithms such as SAC [9, 17] and DDPG [27]. The interplay with experience sampling methods other than PER, such as HER [3] would also be interesting. If the improvements we see in this work hold for other algorithms and domains as well, this could improve multi-agent RL performance in many settings.

## References

[1] Sanjeevan Ahilan and Peter Dayan. Correcting experience replay for multi-agent communication. *arXiv preprint arXiv:2010.01192*, 2020.

[2] Stefano V Albrecht and Peter Stone. Autonomous agents modelling other agents: A comprehensive survey and open problems. *Artificial Intelligence*, 258:66–95, 2018.

[3] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

[4] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123. PMLR, 2013.

[5] Filippos Christianos, Lukas Schäfer, and Stefano Albrecht. Shared experience actor-critic for multi-agent reinforcement learning. *Advances in Neural Information Processing Systems*, 33:10707–10717, 2020.

[6] Jakob Foerster, Ioannis Alexandros Assael, Nando De Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. *Advances in neural information processing systems*, 29, 2016.

[7] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[8] Jayesh K Gupta, Maxim Egorov, and Mykel Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems*, pages 66–83. Springer, 2017.

[9] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[10] Pablo Hernandez-Leal, Bilal Kartal, and Matthew E Taylor. A survey and critique of multiagent deep reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 33(6):750–797, 2019.

[11] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[12] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

[13] Natasha Jaques, Angeliki Lazaridou, Edward Hughes, Caglar Gulcehre, Pedro Ortega, DJ Strouse, Joel Z Leibo, and Nando De Freitas. Social influence as intrinsic motivation for multi-agent deep reinforcement learning. In *International conference on machine learning*, pages 3040–3049. PMLR, 2019.

[14] Angeliki Lazaridou, Alexander Peysakhovich, and Marco Baroni. Multi-agent cooperation and the emergence of (natural) language. *arXiv preprint arXiv:1612.07182*, 2016.

[15] Joel Z. Leibo, Vinicius Zambaldi, Marc Lanctot, Janusz Marecki, and Thore Graepel. Multi-agent reinforcement learning in sequential social dilemmas. In *Proceedings of the 16th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2017)*, AAMAS '17, page 464–473, Richland, SC, 2017. International Foundation for Autonomous Agents and Multiagent Systems.

[16] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2018.

[17] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[18] Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *Advances in neural information processing systems*, 30, 2017.

[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*. 2013.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[21] Peng Peng, Ying Wen, Yaodong Yang, Quan Yuan, Zhenkun Tang, Haitao Long, and Jun Wang. Multiagent bidirectionally-coordinated nets: Emergence of human-level coordination in learning to play starcraft combat games. *arXiv preprint arXiv:1703.10069*, 2017.

[22] Emanuele Pesce and Giovanni Montana. Improving coordination in multi-agent deep reinforcement learning through memory-driven communication. *CoRR*, abs/1901.03887, 2019.

[23] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. In *International conference on machine learning*, pages 4295–4304. PMLR, 2018.

[24] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder de Witt, Gregory Farquhar, Jakob N Foerster, and Shimon Whiteson. Monotonic value function factorisation for deep multi-agent reinforcement learning. *Journal of Machine Learning Research*, 21, 2020.

[25] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[26] Jürgen Schmidhuber, Jieyu Zhao, and Nicol N Schraudolph. Reinforcement learning with self-modifying policies. In *Learning to learn*, pages 293–309. Springer, 1998.

[27] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. PMLR, 2014.

[28] Sainbayar Sukhbaatar, Rob Fergus, et al. Learning multiagent communication with backpropagation. *Advances in neural information processing systems*, 29, 2016.

[29] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[30] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning*, 4(1):1–103, 2010.

[31] Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. Multiagent cooperation and competition with deep reinforcement learning. *PloS one*, 12(4):e0172395, 2017.

[32] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993.

[33] J. K Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sulivan, Luis Santos, Rodrigo Perez, Caroline Horsch, Clemens Dieffendahl, Niall L Williams, Yashas Lokesh, Ryan Sullivan, and Praveen Ravi. Pettingzoo: Gym for multi-agent reinforcement learning. *arXiv preprint arXiv:2009.14471*, 2020.

[34] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[35] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[36] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[37] Annie Xie, Dylan P Losey, Ryan Tolsma, Chelsea Finn, and Dorsa Sadigh. Learning latent representations to influence multi-agent interaction. *arXiv preprint arXiv:2011.06619*, 2020.

[38] Lianmin Zheng, Jiacheng Yang, Han Cai, Weinan Zhang, Jun Wang, and Yong Yu. Magent: A many-agent reinforcement learning platform for artificial collective intelligence. *CoRR*, abs/1712.00600, 2017.

[39] Changxi Zhu, Mehdi Dastani, and Shihan Wang. A survey of multi-agent reinforcement learning with communication. *arXiv preprint arXiv:2203.08975*, 2022.

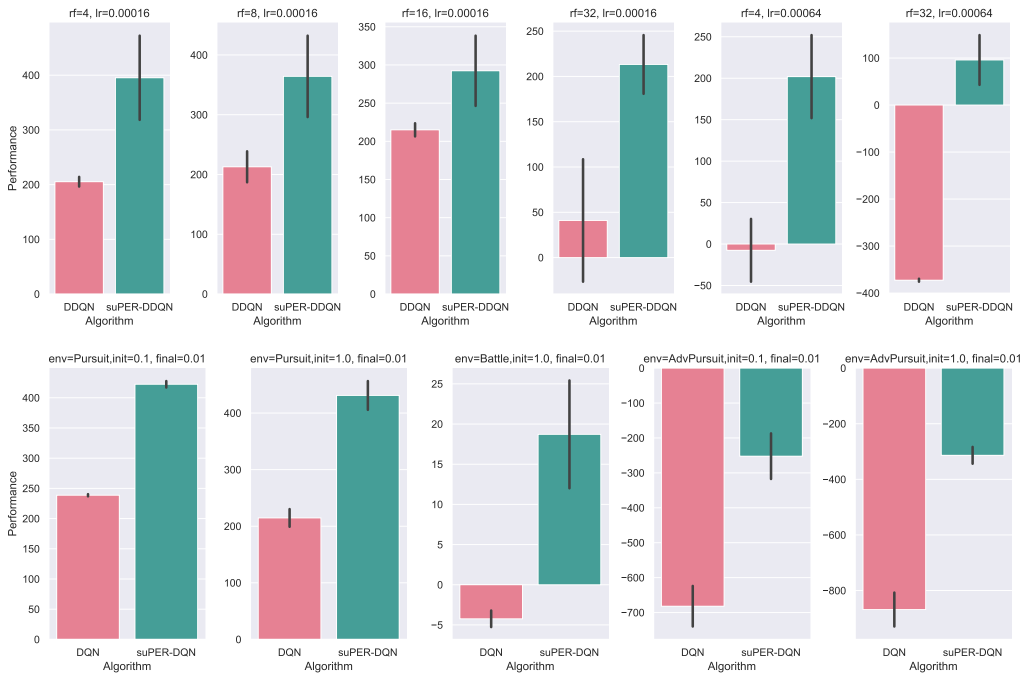

Figure 6: Performance of DDQN and suPER-DDQN (gaussian experience selection, target bandwidth 0.1) for differing hyperparameter settings of the underlying DDQN algorithm. Top: Different learning rates and rollout fragment lengths in Pursuit. Bottom: Different exploration settings in Pursuit and co-evolving variants of Batle and Adversarial-Pursuit. Hyperparameters otherwise identical to those used in Figure 2. Performance measured at 1M timesteps in Pursuit, 300k timesteps in Battle, 400k timesteps in Adversarial-Pursuit.

## A Stability across hyperparameters

Figure 6 shows performance of no-sharing DDQN and suPER-DDQN for different hyperparameters. As we can see, suPER-DDQN outperforms no-sharing DDQN consistently across all the hyperparameter settings considered.

## B Additional Analysis of Bandwidth Sensitivity

We present here a more detailed analysis of bandwidth sensitivity of suPER-DDQN in the three experience selection modes we discuss in the main text. Figure 7 shows the mean performance across five seeds for gaussian (left), quantile (middle) and stochastic (right) experience selection, at 1-2M timesteps (top) and at 250k timesteps (bottom). We can see that at 1-2M timesteps and a target bandwidth of 0.1, all three experience selection criteria perform similary. One thing that stands out is that stochastic selection has much lower performance at other target bandwidths, and also much less performance uplift compared to no-sharing DDQN at 250k timesteps at any bandwidth.

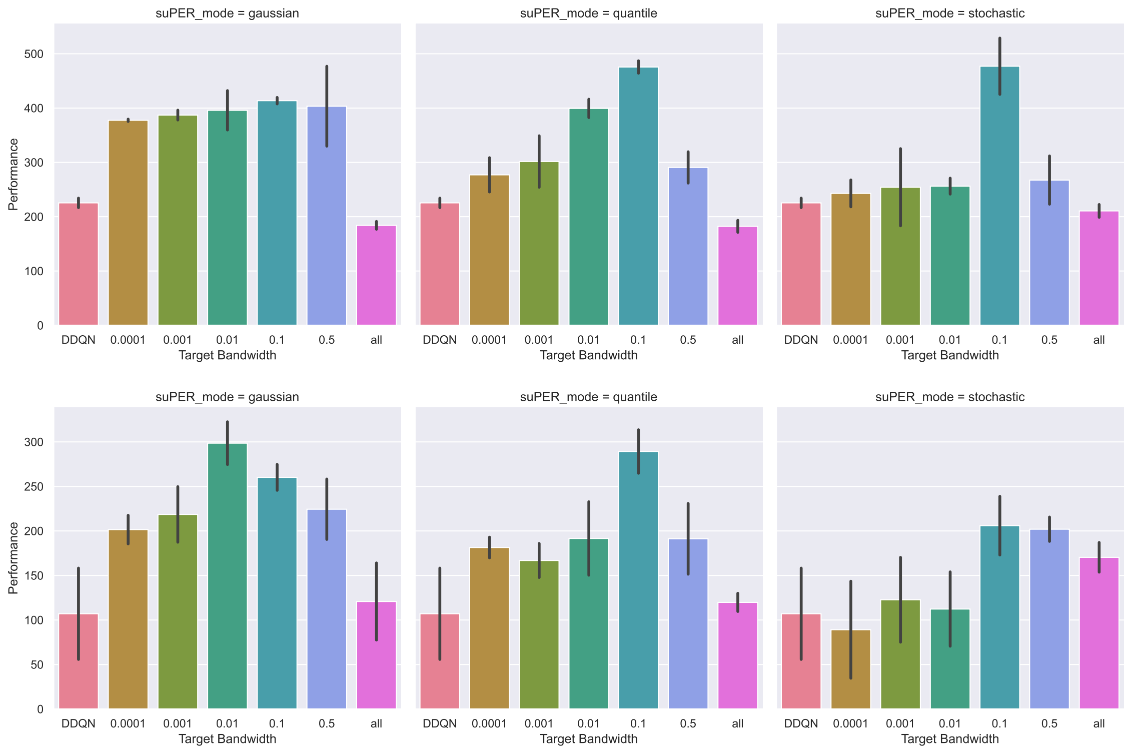

Figure 7: Performance of suPER with different experience selection and varying bandwidth in Pursuit at 1-2M timesteps (top) and at 250k timesteps (bottom).

Gaussian experience selection appears to be less sensitive to target bandwidth, but upon closer analysis we found that it also was much less responsive in terms of how much actual bandwidth it used at different settings. Figure 8 (left) shows the actual bandwidth used by each selection criterion at different target bandwidths. We can see that quantile and stochastic experience hit their target bandwidth very well in general.[2] What stands out, however, is that gaussian selection vastly overshoots the target bandwidth at lower settings, never going significantly below 0.01 actual bandwidth.

What is a fairer comparison therefore is to look at performance versus actual bandwidth used for each of the approaches, which we do in Figure 8 (middle, at 1-2M timesteps, and right, at 250k timesteps). For these figures, we did the following: First, for each experience selection approach and target bandwidth, we computed the mean performance and mean actual bandwidth across the five seeds. Then, for each experience selection mode, we plotted these $(mean actual bandwidth, mean performance)$ (one for each target bandwidth) in a line plot.[3] The result gives us a rough estimate of how each

[2]Quantile selection overshoots at 1e-4 (0.0001) target bandwidth and is closer to 1e-3 actual bandwidth usage, which we attribute to a rounding error, as we ran these experiments with a window size of 1500 (1.5e+3), and a quantile of less than a single element is not well-defined.

[3]Because each data point now has variance in both $x$- and $y$-directions, it is not possible to draw error bars for these.

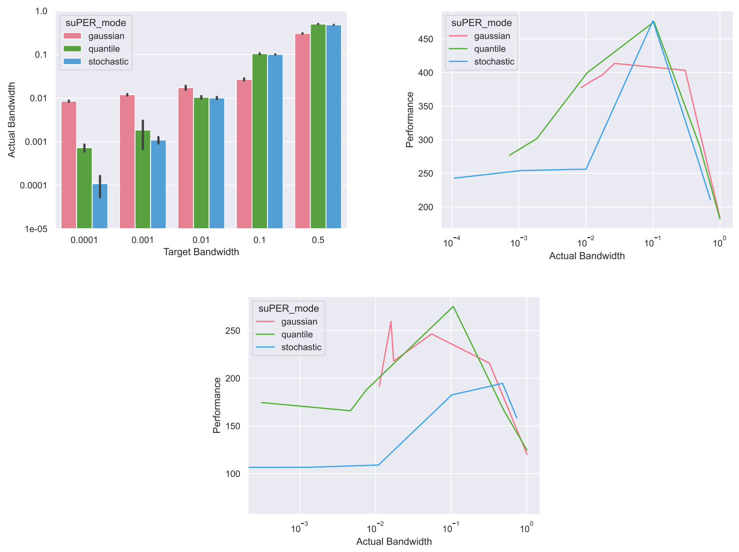

Figure 8: Left: Actual bandwidth used (fraction of experiences shared) at different target bandwidths. Middle, right: Performance compared to actual bandwidth used at 1-2M and 250k timesteps.

approach's performance varies with actual bandwidth used. We see again that stochastic selection shows worse performance than quantile at low bandwidths, and early in training. We also see that gaussian selection very closely approximates quantile selection. Notice that gaussian selection never hits an exact actual bandwidth of 0.1, and so we cannot tell from these data if it would match quantile selection's performance at its peak. However, we can see that at the actual bandwidths that gaussian selection does hit, it shows very similar performance to quantile selection. As stated in the main text, our interpretation of this is that using mean and variance to approximate the exact distribution of absolute td-errors is a reasonable approximation, but that we might need to be more clever in selecting $c$ in equation 3.

## C Experiment Hyperparameters & Details

We performed all experiments using the open-source library **RLlib** [16]. Experiments in Figure 2 and 3 were ran using RLlib version 2.0.0; experiments in other figures were run using version 1.13.0. Environments used are from PettingZoo [33], including SISL [8] and MAgent [38]. The suPER algorithm was implemented by modifying RLlib's standard DQN algorithm to perform the suPER experience sharing between rollout and training. Table 3 lists all the algorithm hyperparameters and environment settings we used for all the experiments. Experiments in the "Stability across hyperparameters" section had hyperparameters set to those listed in Table 3 except those specified in Figure 6. Any parameters not listed were left at their default values. Hyperparameters were tuned using a grid search; some of the combinations tested are also discussed in the "Stability across hyperparameters" section. For DQN, DDQN and their suPER variants, we found hyperparameters using a grid search on independent DDQN in each environment, and then used those hyperparameters for all DQN/DDQN and suPER variants in that environment. For all other algorithms we performed a grid search for each algorithm in each environment. For MADDPG and QMIX, we attempted further optimization using the Python **HyperOpt** package [4], however yielding no significant improvement over our manual grid search. For SEAC, we performed a grid search in each environment, but found no better hyperparameters than the default. We found a CNN network architecture using manual experimentation in each environment, and then used this architecture for all algorithms except MADDPG and QMIX where we used a fully connected net for technical reasons. We tested all other algorithms using both the hand-tuned CNN as well as a fully connected network, and found that the latter performed significantly worse, but still reasonable (and in particular significantly better than MADDPG and QMIX using the same fully connected network, on all domains).

All experiments were repeated with three seeds. All plots show the mean and standard deviation of these seeds at each point in training. For technical reasons, individual experiment runs did not always report data at identical intervals. For instance, one run might report data when it had sampled 51000 environment timesteps, and another run might report at 53000 environment timesteps. In order to still be able to report a meaningful mean and standard deviation across repeated runs, we rounded down the timesteps reported to the nearest $k$ steps, i.e. taking both the data above to represent each run's performance at 50000 steps. We set $k$ to the target reporting interval in each domain (8000 timesteps in Pursuit, 6000 timesteps in the other two domains). Where a run reported more than once in a 10000 step interval, we took the mean of its reports to represent that run's performance in the interval. Mean and standard deviation were calculated across this mean performance for each of the five seeds. To increase legibility, we applied smoothing to Figures 2 and 3 using an exponential window with $\alpha = 0.3$ for Pursuit, $\alpha = 0.1$ for Battle, and $\alpha = 0.25$ for Adversarial-Pursuit. This removes some noise from the reported performance, but does not change the relative ordering of any two curves.

## D Implementation & Reproducibility

All source code is included in the supplementary material and will be made available on publication under an open-source license. We refer the reader to the included README file, which contains instructions to recreate the experiments discussed in this paper.

Table 1: Hyperparameter Configuration Table - SISL: Pursuit

| **Environment Parameters** | | | |
|---|---|---|---|
| **HyperParameters** | **Value** | **HyperParameters** | **Value** |
| max cycles | 500 | x/y sizes | 16/16 |
| shared reward | False | num evaders | 30 |
| horizon | 500 | n catch | 2 |
| surrounded | True | num agents(pursuers) | 8 |
| tag reward | 0.01 | urgency reward | -0.1 |
| constrained window | 1.0 | catch rewards | 5 |
| obs range | 7 | | |
| **CNN Network** | | | |
| CNN layers | [32,64,64] | Kernel size | [2,2] |
| Strides | 1 | | |
| **suPER / DQN / DDQN** | | | |
| learning rate | 0.00016 | final exploration epsilon | 0.001 |
| batch size | 32 | nframework | torch |
| prioritized replay_alpha | 0.6 | prioritized replay eps | 1e-06 |
| dueling | True | target network update_freq | 1000 |
| buffer size | 120000 | rollout fragment length | 4 |
| initial exploration epsilon | 0.1 | | |
| **MADDPG** | | | |
| Actor lr | 0.00025 | Critic lr | 0.00025 |
| NN(FC) | [64,64] | tau | 0.015 |
| framework | tensorflow | actor feature reg | 0.001 |
| **SEAC** | | | |
| learning rate | 3e-4 | adam eps | 0.001 |
| batch size | 5 | use gae | False |
| framework | torch | gae lambda | 0.95 |
| entropy coef | 0.01 | value loss coef | 0.5 |
| max grad norm | 0.5 | use proper time limits | True |
| recurrent policy | False | use linear lr decay | False |
| seac coef | 1.0 | num processes | 4 |
| num steps | 5 | | |
| **QMIX** | | | |
| learning rate | 0.00016 | mixing embed dim | 32 |
| optim alpha | 0.99 | optim eps | 0.00001 |
| grad clip | 10 | NN architecture | RLlib default |

Table 2: Hyperparameter Configuration Table- MAgent: Battle

| **HyperParameters** | **Value** | **HyperParameters** | **Value** |
|---|---|---|---|
| **Environment Parameters** | | | |
| minimap mode | False | step reward | -0.005 |
| Num blue agents | 6 | Num red agents | 6 |
| dead penalty | -0.1 | attack penalty | -0.1 |
| attack opponent reward | 0.2 | max cycles | 1000 |
| extra features | False | map size | 18 |
| **CNN Network** | | | |
| CNN layers | [32,64,64] | Kernel size | [2,2] |
| Strides | 1 | | |
| **suPER / DQN / DDQN** | | | |
| learning rate | 1e-4 | batch size | 32 |
| framework | torch | prioritized replay_alpha | 0.6 |
| prioritized replay eps | 1e-06 | horizon | 1000 |
| dueling | True | target network update_freq | 1200 |
| rollout fragment length | 5 | buffer size | 90000 |
| initial exploration epsilon | 0.1 | final exploration epsilon | 0.001 |
| **MADDPG** | | | |
| Actor lr | 0.00025 | Critic lr | 0.00025 |
| NN(FC) | [64,64] | tau | 0.015 |
| framework | tensorflow | actor feature reg | 0.001 |
| **SEAC** | | | |
| learning rate | 3e-4 | adam eps | 0.001 |
| batch size | 5 | use gae | False |
| framework | torch | gae lambda | 0.95 |
| entropy coef | 0.01 | value loss coef | 0.5 |
| max grad norm | 0.5 | use proper time limits | True |
| recurrent policy | False | use linear lr decay | False |
| seac coef | 1.0 | num processes | 4 |
| num steps | 5 | | |
| **QMIX** | | | |
| learning rate | 0.00016 | mixing embed dim | 32 |
| optim alpha | 0.99 | optim eps | 0.00001 |
| grad clip | 10 | NN architecture | RLlib default |

Table 3: Hyperparameter Configuration Table - MAgent: Adversarial Pursuit

| **Environment Parameters** | | | |
|---|---|---|---|
| **HyperParameters** | **Value** | **HyperParameters** | **Value** |
| Number predators | 4 | Number preys | 8 |
| minimap mode | False | tag penalty | -0.2 |
| max cycles | 500 | extra features | False |
| map size | 18 | | |
| **Policy Network** | | | |
| CNN layers | [32,64,64] | Kernel size | [2,2] |
| Strides | 1 | | |
| **suPER / DQN / DDQN** | | | |
| learning rate | 1e-4 | batch size | 32 |
| framework | torch | prioritized replay alpha | 0.6 |
| prioritized replay eps | 1e-06 | horizon | 500 |
| dueling | True | target network update_freq | 1200 |
| buffer size | 90000 | rollout fragment length | 5 |
| initial exploration epsilon | 0.1 | final exploration epsilon | 0.001 |
| **MADDPG** | | | |
| Actor lr | 0.00025 | Critic lr | 0.00025 |
| NN(FC) | [64,64] | tau | 0.015 |
| framework | tensorflow | actor feature reg | 0.001 |
| **SEAC** | | | |
| learning rate | 3e-4 | adam eps | 0.001 |
| batch size | 5 | use gae | False |
| framework | torch | gae lambda | 0.95 |
| entropy coef | 0.01 | value loss coef | 0.5 |
| max grad norm | 0.5 | use proper time limits | True |
| recurrent policy | False | use linear lr decay | False |
| seac coef | 1.0 | num processes | 4 |
| num steps | 5 | | |
| **QMIX** | | | |
| learning rate | 0.00016 | mixing embed dim | 32 |
| optim alpha | 0.99 | optim eps | 0.00001 |
| grad clip | 10 | NN architecture | RLlib default |