Towards Automatic Actor-Critic Solutions to Continuous Control

Anonymous Author(s)
Affiliation
Address
email

Abstract

Model-free off-policy actor-critic methods are an efficient solution to complex continuous control tasks. However, these algorithms rely on a number of design tricks and hyperparameters, making their application to new domains difficult and computationally expensive. This paper creates an evolutionary approach that automatically tunes these design decisions and eliminates the RL-specific hyperparameters from the Soft Actor-Critic algorithm. Our design is sample efficient and provides practical advantages over baseline approaches, including improved exploration, generalization over multiple control frequencies, and a robust ensemble of high-performance policies. Empirically, we show that our agent outperforms well-tuned hyperparameter settings in popular benchmarks from the DeepMind Control Suite. We then apply it to less common control tasks outside of simulated robotics to find high-performance solutions with minimal compute and research effort.

1 Introduction

Deep Reinforcement Learning (RL) has had great success in many diverse and challenging domains, from robotics [24] to the game of Go [44] and autonomous balloon navigation [5]. However, day-to-day progress in the field is measured in a limited number of benchmark tasks and tends to be dominated by a small group of algorithms. The model-free off-policy actor-critic literature includes dozens of papers that compare their methods on simulated locomotion tasks that have been popular for half a decade [40] [6]. In that time, the research community has settled on a set of accepted hyperparameters and design heuristics that rarely changes. While this may save time and compute when comparing methods on common benchmarks, it makes approaching a brand new domain a daunting and computationally expensive challenge. Many of the most important hyperparameters in state-of-the-art actor-critic algorithms are unintuitive; even experienced RL practitioners may need to resort to grid search and other expensive hyperparameter optimization techniques.

This paper looks to automate the process of tuning an actor-critic algorithm and creates an out-of-the-box solution to dense-reward continuous control problems. The result is a general algorithm that tunes almost every RL design decision and returns an ensemble of high-performance policies while remaining sample-efficient. First, we compare against baseline approaches in common control benchmarks. Then we evaluate our method's ability to reduce engineering effort in new domains by applying it to complex tasks inspired by real-world industrial challenges and operations research. Our solution, which we call Automatic Actor-Critic (**AAC**), is easy to implement and leaves several promising directions for future improvement.

2 Background

2.1 Model-Free Off-Policy Actor-Critics for Continuous Control

We assume the reader is familiar with Reinforcement Learning and the general Markov Decision Process (MDP) setting; see Appendix A for an overview of notation. This paper focuses on solving control tasks where actions are continuous vectors. Deep RL research in continuous control was partly inspired by the success of model-free off-policy actor-critic methods like DDPG [32]. DDPG and later variations iteratively learn a policy π and action-value function(s) Q, parameterized by a deep neural network actor and critic(s), denoted π_θ and Q_ϕ respectively. Interactions are sampled from the environment and added to a large replay buffer $\mathcal{D}$. Typically we encourage the exploration of new behavior by injecting random noise into the policy or sampling from a stochastic actor. The critic networks are updated by approximate dynamic programming to learn the value of taking action a in a given state s. This is accomplished by regression to a temporal difference (TD) target:

$$\mathcal{L}_{critic} = \mathop{\mathbb{E}}_{(s,a,r,s',d)\sim\mathcal{D}} \left[\left(Q_\phi(s,a) - (r + (1-d)\gamma Q_{\phi'}(s', \tilde{a}')) \right)^2 \right] \tag{1}$$

Where $\tilde{a}' \sim \pi_\theta(s')$, and ϕ' term refers to target networks, which are discussed in more detail in Section 2.2. The continuous action space makes it intractable to recover an optimal policy directly from the Q function. Instead, we train the actor network to maximize the output of the critic network:

$$\mathcal{L}_{actor} = \mathop{\mathbb{E}}_{s\sim\mathcal{D}} \left[-Q_\phi(s, \pi_\theta(s)) \right] \tag{2}$$

State-of-the-art algorithms typically learn an ensemble of critic networks to reduce update bias due to function approximation error [15]. In the case of two critic networks, this is known as the “clipped-double-Q-trick" - though it can be expanded to an arbitrary number of networks [8].

The community has generated a vast literature discussing various techniques to improve sample efficiency and performance. Among these are algorithms such as TD3 [15], SAC [16], SUNRISE [30], DisCor [27], REDQ [8], and GRAC [42] - to name only a few. Model-free off-policy actor-critics retain their popularity because their performance is near state-of-the-art on common benchmarks while being widely available and reproducible. Deep Actor-Critics have wide-ranging applications, but the incremental progress in the field is primarily measured on a small set of benchmark tasks. In the process of inventing, comparing, and re-implementing dozens of alternative approaches on this small task set, the research community has settled on several heuristic design choices that are critical to high performance. The reliance on these settings makes it difficult to apply this family of algorithms to new domains. In the next section, we discuss what we consider to be the most important hyperparameters in off-policy actor-critics and the challenges that come with tuning them.

2.2 Design Decisions in Deep Actor-Critic Algorithms

Target networks, Learning Schedules and Replay Ratios: Target networks (Eq 1) prevent the update to $Q_\phi(s,a)$ from immediately impacting the value of $Q_\phi(s',a')$, which destabilizes learning by altering the value of the Q-network's TD target. We can control our targets' rate of change by using a separate network to output $Q_{\phi'}(s',a')$, and updating that network periodically [35] or as a moving average of the online critics' weights with polyak parameter τ [32]. In either case, this creates an important hyperparameter decision. Updating the target network too quickly or too often will destabilize learning, while updating it too slowly or infrequently leads to an unnecessary drop in sample efficiency. GRAC [42] proposes a “self-regularized" update that preserves stability by explicitly penalizing changes in $Q_\phi(s',a')$ - removing the need for target networks and their hyperparameters. However, the penalized update has the side-effect of being much more conservative, requiring several critic gradient updates per training step. The question of precisely how many more updates are needed introduces additional hyperparameters. We need to identify the optimal ratio of 1) environment samples collected, 2) actor gradient updates, 3) critic gradient updates. These values create a *learning schedule* or *replay ratio*. Recent work has found that increasing the replay ratio can boost sample efficiency and performance [13] [8].

Action Persistence and Control Frequency: The control frequency is the rate at which the agent receives states from the environment and is required to provide a new action. In real-world robotics, this might be determined by sensor delay or other natural barriers. The control frequency of simulated research environments is often a relatively arbitrary constant set inside the parameters of the physics simulator (e.g., MuJoCo [48]). As control frequency increases, the time between actions decreases, and therefore the consequences of individual actions become difficult to distinguish. More formally, the advantage $Q(s,a) - V(s)$ of an action a approaches zero as the time between states approaches zero [45] [33]. This presents a challenge to Q-learning methods because the critic value landscape we are maximizing will appear to be mostly flat. A simple way to address this is to enforce a lower control frequency by repeating the agent's past action k times before asking for a new decision. This reduces the control frequency by a factor of k and increases the advantage of optimal actions. The value of k is called the "action persistence", "action repeat" or "frame skip" parameter. Increasing the action persistence can also improve exploration and reduce forward pass computation during deployment [25] [21].

A properly tuned action persistence can have a significant impact on performance [39]. However, high values of k can cause the gap between action choices to be too long to adapt to sudden changes in the environment. There have been several proposed ways to tune k. One approach is to make the action repeat an aspect of the action itself. In discrete settings, this may involve multiplying the action space to create a new action for each original choice but at several different values of k [28]. In continuous spaces, the persistence can become a new dimension of the action vector [43]. Another approach (TASAC [52]) learns a second policy whose only action choices are to repeat the previous action of the main policy or select a new one. More complicated methods directly estimate the optimal action persistence [33] and can handle different control frequencies for each component of the action [29].

Discount Factor: The MDP discount factor γ controls the time horizon over which the agent is maximizing returns. This value is usually treated as a fixed element of the benchmark and set at .99. However, agents are almost always evaluated based on their undiscounted $(\gamma = 1.0)$ returns, which makes γ more of an agent-side hyperparameter than an environmental constraint [21]. Prior work has considered hyperparameter schedules for γ that boost performance by regularizing learning [1].

Entropy Regularization: One way to ensure diverse experience collection is to optimize for policy entropy, balanced by an additional hyperparameter α. PPO [41] and A2C/A3C [34] set α to be a small fixed constant. SAC [17] uses a MaxEnt-RL framework that makes entropy part of the value function. This increases policy entropy and is thought to have other benefits, including robustness to environmental uncertainty and partially observed reward functions [12]. Entropy regularization also keeps the exploration policy more centered around zero as opposed to the random noise heuristics of TD3 and DDPG, which can cause actions to be repeatedly clipped at $(-1, 1)$ [49]. The follow-up version of SAC [16] tunes α to dynamically approach a target entropy level with gradient descent. This target entropy level is denoted H and set to $-|\mathcal{A}|$ - a value that works empirically on benchmark tasks but becomes an unintuitive hyperparameter in new domains. Meta-SAC [50] tunes the target entropy value with a meta-gradient approach.

3 Method: Automatic Actor Critics

The issues above require a number of heuristic solutions that would be expensive to re-tune and re-evaluate on a new domain. We attempt to address this by creating a unified approach that automatically discovers new heuristics for each task and sheds as many hyperparameters as possible. We find inspiration in Population Based Training (PBT) [23]. In PBT, a population of independent training runs with different hyperparameters are conducted independently. At regular intervals, the performance of each run is used to generate a more optimal set of hyperparameters according to an evolutionary strategy where the parameters of the highest-performing runs are used to re-initialize the worst-performing setups. Hyperparameters are randomly perturbed to explore the parameter space. In the off-policy RL context, PBT is quite sample-inefficient because each training run in the population collects a full buffer of samples independently despite being designed to recycle data from a variety of policies. Our first modification is to share environment experience across members of the population. Because the replay buffer we are optimizing over now consists of the experience of many different agents with different parameters, we are diversifying experience collection and recovering the exploration advantages of multi-actor setups like D4PG [4].

We initialize a population of M SAC-style actor-critic agents and begin by searching over γ and the target entropy coefficient H. We make a change of notation from γ and H. g substitutes γ, where $\gamma = 1 - exp(g)$; this gives the agent more control over the small differences between discount values approaching 1.0. h substitutes for the target entropy H, where $H = h(-|\mathcal{A}|)$, meaning it is a coefficient for the default SAC heuristic of $H = -|\mathcal{A}|$.

From there, we add two adjustments to the core agent intended to reduce hyperparameters and find higher-performance policies. We eliminate the need for target networks by utilizing the self-regularizing critic update from GRAC [42]. Updates are stabilized by minimizing the impact that changes to $Q(s, a)$ that have on the target value $Q(s', a')$:

$$\mathcal{L}_{sr} = \mathcal{L}_{critic} + (\not\nabla Q_\phi(s', a') - Q_\phi(s', a')) \tag{3}$$

where $\not\nabla$ denotes a stop gradient operator. This process can be replicated across each critic network when using the clipped-double-Q trick or another bias-reducing method. This loss function slows critic learning by reducing the impact of each gradient update. The GRAC authors address this by introducing an additional heuristic whereby the critic optimization loop continues until the critic loss is less than some percentage of its initial value on that training step. That percentage is annealed throughout training as the critic is more accurate and is less able to reduce its loss function. Experiments in [42] demonstrate that this is a sensitive hyperparameter. Our goal is to eliminate sensitive hyperparameters, so we add the number of actor and critic updates per gradient step as separate PBT-tuned parameters. We denote these as a and c, respectively. Searching over both a and c creates an adaptive replay ratio schedule that can improve sample efficiency.

The action persistence value k discussed in Sec 2.2 can have a critical impact on performance. Rather than adding additional action outputs to adjust action repetitions k, we experiment with the simpler solution of making k a tunable parameter of the environment and add it to the PBT search. However, adjusting the control frequency of the population's experience over time complicates the use of replay buffer data. We address this by concatenating the current value of k to the state vector of the environment. This allows the actor and critic networks to recognize changes in control frequency and adapt their output accordingly while replaying transitions from the buffer as usual. A side effect of this approach is that it allows the agent to generalize across control frequencies and adapt to changes during deployment. There are some additional details related to how we compensate for changes in γ to the reward of frame-skipped transitions. A thorough discussion of this approach to "persistence-aware actor-critics" is provided in Appendix B along with additional experiments focused on this idea.

In total, we are now automatically searching over five key hyperparameters (a, c, k, g, h). Each member of the population trains for one evolutionary epoch with its own hyperparameter values. The population is then evaluated in the environment to determine each member's "fitness" (f). We set the fitness of agent i, denoted f_i, to its mean return with action persistence k_i[1]. However, more complex novelty-related bonuses could be incorporated to improve exploration (Sec 6). The highest-fitness members are randomly paired with the lowest-fitness members to transfer and then perturb their hyperparameter values. Network parameters and optimizer states are also transferred.

All that is left to define are the ranges of hyperparameter values that we would like to search. While this may seem like we have traded each parameter for three new ones (the lower bound, upper bound, and random perturbation range), these are intuitive to define in practice. If our range is too broad, the evolutionary algorithm may take more time to find the correct values, but we can be reasonably confident that it will. The only difficult hyperparameters that we have introduced are the frequency of evolutionary updates and the population size. However, both have intuitive runtime/performance tradeoffs - increasing the population size and length of individual training runs makes us more likely to find the correct parameters and more likely to notice the performance gap between them. We can set these meta-parameters in advance based on available time, compute, and problem difficulty.

Pseudocode is provided in Algorithm 1 and additional implementation details are discussed in Appendix C.1. We will refer to this method as "Automatic Actor-Critic" (**AAC**). To summarize, this agent:

- Does not use target networks and their associated hyperparameters. We automatically learn the replay ratio and additional critic update schedule.
- Dynamically adjusts the action persistence but arrives at a fixed control frequency. As a side effect, it is also capable of adapting to sudden changes in control frequency.
- Does not rely on random noise heuristics for exploration. We sample from a high-entropy stochastic policy that is automatically tuned to approach a target entropy level that is also automatically tuned.
- Does not treat the discount factor as a fixed environment parameter and can dynamically adjust γ to improve evaluation performance.

[1]It is common for environments to have strict maximum episode lengths that directly influence the final return. In these cases, we compensate for differences in action repetition by dividing the step limit by k.

Algorithm 1 Automatic Actor Critic Training

```
Require: Population size $M$, evolutionary epoch $E$, steps per epoch $T$, min and
    max values for $a, c, h, k, g$ (denoted as $a_{\min}, a_{\max}, \ldots, g_{\min}, g_{\max}$).
 1: $\mathcal{D} \leftarrow$ replay buffer initialized with random samples
 2: for $i = 1, \ldots, M$ in population do
 3:     $a_i \sim \mathcal{U}(a_{\min}, a_{\max})$
 4:     $c_i \sim \mathcal{U}(c_{\min}, c_{\max})$
 5:     $h_i \sim \mathcal{U}(h_{\min}, h_{\max})$
 6:     $k_i \sim \mathcal{U}(k_{\min}, k_{\max})$
 7:     $g_i \sim \mathcal{U}(g_{\min}, g_{\max})$
 8:     $P_i^0 \leftarrow (\theta_i, \phi_i, a_i, c_i, h_i, k_i, g_i)$
 9: end for
10: for $e = 1, \ldots, E$ epochs do
11:     for $t = 1, \ldots, T$ steps per epoch do
12:         for $i = 1, \ldots, M$ in population (in parallel) do
13:             Collect exp. from env with $k_i$ and add to $\mathcal{D}$
14:         end for
15:         for $i = 1, \ldots, M$ in population (in parallel) do
16:             for $c = 1, \ldots, c_i$ do
17:                 $\gamma_i = 1 - e^{g_i}$
18:                 critic_update($\phi_i$, $\gamma_i$, $\mathcal{D}$)                ▷ (Eq 3)
19:             end for
20:             for $a = 1, \ldots, a_i$ do
21:                 $H_i = h_i(-|\mathcal{A}|)$
22:                 actor_update($\theta_i$, $H_i$, $\mathcal{D}$)                ▷ (Eq 2)
23:             end for
24:         end for
25:     end for
26:     for $i = 1, \ldots, M$ in population (in parallel) do
27:         Evaluate $P_i^e$ for fitness $f_i$ with persistence $k_i$
28:     end for
29:     Sort population $P$ by $f_i$
30:     "Bad" members $\leftarrow$ bottom 20% of $P$
31:     "Elite" members $\leftarrow$ top 20% of $P$
32:     Randomly shuffle "Bad" and "Elite"
33:     for $bad \in$ "Bad" and $elite \in$ "Elite" do
34:         Copy $elite$'s parameters & weights to $bad$
35:         Perturb $bad$'s $a_i, c_i, h_i, k_i, g_i$
36:     end for
37: end for
```

- Improves exploration by sampling experience from a variety of diverse policies.
- Has just two important hyperparameters, both of which have intuitive performance/runtime tradeoffs that can be considered in advance.
- Returns a population of high-performance solutions that can be ensembled to form a robust final policy.

4 Experiments

We consider the following baselines:

- **SAC**. Soft Actor-Critic with tunable entropy and literature-standard hyperparameters; a table of these standard parameters is available in Appendix D.
- **Persistence-Aware SAC (k-SAC)**. Soft Actor-Critic with tunable entropy and literature-standard hyperparameters, but trained with varying action persistence. We evaluate the agent on a range of k values and report the highest performance.

- **Self-Regularized SAC (SR-SAC)**. We incorporate the self-regularized critic update (Eq 3) into standard SAC[2]. The number of critic updates per training step is determined with the heuristic in [42] - we update on a given batch until the loss drops below $\beta\%$ of its initial value. All other hyperparameters are set to the literature standards.
- **Random Parameter SAC (Rand-SAC)**. Soft Actor-Critic with hyperparameters uniformly chosen from AAC's search space[3]. Each run generates a new set of random hyperparameters. This highlights the hyperparameter sensitivity of SAC and shows the range of performance achieved by naively picking reasonable values to approach each environment.

The total network parameters are kept comparable by adding the action persistence value to the input state whether or not this value is varied during training. For example, SAC runs as normal with a state vector that has an additional element that is fixed at 1. Results are listed as the mean and standard deviation of 5 random seeds. Rand-SAC has high variance by design - we compensate for the extra randomness with 15 total trials. The mean return of Rand-SAC is not as interesting as the variance because sufficient samples represent the performance of the mean of our random parameter distributions.

The baselines are tested alongside AAC in five common tasks of varying difficulty from the DeepMind Control Suite [47]. The results are shown in Figure 1. The randomized hyperparameters are consistently low-performance and high-variance, as expected. The standard SAC defaults are heavily tested on these tasks, so it is not surprising that they perform quite well. SR-SAC and k-SAC are special-purpose techniques designed to compensate for specific design choices in SAC. Their relative performance varies across each task and depends on whether the hyperparameter they address happens to be a significant factor. For example, $k = 1$ is suboptimal in “Fish, Swim", so k-SAC performs well. On the other hand, the critic learning schedule in default SAC is too conservative for “Cheetah, Run", so SR-SAC offers a large improvement. AAC can adapt both of these parameters and discover the correct settings on a task-by-task basis; it matches the performance of the highest-performing baseline, although it may take more samples to sort out the optimal settings. Note that one reason for AAC's slight drop in sample efficiency is the value of a_{max} and c_{max} used in our experiments. We are not allowing our algorithm to fully compensate for the increase from 1 environment sample per training step to AAC's distributed sampling. A member of the population that maxes out its actor and critic updates per step still cannot reach the replay ratio of SR-SAC. This choice was made because our implementation is synchronous, and allowing for a wide range in gradient update counts results in poor compute utilization. We discuss some workarounds for this in Section 6.

Figure 2 shows the evolution of the highest-performing parameters over time. We plot the default parameter value as a light blue line for reference. AAC rediscovers the tuned default setting when it happens to be optimal for the task, e.g., γ and H. Other parameters vary more across tasks, particularly k.

While it is helpful to know that AAC can find quality solutions to popular benchmarks, the real purpose of our algorithm is to simplify the use of actor-critic methods in less common domains. We put this to the test by evaluating AAC outside of simulated robotic locomotion.

	Random Policy	**Qin et al. [38]**	**Rand-SAC**	**SAC**	**AAC**
Setpoint 70	-322 ± 57	-180	-399 ± 99	-216 ± 16	$\mathbf{-175 \pm 3}$
Setpoint 100	-439 ± 129	-	-432 ± 147	-314 ± 56	$\mathbf{-257 \pm 43}$

Table 1: **Industrial Benchmark Results.** Total returns scaled by 1e-3 for readability. The “setpoint" parameter controls the difficulty of the environment and is bounded in $[0, 100]$. We add the setpoint 70 results from [38] to verify our implementation.

The Industrial Benchmark [19] is a synthetic control task designed to imitate the challenges that arise in managing industrial systems. The agent controls three “steering" variables and is rewarded for minimizing the cost and “fatigue" associated with operating the system. The environment has stochastic and delayed rewards along with a partially observable state. We evaluate SAC, Rand-SAC,

[2]We note that this agent is not equivalent to GRAC because it does not include its additional tricks (e.g., CEM action improvement). We are simply adding the self-regularized critic update to SAC to eliminate target networks.

[3]Default SAC has hyperparameters that AAC does not, e.g. τ. In these cases, the value is chosen from a range around the literature default.

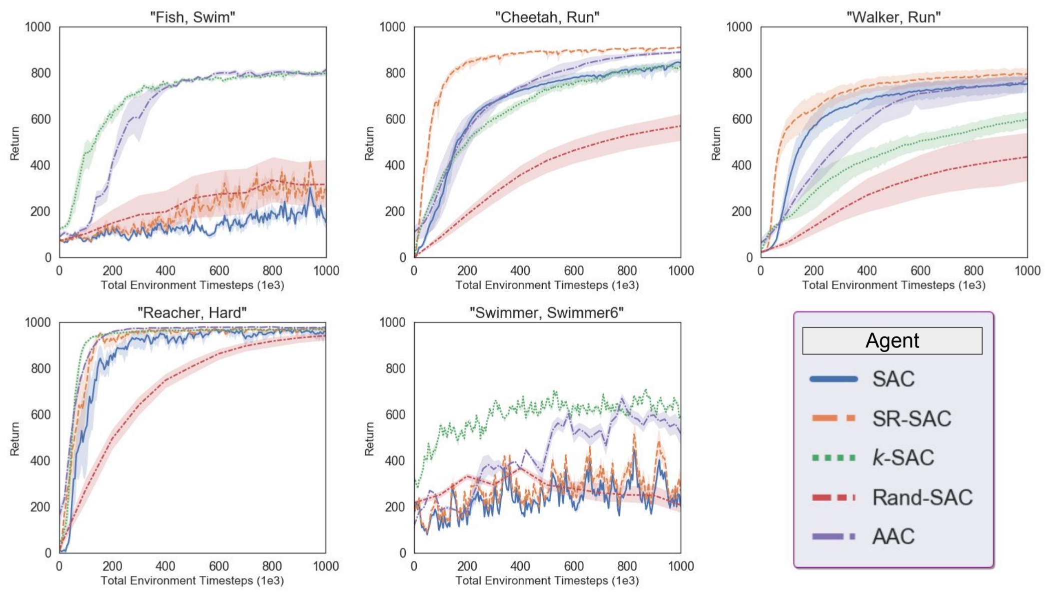

Figure 1: **AAC on benchmark continuous control tasks.** [Best viewed in color]

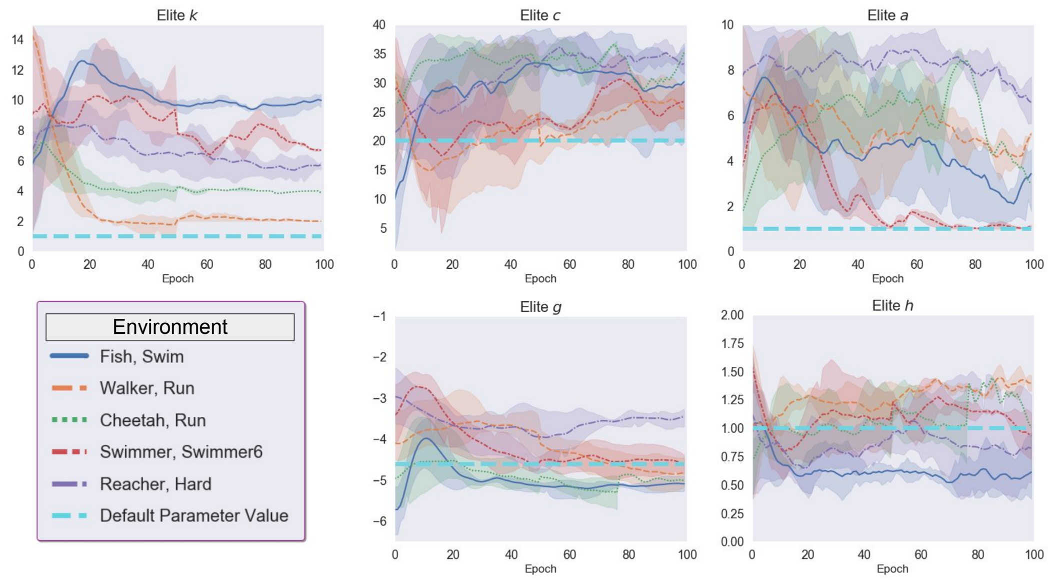

Figure 2: **The hyperparameters learned by AAC.** The common default value is indicated with a horizontal blue line. h and g are substitute variables for H and γ, respectively; see Sec 3 for an explanation. [Best viewed in color]

Random Policy	**Rand-SAC**	**SAC**	**Hubbs et al. [22] RL (PPO)**	**Hubbs et al. [22] Oracle**	**AAC**
8.8 ± 81.1	118 ± 186	342 ± 11	$\mathbf{409.8 \pm 17.9}$	$\mathit{542.7 \pm 29.9}$	$\mathbf{415 \pm 1.5}$

Table 2: **Inventory Management Results.** Rand-SAC, SAC and AAC collect $500,000$ steps of experience. The RL result from [22] uses Proximal Policy Optimization (PPO) [41]. The oracle method generates a theoretical upper bound by solving an optimization problem with information not available to the agent.

and AAC in two different situations of increasing difficulty. The results are displayed in Table 1. AAC reduces the operating cost, and its effects are more noticeable at the greater difficulty.

Next we consider two inventory management problems (IMPs) proposed by [22] and [3]. IMPs involve managing a supply chain to meet customer demand while balancing costs associated with

	Random Policy	SAC	Rand-SAC	AAC
Mean	$-21,669$	-79	$-9,901$	$\mathbf{2.68}$
Std.	$21,032$	25	$11,008$	$\mathbf{2.33}$
Max	$-1,015$	-23	0.57	$\mathbf{6.83}$

Table 3: **Newsvendor Results.** Total return (scaled by 1e$-$4) in the environment over a 40 day interval after $500,000$ environment steps. We also report the maximum score because asymmetric returns make the standard deviation a misleading estimate of the upper performance bound.

ordering and carrying new materials[4]. We assume no prior knowledge of the IMP and instead attempt to solve it using our automatic method. To demonstrate the benefits of an automated tuning system, we do not use an iterative development cycle[5]; we ran AAC for five random seeds on each environment and report the initial results. The scores for the `InvManagement-v1` and `Newsvendor-v0` environments are listed in Tables 2 and 3, respectively. AAC outperforms our baselines and matches the performance of tuned RL results reported in [22].

Finally, we demonstrate the practical advantages of AAC's population of diverse and persistence-aware policies. Results on the DeepMind Control Suite environments are shown in Figure 3. Ensembling the AAC population of actor networks greatly improves performance at sub-optimal control frequencies. Further experiments verify that the state representation of k is correctly used to adapt the policy to changes in action persistence.

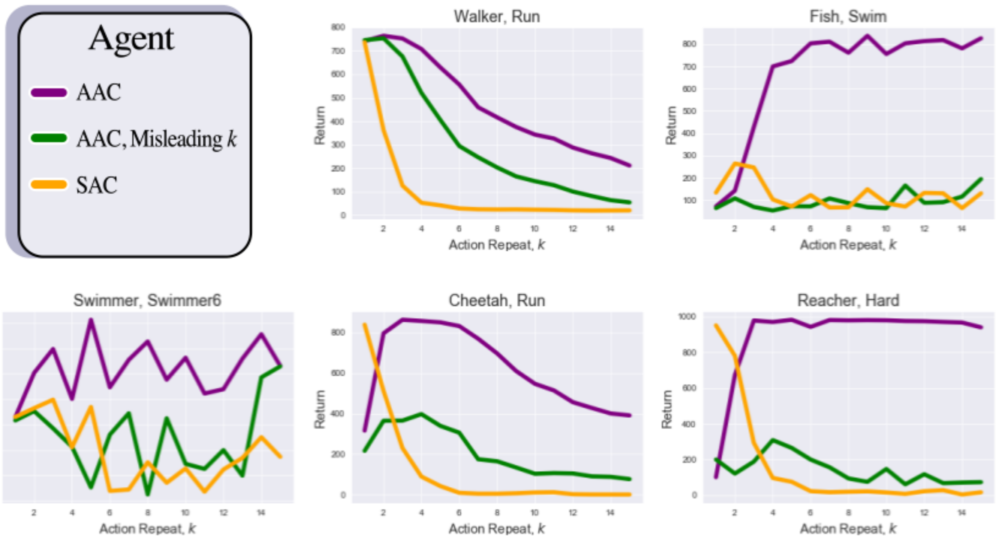

Figure 3: **Generalization of AAC across control frequencies**. Trained agents are evaluated across a range of control frequencies. We utilize AAC's ensemble of policies by returning the mean action across the population. AAC is more robust to changes in control frequency than SAC. In the "Misleading k" experiments, the control frequency of the underlying environment is altered while the state representation of k is fixed at 1; the policy networks have learned to interpret the k value to improve performance at sub-optimal control frequencies.

5 Related Work

This work contributes to the AutoRL literature of online hyperparameter tuning in Deep RL. [21] discusses algorithms' reliance on the inductive biases introduced by popular benchmarks and demonstrates that adaptive methods can match and exceed the performance of well-tuned baselines. OMPAC [10] uses a genetic algorithm to select the policy's softmax temperature and TD(λ) parameters in discrete environments such as Atari and Tetris. HOOF [37] generates a population of policy gradient

[4]We refer the reader to the original references [22] & [3] for thorough descriptions of the environments.

[5]We make two changes to the DMC experiment settings before launch: k_{max} is lowered to 5 from 15 because these environments have short time horizons of 40 and 30, and the epoch length is lowered to 500 from 1000 because we are testing after $500,000$ total timesteps.

updates with different loss function parameters and selects the best combination to continue training with weighted importance sampling. Agent57 [2] uses hyperparameter selection by a multi-armed bandit to improve exploration and surpass human performance on the Atari benchmark. STAX [53] uses meta-gradients to tune the differentiable hyperparameters of the IMPALA [11] algorithm.

The two most similar works to our own are OHT-ES [46] and SEARL [14]. Both are PBT-inspired hyperparameter tuning methods for off-policy RL with a shared replay buffer. OHT-ES adapts the learning rates and discount factor of TD3 while SEARL primarily adjusts the architecture of TD3's networks. Our method eliminates more RL hyperparameters using a MaxEnt framework, self-regularized critic update, dynamic learning schedule, and action repetition. However, we do not attempt to tune optimization-related hyperparameters like learning rate and network size. These methods are quite compatible, and it would be interesting to investigate extensions that combine our core agent with, e.g., the network search and tournament selection of SEARL. We do not compare against these works here because the differences in the core RL agent optimized and the parameters considered make the comparison unmeaningful. The action repetition in our method also complicates comparisons based on total timesteps. AAC prioritizes reducing hyperparameters and easing the development process, which is why our comparisons focus on variations of the RL agent we are optimizing.

6 Limitations and Future Directions

While our method successfully reduces RL-specific hyperparameters and design heuristics, we have not fully realized at least two of its promising advantages. First, distributed and diverse experience collection has the potential to increase exploration in sparse-reward environments. Diversity could be further improved by introducing exploration techniques from both the RL and evolutionary computation literature. We could motivate the exploration of individual actors by incorporating intrinsic rewards [7], and improve the parameter and behavioral diversity of the population as a whole with ideas from Novelty Search [31] [9].

We have also opted to avoid the meta-optimization of network-related hyperparameters such as model architecture and learning rate. The automatic discovery of optimal network architectures is an active area of research in the broader field of AutoML - see [18] for a survey. Many of these approaches could be added to our evolutionary algorithm with the help of an effective indirect encoding for model architecture and safe mutation operations. This is likely to increase the number of evolutionary epochs required to converge on a solution. However, there is plenty of evidence that network design can significantly increase the performance of actor-critic methods [20].

There is also room for improvement in terms of runtime and scalability. The synchronous implementation (see Appendix C.1) used in our experiments limits our ability to adapt time-consuming parameters like the number of actor and critic gradient steps. The original PBT work [23] used an asynchronous framework where elite population members were checkpointed during training and could be read from disk when replacing low-performance members. A similar system could be adapted for the AAC algorithm. This would likely lead to a drop in sample efficiency but may open up the opportunity to scale the method to large clusters and search over more hyperparameters - especially network architectures.

7 Conclusion

This work has presented an automatic framework for online hyperparameter optimization in off-policy actor-critic algorithms. We have shown that our adaptive method can exceed the performance of tuned baselines in common benchmark tasks. However, the true promise of AutoRL methods lies in their ability to automate the process of engineering RL solutions to new domains. We demonstrated our algorithm's ability to succeed in less-studied industrial and operations research environments and are hopeful that this line of work will help enable the adoption of RL to a broader range of real-world problems.

References

[1] Ron Amit, Ron Meir, and Kamil Ciosek. *Discount Factor as a Regularizer in Reinforcement Learning*. 2020. arXiv: 2007.02040 [cs.LG].

[2] Adrià Puigdomènech Badia et al. *Agent57: Outperforming the Atari Human Benchmark*. 2020. arXiv: 2003.13350 [cs.LG].

[3] Bharathan Balaji et al. *ORL: Reinforcement Learning Benchmarks for Online Stochastic Optimization Problems*. 2019. arXiv: 1911.10641 [cs.LG].

[4] Gabriel Barth-Maron et al. *Distributed Distributional Deterministic Policy Gradients*. 2018. arXiv: 1804.08617 [cs.LG].

[5] Marc G. Bellemare et al. "Autonomous navigation of stratospheric balloons using reinforcement learning". In: *Nature* 588.7836 (Dec. 1, 2020), pp. 77–82. ISSN: 1476-4687. DOI: 10.1038/s41586-020-2939-8. URL: https://doi.org/10.1038/s41586-020-2939-8.

[6] Greg Brockman et al. *OpenAI Gym*. 2016. arXiv: 1606.01540 [cs.LG].

[7] Yuri Burda et al. *Exploration by Random Network Distillation*. 2018. arXiv: 1810.12894 [cs.LG].

[8] Xinyue Chen et al. *Randomized Ensembled Double Q-Learning: Learning Fast Without a Model*. 2021. arXiv: 2101.05982 [cs.LG].

[9] Edoardo Conti et al. "Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents". In: *arXiv preprint arXiv:1712.06560* (2017).

[10] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. *Online Meta-learning by Parallel Algorithm Competition*. 2017. arXiv: 1702.07490 [cs.LG].

[11] Lasse Espeholt et al. *IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures*. 2018. arXiv: 1802.01561 [cs.LG].

[12] Benjamin Eysenbach and Sergey Levine. *If MaxEnt RL is the Answer, What is the Question?* 2019. arXiv: 1910.01913 [cs.LG].

[13] William Fedus et al. *Revisiting Fundamentals of Experience Replay*. 2020. arXiv: 2007.06700 [cs.LG].

[14] Jörg K. H. Franke et al. *Sample-Efficient Automated Deep Reinforcement Learning*. 2021. arXiv: 2009.01555 [cs.LG].

[15] Scott Fujimoto, Herke van Hoof, and David Meger. *Addressing Function Approximation Error in Actor-Critic Methods*. 2018. arXiv: 1802.09477 [cs.AI].

[16] Tuomas Haarnoja et al. *Soft Actor-Critic Algorithms and Applications*. 2019. arXiv: 1812.05905 [cs.LG].

[17] Tuomas Haarnoja et al. *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. 2018. arXiv: 1801.01290 [cs.LG].

[18] Xin He, Kaiyong Zhao, and Xiaowen Chu. "AutoML: A survey of the state-of-the-art". In: *Knowledge-Based Systems* 212 (Jan. 2021), p. 106622. ISSN: 0950-7051. DOI: 10.1016/j.knosys.2020.106622. URL: http://dx.doi.org/10.1016/j.knosys.2020.106622.

[19] Daniel Hein et al. *Introduction to the "Industrial Benchmark"*. 2017. arXiv: 1610.03793 [cs.LG].

[20] Peter Henderson et al. *Deep Reinforcement Learning that Matters*. 2019. arXiv: 1709.06560 [cs.LG].

[21] Matteo Hessel et al. *On Inductive Biases in Deep Reinforcement Learning*. 2019. arXiv: 1907.02908 [cs.LG].

[22] Christian D. Hubbs et al. *OR-Gym: A Reinforcement Learning Library for Operations Research Problems*. 2020. arXiv: 2008.06319 [cs.LG].

[23] Max Jaderberg et al. *Population Based Training of Neural Networks*. 2017. arXiv: 1711.09846 [cs.LG].

[24] Dmitry Kalashnikov et al. *QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation*. 2018. arXiv: 1806.10293 [cs.LG].

[25] Shivaram Kalyanakrishnan et al. *An Analysis of Frame-skipping in Reinforcement Learning*. 2021. arXiv: 2102.03718 [cs.LG].

[26] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[27] Aviral Kumar, Abhishek Gupta, and Sergey Levine. *DisCor: Corrective Feedback in Reinforcement Learning via Distribution Correction*. 2020. arXiv: 2003.07305 [cs.LG].

[28] Aravind S. Lakshminarayanan, Sahil Sharma, and Balaraman Ravindran. "Dynamic Action Repetition for Deep Reinforcement Learning". In: *AAAI*. 2017, pp. 2133–2139. URL: `http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14866`.

[29] Jongmin Lee, Byung-Jun Lee, and Kee-Eung Kim. "Reinforcement Learning for Control with Multiple Frequencies". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 3254–3264. URL: `https://proceedings.neurips.cc/paper/2020/file/216f44e2d28d4e175a194492bde9148f-Paper.pdf`.

[30] Kimin Lee et al. *SUNRISE: A Simple Unified Framework for Ensemble Learning in Deep Reinforcement Learning*. 2020. arXiv: `2007.04938 [cs.LG]`.

[31] Joel Lehman and Kenneth O Stanley. "Abandoning objectives: Evolution through the search for novelty alone". In: *Evolutionary computation* 19.2 (2011), pp. 189–223.

[32] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2015. arXiv: `1509.02971 [cs.LG]`.

[33] Alberto Maria Metelli et al. *Control Frequency Adaptation via Action Persistence in Batch Reinforcement Learning*. 2020. arXiv: `2002.06836 [cs.LG]`.

[34] Volodymyr Mnih et al. *Asynchronous Methods for Deep Reinforcement Learning*. 2016. arXiv: `1602.01783 [cs.LG]`.

[35] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 1, 2015), pp. 529–533. ISSN: 1476-4687. DOI: `10.1038/nature14236`. URL: `https://doi.org/10.1038/nature14236`.

[36] Fabio Pardo et al. *Time Limits in Reinforcement Learning*. 2018. arXiv: `1712.00378 [cs.LG]`.

[37] Supratik Paul, Vitaly Kurin, and Shimon Whiteson. *Fast Efficient Hyperparameter Tuning for Policy Gradients*. 2019. arXiv: `1902.06583 [cs.LG]`.

[38] Rongjun Qin et al. *NeoRL: A Near Real-World Benchmark for Offline Reinforcement Learning*. 2021. arXiv: `2102.00714 [cs.LG]`.

[39] Daniele Reda, Tianxin Tao, and Michiel van de Panne. "Learning to Locomote: Understanding How Environment Design Matters for Deep Reinforcement Learning". In: *Motion, Interaction and Games* (Oct. 2020). DOI: `10.1145/3424636.3426907`. URL: `http://dx.doi.org/10.1145/3424636.3426907`.

[40] John Schulman et al. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2015. arXiv: `1506.02438 [cs.LG]`.

[41] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: `1707.06347 [cs.LG]`.

[42] Lin Shao et al. *GRAC: Self-Guided and Self-Regularized Actor-Critic*. 2020. arXiv: `2009.08973 [cs.LG]`.

[43] Sahil Sharma, Aravind Srinivas, and Balaraman Ravindran. *Learning to Repeat: Fine Grained Action Repetition for Deep Reinforcement Learning*. 2020. arXiv: `1702.06054 [cs.LG]`.

[44] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. arXiv: `1712.01815 [cs.AI]`.

[45] Corentin Tallec, Léonard Blier, and Yann Ollivier. *Making Deep Q-learning methods robust to time discretization*. 2019. arXiv: `1901.09732 [cs.LG]`.

[46] Yunhao Tang and Krzysztof Choromanski. *Online Hyper-parameter Tuning in Off-policy Learning via Evolutionary Strategies*. 2020. arXiv: `2006.07554 [cs.LG]`.

[47] Yuval Tassa et al. *DeepMind Control Suite*. 2018. arXiv: `1801.00690 [cs.AI]`.

[48] 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*. 2012, pp. 5026–5033. DOI: `10.1109/IROS.2012.6386109`.

[49] Che Wang et al. *Striving for Simplicity and Performance in Off-Policy DRL: Output Normalization and Non-Uniform Sampling*. 2020. arXiv: `1910.02208 [cs.LG]`.

[50] Yufei Wang and Tianwei Ni. *Meta-SAC: Auto-tune the Entropy Temperature of Soft Actor-Critic via Metagradient*. 2020. arXiv: `2007.01932 [cs.LG]`.

[51] Denis Yarats and Ilya Kostrikov. *Soft Actor-Critic (SAC) implementation in PyTorch*. `https://github.com/denisyarats/pytorch_sac`. 2020.

[52] Haonan Yu, Wei Xu, and Haichao Zhang. *TASAC: Temporally Abstract Soft Actor-Critic for Continuous Control*. 2021. arXiv: `2104.06521 [cs.LG]`.

[53] Tom Zahavy et al. *A Self-Tuning Actor-Critic Algorithm*. 2021. arXiv: `2002.12928 [stat.ML]`.

A RL Notation

- (s, a, r, s', d). One transition of experience from the environment. Consists of a state s, the action selected by the behavior policy a, the reward returned by the environment r along with the next state s' and boolean d indicating the end of an episode.
- $\mathcal{A}$ is the action space. $|\mathcal{A}|$ refers to the dimension of the action space, or the number of elements in each action vector.
- γ. The discount factor that determines the agent's focus on long-term rewards. Gamma values approaching 1.0 place encourage long-horizon planning while smaller values prioritize greedy behavior. The discounted expected return is defined:

$$G(t) = \sum_{i=t}^{\infty} \gamma^i r_i \tag{4}$$

- π. The policy function mapping states to a distribution over actions.
- $Q_\pi(s, a)$. The state-action value function, representing the expected discounted returns starting in state s, taking action a and following policy π thereafter.
- $V(s)$. The value function, representing the expected discounted returns starting in state s and following policy π.

B Addressing Control Frequency with Persistence-Aware Actor-Critics

As discussed in Sec 2.2, the default control frequency of many environments is an arbitrary choice that can hinder optimization and exploration. When this is confronted in the literature, it is typically solved by making the action persistence a learned output of the actor or an additional set of discrete actions. However, the latter approach does not scale well with action size or maximum action repetition; providing a wide range of k values for the Atari domain, for example, would require dozens of additional actions, greatly increasing the complexity of exploration. Instead, we typically pick one or two higher k values above the single-step default. This replaces the control frequency hyperparameter with several new hyperparameters that likely require a grid search. Making k a direct output of the actor network is a better solution for continuous domains, but it comes with implementation challenges of its own. For one, the discount γ needs to be factored into the k action repetitions - otherwise, the agent will favor high persistence values because they allow for the undiscounted accumulation of rewards. This can make it difficult to adjust γ over the course of off-policy training. Furthermore, dynamic action repetition begins to trespass on the territory of hierarchical RL; a k-repetition policy can be formalized as a k-step *option* of a MDP. A policy that operates on an unpredictable timescale may complicate integration with higher-level policies.

We consider an alternative solution in which the action persistence value k is an element of the state vector. The actor and critic networks are able to interpret the current persistence and adapt accordingly. We can then vary the persistence throughout training and evaluate the same actor network on multiple k values in order to determine the appropriate setting at test time. We make another modification and have the environment return an array of rewards that correspond to each of the k possible timesteps. This lets us adjust the value of γ and recompute accurate TD targets during the critic update by discounting each element of the array and discounting the subsequent Q value by an additional timestep.

We demonstrate this technique in three tasks from the DeepMind Control Suite. "Learned Persistence" adds k as an additional element of the action space. "Incremental Schedule" methods use the naive approach of iterating through all reasonable k values during training. "Sampled Schedule" uses Thompson sampling over the returns at the previous evaluation period to pick k for the next training phase. "Delayed Sampled Schedule" begins by iterating through all k values as a crude exploration mechanism before sampling the setting later in training to avoid wasting time on clearly sub-optimal settings. All methods are evaluated on all k values and the highest return is reported. Results are shown in Figure 4.

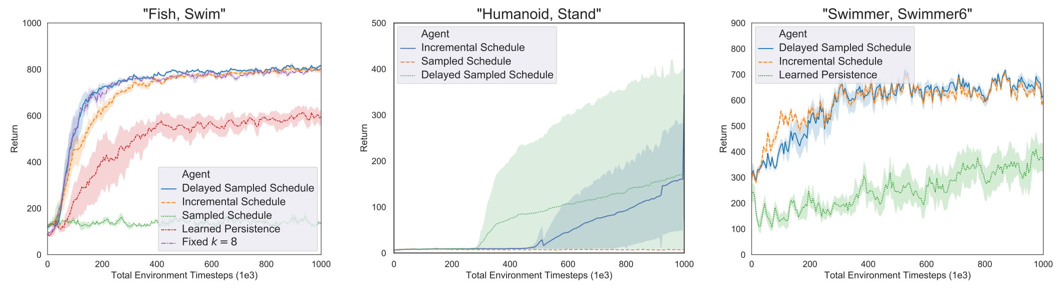

Figure 4: **Action Persistence Experiments.**

A side effect of this approach is that it allows the agent to generalize across control frequencies and therefore adapt to changes that may occur during deployment. If the training phase determines that the optimal action persistence is 5, for example, then a sensor malfunction or slowdown that cuts our control frequency in half can be compensated for by setting the persistence element of the agent's state vector to 10. This is still likely to decrease performance, but in our case the agent has seen values of 10 during training and is more capable of generalizing to the new situation. This effect is demonstrated by experiments in Figure 3.

C AAC Algorithm Details

C.1 Implementation Details

We initialize a population of 20 members with hyperparameters chosen uniformly from the range provided in Table 4. This range was determined by simple intuition about the range of reasonable parameter settings that we might otherwise grid search over. The only unintuitive choice is the difference between $c_{\max}$ and $a_{\max}$, which is based on the need for the self-regularized TD update (Eq 3) to perform many more critic updates than is standard. The ith member of the population has parameters $(\theta_i, \phi_i, a_i, c_i, h_i, k_i, g_i)$. We use the standard clipped-double-Q-trick [15] such that each agent actually has two critic networks ϕ_{1_i} and ϕ_{2_i}.

We collect $10,000$ random environment samples split evenly among the full range of k values to initialize the replay buffer. Each agent is trained in parallel, adding experience from the environment with an action persistence of k_i to a collective replay buffer that holds $2,000,000$ samples before overwriting the oldest experience[6]. Therefore each iteration of the AAC algorithm collects 20 new transitions.

During each training step, agent i updates its critic networks c_i times and its actor network a_i times. Each of these training steps samples a fresh batch of experience from the buffer with a batch size of 512 for DMC experiments and 128 otherwise. This is a slight divergence from the self-regularized heuristic of GRAC where the same batch is optimized repeatedly. We utilize an environment wrapper that returns an array of rewards $\hat{\mathbf{r}}$ representing the reward at each timestep up to the maximum possible k value. If $k_i < k_{\max}$ the extraneous entries are set to 0. When computing temporal difference targets, we multiply the jth entry of $\hat{\mathbf{r}}$ by γ^j and sum the resulting array to get the r term in Eq 1. We then multiply the $Q_{\phi_i}(s', a')$ term by γ^{k_i+1} to keep the time horizon consistent across different action repetitions. We also override terminal signals for episodes that end as a result of reaching the max step limit[7]. The α entropy coefficient is updated using the gradient descent technique from SAC [16] with target entropy $h_i(-|\mathcal{A}|)$ - this takes place inside the actor update function, meaning α is updated a_i times per training step.

Each agent continues to train independently for one evolutionary epoch. The length of each epoch is a new hyperparameter of our method. We set this length to be $1,000$ steps, which corresponds to 1 episode of training in the DeepMind Control Suite environments. This choice is arbitrary and likely to be sub-optimally short but achieves good results and reduces training time. At the end of each epoch, the population is evaluated. The fitness of each member in the population is set to the mean return across these evaluations. We sort the population by fitness and separate the best 20% and worst 20% of agents. The choice of 20% as a threshold is so arbitrary that attempting to tune it would be against the spirit of our "automatic" approach - this value is simply copied from Population Based

[6]We train the agent for a maximum of $2,000,000$ total samples, and only displayed the results after $1,000,000$, so experience is never overwritten in practice.

[7]This implementation detail is known as "infinite bootstrapping" and analyzed in [36].

Param	**Min**	**Max**	δ
a	1	10	2
c	1	40	5
h	0.25	1.75	0.25
k	1	15 (DMC, IB), 5 (OR-Gym)	2
g	-6.5	-1	0.5

Table 4: **Parameter search ranges.** See explanation of each in Sec 3.

Training [23] and was never changed during our research process. These "bad" and "elite" groups are randomly paired for evolutionary updates. Each pair copies the values $(\theta_i, \phi_i, a_i, c_i, g_i, h_i, k_i)$ along with the current α and Adam optimizer [26] settings from the elite agent to the bad agent. The hyperparameters a_i, c_i, g_i, h_i, k_i are then altered by adding a perturbation sampled uniformly from the range $[-\delta, \delta]$[8]. The values of δ for each parameter are also listed in Table 4. During our development process, these values were determined by guessing an appropriate range and then given a slight boost to generate a satisfactory shift in the parameter distributions over time.

Our synchronous implementation runs at around 2 iterations per second (T in Algorithm 1), leading to a training time of 7 hours for the main DeepMind Control Suite experiments. The population is split across 2 GPUs. At 2 GPUs and 7 hours per trial for at least 3 trials in 9 environments, the AAC results in this paper consume approximately 378 GPU hours. The SAC, Rand-SAC and k-SAC baselines train in roughly 3 hours on a single GPU. With 3 algorithms in 9 environments running for 5 random seeds[9], the baselines consume approximately 216 GPU hours.

D Baseline Implementation Details

Our default hyperparameters for SAC and SR-SAC are listed in Table 5. These settings are chosen based on [51], [15], [16], and other publicly available implementations.

Param	**Value**
Batch Size	128, 512 (DMC)
τ	.005
actor lr	$3e$-4
critic lr	$3e$-4
α lr	$1e$-4
γ	0.99
Warmup Steps	1000
Target Delay	2
Critic Updates Per Step	1
Actor Updates Per Step	1
SR β Init	90
SR β Final	70
H	$-\lvert\mathcal{A}\rvert$
Architecture	256, ReLU, 256, ReLU
Action Log Std Range	$(-10, 2)$

Table 5: **Standard hyperparameters for SAC, SR-SAC, and k-SAC used in our experiments.**

[8]The perturbations for integer hyperparameters like a, c and k are sampled uniformly from the integers in the range $(-\delta, \delta)$.

[9]We use 15 random seeds for Rand-SAC, which has much higher variance by design.