Scaling Multi-Agent Reinforcement Learning with Selective Parameter Sharing

Filippos Christianos [1]   **Georgios Papoudakis** [1]   **Arrasy Rahman** [1]   **Stefano V. Albrecht** [1]

Abstract

Sharing parameters in multi-agent deep reinforcement learning has played an essential role in allowing algorithms to scale to a large number of agents. Parameter sharing between agents significantly decreases the number of trainable parameters, shortening training times to tractable levels, and has been linked to more efficient learning. However, having all agents share the same parameters can also have a detrimental effect on learning. We demonstrate the impact of parameter sharing methods on training speed and converged returns, establishing that when applied indiscriminately, their effectiveness is highly dependent on the environment. We propose a novel method to automatically identify agents which may benefit from sharing parameters by partitioning them based on their abilities and goals. Our approach combines the increased sample efficiency of parameter sharing with the representational capacity of multiple independent networks to reduce training time and increase final returns.

1. Introduction

Multi-agent reinforcement learning (MARL) aims to jointly train multiple agents to solve a given task in a shared environment. Recent work has focused on novel techniques for experience sharing (Christianos et al., 2020), agent modelling (Albrecht & Stone, 2018), and communication between agents (Rangwala & Williams, 2020; Zhang et al., 2020) to address the non-stationarity and multi-agent credit assignment problems (Papoudakis et al., 2019). A problem that has received less attention to date is how to scale MARL algorithms to many agents, with typical numbers in previous works ranging between two and ten agents.

One common implementation technique to facilitate training with a larger number of agents is *parameter sharing*

(e.g. (Gupta et al., 2017)) whereby agents share some or all parameters in their policy networks. In the literature, parameter sharing is typically applied indiscriminately across all agents, which we call *naive*. Naive parameter sharing has been effective primarily due to the similar (if not identical) observation and reward functions between agents found in many multi-agent environments. This similarity allows agents to share representations in intermediate neural network layers. Despite the occasional effectiveness of naive parameter sharing, it is not supported by theoretical work and has not received much attention beyond being mentioned as an implementation detail. Indeed, naive parameter sharing can decrease training time, but we show that it can be detrimental to final convergence in many environments, even when paired with implementation details that generally accompany it. We observe in our experiments that when the transition or the reward functions are distinct across agents, hidden representations that can be shared are harder to form, and fully-shared parameters are not effective.

These limitations, however, do not imply that parameter sharing does not have a place in MARL. In contrast, we believe that it is an essential tool in scaling deep MARL algorithms to large numbers of agents, provided it can be done selectively, so as not to limit final performance. Therefore, we aim to benefit from parameter sharing when possible but also avoid potential bottlenecks. We introduce a method, Selective Parameter Sharing (SePS)[*], to automatically identify agents which may benefit from sharing parameters by partitioning them based on their abilities and goals. This partitioning is performed by encoding each agent to an embedding space by observing their trajectories, and then applying an unsupervised clustering algorithm to the encodings.

We can acquire an intuition of the setting this paper discusses by imagining a team of robots that must learn to run a restaurant, fulfilling both waiters and cooks' roles. Of course, agents belonging to the same group have to learn similar policies and therefore shared latent representations can significantly decrease learning requirements (i.e. there is no need for each cook to learn separately how to chop ingredients). Nevertheless, waiters and cooks have almost no common functionalities, and the representational capacity

---

[1]School of Informatics, University of Edinburgh, Edinburgh, United Kingdom. Correspondence to: Filippos Christianos <f.christianos@ed.ac.uk>.

---

[*]We provide an open-source implementation of SePS here: `https://github.com/uoe-agents/seps`

of a single neural network poses a bottleneck when attempting to learn all distinct roles. Furthermore, we show that agents tend to forget information needed by others when updating the parameters with their own objectives, actively interfering with other agents' learning.

We provide comparisons of typical usage of parameter sharing (sharing across all agents, appending agent indices, or not sharing at all) and show that i) SePS can converge to higher returns than both not sharing parameters and sharing them naively, ii) SePS is more sample efficient and executes considerably faster than not sharing parameters. Moreover, in contrast to baseline methods, SePS scaled to hundreds of agents (we experimented with up to 200) in our environments that contained non-homogeneous agents.

2. Background

Markov Games: A Markov game (e.g. (Littman, 1994); also known as a stochastic game (Shapley, 1953)) with partial observability is defined by the tuple $(\mathcal{N}, \mathcal{S}, \{O^i\}_{i \in \mathcal{N}}, \{A^i\}_{i \in \mathcal{N}}, \mathcal{P}, \{R^i\}_{i \in \mathcal{N}})$, with agents $i \in \mathcal{N} = \{1, \ldots, N\}$, state space $\mathcal{S}$, joint observation space $\mathcal{O} = O^1 \times \ldots \times O^N$, and joint action space $\mathcal{A} = A^1 \times \ldots \times A^N$. Each agent i only perceives local observations $o^i \in O^i$ which depend on the current state. Function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \mapsto \Delta(\mathcal{S})$ returns a distribution over successor states given a state and a joint action; $R^i : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ is the reward function giving agent i's individual reward r_t^i at timestep t. The objective is to find policies $\pi = (\pi_1, \ldots, \pi_N)$ for all agents such that the discounted return of each agent i, $G^i = \sum_{t=0}^{T} \gamma^t r_t^i$, is maximised with respect to other policies in π, formally $\forall_i : \pi_i \in \arg\max_{\pi_i'} \mathbb{E}[G^i | \pi_i', \pi_{-i}]$ where $\pi_{-i} = \pi \setminus \{\pi_i\}$, γ is the discount factor, and T the total timesteps of an episode.

Unlike some recent MARL work we do not assume identical action, observation spaces, or reward functions between agents (Christianos et al., 2020; Rashid et al., 2018; Foerster et al., 2018).

Policy Gradient and Actor-Critic: The goal of reinforcement learning is to find strategies that optimise the returns of the agents. Policy gradient, a class of model-free RL algorithms, directly learn and optimise a policy π_ϕ parameterised by ϕ. The REINFORCE algorithm (Williams, 1992), follows the gradients of the objective $\nabla_\phi J(\phi) = \mathbb{E}_{\pi_\phi} [G_t \nabla_\phi \ln \pi_\phi(a_t|s_t)]$ to find a policy that maximises the returns. To further reduce the variance of gradient estimates, actor-critic algorithms replace the Monte Carlo returns with a value function $V_\pi(s; \upsilon)$. In a multi-agent, partially observable setting, a simple actor-critic algorithm defines the policy loss function for an agent i as:

$$\mathcal{L}(\phi_i) = -\log \pi(a_t^i|o_t^i; \phi_i)(r_t^i + \gamma V(o_{t+1}^i; \upsilon_i) - V(o_t^i; \upsilon_i))$$

and the respective value loss function as:

$$\mathcal{L}(\upsilon_i) = ||V(o_t^i; \upsilon_i) - y_i||^2 \text{ with } y_i = r_t^i + \gamma V(o_{t+1}^i; \upsilon_i)$$

In this paper, for reinforcement learning, we use A2C (Mnih et al., 2016), an actor-critic algorithm that additionally uses n-step rewards, environments that run in parallel, and improved exploration with entropy regularisation.

Variational Autoencoders: Variational autoencoders (VAEs) are generative models that explicitly learn a density function over some unobserved latent variables Z given an input $x \in X$, where X is a dataset. Given the unknown true posterior $p(z|x)$, VAEs approximate it with a parametric distribution $q_\theta(z|x)$ with parameters θ. Computing the KL-divergence from the parametric distribution to the true posterior results in:

$$\begin{aligned} \mathrm{D_{KL}}(q_\theta(z|x)\|p(z|x)) &= \log p(x) \\ &- \mathbb{E}_{z \sim q_\theta(z|x)}[\log p_u(x|z)] + \mathrm{D_{KL}}(q_\theta(z|x)\|q(z)) \end{aligned}$$

The term $\log p(x)$ is called log-evidence and it is constant. The other two terms are the negative evidence lower bound (ELBO). Minimising the ELBO is equivalent to minimising the KL-divergence between the parametric and the true posterior.

3. Selective Parameter Sharing

To improve the effectiveness of parameter sharing, and allow for several distinct roles to be learned, we attempt to group agents that should be sharing their parameters during training. In an environment, we assume that N agents can be partitioned into K sets ($K < N$) but without knowing K nor the partitioning. With $\mathcal{K} = \{\pi_1, \ldots, \pi_K\}$, each agent in a cluster k uses and updates the shared policy π_k. As we show in our experiments, such distinct shared policies can often be trained more efficiently while offering enough representational capacity to successfully solve the environment, and may even reach higher overall returns than alternative methods (see Section 4). Figure 1 depicts a top-level diagram of the components in our architecture.

To assign agents to partitions, we propose the use of a deterministic function $\mu : \mathcal{N} \mapsto \mathcal{K}$ that maps each agent i to a parameterised policy (or partition) π_k. This partitioning is learned prior to RL training. Therefore, agents that share parameters get to benefit from shared representations in the latent layers of their neural networks, while not interfering with agents using other parameters.

Recall the transition $\mathcal{P}$ and reward R^i functions that define an environment's dynamics (Section 2). We aim to determine μ and partition agents such that agents which try to solve similar tasks use shared policies. Therefore, we introduce another concept: a set of functions $\hat{\mathcal{P}}^i$ and $\hat{R}^i$ that
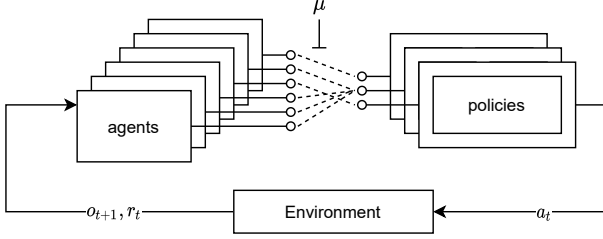
Figure 1: A top-level diagram of the selective parameter sharing architecture. N agents are operating in one environment and receive observations and rewards. With SePS we are training $K < N$ policies and use a deterministic function μ to select which policy controls which agent.
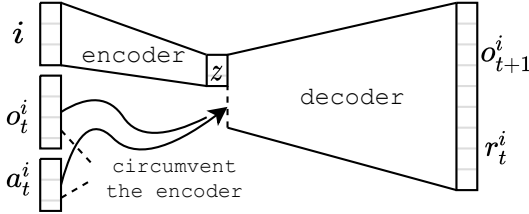

Figure 2: The encoder-decoder model. The encoder learns to encode the id of an agent in an embedding space while the decoder predicts the reward and next observation.

attempt to approximate $\mathcal{P}$ and R^i, but from the agents' limited perspective of the world. An agent does not observe the state nor the actions of another agent, and hence we define $\hat{\mathcal{P}}^i : O^i \times A^i \mapsto \Delta(O^i)$ and $\hat{R}^i : O^i \times A^i \mapsto \mathbb{R}$, that model the next observation and reward respectively, based only on the observation and action of an agent i. When learning these functions our goal is not to ensure their accuracy as approximators of the dynamics; but rather that they identify similar agents to provide a basis for partitioning. We speculate that agents that should be grouped have similar reward and observation functions. Thus, the reasoning behind the following method is our desire to identify agents with identical $\hat{\mathcal{P}}^i$ and $\hat{R}^i$, and have them share their network parameters.

3.1. Encoding Agent Identities

We define an encoder f_e and a decoder f_p (Fig. 2) parameterised by θ and u respectively. The encoder, conditioned solely on the agent id, outputs the parameters that define an m-dimensional Gaussian distribution we can sample from. We refer to samples from this latent space as z. The decoder, further divided into an observation and reward decoder f_p^o and f_p^r respectively, receives the observation, action, and sampled encoding z of agent i, and attempts to predict the next observation and reward. In contrast to the classical definition of autoencoders, o_t^i and a_t^i *bypass* the encoder and are

only received by the decoder. Thus, due to the bottleneck, z can only encode information about the agent, such as its reward function $\hat{R}^i$ or observation transition model $\hat{\mathcal{P}}^i$.

To formalise the process, we assume that for each agent its identity i is representative of its observation transition distribution and reward function. Additionally, we assume that both the identity of each agent and its observation transition distribution can be projected in a latent space Z through the posteriors $q(z|i)$ and $p(z|\text{tr} = (o_{t+1}, o_t, r_t, a_t))$. The goal is to find the posterior $q(z|i)$. We assume a variational family of parameterised Gaussian distributions with parameters θ: $q_\theta(z|i) = \mathcal{N}(\mu_\theta, \Sigma_\theta; i)$. To solve this problem we use the variational autoencoding (Kingma & Welling, 2014) framework to optimise the objective $D_{KL}(q_\theta(z|i)||p(z|\text{tr}))$.

We derive a lower bound on the log-evidence (ELBO) of the transition $\log p(tr)$ as:

$$\log p(\text{tr}) \geq \mathbb{E}_{z \sim q_\theta(z|i)}[\log p_u(\text{tr}|z)]$$
$$- D_{KL}(q_\theta(z|i)||p(z)) \quad (1)$$

The reconstruction term of the ELBO factorises as:

$$\log p_u(\text{tr}|z) = \log p_u(r_t, o_{t+1}|a_t, o_t, z)p(a_t, o_t|z) =$$
$$\log p_u(r_t|o_{t+1}, a_t, o_t, z) + \log p_u(o_{t+1}|a_t, o_t, z) + c$$

The last term is discarded as a_t and o_t do not depend on the latent variable z. In these instances, q_θ and p_u function as f_e and f_p respectively.

For the encoder-decoder model to learn from the experience of all agents, it is trained with samples from *all* agents and will represent the collection of the agent-centred transition and reward functions $\hat{\mathcal{P}}^i$ and $\hat{R}^i$ for all $i \in \mathcal{N}$. Given the inputs of the decoder, the information of the agent id can only pass through the sample z.

Minimising the model loss (Eq. (1)) can be done prior to reinforcement learning. We sample actions $a^i \sim A^i$ and store the observed trajectories in a shared experience replay with all agents. We have empirically observed that the data required for this procedure is orders of magnitude less than what is usually required for reinforcement learning, and can even be reused for training the policies, thus not adding to the sample complexity.

The final step of the pre-training procedure is to run a clustering algorithm on the means generated from the encoder $f_e(i)$ for all $i \in N$, and use the agent indices clustered together to define μ. In the experiments that follow, we use k-means for simplicity. After the partitioning is completed, a static computational graph (for automatic differentiation) can be generated to train the policies with significant speed advantages.

Table 1: Brief description of environments, including how agents are distributed to different agent types: colours in BPS and C-RWARE, levels in LBF or different units in MMM2. Environments with † or ‡ mark different observation spaces or action spaces across types respectively, while § marks a cooperative (shared) reward.

	# Agents	# Types	Type Distribution
BPS (1)	15	3	5–5–5
BPS (2)	30	3	10–10–10
BPS (3)	30	5	6–6–6–6–6
BPS (4)	30	5	2–2–2–15–9
BPS-h (1)	15	3†	5–5–5
BPS-h (2)	30	5†	6–6–6–6–6
BPS-h (3)	200	4†	50–50–50–50
C-RWARE (1)	4	2‡	2–2
C-RWARE (2)	8	2‡	4–4
C-RWARE (3)	16	2‡	8–8
LBF	12	3	4–4–4–4
MMM2	10§	3	7–2–1

4. Experimental Evaluation

In this section, we evaluate both whether SePS performs as intended by correctly partitioning the agents and whether this partitioning helps in improving the overall returns, sample complexity, and training time. For RL, we use the A2C (Mnih et al., 2016) algorithm and report the sum of returns of all agents. A search was performed for A2C's hyperparameters across all baselines, while hyperparameters of the clustering portion of SePS were easily found manually, and kept identical across all environments (more details in Section 4.8).

4.1. Multi-Agent Environments

We use four multi-agent environments (Fig. 3) which are described below and summarised in Table 1.

Blind-particle Spread: Our motivating toy environment is a custom scenario created with the Multi-agent Particle Environment (MPE) (Lowe et al., 2017). Blind-particle spread (BPS, Fig. 3a) consists of landmarks of different colours and numerous agents that have been also assigned colours. The agents are unable to see their own colour (or of the other agents) but need to move towards the correct landmark. This environment enables us to investigate the effects of parameter sharing by allowing us to control two important variables: i) the number of agents and ii) the number of colours (distinct behaviours that must be learned). In a further, more difficult variation which we name BPS-h, each group of agents also has a different observation space (e.g. the agents could be equipped with different sensors).

Coloured Multi-Robot Warehouse: The Coloured Multi-Robot Warehouse (C-RWARE, Fig. 3b) is a variation of the RWARE environment (Christianos et al., 2020), where multiple robots have different functionalities and are rewarded only for delivering specific shelves (denoted by different colours) and have different action spaces. The agents can rotate or move forward and pick up or drop a shelf. The observation consists only of a 3×3 square centred around the agent. Agents are only rewarded (with 1.0) when successfully arriving at the goal with a requested shelf of the correct colour, making the reward sparse. RWARE is known (Christianos et al., 2020; Papoudakis et al., 2020) to be an environment with difficult exploration, and independent learners have been shown to struggle on it.

Level-based Foraging: Level-based Foraging (LBF, Fig. 3c) (Albrecht & Ramamoorthy, 2013) is a multi-agent environment where agents are placed in a grid, and required to forage randomly scattered food. Each agent is assigned a level, and each food also is assigned a level at the beginning of the episode. The agents can move in four directions and attempt to forage an adjacent food. For foraging to be successful, the sum of the agent levels foraging the food must be equal or greater than its level. LBF is partially observable, and while the agents can see the positioning of agents and food, as well as the food levels, they cannot see any of the agent levels. The reward is proportionate to the agents' contribution when a food is successfully loaded.

Starcraft Multi-Agent Challenge: While the multi-agent Starcraft (SMAC) (Samvelyan et al., 2019) environment might not be the archetype for displaying the strengths of selective parameter sharing, it is a widely used setting where multiple agents of distinct types co-exist and must learn together. For instance, the "MMM2" environment (Fig. 3d) contains three types of units (marines, marauders, and medivacs) with distinct attributes. One of those unit types, medivacs, is especially different since it needs to learn how to heal friendly units instead of attacking enemies.

4.2. Baselines

We compare SePS against several other methods of parameter sharing described below.

No Parameter Sharing (NoPS): In our NoPS baseline, all agents have their own set of parameters, and there is no overlap of gradients. This approach is common in the literature and usually encountered when there is no mention of parameter sharing, e.g. MADDPG (Lowe et al., 2017).

Full Parameter Sharing (FuPS): The second baseline, FuPS consists of a single set of parameters that will be shared between all agents. FuPS is a naive baseline since it does not allow agents to ever develop any differences in their behaviour.

(a) Blind-Particle Spread (BPS)

(b) Coloured Multi-Robot Warehouse (C-RWARE)

(c) Level-based Foraging (LBF)

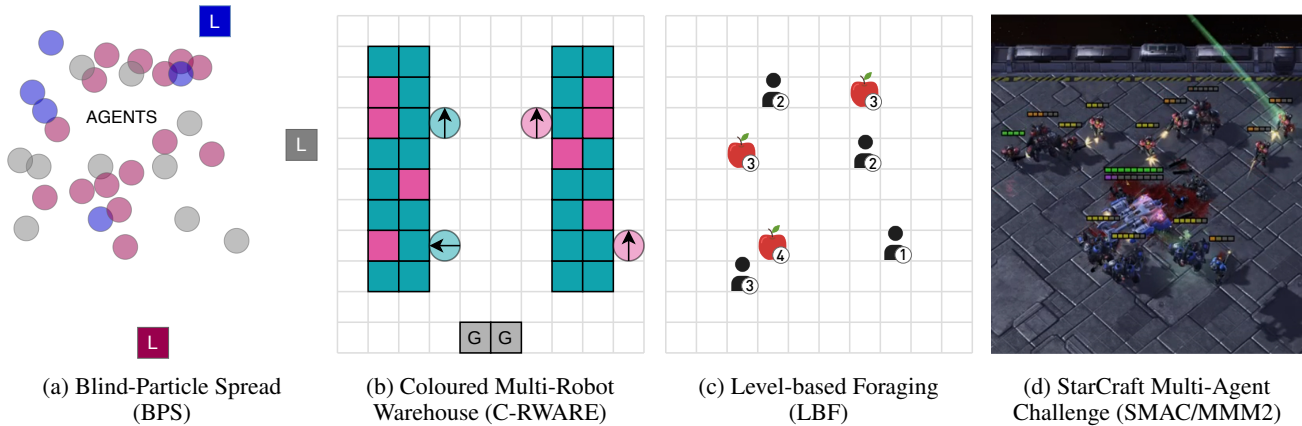(d) StarCraft Multi-Agent Challenge (SMAC/MMM2)

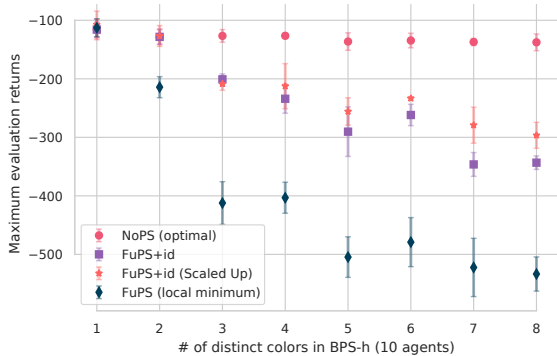Figure 3: Visualisation of environments used in experiments.

Figure 4: BPS-h with 10 agents and a variable number of colours. The maximum evaluation returns are recorded for each environment and algorithm, and vertical bars indicate standard deviation across seeds.

Full Parameter Sharing with index (FuPS+id): Finally, we test a variation of the previous method, where the policy is also conditioned on the agent id. While the use of FuPS is limited and our expectations are not high, since there is no way to differentiate between agents, FuPS+id is encountered very often in the literature (Rashid et al., 2018; Foerster et al., 2018; Gupta et al., 2017).

4.3. An Experimental Evaluation of FuPS+id

The dense reward signal in our toy environment, BPS, makes it sufficiently simple for non-parameter sharing agents to learn how to reach their respective landmarks. However, when parameter sharing is involved, we expect the task to become considerably harder. FuPS+id presumably solves this by allowing each agent to develop a distinct policy based on agent id. To investigate this, we tested NoPS, FuPS, and FuPS+id, in a series of BPS tasks, where the number of agents will remain constant, and the number of colours (landmarks) increases. NoPS should have no issue learning

immediately since each agent needs to learn to navigate to a specific landmark. In contrast, sharing parameters with FuPS can not work because the agents lack the information required to determine their colour and move accordingly. Therefore, agents trained with NoPS tend to fall into the *local minimum* of moving to a location that minimises the distance between all landmarks.

We are, however, very interested in what FuPS+id can learn. The agents have all the information needed to learn how to move to the correct landmark. But, as we have hypothesised in earlier sections, the overlap of different policies which must be represented on the same parameters, poses a significant bottleneck for learning. Indeed, as Fig. 4 indicates, the performance of FuPS+id deteriorates sharply, even with only three colours.

An argument could be made that an increased number of colours in the BPS task should be accompanied by an increase in FuPS+id's model size. Such an increase in the number of parameters increases the representation capacity and could allow for multiple distinct policies to be learned. To test this hypothesis we include in Fig. 4 the FuPS+id (Scaled Up) baseline which scales the width of the network such that the number of parameters grows with the number of colours shown in the x axis. Specifically, in this experiment the FuPS and FuPS+id baselines use approximately $18K$ parameters (two layers of 128 units). The FuPS+id (Scaled Up) baseline, however, uses approximately $\#Colors * 18K$ parameters for the different BPS-h tasks (for exact sizes of the layers in each task see Section 4.8).

However, we observe that even the FuPS+id (Scaled Up) baseline does not successfully learn these otherwise simple BPS tasks. Therefore we conjecture that the issue with FuPS+id is not the model capacity since it could have enough parameters to learn all behaviours. Instead, learning on shared parameters interferes with the learning of other agents. In the following sections, we will instead optimise

the network size as a hyperparameter.

4.4. Reinforcement Learning with Parameter Sharing

Next, we investigate how selectively sharing parameters affects reinforcement learning performance. Our hypothesis remains that agents can benefit from sharing parameters if they have been clustered together by SePS. Our results, detailed in Table 2 support this hypothesis. We also present the learning curves on a selection of environments in Fig. 5.

BPS: The BPS tasks (Table 2 and Figs. 5a and 5b) are trivial for independent learners given the dense reward: each agent learns to always move towards a specific colour. However, if all agents share a policy, then the agents (not being able to perceive their own colours) only learn to move to a local minimum. FuPS+id, which supposedly circumvents the problem, still has issues correctly learning this problem. Due to the high computational requirements of NoPS, which requires N different sets of parameters, running it on BPS-h (4) with 200 agents was infeasible.

C-RWARE: Our results in C-RWARE (Table 2 and Figs. 5c and 5d) are more surprising. NoPS which was a strong contender in BPS, completely failed to learn in C-RWARE (2) and (3). These tasks are very sparsely rewarded, which seems to make independent learning ineffective. Instead, sharing parameters also combined the received rewards, providing a useful learning direction to the optimiser. Also, similarly to BPS, SePS outperforms the other naive parameter sharing methods.

LBF: Similarly to other environments, SePS agents in LBF achieve higher returns with more efficient use of environment samples (Fig. 5e). The optimal performance in LBF is 1.0, and while FuPS+id is close, it does not achieve the same returns as SePS. NoPS takes considerably more samples to train, but given our BPS results, it is possible it eventually converges to the same returns as SePS.

MMM2: In one of the hardest SMAC environments, MMM2 (Fig. 5f), the most surprising result was the difference in converged returns between NoPS and parameter sharing methods. Even fully shared parameters - and after making sure the identity of the agents does not leak through the observation - outperforms NoPS. Our hypothesis on these results is that i) this task requires agents to act in a very similar way (e.g. only targeting the same opponents) and ii) parameter sharing plays a previously underrated role in decomposing (or reasoning over) a shared reward. The rest of the methods behave similarly to other environments, but with a minimal improvement of SePS over FuPS+id.

4.5. A Peek into the Embedding Space of SePS

Next, an important step in evaluating SePS is to verify that meaningful clusters appear after optimising the objectives discussed in Section 3.

Our goal is to compare the clustering of our algorithm, with one decided by a human who is given knowledge of the environment. We visualise the embedding for each of the units on a SMAC task (Fig. 6). We can assert that clearly visible clustering is precisely what we would expect. Different types of units all have different properties (e.g. movement speed, health, or damage) and thus a distinct interaction with the environment. These differences were picked up by the encoder that subsequently spread them in the embedding space.

A question that might arise is why the agents of the same unit type (and cluster) are spread out in the z axis of Fig. 6. In the SMAC environment, there is another difference between the agents: their starting position. Therefore, the initial observations o_0^i are sampled from different sets for each of the agents. The encoder picks up on this feature and further spreads the latent encoding. While k-means clustered the agents by unit type, it could be argued that more clusters could have been formed, which goes beyond the scope of our work and is considered a feature (and open problem) of unsupervised clustering (Kaufman & Rousseeuw, 2009). However, this was the exception in our tests, and in all other environments, where the starting observations are sampled from the same set, the latent variables of similar agents are overlapping one another. Nevertheless, in Section 4.6 we explore how the number of clusters can be determined, and how wrong choices can affect learning.

The clustering process across all environments and seeds matched the various types of agents. For instance, in C-RWARE and BPS, each cluster contains only agents of the same colour. Importantly, this information is not included in the observation space and therefore not observed by the agents or even the encoder; it is only understood after observing the transitions and rewards of each agent.

4.6. Determining the Number of Clusters

Several ways to determine the number of clusters in the embedding space exist. Arguably the most straightforward is the use of domain knowledge. But, having an estimate of the value of K, could also mean knowledge of which agents should be assigned in clusters in the first place. Despite the diminished importance of the pretraining SePS stage in this situation, we believe that understanding the effectiveness of shared parameters between clusters is still of value.

A second approach would consist of treating K as a tunable hyperparameter. In Fig. 7, we present the returns during training on C-RWARE when SePS is forced to create a varied amount of clusters. It is clear from the results, that overestimating K is of little significance. However, trying to form fewer clusters than needed lowers the achieved returns,

(a) BPS (3)

(b) BPS-h (2)

(c) C-RWARE (1)

(d) C-RWARE (3)
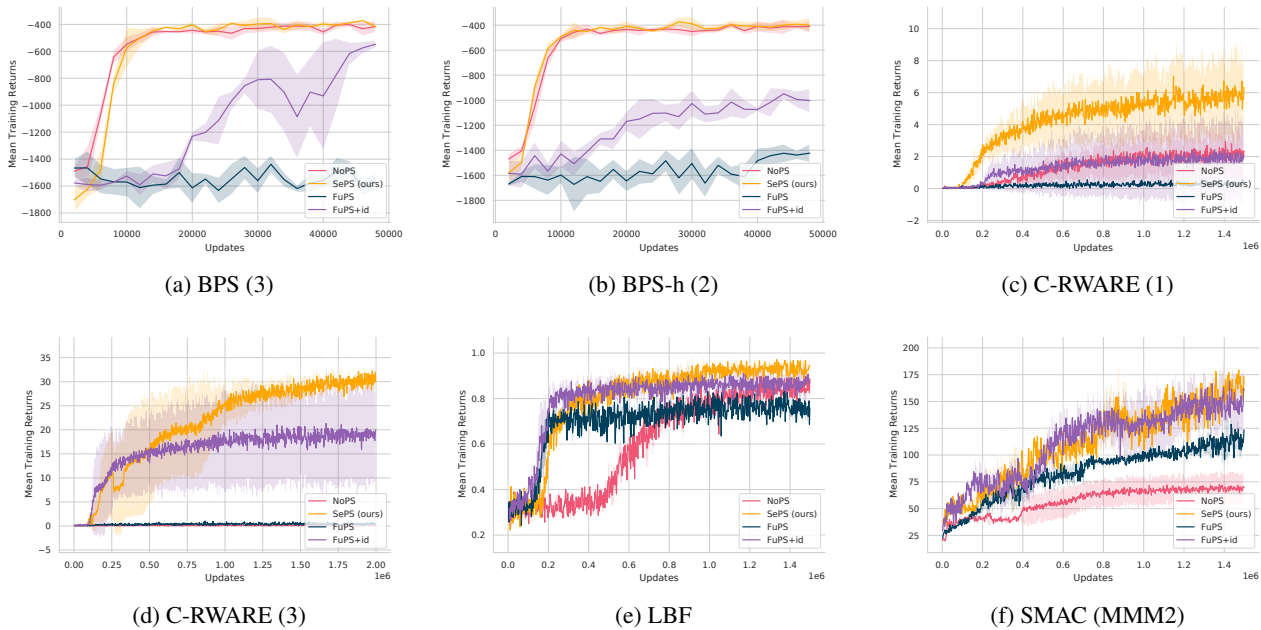
(e) LBF

(f) SMAC (MMM2)

Figure 5: Learning curves showing the mean returns during training for a selection of the environments. The shaded area represents standard deviation across seeds.
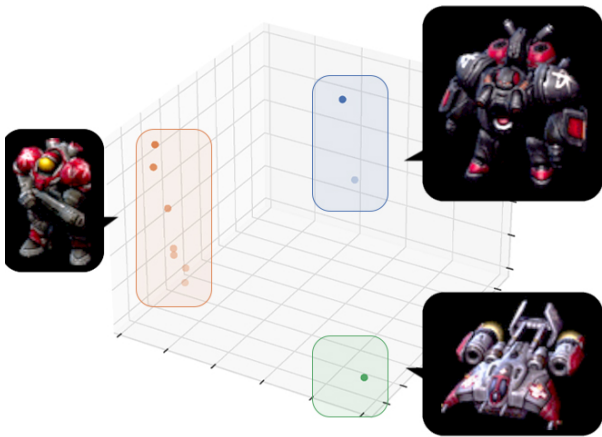

Figure 6: Visualisation of the means of z for each of the 10 agents in SMAC (MMM2 task) in a 3-dimensional space. The colours identify the clusters created by k-means.

Figure 7: Mean returns during training for different number of clusters on C-RWARE (9 agents and 3 colours).

and collapses to NoPS when $K = 1$.

Finally, there is a plethora of well-studied heuristics for separating clusters when the embedding space is known. The elbow method (Thorndike, 1953), the silhouette method (Rousseeuw, 1987), or the Davies–Bouldin index (Davies & Bouldin, 1979), all could be used to determine the number of clusters since our method tends to produce well-separated values. We have implemented and tested the Davies-Bouldin index, and we have found that
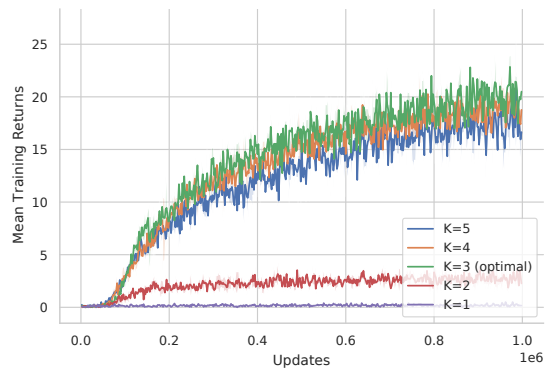
coupled with k-means, reliably finds the same clusters an expert would in our tested environments (i.e. the second column in Table 1).

4.7. Computational Benefits

In the previous sections, we showed the effectiveness of learning, showing that SePS achieves the highest returns among the baselines. However, we have not addressed how SePS computationally benefits MARL when applied to multiple agents. To examine this, we have created Fig. 8, which presents the median time for a timestep during training. It is

Table 2: Maximum evaluation returns with std across seeds. Highest means (within one std) are shown in bold.

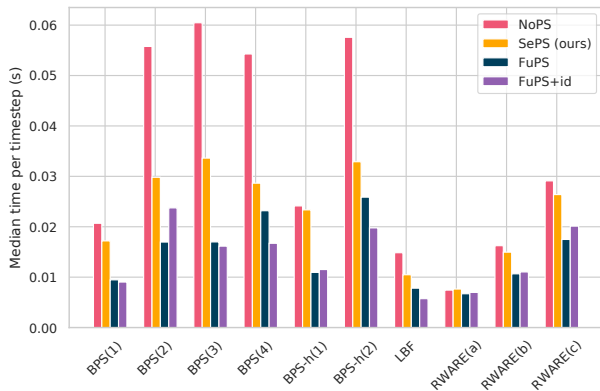|	NoPS	SePS (Ours)	FuPS	FuPS+id
BPS (1)	-189.99 ± 12.96	-179.69 ± 7.02	-612.18 ± 52.14	-221.44 ± 9.72
BPS (2)	-385.18 ± 34.91	-369.69 ± 38.70	-1301.35 ± 164.84	-443.84 ± 20.74
BPS (3)	-401.09 ± 20.37	-371.99 ± 6.92	-1438.41 ± 68.41	-546.95 ± 29.52
BPS (4)	-388.21 ± 41.35	-378.50 ± 77.75	-1339.14 ± 43.95	-722.33 ± 103.92
BPS-h (1)	-187.92 ± 15.78	-189.93 ± 31.29	-571.46 ± 52.12	-293.31 ± 26.89
BPS-h (2)	-398.11 ± 23.64	-373.92 ± 39.19	-1422.80 ± 74.70	-948.44 ± 43.67
BPS-h (3)	N/A	-2522.61 ± 276.29	-6825.47 ± 115.74	-4085.51 ± 3.71
C-RWARE (1)	2.93 ± 2.25	7.03 ± 3.72	0.57 ± 0.28	2.42 ± 3.52
C-RWARE (2)	0.28 ± 0.28	20.88 ± 1.15	0.55 ± 0.40	10.35 ± 8.65
C-RWARE (3)	0.33 ± 0.15	32.27 ± 3.16	1.03 ± 0.64	21.30 ± 15.13
LBF	0.91 ± 0.05	0.97 ± 0.03	0.83 ± 0.02	0.91 ± 0.05
MMM2	72.32 ± 15.00	179.45 ± 7.11	128.75 ± 18.37	168.39 ± 29.99

Figure 8: Median running time of a timestep during training over all the the environments and methods.

clear that while SePS adds computational complexity over the fully shared networks, it scales significantly better than NoPS does. In the BPS environments with 30 agents, SePS almost requires half the training time of NoPS due to the substantially fewer trainable parameters. In BPS-h(3), training with NoPS was infeasible since it requires 200 sets of parameters (50 more times than SePS).

4.8. Implementation Details

In our experiments, we used Adam with learning rate of $3e-4$, optimiser epsilon $1e-5$, entropy coefficient $1e-2$, and value, critic, and encoder-decoder networks with two layers of 64 or 128 units. Eight environments were sampled concurrently and 5-step returns were computed. The hyperparameters of entropy, learning rate, network width (but not depth), and n-step returns have been optimised using a coarse grid search, on one task per environment. A finer grid search on the entropy coefficient for the SMAC task

was performed. For the encoder-decoder training, m was set at 5, the KL loss was scaled by $1e-4$, and we used batch size 128. For Fig. 4, FuPS and FuPS+id use two layers of 128 units, but FuPS+id (Scaled Up) uses two layers with a width of 128, 189, 236, 277, 313, 345, 375, and 401 for BPS-h with one to eight colors respectively. Figure 8 was generated in an AMD Epyc 7702 running Python 3 with environments sampled in parallel threads.

5. Related Work

Centralised Training with Decentralised Execution (CTDE): A paradigm popular in cooperative MARL, assumes that during training all agents can access data from all other agents. After the training is completed, the agents stop having access to external data, and can only observe their own perspective of the environment. CTDE algorithms such as MADDPG (Lowe et al., 2017), Q-MIX (Rashid et al., 2018), and SEAC (Christianos et al., 2020) all benefit from the centralised training stage and have been repeatedly shown to outperform non-CTDE baselines. SePS also adheres to the CTDE paradigm and assumes that during training all information is shared.

Parameter Sharing: Sharing parameters between agents has a long history in MARL. Tan (1993) investigates sharing policies between cooperative settings in non-deep RL settings. More recently, algorithms such as COMA (Foerster et al., 2018), Q-Mix (Rashid et al., 2018), or Mean Field RL (Yang et al., 2018) share the parameters of neural networks similarly to our FuPS and FuPS+id baselines. ROMA (Wang et al., 2020) learns dynamic roles to share experience between agents that perform similar tasks. With SePS we do this operation statically in order to maximise computational efficiency, but we arrive at similar partitioning of agents in heterogenous SMAC tasks (Fig. 6). The novelty of SePS does not come from sharing parameters,

which is a well-established method in MARL, but that it creates neural network architectures in advance, allowing more efficient and effective sharing.

Sharing Experience: SEAC (Christianos et al., 2020) shares experience between agents while maintaining separate policy and value networks. While SEAC achieves state-of-the-art performance, not only does it require one network per agent (i.e. NoPS), it also stacks the experience of the agents leading to increased batch sizes. With SePS we forfeit the exploration benefits of SEAC but arrive at a method that may scale to hundreds of agents.

Scaling MARL to more Agents: Mean Field (Yang et al., 2018) tackles MARL with numerous agents by approximating interactions between a single agent and the average effect of the population. While it is shown that convergence is improved, Mean Field RL shares parameters in a fashion similar to FuPS. Our method operates as a pre-training step and attempts to find a network architecture configuration that improves learning. SePS can be combined with MARL algorithms (centralised critic, value decomposition, or others) since it improves a different part of the RL procedure.

6. Limitations and Future Work

Partitioning the agents using samples collected before agents are allowed to learn a policy does come with a disadvantage. In situations where agents share dynamics and reward functions ($\hat{\mathcal{P}}^i$ and $\hat{R}^i$) early in the policies' training but diverge later (e.g. agents are required to do the same task and then a different task in the same episode), learning the encoder-decoder with the initially collected samples may fail to properly partition agents. While in that case SePS will operate similarly to the full parameter sharing baselines like FuPS, it could be further improved by regularly retraining the encoder-decoder model with newer experience and redistributing agents to clusters if they have diverged.

A more complicated situation arises when agents have identical dynamics and rewards but are meant to take on different roles. The benefit of sharing parameters (or not) in such a case is highly dependent on the nature of the specific environment. While it may be possible that roles can be found and used to further partition agents if the SePS procedure is performed with trained policies (by recognising the difference in the sampling distributions), we leave such experiments and potential improvements to future work.

7. Conclusion

This paper explored existing methods for parameter sharing in MARL, identifying situations where they were ineffective. Our experiments suggested that sharing parameters indiscriminately between agents made learning harder since

agents interfered with the learning of others (Section 4.3). Therefore, we proposed a method for selective parameter sharing, that identified groups of agents that may benefit from sharing parameters. SePS was shown to successfully recognise heterogeneous agents and assign them to different parameter sets, allowing MARL training to scale to hundreds of agents even when they were not homogeneous. Our method was shown to outperform other parameter sharing baselines in converged returns, and a non parameter sharing baseline both in converged returns and training speed.

8. Funding Disclosure

References

Albrecht, S. V. and Ramamoorthy, S. A Game-Theoretic Model and Best-Response Learning Method for Ad Hoc Coordination in Multiagent Systems. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems*, AAMAS '13, pp. 1155–1156, Richland, SC, May 2013.

Albrecht, S. V. and Stone, P. Autonomous Agents Modelling Other Agents: A Comprehensive Survey and Open Problems. *Artificial Intelligence*, 258:66–95, 2018.

Christianos, F., Schäfer, L., and Albrecht, S. V. Shared Experience Actor-Critic for Multi-Agent Reinforcement Learning. In *Advances in Neural Information Processing Systems*, volume 33, pp. 10707–10717. Curran Associates, Inc., December 2020.

Davies, D. L. and Bouldin, D. W. A Cluster Separation Measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):224–227, April 1979.

Foerster, J. N., Farquhar, G., Afouras, T., Nardelli, N., and Whiteson, S. Counterfactual Multi-Agent Policy Gradients. In *Thirty-Second AAAI Conference on Artificial Intelligence*, April 2018.

Gupta, J. K., Egorov, M., and Kochenderfer, M. Cooperative Multi-agent Control Using Deep Reinforcement Learning. In *Autonomous Agents and Multiagent Systems*, Lecture Notes in Computer Science, pp. 66–83, Cham, 2017. Springer International Publishing.

Kaufman, L. and Rousseeuw, P. J. *Finding Groups in Data: An Introduction to Cluster Analysis*, volume 344. John Wiley & Sons, 2009.

Kingma, D. P. and Welling, M. Auto-Encoding Variational Bayes. In *International Conference on Learning Representations*, 2014.

Littman, M. L. Markov Games as a Framework for Multi-Agent Reinforcement Learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, 1994.

Lowe, R., Wu, Y., Tamar, A., Harb, J., Pieter Abbeel, O., and Mordatch, I. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. In *Advances in Neural Information Processing Systems*, volume 30, pp. 6379–6390, 2017.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous Methods for Deep Reinforcement Learning. In *International Conference on Machine Learning*, pp. 1928–1937. PMLR, June 2016.

Papoudakis, G., Christianos, F., Rahman, A., and Albrecht, S. V. Dealing with Non-Stationarity in Multi-Agent Deep Reinforcement Learning. *arXiv:1906.04737 [cs, stat]*, June 2019.

Papoudakis, G., Christianos, F., Schäfer, L., and Albrecht, S. V. Comparative Evaluation of Multi-Agent Deep Reinforcement Learning Algorithms. *arXiv:2006.07869 [cs, stat]*, June 2020.

Rangwala, M. and Williams, R. Learning Multi-Agent Communication through Structured Attentive Reasoning. In *Advances in Neural Information Processing Systems*, volume 33, pp. 10088–10098. Curran Associates, Inc., December 2020.

Rashid, T., Samvelyan, M., Schroeder, C., Farquhar, G., Foerster, J., and Whiteson, S. QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning. In *International Conference on Machine Learning*, pp. 4295–4304. PMLR, July 2018.

Rousseeuw, P. J. Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis. *Journal of Computational and Applied Mathematics*, 20:53 – 65, 1987.

Samvelyan, M., Rashid, T., De Witt, C. S., Farquhar, G., Nardelli, N., Rudner, T. G. J., Hung, C. M., Torr, P. H. S., Foerster, J., and Whiteson, S. The StarCraft Multi-Agent Challenge. In *International Joint Conference on Autonomous Agents and Multi-Agent Systems*, volume 4, pp. 2186–2188, 2019.

Shapley, L. S. Stochastic games. *Proceedings of the National Academy of Sciences*, 39(10):1095–1100, 1953.

Tan, M. Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents. In *International Conference on Machine Learning*, 1993.

Thorndike, R. L. Who Belongs in the Family? *Psychometrika*, 18(4):267–276, December 1953.

Wang, T., Dong, H., Lesser, V., and Zhang, C. ROMA: Multi-Agent Reinforcement Learning with Emergent Roles. In *International Conference on Machine Learning*, pp. 9876–9886. PMLR, November 2020.

Williams, R. J. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8(3–4):229–256, May 1992.

Yang, Y., Luo, R., Li, M., Zhou, M., Zhang, W., and Wang, J. Mean Field Multi-Agent Reinforcement Learning. In *International Conference on Machine Learning*, pp. 5571–5580. PMLR, July 2018.

Zhang, S. Q., Zhang, Q., and Lin, J. Succinct and Robust Multi-Agent Communication With Temporal Message Control. In *Advances in Neural Information Processing Systems*, volume 33, pp. 17271–17282. Curran Associates, Inc., 2020.