SHAC++: A Neural Network to Rule All Differentiable Simulators

Paper #6464

Abstract. Reinforcement learning (RL) algorithms show promise in robotics and multi-agent systems but often suffer from low sample efficiency. While methods like SHAC leverage differentiable simulators to improve efficiency, they are limited to specific settings: they require fully differentiable environments (including transition and reward functions) and have primarily been demonstrated in single-agent scenarios. To overcome these limitations, we introduce SHAC++, a novel framework inspired by SHAC. SHAC++ removes the need for differentiable simulator components by using neural networks to approximate the required gradients, training these networks alongside the standard policy and value networks. This allows the core SHAC approach to be applied in both non-differentiable and multi-agent environments. We evaluate SHAC++ on challenging multi-agent tasks from the VMAS suite, comparing it against SHAC (where applicable) and PPO, a standard algorithm for non-differentiable settings. Our results demonstrate that SHAC++ significantly outperforms PPO in both single- and multi-agent scenarios. Furthermore, in differentiable environments where SHAC operates, SHAC++ achieves comparable performance despite lacking direct access to simulator gradients, thus successfully extending SHAC's benefits to a broader class of problems.

1 Introduction

Learning control policies is crucial for applications like robotics [48], autonomous driving [17], and swarm intelligence [50]. These domains often present challenges such as sparse rewards, complex dynamics, and multi-agent interactions. Many current RL approaches focus specifically on either single- or multi-agent settings, lacking a unified solution. Furthermore, RL algorithms are often *sample inefficient*, requiring extensive interaction with the environment to learn effective policies.

A promising direction to improve sample efficiency involves utilizing *differentiable simulators*, namely simulators that allow for backpropagation through their components. SHAC [56] exemplifies this approach, using gradients obtained directly from the simulator to accelerate policy optimization. However, SHAC's applicability is constrained by two key limitations. First, it necessitates both a differentiable transition function *and* a differentiable reward function, which is often impractical, especially with sparse or complex reward structures. Second, while powerful, backpropagation through complex dynamics, like object collisions [29] or intricate multi-agent interactions, can lead to unstable gradients [9, 40], potentially hindering learning. The original SHAC work also focused primarily on single-agent tasks.

In this work, we propose SHAC++ to address these limitations and extend gradient-based policy optimization beyond SHAC's constraints. Our goal is to develop a sample-efficient reinforcement learning framework that maintains the benefits of gradient-based optimization without requiring differentiable simulators. Drawing inspiration from model-based RL [19, 20], SHAC++ substitutes direct simulator differentiation with neural approximations of the gradient functions. By training networks that approximate transition and reward gradients concurrently with policy optimization, our approach preserves the sample efficiency benefits of gradient information while eliminating strict differentiability requirements. This design enhances robustness to complex dynamics and enables application in both non-differentiable environments and multi-agent scenarios, albeit with additional computational overhead compared to SHAC.

We evaluate SHAC++ against two established baselines: PPO [45] (including its multi-agent variant MAPPO [18]) as the standard for non-differentiable environments, and the original SHAC [56] where applicable. Our experiments utilize the VMAS simulator [10], which provides physics-based multi-agent environments with varying levels of complexity. Through this comparative analysis on cooperative tasks with different agent populations, we investigate the following research questions:

- **RQ$_1$** Can we train a neural network to approximate the gradients of a differentiable simulator?
- **RQ$_2$** How does our algorithm compare to PPO and SHAC in both single-agent and multi-agent settings?
- **RQ$_3$** How does the performance of these algorithms change as the search space increases?

Our contributions include:

- a novel RL framework (SHAC++) that employs learned gradient approximations, eliminating the need for differentiable simulators;
- the first empirical evaluation of SHAC in multi-agent environments, establishing a baseline for differentiable MARL;
- a comprehensive comparison across environments with varying agent counts and differentiability properties;
- experimental evidence demonstrating that SHAC++ matches SHAC's performance in differentiable settings while substantially outperforming PPO in sample efficiency and final reward across both single-agent and complex multi-agent scenarios.

2 Background & Notation

2.1 Markov Decision Processes

To provide context for our work, we first introduce the reinforcement learning (RL) framework. Particularly, we focus on multi-agent settings, where multiple agents interact with each other and the environment. In doing so, each task is then modeled as a partially observable, decentralized, finite-horizon, Markov Decision Process (MDP) [46], $\mathcal{M}$: $(\mathcal{N}, \mathcal{S}, \mathcal{O}, \mathcal{A}, \pi, \mathcal{R}, F, H, \gamma, \mu)$. Where $\mathcal{N} = \{1, 2, \ldots, n\} = [n]$ and $\mathcal{S}$ denote the set of agents and the state

space, respectively. $\mathcal{O} = \{\mathcal{O}_i\}_{i\in\mathcal{N}}$ and $\mathcal{A} = \{\mathcal{A}_i\}_{i\in\mathcal{N}}$ are the observation space and action space for each agent. $\{P_i : \mathcal{S} \to \mathcal{O}_i\}_{i\in\mathcal{N}}$ are the projections from the state to the observation space and $\{R_i : \mathcal{S} \times \mathcal{A}_i \times \mathcal{S} \to \mathbb{R}\}_{i\in\mathcal{N}}$ are the reward functions for each agent. $F : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ denotes the transition function. γ and H denote the discount factor and the time horizon respectively. Finally, μ is the initial state distribution.

Further, let us denote the space of trajectories, $\mathcal{T} = (\mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathbb{R}^n)^*$, where each trajectory consists of a sequence of: starting state, action per agent, subsequent state, and reward per agent. Given a generic trajectory $\tau \in \mathcal{T}$, we denote its i-th starting state with s_i^τ, its i-th action for agent j with a_{ij}^τ, and its i-th reward for agent j with r_{ij}^τ.

A policy for agent i is a function $\pi_i : \mathcal{O}_i \to \mathcal{A}_i$ that maps its observation space to its action space. We will denote the collective policy, determined by the collection of agent-specific policies simply with $\pi = \{\pi_i\}_{i\in\mathcal{N}}$. We can use the policy to sample trajectories from the MDP by iteratively applying the transition, reward, projection and policy functions. In this case, for trajectory τ, we have that the i-th action for the j-th agent is $a_{ij}^\tau = \pi_j(P_j(s_i^\tau))$ and the next state is $s_{i+1}^\tau = F(s_i^\tau, a_{i1}, \ldots, a_{in})$. The i-th reward for the j-th agent is $r_{ij}^\tau = R_j(s_i^\tau, a_{ij}^\tau, s_{i+1}^\tau)$. Finally, s_0 is sampled from the initial state distribution μ. For brevity, we will simply denote with $\tau \sim \pi$ a trajectory sampled following the policies π.

The goal of the MDP is to find a policy that maximizes the expected return

$$\mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{H-1}\sum_{i\in\mathcal{N}}\gamma^t r_{ti}^\tau\right]$$

2.2 Neural Networks

A neural network is a function $f_\theta : \mathbb{R}^n \to \mathbb{R}^m$ parameterized by its weights $\theta \in \mathbb{R}^w$, which maps an input vector $x \in \mathbb{R}^n$ to an output vector $y \in \mathbb{R}^m$. The network parameters θ are optimized by minimizing the loss function $\mathcal{L}_\theta$, which quantifies the discrepancy between the network's predictions and the target outputs based on the training data.

In this work, we will often approximate functions (such as transition F) with neural networks. Therefore, for each function g, we will denote the corresponding neural network with g_θ. For instance, the transition and its corresponding neural network are denoted as F and F_θ, respectively. The corresponding loss will be denoted as $\mathcal{L}_\theta^g$ and it will aim to minimize the differences between g and g_θ.

While we use a unique set of parameters θ for all the networks. Each network can either use its own subset of θ or it can share parameters with other networks. In practice, the parameters will be shared only for the policies.

For simplicity, we formally consider only the case where policies have access to the full state, meaning that P_i is the identity function. This allows us to focus on the fundamental aspects of our approach without the added complexity of partial observability. Extending our framework to handle partial observability is straightforward; however, we choose not to address it here to maintain clarity and simplicity in our presentation. Nonetheless, our experiments do include scenarios where policies do not have access to the complete state information.

2.3 SHAC

In its trivial multi-agent extension, SHAC trains policies $\pi_{1\theta}, \ldots, \pi_{n\theta}$ by minimizing the loss function $\mathcal{L}_\theta^\pi : \mathcal{T}^N \to \mathbb{R}$ for a batch of N trajectories $B = \{\tau_1, \ldots, \tau_N\}$ sampled using π. The loss $\mathcal{L}_\theta^\pi(B)$ is defined as:

$$\frac{1}{Nnh}\sum_{\tau\in B}\sum_{t=t_0}^{t_0+h-1}\sum_{i\in\mathcal{N}}\gamma^{t-t_0}R(s_t^\tau, a_{ti}^\tau) + \gamma^h V_\theta(s_{t_0+h}^\tau)$$

where $h \ll H$ is the rollout size (namely the number of steps used to estimate the return), t_0 is the initial time step of the rollout, $V_\theta : \mathcal{S} \to \mathbb{R}$ is the value function estimating the expected return. It is apparent that SHAC assumes that the policy, reward, value, projection, and transition functions are all differentiable, i.e., $\{\pi_i\}_{i\in\mathcal{N}}, \{R_i\}_{i\in\mathcal{N}}, V_\theta$, and $F \in \mathcal{C}^1$. While this is a common requirement for policy and value functions, differentiable transition and reward functions are rarely available.

The value function V_θ is trained to approximate a temporal difference-λ formulation [49] V, which in SHAC translates to minimize the loss $\mathcal{L}_\theta^V : \mathcal{T}^N \to \mathbb{R}$.

$$\mathcal{L}_\theta^V(B) = \frac{1}{Nh}\sum_{\tau\in B}\sum_{t=0}^{h}||V_\theta(s_t^\tau) - V(s_t^\tau)||^2$$

$$V(s_t) = (1-\lambda)\left(\sum_{k=1}^{h-t-1}\lambda^{k-1}G_t^k\right) + \lambda^{h-t-1}G_t^{h-t}$$

$$G_t^k = \left(\sum_{l=t}^{k-1+t}\sum_{i\in\mathcal{N}}\gamma^{l-t}r_{li}\right) + \gamma^k V(s_{t+k})$$

G_k^t represents the discounted sum of rewards up to step k, plus the bootstrapped value of the state at step $t+k$. The estimates V is computed by discounting longer (higher k) discounted sums, G_t^k.

Finally, Fig. 1a presents an unrolled view of the SHAC algorithm for a single-agent scenario.

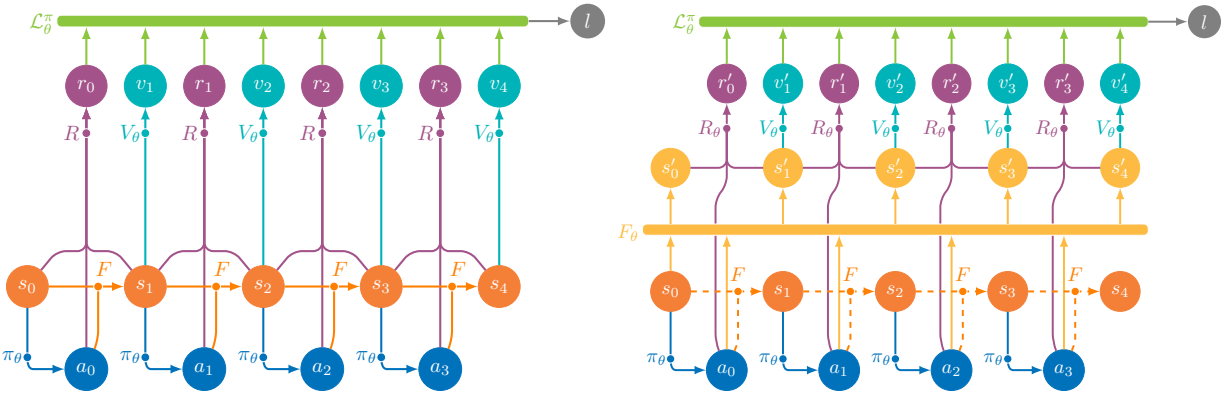
Algorithm 1 SHAC++ minimal (no cache and no cool-down) pseudocode. $s_{:,0}$ denotes the first step of each trajectory in s. a and r denote the actions and rewards for all agents.

Initialize $\pi_\theta, V_\theta, R_\theta, F_\theta$ networks
Initialize learning rates $\eta_\pi, \eta_R, \eta_V, \eta_F$
Initialize environment $\mathcal{E}$
Initialize γ, α
for episode in $1, \ldots, N_{\text{episodes}}$ **do**
  $s, a, r, v \sim \mathcal{E}(\pi)$
  $\hat{s} = F_\theta(s_{:,0}, a)$
  $\hat{r} = R_\theta(\hat{s})$
  $\hat{v} = V_\theta(\hat{s})$
  $\theta \leftarrow \theta - \eta_\pi \nabla \mathcal{L}'^\pi_\theta(\hat{r}, \hat{v}, \gamma, \alpha)$
  $\theta \leftarrow \theta - \eta_V \nabla \mathcal{L}_\theta^V(\hat{v}, s, v)$
  $\theta \leftarrow \theta - \eta_R \nabla \mathcal{L}_\theta^R(\hat{r}, s, a, r)$
  $\theta \leftarrow \theta - \eta_F \nabla \mathcal{L}_\theta^F(\hat{s}, s, a)$
end for

3 SHAC++

As mentioned previously, we aim to address the limitations of SHAC by approximating the gradients of a differentiable environment (R and F) with neural networks (R_θ and F_θ). In particular, we aim to train a neural network to approximate the reward function by minimizing:

$$\mathop{E}_{\tau\sim\pi}\left[\sum_{s_i, a_{ij}, s_{i+1}\in\tau}\left(R_{i\theta}(s_i, a_{ij}, s_{i+1}) - R_i(s_i, a_{ij}, s_{i+1})\right)^2\right]$$



(a) SHAC (b) SHAC++

Figure 1: Comparison between SHAC and SHAC++ during a rollout of 4 steps. Nodes with labels (such as r_i or r'_i) indicate values, while nodes without labels (such as π or R) represent functions. Dashed lines denote non-differentiability. a_i and r_i refer to the collection of actions and rewards for all agents. Note that the main advantage of SHAC++ is that the transition function F on the right can be non-differentiable, unlike the left F in SHAC, which must be differentiable.

In practice, this is achieved by leveraging a *replay buffer* to cache the state-action-reward triplets obtained from the simulator. The neural network is then trained to minimize the mean squared error (MSE) loss. Specifically, we minimize the loss $\mathcal{L}_\theta^{R_i} : (\mathcal{S} \times \mathcal{A}_i \times \mathcal{S})^N \to \mathbb{R}$:

$$\mathcal{L}_\theta^{R_i}(B) = \frac{1}{N} \sum_{s,a,s',r \in B} (R_{i\theta}(s,a) - r)^2$$

Where B represents a batch of state-action-state-reward tuples sampled uniformly from the cache.

Similarly, we aim to train a neural network to approximate the transition function. However, to avoid vanishing or exploding gradients, we employ a non-recursive approach similar to Action World Model [39]. This corresponds to train a neural network that given a starting observation from $\mathcal{S}$ and a sequence of h actions from $\mathcal{A}^h$, predicts the next h states from $\mathcal{S}^h$. We aim to minimize:

$$\mathop{E}_{\tau \sim \pi} \left[\sum_{\substack{s_t,\dots,s_{t+h+1}, \\ a_{t*},\dots,a_{t+h*} \in \tau}} ||F_\theta(s_t, a_{t*}, \dots, a_{i+h*}) - [s_t, \dots, s_{t+h+1}]||^2 \right]$$

Where $a_{t*} = \{a_{ti}\}_{i \in \mathcal{N}}$. In practice, this is achieved by caching trajectories and then training the neural network to minimize the MSE loss. That is, we minimize the loss $\mathcal{L}_\theta^F : \mathcal{T}^N \to \mathbb{R}$:

$$\frac{1}{Nh} \sum_{\tau \in B} ||F_\theta(s_0^\tau, a_{1*}^\tau, \dots, a_{h*}) - [s_1^\tau, \dots, s_h^\tau]||^2$$

Where B represents a batch of trajectories of length h sampled uniformly from the cache.

Further, we modify the SHAC policy loss, $\mathcal{L}_\theta^\pi$, to include a term that penalizes actions outside the action space. Suppose the action space is (l, r) (with $l < r$) for all agents, we minimize the loss $\mathcal{L'}_\theta^\pi$:

$$\mathcal{L'}_\theta^\pi(B) = \mathcal{L}_\theta^\pi(B) + \alpha \frac{1}{NH} \sum_{\tau \in B} \sum_{a \in \tau} d(a^\tau, (l, r))$$

$$d(x, (l, r)) = \begin{cases} (x - l)^2 & \text{if } x < l \\ (x - r)^2 & \text{if } x > r \\ 0 & \text{otherwise} \end{cases}$$

Where α is a hyper-parameter that controls the strength of the penalty. In our experiments, we set $\alpha = 1$. Usually, actions are either clipped to the allowed action space or, an activation function (such as hyperbolic tangent or sigmoid) is used to ensure that the actions are within the allowed range. While this approach is acceptable for algorithms like PPO and SHAC, the penalty we employ is crucial in our specific scenario. This necessity arises from the fact that we are training the policy to optimize two neural networks that serve as environment approximations. Because these networks are never trained with data outside the permissible action space, they are likely to be maximized beyond these bounds. Consequently, the policy may generate actions far outside the allowed range. This can ultimately result in 0-gradients when action clipping is applied or high pre-activation actions when activation function is used.

To summarize, our algorithm, aims to minimize the 4 presented losses, $\mathcal{L'}_\theta^\pi$, $\mathcal{L}_\theta^V$, $\mathcal{L}_\theta^R$, and $\mathcal{L}_\theta^F$. A minimal pseudocode of our algorithm is presented in Algorithm 1. Fig. 1b presents an unrolled view of the SHAC++ algorithm.

However, consider that, to ensure fast convergence of the reward and transition networks, we employ a cache to store seen trajectories. To ensure that the cache is balanced, and not filled with 0-reward trajectories (which are common in case of sparse rewards), we divide the cache in K equally sized bins. Each bin is filled with trajectories that have a similar reward.

Additionally, to minimize the time spent between swapping among the different training procedures (policy, value, reward, transition), we apply a cool-down of $K^V/K^R/K^F$ episodes between running training procedure for the value/reward/transition networks.

The full implementation is openly available at https://anonymous.4open.science/r/diffagents-8653[1].

4 Experiments

In this section, we evaluate SHAC++ against both PPO and SHAC across various multi-agent scenarios and architectures. An ablation study replacing only the transition function (SHAC+, where the reward function remains differentiable) is provided in Section D. We select PPO as our baseline because, despite the emergence of more recent algorithms HAPPO [36], and SAC [33], it remains one of the few approaches that scales effectively to multi-agent settings without substantial modifications (e.g., MAPPO [18]). This makes PPO

[1] The link of the original repository will be provided upon acceptance.

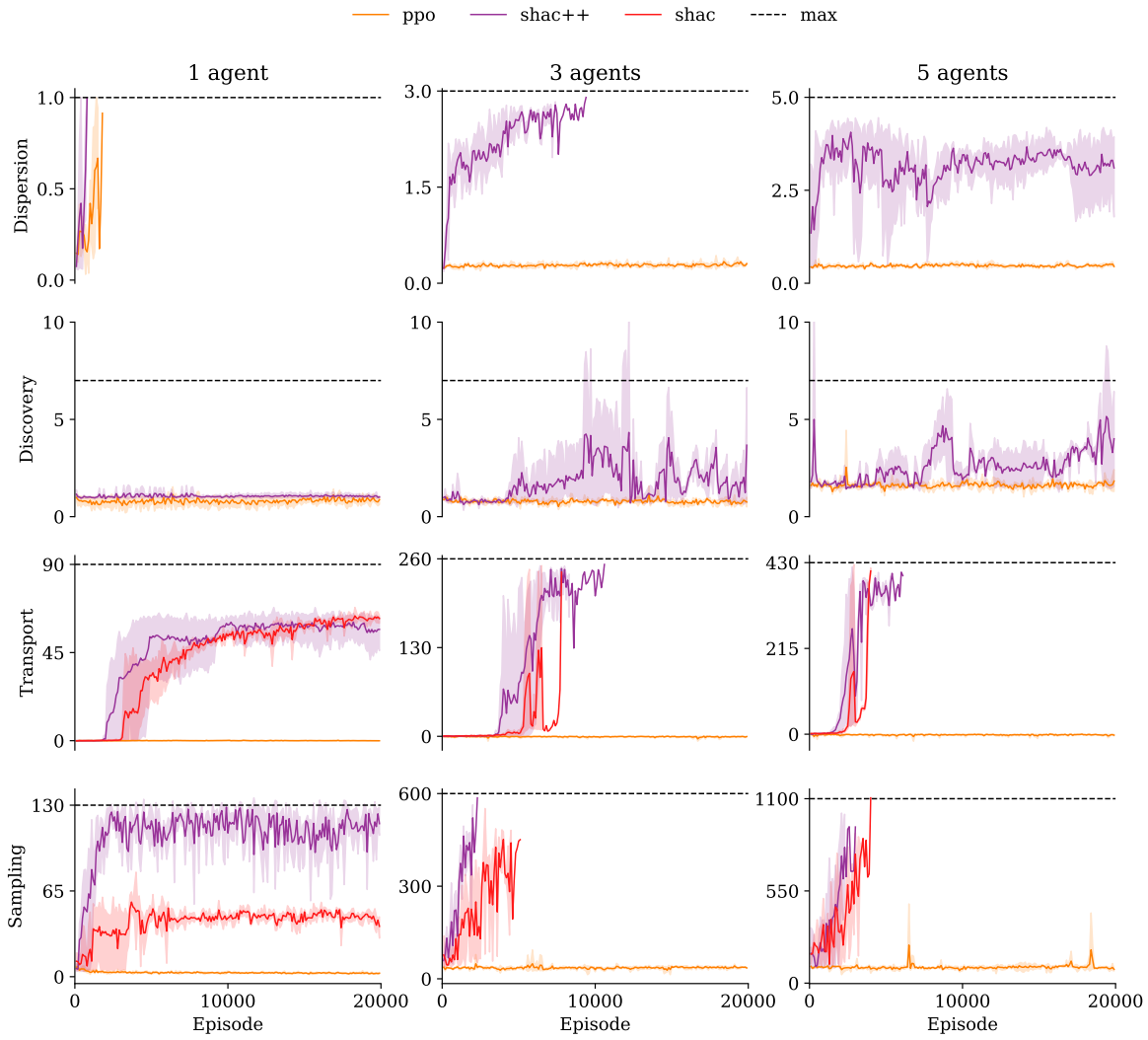


Figure 2: Performance comparison of SHAC++, PPO, and SHAC across different scenarios (Dispersion, Transport, Discovery, and Sampling) using the Transformer architecture for multi-agent settings and the MLP architecture for single-agent settings. The results show the mean and standard deviation of rewards over 3 runs. For results using the MLP architecture in all scenarios, refer to Fig. 7.

a representative benchmark for state-of-the-art performance in non-differentiable environments.

Scenarios. We evaluate our algorithms on several multi-agent scenarios from VMAS [10], which we selected over alternatives like PettingZoo [52], SMAC [44], MA-MuJoCo [31], and others due to its key advantages: differentiable physics (enabling gradient-based optimization), multi-agent design, vectorized implementation (for efficient parallel training), realistic physics-based simulation, and open-source accessibility. For more details about the choice of environments, see Section A. We selected four distinct scenarios: Dispersion, Discovery, Transport, and Sampling, specifically for their cooperative nature and varying levels of required coordination (see Fig. 5 and Section A for details). These scenarios can be categorized along two dimensions:

- **Reward differentiability:** Dispersion and Discovery feature non-differentiable reward functions (making SHAC inapplicable), while Transport and Sampling have differentiable rewards (allowing for SHAC implementation).
- **Observation completeness:** Transport and Dispersion provide agents with complete observation spaces relative to the environment state. In contrast, Discovery and Sampling have incomplete observation spaces, which presents an additional challenge for SHAC++ compared to SHAC, as SHAC++ lacks access to the complete environmental state.

This selection of scenarios enables a thorough evaluation of how each algorithm handles different types of multi-agent cooperation challenges.

Architectures. We test SHAC, PPO, and SHAC++ with both an MLP and a Transformer architecture. Specifically, we use a 1-layer MLP or a 1-layer single-head Transformer for policy, reward, and value network. The transition network used by SHAC++ is always a 3-layer single-head Transformer

While the MLP architecture is the simplest, the Transformer baseline benefits from positional invariance property of the agents observations. We apply the transformer architecture only if the number of agents is greater than 1, otherwise we use only the MLP architecture. For more details on the architectures, see Section B.

Hyperparameters. For all networks we use learning rate of $1e$-3 with Adam Optimizer [35]. For all scenarios, each training episode consists of 512 environments of 32 steps each. Each vali-

Environment	agents	PPO	SHAC	SHAC+ MLP	SHAC++	PPO	SHAC	SHAC+ Transformer	SHAC++
dispersion	1	**1.00**	-	0.99	**1.00**	-	-	-	-
	3	0.22	-	0.64	0.99	0.15	-	1.00	**1.00**
	5	0.16	-	0.85	**1.00**	0.14	-	0.93	0.93
transport	1	0.01	**1.00**	0.89	**1.00**	-	-	-	-
	3	0.01	0.84	0.95	0.41	0.00	0.95	**1.00**	0.95
	5	0.01	0.14	0.95	0.45	0.00	**1.00**	0.99	0.99
discovery	1	0.33	-	**1.00**	0.35	-	-	-	-
	3	0.03	-	0.08	0.08	0.15	-	**1.00**	0.95
	5	0.15	-	0.14	0.14	0.39	-	0.61	**1.00**
sampling	1	0.07	0.58	0.59	**1.00**	-	-	-	-
	3	0.09	0.86	0.93	0.92	0.16	0.94	1.00	**1.00**
	5	0.12	0.83	0.92	0.86	0.43	**1.00**	0.89	0.92

Table 1: Normalized rewards (relative to the best performing model) for the different scenarios. Best results are in bold.

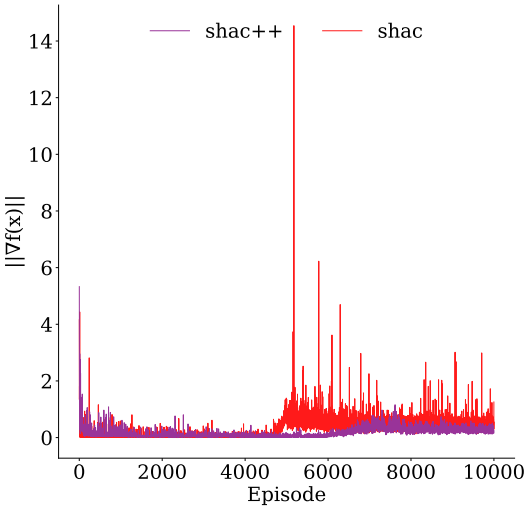


Figure 3: Policy gradient norms in the Transport environment over 5000 epochs. Spikes in gradient norm for SHAC (epoch 2700) and SHAC++ (epoch 3200) correspond to generalization events when agents begin frequent interactions with the package and each other.

Figure 4: Snapshots of trained agents with SHAC++ in the Transport scenario. Colored circles represent agents pushing a red package (square) toward the green goal.

dation episode is composed of 512 environments of 512 steps each. We employ early stopping when the agents achieve 90% of the maximum reward in 90% of the episode's environments. If early stopping is not triggered, we stop training after 20,000 episodes. The discount factor is set to 0.99 and lambda factor is set to 0.95. We report results for increasing number of agents $n \in \{1, 3, 5\}$. We repeat and report results for 3 runs with different seeds. See Table 4 and Table 3 for a complete list of hyperparameters.

Overall, considering different training algorithms, architectures, scenarios, hyperparameters, we completed a total of 254 runs lasting between 1 to 8 hours each.

Hardware Setup. Our experiments were conducted on a cluster of machines composed of 2 computational nodes, each with one NVIDIA Tesla V100 GPU (with 32GB of memory), 200GB of RAM, and two Intel Xeon Gold 6226R processors (32 cores each).

4.1 Results

In this section, we compare the performance of SHAC++, PPO, and SHAC across different scenarios using the Transformer architecture for multi-agent settings and the MLP architecture for single-agent settings. The results show the mean and standard deviation of achieved rewards over 3 runs. An overview of the results is shown in Fig. 2 and Table 1. For results using the MLP architecture in all scenarios, refer to Fig. 7.

Dispersion. In Fig. 2, the first row displays results for the dispersion scenario (non-differentiable, thus SHAC is not applicable). In the first column (single agent, see also Table 1 and Fig. 7), both PPO and SHAC++ perform well, though SHAC++ converges faster thanks to gradient approximation. In the second column (3 agents), PPO fails to converge, whereas SHAC++ converges in fewer than 10,000 episodes, suggesting emergent cooperative behavior. In the third column (5 agents), PPO fails to converge, while SHAC++ achieves high rewards (with no early stopping) but shows high variance due to increased environment complexity.

Discovery. The second row of Fig. 2 shows a scenario with partial observability, namely discovery. In the first column (single agent), SHAC++ outperforms PPO, though it does not reach the maximum reward, suggesting the world model struggles to capture dynamics. In the second and third columns (3 and 5 agents), PPO again fails to converge, whereas SHAC++ maintains superior results, indicating cooperative behavior.

Transport. The third row of Fig. 2 presents a differentiable scenario, transport. In the first column (single agent), PPO fails to converge, while SHAC and SHAC++ achieve comparable performance. SHAC++ converges faster, but SHAC is more consistent. In the second and third columns (3 and 5 agents), PPO fails as in the single-agent case, while SHAC and SHAC++ both complete the task around 10,000 episodes. With more agents, convergence is faster because the task becomes easier (as more agents can push the package). The emergent of cooperation (see Fig. 4) can be also display but looking to the gradient norm in Fig. 3 we can see that the norm of the gradients for SHAC and SHAC++ increases when the agents start to collide with each other and the package. This is a sign of generalization, as the agents learn to avoid collisions and push the package together.

Sampling. The fourth row of Fig. 2 illustrates the sampling sce-

nario, which is differentiable and thus applicable to SHAC. In the first column (single agent), PPO fails to converge, while SHAC++ outperforms SHAC. In the second and third columns (3 and 5 agents), PPO again fails to converge, while SHAC and SHAC++ both complete training in fewer than 5,000 episodes. Notably, SHAC does not exhibit the same stability advantage it displayed in other scenarios, possibly because gradients derived directly from the environment are noisier compared to those learned by the transition network.

5 Discussion

In the following, we discuss the results obtained in Section 4 and provide some insights into the performance of SHAC++ compared to SHAC and PPO, addressing the research questions posed in Section 1. Note that SHAC++ does not aim to outperform SHAC, but rather to perform comparably while being applicable even in non-differentiable environments.

5.1 Gradient Approximation with Neural Networks

Research Question

Can we train a neural network to approximate the gradients of a differentiable simulator?

The proposed framework, SHAC++, appears to be successful in emulating the behavior of SHAC in differentiable environments such as transport and sampling. Therefore, we can conclude that it is definitely possible to approximate the gradients of a differentiable environment with a neural network. However, we note that this approach, while obtaining comparable results in most setups, tends to display higher variance. Both neural networks representing the reward and transition functions require sufficient training before becoming effective, which may entail multiple episodes. Interestingly, SHAC++ outperforms SHAC in the Sampling scenario. A possible explanation is that the complex dynamics of the environment may lead to unstable gradients.

Reply Summary

Yes, SHAC++ approximates simulator gradients, achieving comparable performance to SHAC in differentiable settings, occasionally outperforming it despite initially higher variance.

5.2 Comparison to PPO and SHAC

Research Question

How does our algorithm compare to PPO and SHAC in both single-agent and multi-agent settings?

The proposed algorithm appears capable of outperforming PPO across all scenarios, also showing better sample efficiency.

Single-agent:. In single-agent cases, while PPO manages to find near-good policies (e.g., in the Dispersion scenario), it often fails to complete scenarios within the same number of steps. Indeed, SHAC++ achieves better results in all environments (see Table 1). Compared to SHAC (where applicable), SHAC++ performs comparably.

Multi-agent:. This pattern becomes even more evident in multi-agent scenarios. In these cases, PPO often struggles to find even marginally good policies. SHAC++, on the other hand, eventually succeeds in understanding the environment and subsequently improves the policy to enable cooperation—see both Table 1 and Fig. 2. Its performance remains comparable to SHAC where SHAC can be applied. Overall, despite having to train multiple networks, SHAC++ converges much faster to optimal policies than PPO, as gradient guidance enables quicker and more accurate convergence to the desired outcome.

Reply Summary

SHAC++ outperforms PPO across all scenarios, with PPO struggling especially in multi-agent settings. When compared to SHAC (where applicable), SHAC++ achieves comparable performance while enabling emergent cooperative behaviors.

5.3 Scaling with Search Space

Research Question

How does the performance of these algorithms change as the search space increases?

To analyze the impact of increasing search spaces, we primarily focused on scenarios with varying numbers of agents (up to 5 in the main experiments). The results demonstrate that performance patterns differ across scenarios. In the Dispersion scenario, SHAC++ shows increasing difficulty in finding optimal policies as the number of agents grows. Conversely, in Transport and Sampling scenarios, the system appears to converge more quickly with more agents. This counterintuitive behavior likely stems from the fact that, despite larger search spaces, the fundamental tasks in Sampling and Transport become relatively easier to solve with multiple agents, making policy discovery more straightforward. Even with approximated world dynamics (transition and reward functions), the guidance provided is sufficient to lead policies toward optimal behavior.

This scaling advantage is not observed in the Dispersion scenario, where complexity increases quadratically with agent count. Nevertheless, we observe consistent emergence of cooperative behavior patterns across all experimental scenarios, suggesting potential scalability to larger agent populations (quantitative analysis with up to 20 agents is provided in Section C). The primary computational constraint lies in the transformer architecture employed for the action world model, where sequence length scales linearly with the number of agents (specifically, sequence length = $n \times s$, where n is the number of agents and s is the number of training steps). With agent populations exceeding approximately 50 agents, the self-attention mechanism's $O(n^2)$ complexity leads to prohibitive memory requirements. For such large-scale multi-agent systems, linear attention mechanisms (e.g., [8]) represent a viable architectural alternative.

Reply Summary

Performance varies with scale: Transport/Sampling converge faster with more agents, while Dispersion becomes harder. Cooperation emerges across scenarios. Main scalability limitation is transformer attention complexity, becoming prohibitive at large agent counts (50), though moderate populations remain feasible.

6 Related Work

Differentiable Engines. Game engines [7, 6] and Physics engines [53, 15] have been widely used in the field of policy learning. Recently, there has been a trend towards differentiable engines, which allows for the use of gradient-based optimization methods, such as backpropagation. Notable examples are [28, 51, 4]. These frameworks may be built on top of auto differentiation libraries such [11, 3] (this is the case of VMAS) or directly with analytical gradients [12, 55].

Further, several works tackle the challenge of the non-smoothness of the contact dynamics [16, 43]

Deep Reinforcement Learning. Deep reinforcement learning has revolutionized policy learning tasks by combining reinforcement learning with deep neural networks. Foundational works like DQN [41] (and its variants [23]) and AlphaZero [47] demonstrated human-level performance in complex environments such as Atari games and chess. Over the years, several algorithms have been proposed to create more stable and efficient learning methods. Notable examples include A3C for asynchronous learning [26], DDPG and PPO for continuous action spaces [25, 45] (also used in modern chatbot like ChatGPT), and SAC for continuous action spaces with off-policy learning [32]. Following the initial era of model-free algorithms, model-based algorithms have been introduced to enhance sample efficiency. Early works such as Dreamer [20] and PlaNet [19] have demonstrated promising results in terms of both sample efficiency and performance.

Multi-Agent Reinforcement Learning. Multi-Agent Reinforcement Learning (MARL) is a subfield of reinforcement learning that focuses on learning policies for multiple agents that interact with each other [1]. This field has a wide range of applications, including robotics [14], traffic control [13], and game playing [5, 34], and involves different types of agents such as cooperative, competitive, and mixed. In this paper, we primarily focus on cooperative multi-agent scenarios, where agents need to collaborate to achieve a common goal. Prominent works in this area include extensions of single-agent algorithms to multi-agent settings, such as MADDPG [37] and MAPPO [18], as well as novel solutions to unique challenges in multi-agent settings. Examples of these solutions are COMA [21] for multi-agent credit assignment, QMIX [24] for coordination in cooperative multi-agent settings, and Mean Field MARL [27] for handling a large number of agents.

Reinforcement Learning on Differentiable Engines. Precursors of SHAC with similar intuition are PODS [42] and CE-APG [30]. In [38], the authors introduce rendering to enhance model-based RL. In [57] authors employ a differentiable symbolic expression search engine. Trajectory optimization approach is taken in [54]. In [29] authors tackle unstable gradient due to stiff dynamics by adaptively truncating trajectories when these occur. With respect to this approach, we take a substantially different method, by emulating the differentiable environment with a trained neural network.

7 Limitations

Environments often exhibit vastly different characteristics and challenges, and the proposed method is not a universal solution suitable for all scenarios. One of the most difficult components to learn is the world model, which relies on meaningful interactions between agents and the environment—interactions that do not necessarily yield immediate rewards. This creates a cold-start problem: the world model cannot be learned until agents are able to interact effectively with the environment. To address this, as in prior works, we use stochastic action sampling instead of a deterministic policy. In the environments considered, this approach eventually leads to the successful learning of a world model, which in turn allows agents to develop a viable policy. However, this strategy may not be effective in more complex environments where random actions fail to produce meaningful progress. In such cases, the proposed method may not succeed in learning a policy.

This limitation can be mitigated by introducing a cold-start dataset—either manually curated or generated using alternative algorithms such as PPO or MPPI [2]. Such data can help initialize the world model and provide a useful foundation for the learning process.

8 Conclusions & Future Works

In this paper, we proposed SHAC++, an extension of SHAC, to provide a reinforcement learning solution that is both sample efficient, able to work in both single-agent and multi-agent scenarios, and capable of handling non-differentiable scenarios. We demonstrated that this solution maintains alignment with SHAC ($\mathbf{RQ_1}$), outperforms the current state-of-the-art PPO in both single-agent and multi-agent scenarios ($\mathbf{RQ_2}$), and exhibits promising scaling capabilities in multi-agent settings with emergent collaborative behaviors ($\mathbf{RQ_3}$).

Despite these encouraging results, several directions for future work remain. First, we aim to explore scalability beyond the current limitation of tens of agents. Additionally, as evidenced in the discovery scenario, the current approach struggles when agents have highly partial observations of the environment. Furthermore, we plan to evaluate our approach in other multi-agent environments, such as those presented from the cooperative AI community [22] and also in adversarial settings.

Acknowledgements

Redacted for anonimity

References

[1] Stefano V Albrecht, Filippos Christianos, and Lukas Schäfer. *Multi-agent reinforcement learning: Foundations and modern approaches*. MIT Press, 2024.

[2] Juan Alvarez-Padilla, John Z Zhang, Sofia Kwok, John M Dolan, and Zachary Manchester. Real-time whole-body control of legged robots with model-predictive path integral control. *arXiv preprint arXiv:2409.10469*, 2024.

[3] Jason Ansel and et al. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, April 2024.

[4] Genesis Authors. Genesis: A universal and generative physics engine for robotics and beyond, December 2024.

[5] Bowen Baker, Ingmar Kanitscheider, Todor M. Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[6] Martin Balla, George EM Long, Dominik Jeurissen, James Goodman, Raluca D Gaina, and Diego Perez-Liebana. Pytag: Challenges and opportunities for reinforcement learning in tabletop games. 2023.

[7] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[8] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[9] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[10] Matteo Bettini, Ryan Kortvelesy, Jan Blumenkamp, and Amanda Prorok. VMAS: A vectorized multi-agent simulator for collective robot learning. In *Distributed Autonomous Robotic Systems - 16th International Symposium, DARS 2022*, volume 28 of *Springer Proceedings in Advanced Robotics*, pages 42–56. Springer, 2022.

[11] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[12] Justin Carpentier and Nicolas Mansard. Analytical derivatives of rigid body dynamics algorithms. In *Robotics: Science and systems (RSS 2018)*, 2018.

[13] Tianshu Chu, Jie Wang, Lara Codecà, and Zhaojian Li. Multi-agent deep reinforcement learning for large-scale traffic signal control. *IEEE Trans. Intell. Transp. Syst.*, 21(3):1086–1095, 2020.

[14] Jaehoon Chung, Jamil Fayyad, Younes Al Younes, and Homayoun Najjaran. Learning team-based navigation: a review of deep reinforcement learning techniques for multi-agent pathfinding. *Artif. Intell. Rev.*, 57(2):41, 2024.

[15] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning, 2016.

[16] Jonas Degrave, Michiel Hermans, Joni Dambre, and Francis Wyffels. A differentiable physics engine for deep learning in robotics. *Frontiers in neurorobotics*, 13:6, 2019.

[17] Badr Ben Elallid, Nabil Benamar, Abdelhakim Senhaji Hafid, Tajjeeddine Rachidi, and Nabil Mrani. A comprehensive survey on the application of deep and reinforcement learning approaches in autonomous driving. *Journal of King Saud University-Computer and Information Sciences*, 34(9):7366–7390, 2022.

[18] Chao Yu et al. The surprising effectiveness of PPO in cooperative multi-agent games. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[19] Danijar Hafner et al. Learning latent dynamics for planning from pixels. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, volume 97 of *Proceedings of Machine Learning Research*, pages 2555–2565. PMLR, 2019.

[20] Danijar Hafner et al. Dream to control: Learning behaviors by latent imagination. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[21] Jakob N. Foerster et al. Counterfactual multi-agent policy gradients. In *32nd AAAI Conference on Artificial Intelligence (AAAI)*, pages 2974–2982, 2018.

[22] John P. Agapiou et al. Melting pot 2.0. *CoRR*, abs/2211.13746, 2022.

[23] Matteo Hessel et al. Rainbow: Combining improvements in deep reinforcement learning. In *Proc. 32nd AAAI Conf. Artificial Intelligence (AAAI)*, pages 3215–3222. AAAI Press, 2018.

[24] Tabish Rashid et al. QMIX: monotonic value function factorisation for deep multi-agent reinforcement learning. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 4292–4301. PMLR, 2018.

[25] Timothy P. Lillicrap et al. Continuous control with deep reinforcement learning. In *4th International Conference on Learning Representations (ICLR)*, 2016.

[26] Volodymyr Mnih et al. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, volume 48, pages 1928–1937, 2016.

[27] Yaodong Yang et al. Mean field multi-agent reinforcement learning. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 5567–5576. PMLR, 2018.

[28] C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - a differentiable physics engine for large scale rigid body simulation, 2021.

[29] Ignat Georgiev, Krishnan Srinivasan, Jie Xu, Eric Heiden, and Animesh Garg. Adaptive horizon actor-critic for policy learning in contact-rich differentiable simulation. In *Forty-first International Conference on Machine Learning*, 2024.

[30] Sean Gillen and Katie Byl. Leveraging reward gradients for reinforcement learning in differentiable physics simulations. *arXiv preprint arXiv:2203.02857*, 2022.

[31] Abhishek Gupta, Rohit Kumar, Maxim Egorov, and Sergey Levine. Multi-agent reinforcement learning in mixed cooperative-competitive environments. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*, pages 11093–11100, 2021.

[32] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1856–1865. PMLR, 2018.

[33] Tuomas et al. Haarnoja. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[34] Max et al. Jaderberg. Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865, 2019.

[35] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[36] Jakub Kuba, Zihan Wang, Kaiqing Zhang, and Tamer Basar. Trust region policy optimization for multi-agent reinforcement learning. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, 2021.

[37] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, pages 6379–6390, 2017.

[38] Jun Lv, Yunhai Feng, Cheng Zhang, Shuang Zhao, Lin Shao, and Cewu Lu. Sam-rl: Sensing-aware model-based reinforcement learning via differentiable physics-based simulation and rendering. *The International Journal of Robotics Research*, page 02783649241284653, 2023.

[39] Michel Ma, Tianwei Ni, Clement Gehring, Pierluca D'Oro, and Pierre-Luc Bacon. Do transformer world models give better policy gradients? In *Forty-first International Conference on Machine Learning*, 2024.

[40] Luke Metz, C Daniel Freeman, Samuel S Schoenholz, and Tal Kachman. Gradients are not all you need. *arXiv preprint arXiv:2111.05803*, 2021.

[41] Volodymyr Mnih. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[42] Miguel Angel Zamora Mora, Momchil Peychev, Sehoon Ha, Martin Vechev, and Stelian Coros. Pods: Policy optimization via differentiable simulation. In *International Conference on Machine Learning*, pages 7805–7817. PMLR, 2021.

[43] Geilinger Moritz, Hahn David, Z Yonas, Bächer Moritz, T Benrhnard, and Coros Stelian. Analytically differentiable dynamics for multibody systems with friction contact. *ACM Transactions on Graphics*, 39(6), 2020.

[44] Mikayel et al. Samvelyan. The starcraft multi-agent challenge. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 2186–2188, 2019.

[45] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[46] Lloyd S Shapley. Stochastic games. *Proceedings of the national academy of sciences*, 39(10):1095–1100, 1953.

[47] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[48] Bharat Singh, Rajesh Kumar, and Vinay Pratap Singh. Reinforcement learning in robotic applications: a comprehensive survey. *Artificial Intelligence Review*, 55(2):945–990, 2022.

[49] Richard S Sutton, Andrew G Barto, et al. Introduction to reinforcement learning. vol. 135, 1998.

[50] Jun Tang, Gang Liu, and Qingtao Pan. A review on representative swarm intelligence algorithms for solving optimization problems: Applications and trends. *IEEE/CAA Journal of Automatica Sinica*, 8(10):1627–1643, 2021.

[51] A. Howell Taylor, Simon Le Cleac'h, Zico Kolter, Mac Schwager, and Zachary Manchester. Dojo: A differentiable simulator for robotics. *arXiv preprint arXiv:2203.00806*, 2022.

[52] J. K. et al. Terry. Pettingzoo: Gym for multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 34, pages 15032–15043, 2021.

[53] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE, 2012.

[54] Weikang et al. Wan. Difftop: Differentiable trajectory optimization for deep reinforcement and imitation learning. *arXiv preprint arXiv:2402.05421*, 2024.

[55] Keenon Werling, Dalton Omens, Jeongseok Lee, Ioannis Exarchos, and C Karen Liu. Fast and feature-complete differentiable physics engine for articulated rigid bodies with contact constraints. In *Robotics: Science and Systems*, 2021.

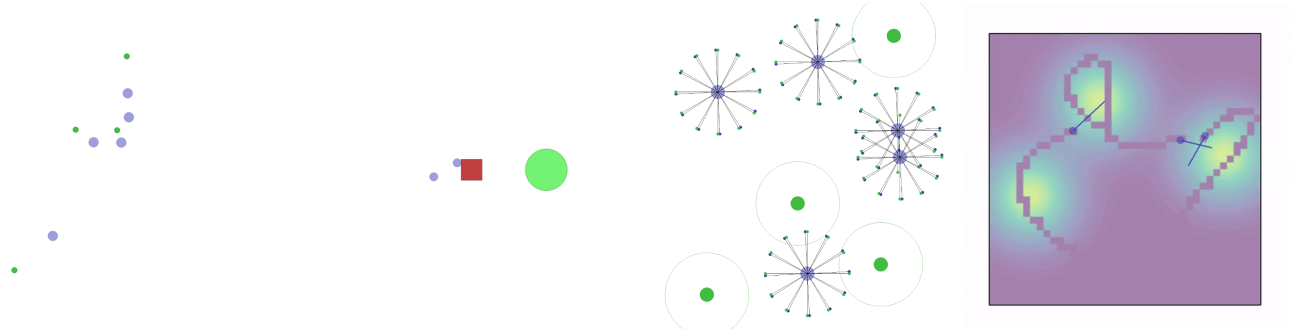
[56] Jie Xu, Miles Macklin, Viktor Makoviychuk, Yashraj Narang, Animesh Garg, Fabio Ramos, and Wojciech Matusik. Accelerated policy learning with parallel differentiable simulation. In *International Conference on Learning Representations*, 2022.

[57] Wenqing Zheng, SP Sharan, Zhiwen Fan, Kevin Wang, Yihan Xi, and Zhangyang Wang. Symbolic visual reinforcement learning: A scalable framework with object-level abstraction and differentiable expression search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

A Environments

Table 2: Comparison of Multi-Agent Reinforcement Learning Environments/Frameworks

Feature	VMAS	PettingZoo	SMAC	MA-MuJoCo	PyBullet	Neural MMO	Melting Pot
Key Focus	Fast, Vectorized 2D Physics (PyTorch)	Standardized MARL API (Collection)	StarCraft II Micromanagement	Multi-Agent Robotics (MuJoCo tasks)	General 3D Physics Engine	Large-Scale Agent Survival/Econ.	Social Dilemmas / Cooperation
Single/Multi	Primarily Multi (can do Single)	Primarily Multi (API standard)	Multi Only	Primarily Multi (adaptations)	Both	Multi Only (Massively)	Multi Only
Differentiable	Yes	No	No	No	No (standard use)	No	No
Generality	Medium (Physics-based, PyTorch)	Very High (via wrapped envs)	Low (Specific game benchmark)	Medium (MuJoCo robotics)	High (General physics)	Medium (Large-scale agent focus)	Medium (Social dilemmas focus)



(a) Dispersion (b) Transport (c) Discovery (d) Sampling

Figure 5: VMAS environments used in the experiments.

In this section, we provide a detailed overview of the environments used to evaluate the SHAC++ algorithm. For our experiments, we selected four vectorized scenarios from VMAS (Vectorized Multi-Agent Simulator), which offers an ideal balance of features for our research objectives.

VMAS stands out among alternative multi-agent reinforcement learning frameworks such as PettingZoo [6], SMAC [4], MA-MuJoCo [2], PyBullet [1], Neural MMO [5], and Melting Pot [3]. As summarized in Table 2, VMAS provides several key advantages for our work:

- **Multi-Agent Framework**: VMAS is specifically designed for multi-agent scenarios, making it appropriate for our experimental setup.
- **Differentiable Physics**: Unlike most alternatives, VMAS offers a differentiable physics engine that enables gradient-based optimization techniques.
- **Vectorized Implementation**: The framework supports parallel execution of multiple environments, significantly improving training efficiency.
- **Physics-based Simulation**: VMAS provides realistic modeling of agent interactions and dynamics through its physics engine.
- **Open Source Accessibility**: The codebase is fully accessible, allowing necessary modifications for our experiments.

While VMAS provides differentiable transition functions across all tasks, its reward functions are generally non-differentiable. To address this limitation, we implemented custom differentiable reward functions where possible to replicate the original VMAS behavior. Consequently, some scenarios are compatible with all tested algorithms (PPO, SHAC, and SHAC++), while others can only be used with PPO and SHAC++.

An additional consideration is that VMAS often provides agents with partial observations rather than complete state information. In some scenarios, these observations contain sufficient information to predict subsequent states, while in others they do not. This partial observability presents a particular challenge for SHAC++, as its world model must operate with incomplete information. In contrast, SHAC computes gradients with access to the full world state. We deliberately maintained these observability constraints to assess SHAC++'s performance under more realistic conditions.

Figure 5 illustrates the four VMAS environments used in our experiments, which are detailed in the following subsections.

A.1 Dispersion

In the Dispersion task, n agents must reach n randomly placed goals. In Fig. 5a, the agents are represented by blue dots, while the goals are depicted as green dots.

Each agent has a continuous 2-dimensional action space, bounded between -1 and 1, to represent acceleration along the x- and y-axes, respectively. Each agent observes its own position, velocity, and relative position to each goal. As a consequence, the world model has sufficient information to accurately predict the next states.

Each agent receives a reward of 1 upon reaching a goal, making the maximum possible reward n. While the transition function is differentiable, the reward function is not. Consequently, only PPO and SHAC++ are applicable to this scenario.

A.2 Transport

In the Transport problem, n agents work together to push a package to a randomly placed target location. In Fig. 5b, the agents are represented by blue dots, the package is shown as a red square, and the goal is depicted as a green circle. The more agents collaborate to push the package, the faster they can reach the goal and achieve the maximum reward.

Each agent has a continuous 2-dimensional action space, bounded between -1 and 1, to represent acceleration along the x- and y-axes, respectively. Each agent observes its own position, velocity, the relative position to the package, and the relative position between the package and the goal. As a consequence, the world model has sufficient information to accurately predict the next states.

Each agent receives a reward proportional to the distance between the initial position of the package and the goal. Consequently, the maximum reward corresponds to the distance between the package and the goal. Since the maximum reward varies across environments, we only plot a reference line representing good performance. Although this reward function is differentiable, the implementation provided by VMAS is not. Therefore, we created a custom reward function to replicate the original one used by all algorithms (SHAC++, SHAC, and PPO).

A.3 Discovery

In the Discovery task, n agents aim to collect as many randomly placed goals as possible. When they successfully collect k goals, another k goals are randomly placed. To collect a goal, at least s agents must be in proximity to it. In Fig. 5c, the agents are represented by blue dots, while the goals are depicted as green dots. The proximity area is indicated by a dashed circle. In our experiments, we set $s = 2$ and $k = 7$ when $n > 1$. When $n = 1$, we set $s = 1$ and $k = 7$.

Each agent has a continuous 2-dimensional action space, bounded between -1 and 1, to represent acceleration along the x- and y-axes, respectively. Each agent observes its own position, velocity, and lidar measurements of other agents and goals. As a consequence, the world model lacks sufficient information to accurately predict the next states. Specifically, the goal positions are not directly observed but are instead sensed through the lidar measurements. We believe this limitation significantly affects the performance of SHAC++.

Each agent receives a reward of 1 when it collects a goal. Thus, the maximum reward depends on the amount of time the agents have to collect the goals. For early stopping purposes, we set the maximum reward to k. Additionally, the reference line is set to k. While the transition function is differentiable, the reward function is not. Therefore, only PPO and SHAC++ are applicable to this scenario.

A.4 Sampling

In the Sampling task, n agents must collect rewards from a grid. The rewards are sampled from k Gaussian distributions. In Fig. 5d, the agents are represented by blue dots, while the k reward-generating Gaussians are displayed as a scalar field, where the intensity of the color represents the respective reward value.

Each agent has a continuous 2-dimensional action space, bounded between -1 and 1, to represent acceleration along the x- and y-axes, respectively. Each agent observes its own position, velocity, lidar values from other agents, and rewards in a 3x3 grid around itself. As a consequence, the world model lacks sufficient information to accurately predict the next states. Specifically, it does not have access to the full reward distribution. We believe this limitation significantly affects the performance of SHAC++.

Each agent receives a reward equal to the value of the reward in the grid cell where it is located. Thus, the maximum reward depends on the placement of the Gaussians. Since the maximum reward varies across environments, we only plot a reference line representing good performance. While the reward function is differentiable, the one provided by VMAS is not. Therefore, we created a custom reward function to match the original one used by all algorithms (SHAC++, SHAC, and PPO).

B Architectures

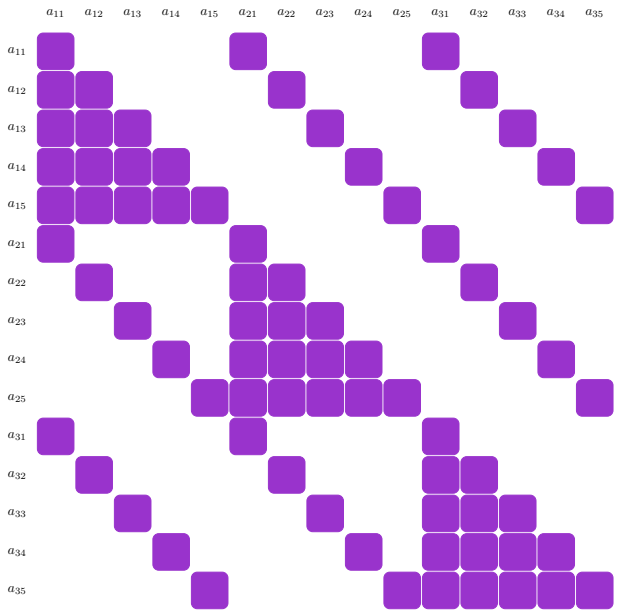
As mentioned earlier, we test PPO, SHAC, SHAC++, and SHAC+ with two different architecture—Transformer and MLP. The former is a popular positional invariant architecture for sequence modeling [7]. While the latter is a simple feed-forward neural network.

The main advantage of the transformer architecture is its positional invariance property. In fact, the order in which the agent's observation are fed into the network should not affect the ultimate outcome, at least, for the tested environments. We believe that this leads to better sample efficiency that it is ultimately observed in policies achieving higher rewards (see Table 1).

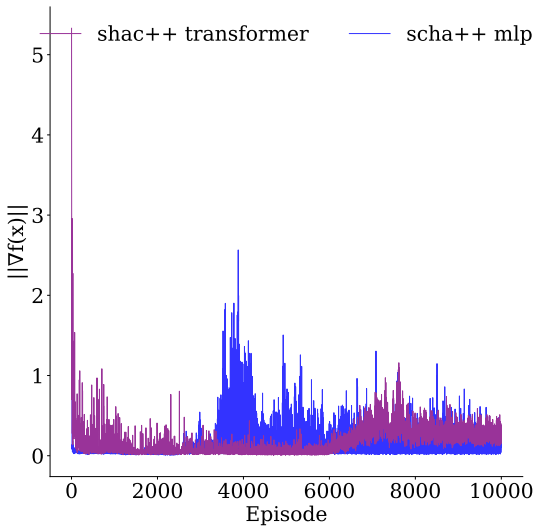
In contrast, the MLP is fed with a concatenation of all agent observations. This results in a simple and lightweight implementation, leading to faster inference times. However, in our experiments, the MLP appears to underperform compared to the transformer architecture.

B.1 Policy/Value/Reward MLP Network

When implementing the policy, value, and reward functions with an MLP, we use a single hidden layer with sizes of 32, 96, and 160, depending on the number of agents. This choice was made to account for the increasing complexity that a higher number of agents should introduce. The activation function is always ReLU, and dropout is used only for the policy network. The learning rate is consistently set to 0.001. The cache for training these networks is set to $30,000$ samples. Gradient clipping is applied only to the policy network, with a threshold of 1. For a full list of hyperparameters, see Table 3 and Table 4.



(a) Attention matrix for the world model.



(b) Gradient norm for a transformer architecture wrt. a MLP architecture.

Figure 6: Network architecture visualizations: (left) attention pattern in the world model; (right) gradient norm comparison between transformer and MLP architectures.

B.2 Policy/Value/Reward Transformer Network

When implementing the policy, value, and reward functions with a transformer, we use a single hidden layer with a size of 64, regardless of the number of agents, and a feed-forward size of 128. We do not scale these sizes with the number of agents, as the transformer is expected to natively scale with the sequence length. The remaining hyperparameters are set consistently with those used in the MLP architecture. For a full list of hyperparameters, see Table 3 and Table 4.

B.3 World Model

When implementing the action world model, we consistently use a transformer architecture. Each action is represented as an embedding fed into the network. To maintain positional invariance between the agents, we employ a custom attention matrix that allows the action for step i of agent j, a_{ij}, to attend only to actions from the same step or previous steps of the same agent. Formally, a_{ij} can attend to all a_{kj} for $k < i$ and to all a_{ik} for $k \in \mathcal{N}$. This attention pattern is shown in Fig. 6a. As a result, the world models process sequences of length $32 \cdot n$. We use a slightly deeper architecture to account for the increased complexity of the task, with 3 layers. The hidden size is set to 64 and the feed-forward size to 128. The remaining hyperparameters are set consistently with those used in the MLP architecture. For a full list of hyperparameters, see Table 3.

C Additional Experiments

Finally, we present the results of the experiments with the MLP architecture in Fig. 7. We observe that the MLP architecture underperforms compared to the Transformer architecture. Nonetheless, SHAC++ often outperforms PPO and generally performs on par with SHAC.

Interestingly, the Discovery scenario appears to be the most challenging, as none of the algorithms (PPO and SHAC++) are able to make any meaningful progress. This may be caused by the sample inefficiency of the MLP architecture, combined with the fact that the world model cannot access the full state.

The only scenario where SHAC outperforms SHAC++ is the Transport scenario. This may be due to the MLP's inability to overcome the gradient noise from the world model.

To check also the scalability of SHAC++ we run the experiments with up to 20 agents in the Transport scenario. The results are shown in Fig. 7. We observe that SHAC++ is able to scale well with the number of agents, achieving good results even with 20 agents, also in this case similarly to SHAC.

D Ablation Study

In the main paper, we focused on simultaneously replacing both the transition and reward functions with neural networks. This approach allows for greater generality and can be applied across a multitude of environments. However, when only the transition or reward function is differentiable and available, it is feasible to replace only one of them. This simplifies the algorithm, requiring the training of only a single additional network alongside the policy and value functions.

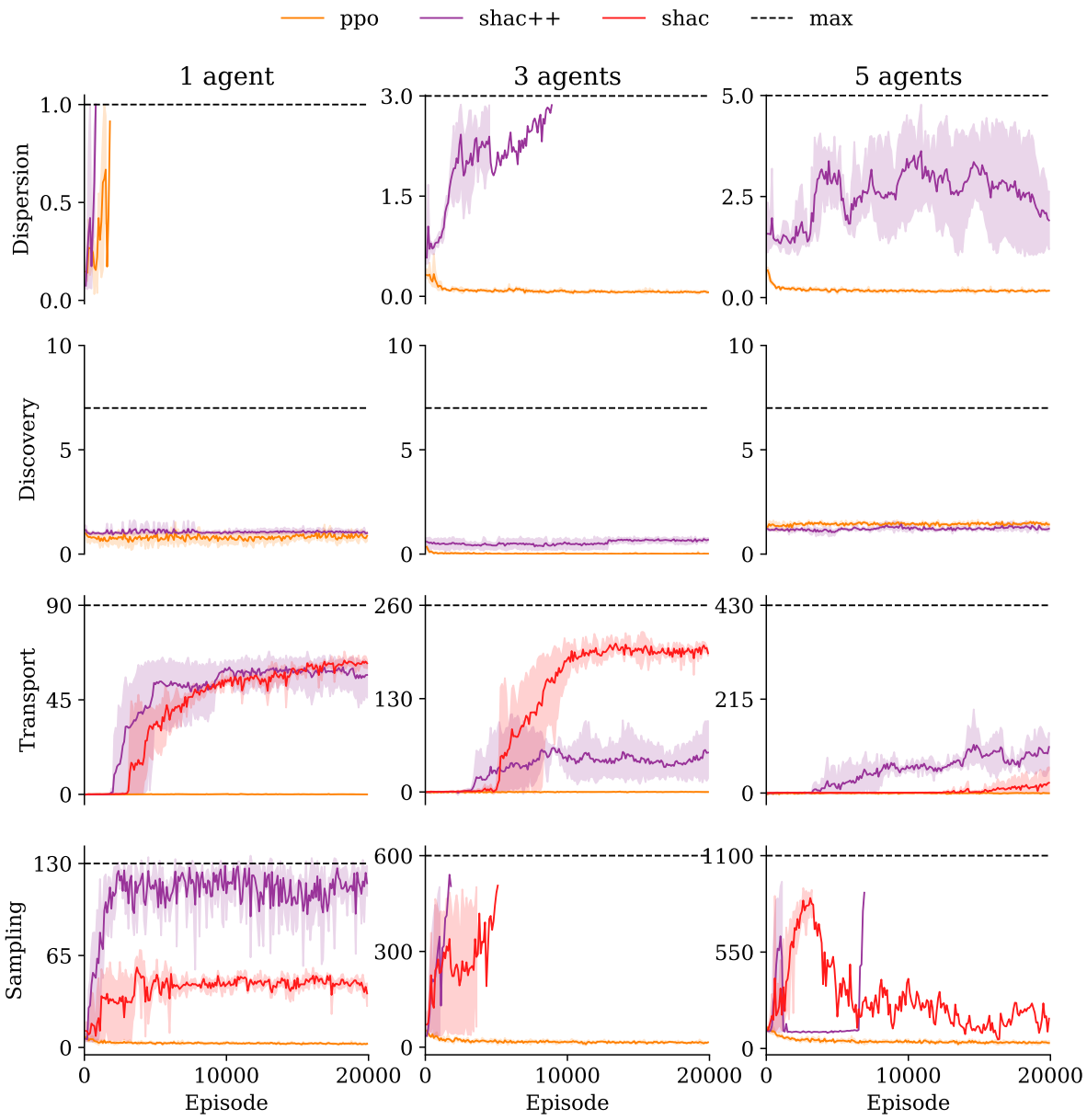


Figure 7: Comparison between SHAC++, PPO, and SHAC for increasing number of agents for Dispersion, Transport, Discovery, and Sampling scenarios with the MLP architecture. We show the mean and standard deviation of the normalized rewards across 3 runs. For the MLP architecture see Fig. 2.

Specifically, we focus on replacing only the reward function. In fact, transition dynamics are often complex and challenging to model with a neural network, especially in multi-agent environments where observations depend not only on the actions of individual agents but also on those of other agents. This, in turn, may necessitate the use of larger architectures with greater computational demands, ultimately limiting the algorithm's practicality.

In contrast, the reward function is often simpler and, therefore, easier to model using lightweight neural networks. Moreover, a high degree of precision is typically not required in most scenarios. As a result, the reward function is an ideal candidate for replacement with a neural network.

This experiment also enables us to compare the final impact of gradients generated by a complex physics engine with those generated by a neural network. While the physics engine may produce more unstable gradients, the neural network may generate imprecise gradients, particularly during the early stages of training.

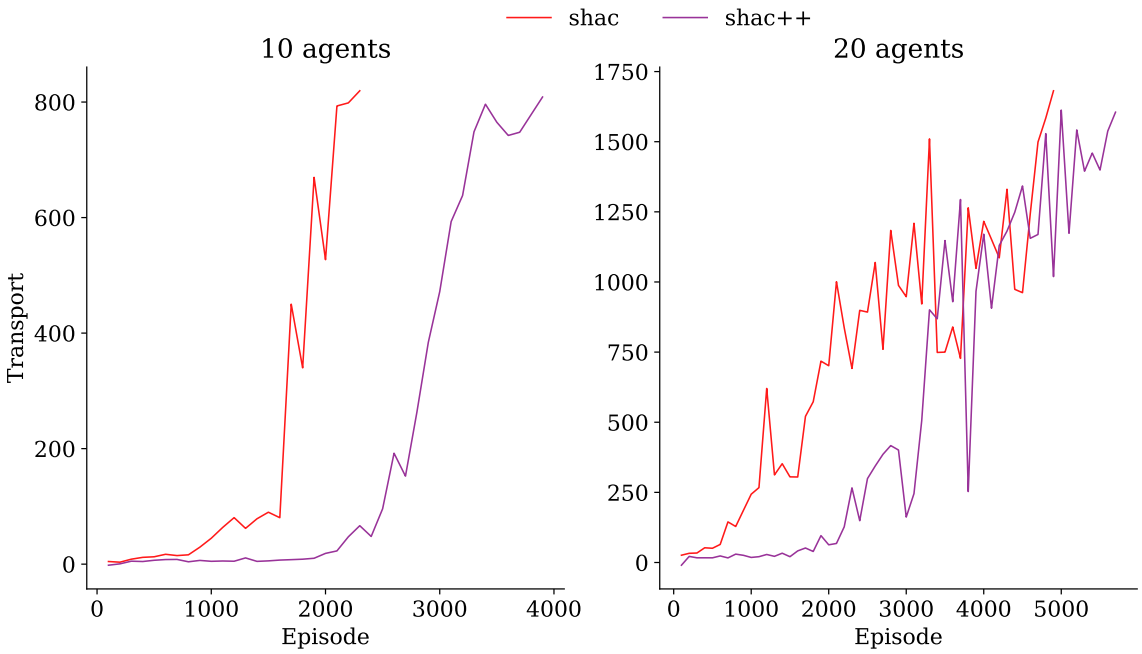


Figure 8: Comparison between SHAC++ and SHAC for increasing number of agents (up to 20) in the Transport scenario with the Transformer architecture. Results demonstrate the scalability of both approaches as the number of agents increases.

D.1 Results

We refer to the SHAC++ algorithm as SHAC+ when the transition function is differentiable and directly available. The results for the transformer architecture are shown in Fig. 9, while the results for the MLP architecture are presented in Fig. 10.

For the transformer architecture, SHAC++ generally appears to generate better policies, particularly in the Sampling and Transport environments. In the Dispersion environment, both algorithms perform comparably. Slightly better results are achieved with the SHAC+ algorithm in the Discovery environment.

For the MLP architecture, the results are more mixed. In the Dispersion environment, SHAC++ appears to achieve better policies. Conversely, in the Transport environment, SHAC+ leads to substantially better policies. In the Discovery environment, both policies struggle to produce meaningful results. Finally, in the Sampling environment, SHAC++ performs on par with SHAC+, except in the single-agent case, where SHAC+ struggles.

D.2 Discussion

We can easily observe that the transformer architecture leads, generally speaking, to better policies. This is also evident by inspecting Table 1. This may be due to the positional invariance property of the transformers leading to more sample efficient networks.

Furthermore, with the transformer architecture, we observe a tendency for SHAC++ to produce better results. This may be attributed to the generated gradients being more stable, as shown in Fig. 6b.

In contrast, for the MLP architecture, the results are more mixed. This could be attributed to the MLP architecture being less sample-efficient, which ultimately leads to less accurate gradients.

References

[1] Erwin Coumans. Pybullet, a python module for physics simulation for games, robotics and machine learning, 2017.

[2] Abhishek Gupta, Rohit Kumar, Maxim Egorov, and Sergey Levine. Multi-agent reinforcement learning in mixed cooperative-competitive environments. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*, pages 11093–11100, 2021.

[3] Joel Z. Leibo, Edgar A. Dueñez-Guzmán, Alexander Sasha Vezhnevets, John P. Agapiou, Peter Sunehag, Raphael Koster, Jayd Matyas, Charles Beattie, Igor Mordatch, and Thore Graepel. Scalable evaluation of multi-agent reinforcement learning with melting pot. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139, pages 6187–6199, 2021.

[4] Mikayel et al. Samvelyan. The starcraft multi-agent challenge. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 2186–2188, 2019.

[5] Joseph Suarez, Yilun Du, Phillip Isola, and Igor Mordatch. Neural mmo: A massively multiagent game environment for training and evaluating intelligent agents. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, pages 4476–4483, 2019.

[6] J. K. et al. Terry. Pettingzoo: Gym for multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 34, pages 15032–15043, 2021.

[7] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

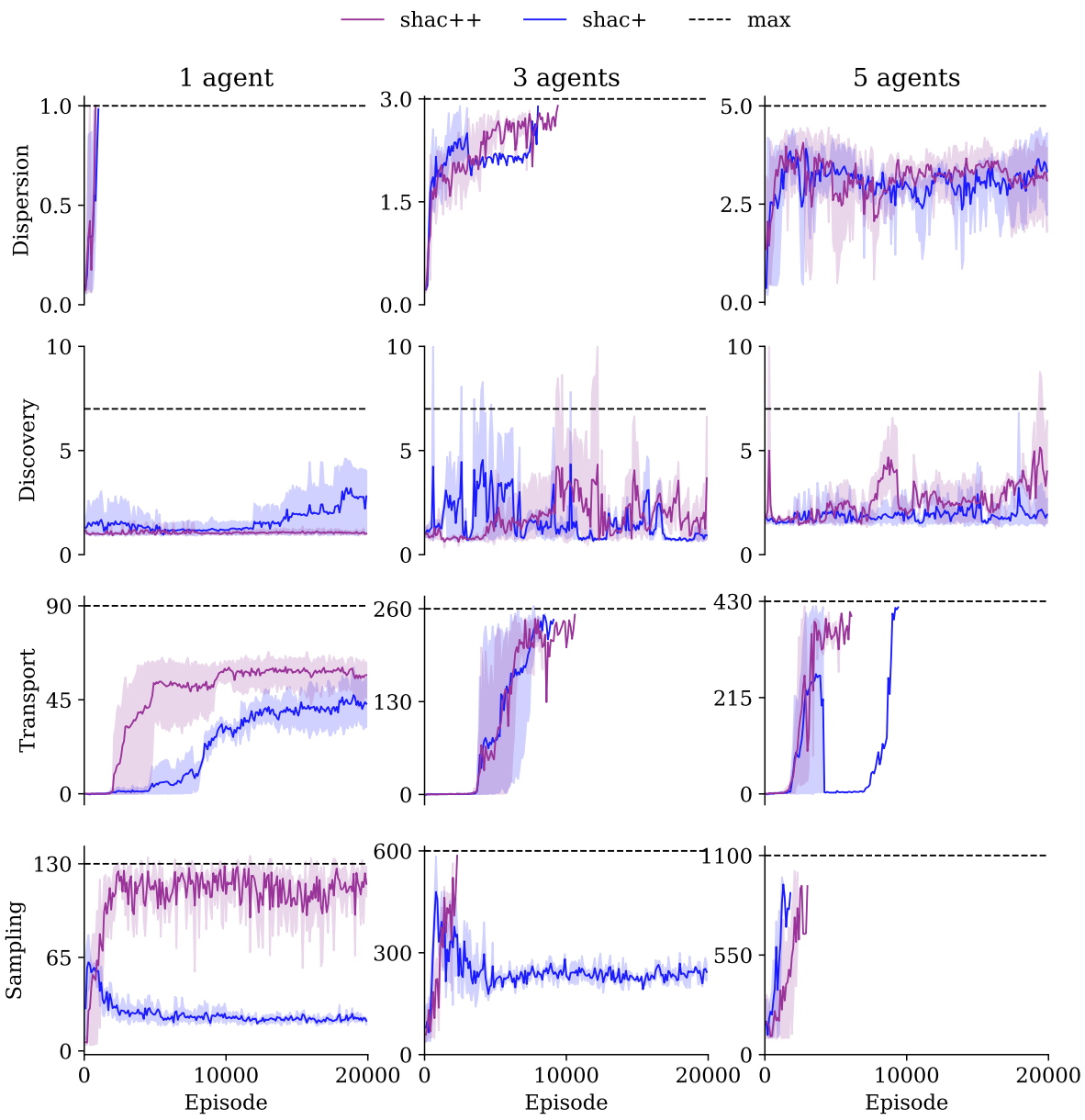


Figure 9: Comparison between SHAC++ with and without transition network, labelled SHAC+, for increasing number of agents for Dispersion, Transport, Discovery, and Sampling scenarios. Transformer architecture.

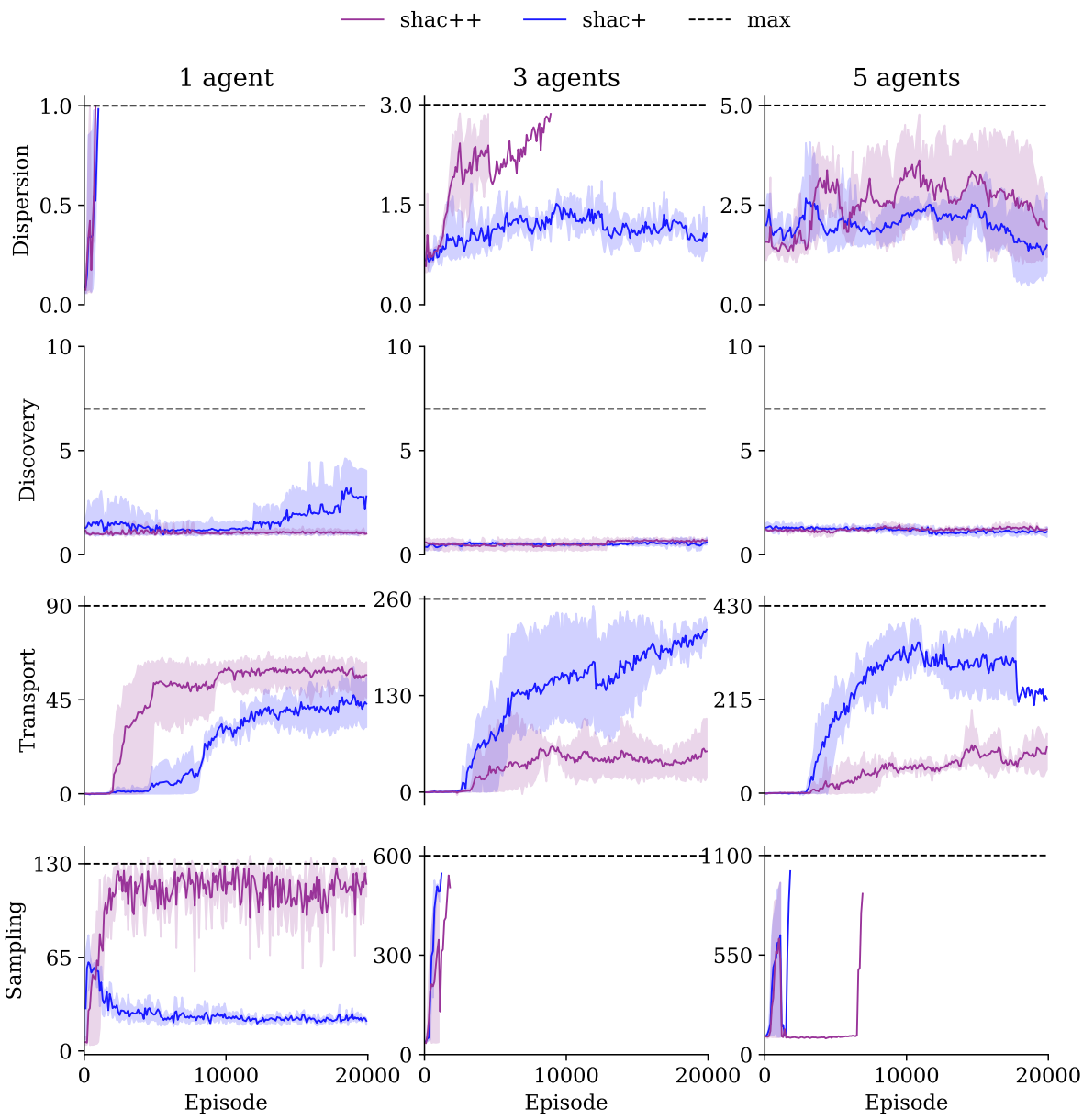


Figure 10: Comparison between SHAC++ with and without transition network, labelled SHAC+, for increasing number of agents for Dispersion, Transport, Discovery, and Sampling scenarios. MLP architecture.

Parameter	Transformer	MLP
number of training environments	512	512
training horizon	32	32
number of evaluation environments	512	512
evaluation horizon	512	512
policy layers	1	1
policy hidden size	64	160, 32, 96
policy feedforward size	128	-
policy heads	1	-
policy dropout	0.1	0.1
policy activation	ReLU	ReLU
policy variance	1	1
value layers	1	1
value hidden size	64	160, 32, 96
value feedforward size	128	-
value dropout	0.0	0.0
value activation	ReLU	ReLU
reward layers	1	1
reward hidden size	64	160, 32, 96
reward feedforward size	128	-
reward dropout	0.0	0.0
reward activation	ReLU	ReLU
world layers	3	3
world hidden size	64	64
world feedforward size	128	128
world dropout	0.0	0.0
world activation	ReLU	ReLU
policy learning rate	0.001	0.001
reward learning rate	0.001	0.001
value learning rate	0.001	0.001
value clip coefficient	None	None
reward clip coefficient	None	None
policy clip coefficient	1	1
world cache size	30000	30000
reward cache size	30000	30000
value cache size	30000	30000
world/value/reward batch size	1000	1000
world/value/reward cooldown epochs	10	10
world cache bins	25	25
reward cache bins	9	9
value cache bins	9	9
α	1.0	1.0
early stopping - max reward fraction	0.9	0.9
early stopping - max envs fraction	0.9	0.9
seed	42, 43, 44	42, 43, 44
max episodes	20000	20000
epochs between environment resets	10	10
epochs between evaluations	100	100
γ	0.99	0.99
λ	0.95	0.95

Table 3: SHAC/SHAC++/SHAC+ Hyperparameters

Parameter	Transformer	MLP
number of training environments	512	512
training horizon	32	32
number of evaluation environments	512	512
evaluation horizon	512	512
policy layers	1	1
policy hidden size	64	160, 32, 96
policy feedforward size	128	-
policy heads	1	-
policy dropout	0.1	0.1
policy activation	ReLU	ReLU
policy variance	1	1
value layers	1	1
value hidden size	64	160, 32, 96
value feedforward size	128	-
value dropout	0.0	0.0
value activation	ReLU	ReLU
policy learning rate	0.001	0.001
value learning rate	0.001	0.001
early stopping - max reward fraction	0.9	0.9
early stopping - max envs fraction	0.9	0.9
seed	42, 43, 44	42, 43, 44
max episodes	20000	20000
epochs between environment resets	10	10
epochs between evaluations	100	100
γ	0.99	0.99
λ	0.95	0.95
α	1.0	1.0

Table 4: PPO Hyperparmeters