WHAT MATTERS FOR ON-POLICY DEEP ACTOR-CRITIC METHODS? A LARGE-SCALE STUDY

Anonymous authors
Paper under double-blind review

ABSTRACT

In recent years, reinforcement learning (RL) has been successfully applied to many different continuous control tasks. While RL algorithms are often conceptually simple, their state-of-the-art implementations take numerous low- and high-level design decisions that strongly affect the performance of the resulting agents. Those choices are usually not extensively discussed in the literature, leading to discrepancy between published descriptions of algorithms and their implementations. This makes it hard to attribute progress in RL and slows down overall progress [27]. As a step towards filling that gap, we implement >50 such "choices" in a unified on-policy deep actor-critic framework, allowing us to investigate their impact in a large-scale empirical study. We train over 250'000 agents in five continuous control environments of different complexity and provide insights and practical recommendations for the training of on-policy deep actor-critic RL agents.

1 INTRODUCTION

Deep reinforcement learning (RL) has seen increased interest in recent years due to its ability to have neural-network-based agents learn to act in environments through interactions. For continuous control tasks, on-policy algorithms such as REINFORCE [2], TRPO [10], A3C [14], PPO [17] and off-policy algorithms such as DDPG [13] and SAC [21] have enabled successful applications such as quadrupedal locomotion [20], self-driving [30] or dexterous in-hand manipulation [20, 25, 32].

Many of these papers investigate in depth different loss functions and learning paradigms. Yet, it is less visible that behind successful experiments in deep RL there are complicated code bases that contain a large number of low- and high-level design decisions that are usually not discussed in research papers. While one may assume that such "choices" do not matter, there is some evidence that they are in fact crucial for or even driving good performance [27].

While there are open-source implementations available that can be used by practitioners, this is still unsatisfactory: In research publications, often different algorithms implemented in different code bases are compared one-to-one. This makes it impossible to assess whether improvements are due to the algorithms or due to their implementations. Furthermore, without an understanding of lower-level choices, it is hard to assess the performance of high-level algorithmic choices as performance may strongly depend on the tuning of hyperparameters and implementation-level details. Overall, this makes it hard to attribute progress in RL and slows down further research [15, 22, 27].

Our contributions. Our key goal in this paper is to investigate such lower level choices in depth and to understand their impact on final agent performance. Hence, as our key contributions, we (1) implement >50 choices in a unified on-policy deep actor-critic implementation, (2) conducted a large-scale (more than 250'000 agents trained) experimental study that covers different aspects of the training process, and (3) analyze the experimental results to provide practical insights and recommendations for the training of on-policy deep actor-critic RL agents.

Most surprising finding. While many of our experimental findings confirm common RL practices, some of them are quite surprising, e.g. the policy initialization scheme significantly influences the performance while it is rarely even mentioned in RL publications. In particular, we have found that initializing the network so that the initial action distribution has zero mean, a rather low standard deviation and is independent of the observation significantly improves the training speed (Sec. 3.2).

The rest of of this paper is structured as follows: We describe our experimental setup and performance metrics used in Sec. 2. Then, in Sec. 3 we present and analyse the experimental results and finish with related work in Sec. 4 and conclusions in Sec. 5. The appendices contain the detailed description of all design choices we experiment with (App. B), default hyperparameters (App. C) and the raw experimental results (App. D - K).

2 STUDY DESIGN

Considered setting. In this paper, we consider the setting of *on-policy deep actor-critic reinforcement learning for continuous control*. We define on-policy learning in the following loose sense: We consider policy iteration algorithms that iterate between generating experience using the current policy and using the experience to improve the policy. This is the standard *modus operandi* of algorithms usually considered on-policy such as PPO [17]. However, we note that algorithms often perform several model updates and thus may operate technically on off-policy data within a single policy improvement iteration. As benchmark environments, we consider five widely used continuous control environments from OpenAI Gym [12] of varying complexity: Hopper-v1, Walker2d-v1, HalfCheetah-v1, Ant-v1, and Humanoid-v1 [1].

Unified on-policy deep actor-critic gradient algorithm. We took the following approach to create a highly configurable unified on-policy deep actor-critic gradient algorithm with as many choices as possible:

1. We researched prior work and popular code bases to make a list of commonly used choices, i.e., different loss functions (both for value functions and policies), architectural choices such as initialization methods, heuristic tricks such as gradient clipping and all their corresponding hyperparameters.
2. Based on this, we implemented a single, unified on-policy deep actor-critic agent and corresponding training protocol starting from the SEED RL code base [28]. Whenever we were faced with implementation decisions that required us to take decisions that could not be clearly motivated or had alternative solutions, we further added such decisions as additional choices.
3. We verified that when all choices are selected as in the PPO implementation from OpenAI baselines, we obtain similar performance as reported in the PPO paper [17]. We chose PPO because it is probably the most commonly used on-policy deep actor-critic RL algorithm at the moment.

The resulting agent implementation is detailed in Appendix B. The key property is that the implementation exposes all choices as configuration options in an unified manner. For convenience, we mark each of the choice in this paper with a number (e.g., `C1`) and a fixed name (e.g. `num_envs (C1)`) that can be easily used to find a description of the choice in Appendix B.

Difficulty of investigating choices. The primary goal of this paper is to understand how the different choices affect the final performance of an agent and to derive recommendations for these choices. There are two key reasons why this is challenging:

First, we are mainly interested in insights on choices for good hyperparameter configurations. Yet, if all choices are sampled randomly, the performance is very bad and little (if any) training progress is made. This may be explained by the presence of sub-optimal settings (e.g., hyperparameters of the wrong scale) that prohibit learning at all. If there are many choices, the probability of such failure increases exponentially.

Second, many choices may have strong interactions with other related choices, for example the learning rate and the minibatch size. This means that such choices need to be tuned together and experiments where only a single choice is varied but interacting choices are kept fixed may be misleading.

[1] It has been noticed that the version of the Mujoco physics simulator [5] can slightly influence the behaviour of some of the environments — `https://github.com/openai/gym/issues/1541`. We used Mujoco 2.0 in our experiments.

Basic experimental design. To address these issues, we design a series of experiments as follows: We create groups of choices around thematic groups where we suspect interactions between different choices, for example we group together all choices related to neural network architecture. We also include `Adam learning rate (C24)` in all of the groups as we suspect that it may interact with many other choices.

Then, in each experiment, we train a large number of models where we randomly sample the choices within the corresponding group [2]. All other settings (for choices not in the group) are set to settings of a competitive base configuration (detailed in Appendix C) that is close to the default PPOv2 configuration[3] scaled up to 256 parallel environments. This has two effects: First, it ensures that our set of trained models contains good models (as verified by performance statistics in the corresponding results). Second, it guarantees that we have models that have different combinations of potentially interacting choices.

We consider two different analyses for each choice (e.g, for `advantage_estimator (C6)`):

Conditional 95th percentile: For each potential value of that choice (e.g., `advantage_estimator (C6)` = `N-Step`), we look at the performance distribution of sampled configurations with that value. We report the 95th percentile of the performance as well as a confidence interval based on a binomial approximation [4]. Intuitively, this corresponds to a robust estimate of the performance one can expect if all other choices in the group were tuned with random search and a limited budget of roughly 20 hyperparameter configurations.

Distribution of choice within top 5% configurations. We further consider for each choice the distribution of values among the top 5% configurations trained in that experiment. The reasoning is as follows: By design of the experiment, values for each choice are distributed uniformly at random. Thus, if certain values are over-represented in the top models, this indicates that the specific choice is important in guaranteeing good performance.

Performance measures. We employ the following way to compute performance: For each hyperparameter configuration, we train 3 models with independent random seeds where each model is trained for one million (Hopper, HalfCheetah, Walker2d) or two million environment steps (Ant, Humanoid). We evaluate trained policies every hundred thousand steps by freezing the policy and computing the average undiscounted episode return of 100 episodes (with the stochastic policy). We then average these score to obtain a single performance score of the seed which is proportional to the area under the learning curve. This ensures we assign higher scores to agents that learn quickly. The performance score of a hyperparameter configuration is finally set to the median performance score across the 3 seeds. This reduces the impact of training noise, i.e., that certain seeds of the same configuration may train much better than others.

3 EXPERIMENTS

We run experiments for eight thematic groups: *Policy Losses* (Sec. 3.1), *Networks architecture* (Sec. 3.2), *Normalization and clipping* (Sec. 3.3), *Advantage Estimation* (Sec. 3.4), *Training setup* (Sec. 3.5), *Timesteps handling* (Sec. 3.6), *Optimizers* (Sec. 3.7), and *Regularization* (Sec. 3.8). For each group, we provide a full experimental design and full experimental plots in Appendices D - K so that the reader can draw their own conclusions from the experimental results. In the following sections, we provide short descriptions of the experiments, our interpretation of the results, as well as practical recommendations for agent training for continuous control.

[2] Exact details for the different experiments are provided in Appendices D - K.

[3] `https://github.com/openai/baselines/blob/master/baselines/ppo2/defaults.py`

[4] We compute confidence intervals with a significance level of $\alpha = 5\%$ as follows: We find $i_l = icdf\left(\frac{\alpha}{2}\right)$ and $i_h = icdf\left(1 - \frac{\alpha}{2}\right)$ where $icdf$ is the inverse cumulative density function of a binomial distribution with $p = 0.95$ (as we consider the 95th percentile) and the number of draws equals the number of samples. We then report the i_lth and i_hth highest scores as the confidence interval.

3.1 POLICY LOSSES (BASED ON THE RESULTS IN APPENDIX D)

Study description. We investigate different policy losses (C14): vanilla policy gradient (PG), V-trace [19], PPO [17], AWR [33], V-MPO[5] [34] and the limiting case of AWR ($\beta \to 0$) and V-MPO ($\epsilon_n \to 0$) which we call Repeat Positive Advantages (RPA) as it is equivalent to the negative log-probability of actions with positive advantages. See App. B.3 for a detailed description of the different losses. We further sweep the hyperparameters of each of the losses (C15, C16, C18, C17, C19), the learning rate (C24) and the number of passes over the data (C3).

The goal of this study is to better understand the importance of the policy loss function in the on-policy deep actor-critic setting considered in this paper. The goal is **not** to provide a general statement that one of the losses is better than the others as some of them were specifically designed for other settings (e.g., the V-trace loss is targeted at near-on-policy data in a distributed setting).

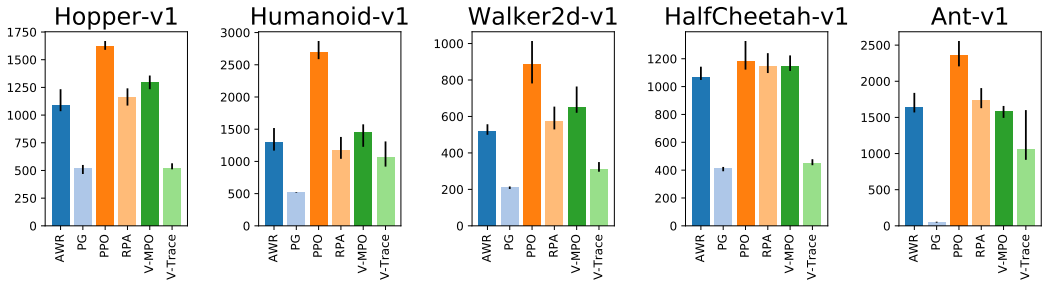

Figure 1: Comparison of different policy losses (C14).

Interpretation. Fig. 1 shows the 95-th percentile of the average policy score during training for different policy losses (C14). We observe that PPO performs better than the other losses on 4 out of 5 environments and is one of the top performing losses on HalfCheetah. As we randomly sample the loss specific hyperparameters in this analysis, one might argue that our approach favours choices that are not too sensitive to hyperparameters. At the same time, there might be losses that are sensitive to their hyperparameters but for which good settings may be easily found. Fig. 5 shows that even if we condition on choosing the optimal loss hyperparameters for each loss[6], PPO still outperforms the other losses on the two hardest tasks — Humanoid and Ant[7] and is one of the top performing losses on the other 3 tasks. Moreover, we show the empirical cumulative density functions of agent performance conditioned on the policy loss used in Fig. 4. Perhaps unsurprisingly, PG and V-trace perform worse on all tasks. This is likely caused by their inability to handle data that become off-policy in one iteration, either due to multiple passes (C3) over experience (which can be seen in Fig. 14) or a large experience buffer (C2) in relation to the batch size (C4). Overall, these results show that trust-region optimization (preventing the current policy from diverging too much from the behavioral one) which is present in all the other policy losses is crucial for good sample complexity. For PPO and its clipping threshold ϵ (C16), we further observe that $\epsilon = 0.2$ and $\epsilon = 0.3$ perform reasonably well in all environments but that lower ($\epsilon = 0.1$) or higher ($\epsilon = 0.5$) values give better performance on some of the environments (See Fig. 10 and Fig. 32).

Recommendation. Use the PPO policy loss. Start with the clipping threshold set to 0.25 but also try lower and higher values if possible.

3.2 NETWORKS ARCHITECTURE (BASED ON THE RESULTS IN APPENDIX E)

Study description. We investigate the impact of differences in the policy and value function neural network architectures. We consider choices related to the network structure and size (C47, C48,

[5]We used the V-MPO policy loss without the decoupled KL constraint as we investigate the effects of different policy regularizers separately in Sec. 3.8.

[6]AWR loss has two hyperparameters — the temperature β (C18) and the weight clipping coefficient $\omega_{\max}$ (C17). We only condition on β which is more important.

[7]These two tasks were not included in the original PPO paper [17] so the hyperparameters we use were not tuned for them.

C49, C50, C51, C52, C52), activation functions (C55), and initialization of network weights (C56, C57, C58). We further include choices related to the standard deviation of actions (C59, C60, C61, C62) and transformations of sampled actions (C63).

Interpretation. Separate value and policy networks (C47) appear to lead to better performance on four out of five environments (Fig. 15). To avoid analyzing the other choices based on bad models, we thus focus for the rest of this experiment only on agents with separate value and policy networks. Regarding network sizes, the optimal width of the policy MLP depends on the complexity of the environment (Fig. 18) and too low or too high values can cause significant drop in performance while for the value function there seems to be no downside in using wider networks (Fig. 21). Moreover, on some environments it is beneficial to make the value network wider than the policy one, e.g. on HalfCheetah the best results are achieved with $16 - 32$ units per layer in the policy network and 256 in the value network. Two hidden layers appear to work well for policy (Fig. 22) and value networks (Fig. 20) in all tested environments. As for activation functions, we observe that tanh activations perform best and relu worst. (Fig. 30).

Interestingly, the initial policy appears to have a surprisingly high impact on the training performance. The key recipe appears is to initialize the policy at the beginning of training so that the action distribution is centered around 0[8] regardless of the observation and has a rather small standard deviation. This can be achieved by initializing the policy MLP with smaller weights in the last layer (C57, Fig. 24, this alone boosts the performance on Humanoid by 66%) so that the initial action distribution is almost independent of the observation and by introducing an offset in the standard deviation of actions (C61). Fig. 2 shows that the performance is very sensitive to the initial action standard deviation with 0.5 performing best on all environments except Hopper where higher values perform better.

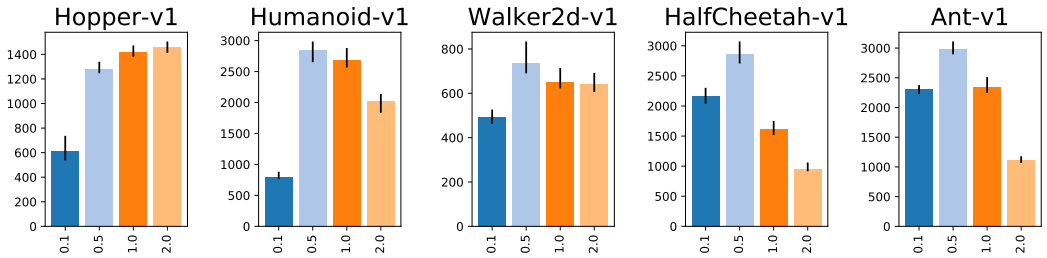

Figure 2: Comparison of different initial standard deviations of actions (C61).

Fig. 17 compares two approaches to transform unbounded sampled actions into the bounded $[-1, 1]$ domain expected by the environment (C63): clipping and applying a tanh function. tanh performs slightly better overall (in particular it improves the performance on HalfCheetah by 30%). Comparing Fig. 17 and Fig. 2 suggests that the difference might be mostly caused by the decreased magnitude of initial actions[9].

Other choices appear to be less important: The scale of the last layer initialization matters much less for the value MLP (C58) than for the policy MLP (Fig. 19). Apart from the last layer scaling, the network initialization scheme (C56) does not matter too much (Fig. 27). Only he_normal and he_uniform [7] appear to be suboptimal choices with the other options performing very similarly. There also appears to be no clear benefits if the standard deviation of the policy is learned for each state (i.e. outputted by the policy network) or once globally for all states (C59, Fig. 23). For the transformation of policy output into action standard deviation (C60), softplus and exponentiation perform very similarly[10] (Fig. 25). Finally, the minimum action standard deviation (C62) seems to matter little, if it is not set too large (Fig. 30).

[8] All environments expect normalized actions in $[-1, 1]$.

[9] tanh can also potentially perform better with entropy regularization (not used in this experiment) as it bounds the maximum possible policy entropy.

[10] We noticed that some of the training runs with exponentiation resulted in NaNs but clipping the exponent solves this issue (See Sec. B.8 for the details).

Recommendation. Initialize the last policy layer with $100\times$ smaller weights. Use `softplus` to transform network output into action standard deviation and add a (negative) offset to its input to decrease the initial standard deviation of actions. Tune this offset if possible. Use `tanh` both as the activation function (if the networks are not too deep) and to transform the samples from the normal distribution to the bounded action space. Use a wide value MLP (no layers shared with the policy) but tune the policy width (it might need to be narrower than the value MLP).

3.3 Normalization and clipping (based on the results in Appendix F)

Study description. We investigate the impact of different normalization techniques: observation normalization (`C64`), value function normalization (`C66`), per-minibatch advantage normalization (`C67`), as well as gradient (`C68`) and observation (`C65`) clipping.

Interpretation. Input normalization (`C64`) is crucial for good performance on all environments apart from Hopper (Fig. 33). Quite surprisingly, value function normalization (`C66`) also influences the performance very strongly — it is crucial for good performance on HalfCheetah and Humanoid, helps slightly on Hopper and Ant and significantly *hurts* the performance on Walker2d (Fig. 37). We are not sure why the value function scale matters that much but suspect that it affects the performance by changing the speed of the value function fitting.[11] In contrast to observation and value function normalization, per-minibatch advantage normalization (`C67`) seems not to affect the performance too much (Fig. 35). Similarly, we have found little evidence that clipping normalized[12] observations (`C65`) helps (Fig. 38) but it might be worth using if there is a risk of extremely high observations due to simulator divergence. Finally, gradient clipping (`C68`) provides a small performance boost with the exact clipping threshold making little difference (Fig. 34).

Recommendation. Always use observation normalization and check if value function normalization improves performance. Gradient clipping might slightly help but is of secondary importance.

3.4 Advantage Estimation (based on the results in Appendix G)

Study description. We compare the most commonly used advantage estimators (`C6`): N-step [3], GAE [9] and V-trace [19] and their hyperparameters (`C7`, `C8`, `C9`, `C10`). We also experiment with applying PPO-style pessimistic clipping (`C13`) to the value loss (present in the original PPO implementation but not mentioned in the PPO paper [17]) and using Huber loss [1] instead of MSE for value learning (`C11`, `C12`). Moreover, we varied the number of parallel environments used (`C1`) as it changes the length of the experience fragments collected in each step.

Interpretation. GAE and V-trace appear to perform better than N-step returns (Fig. 44 and 40) which indicates that it is beneficial to combine the value estimators from multiple timesteps. We have not found a significant performance difference between GAE and V-trace in our experiments. $\lambda = 0.9$ (`C8`, `C9`) performed well regardless of whether GAE (Fig. 45) or V-trace (Fig. 49) was used on all tasks but tuning this value per environment may lead to modest performance gains. We have found that PPO-style value loss clipping (`C13`) hurts the performance regardless of the clipping threshold[13] (Fig. 43). Similarly, the Huber loss (`C11`) performed worse than MSE in all environments (Fig. 42) regardless of the value of the threshold (`C12`) used (Fig. 48).

Recommendation. Use GAE with $\lambda = 0.9$ but neither Huber loss nor PPO-style value loss clipping.

3.5 Training setup (based on the results in Appendix H)

Study description. We investigate choices related to the data collection and minibatch handling: the number of parallel environments used (`C1`), the number of transitions gathered in each iteration (`C2`), the number of passes over the data (`C3`), minibatch size (`C4`) and how the data is split into minibatches (`C5`).

[11] Another explanation could be the interaction between the value function normalization and PPO-style value clipping (`C13`). We have, however, disable the value clipping in this experiment to avoid this interaction. The disabling of the value clipping could also explain why our conclusions are different from [27] where a form of value normalization improved the performance on Walker.

[12] We only applied clipping if input normalization was enabled.

[13] This is consistent with prior work [27].

For the last choice, in addition to standard choices, we also consider a new small modification of the original PPO approach: The original PPO implementation splits the data in each policy iteration step into individual transitions and then randomly assigns them to minibatches (C5). This makes it impossible to compute advantages as the temporal structure is broken. Therefore, the advantages are computed once at the beginning of each policy iteration step and then used in minibatch policy and value function optimization. This results in higher diversity of data in each minibatch at the cost of using slightly stale advantage estimations. As a remedy to this problem, we propose to recompute the advantages at the beginning of each pass over the data instead of just once per iteration.

Results. Unsurprisingly, going over the experience multiple times appears to be crucial for good sample complexity (Fig. 54). Often, this is computationally cheap due to the simple models considered, in particular on machines with accelerators such as GPUs and TPUs. As we increase the number of parallel environments (C1), performance decreases sharply on some of the environments (Fig. 55). This is likely caused by shortened experience chunks (See Sec. B.1 for the detailed description of the data collection process) and earlier value bootstrapping. Despite that, training with more environments usually leads to faster training in wall-clock time if enough CPU cores are available. Increasing the batch size (C4) does not appear to hurt the sample complexity in the range we tested (Fig. 57) which suggests that it should be increased for faster iteration speed. On the other hand, the number of transitions gathered in each iteration (C2) influences the performance quite significantly (Fig. 52). Finally, we compare different ways to handle minibatches (See Sec. B.1 for the detailed description of different variants) in Fig. 53 and 58. The plots suggest that stale advantages can in fact hurt performance and that recomputing them at the beginning of each pass at least partially mitigates the problem and performs best among all variants.

Recommendation. Go over experience multiple times. Shuffle individual transitions before assigning them to minibatches and recompute advantages once per data pass (See App. B.1 for the details). For faster wall-clock time training use many parallel environments and increase the batch size (both might hurt the sample complexity). Tune the number of transitions in each iteration (C2) if possible.

3.6 Timesteps handling (based on the results in Appendix I)

Study description. We investigate choices related to the handling of timesteps: discount factor[14] (C20), frame skip (C21), and how episode termination due to timestep limits are handled (C22). The latter relates to a technical difficulty explained in App. B.4 where one assumes for the algorithm an infinite time horizon but then trains using a finite time horizon [16].

Interpretation. Fig. 60 shows that the performance depends heavily on the discount factor γ (C20) with $\gamma = 0.99$ performing reasonably well in all environments. Skipping every other frame (C21) improves the performance on 2 out of 5 environments (Fig. 61). Proper handling of episodes abandoned due to the timestep limit seems not to affect the performance (C22, Fig. 62) which is probably caused by the fact that the timestep limit is quite high (1000 transitions) in all the environments we considered.

Recommendation. Discount factor γ is one of the most important hyperparameters and should be tuned per environment (start with $\gamma = 0.99$). Try frame skip if possible. There is no need to handle environments step limits in a special way for large step limits.

3.7 Optimizers (based on the results in Appendix J)

Study description. We investigate two gradient-based optimizers commonly used in RL: (C23) – Adam [8] and RMSprop – as well as their hyperparameters (C24, C25, C26, C27, C28, C29, C30) and a linear learning rate decay schedule (C31).

Interpretation. The differences in performance between the optimizers (C23) appear to be rather small with no optimizer consistently outperforming the other across environments (Fig. 66). Unsurprisingly, the learning rate influences the performance very strongly (Fig. 69) with the default value of 0.0003 for Adam (C24) performing well on all tasks. Fig. 67 shows that Adam works better with momentum (C26). For RMSprop, momentum (C27) makes less difference (Fig. 71) but our results

[14] While the discount factor is sometimes treated as a part of the environment, we assume that the real goal is to maximize *undiscounted* returns and the discount factor is a part of the algorithm which makes learning easier.

suggest that it might slightly improve performance[15]. Whether the centered or uncentered version of RMSprop is used (C30) makes no difference (Fig. 70) and similarly we did not find any difference between different values of the ϵ coefficients (C28, C29, Fig. 68 and 72). Linearly decaying the learning rate to 0 increases the performance on 4 out of 5 tasks but the gains are very small apart from Ant, where it leads to 15% higher scores (Fig. 65).

Recommendation. Use Adam [8] optimizer with momentum $\beta_1 = 0.9$ and a tuned learning rate (0.0003 is a safe default). Linearly decaying the learning rate may slightly improve performance but is of secondary importance.

3.8 Regularization (based on the results in Appendix K)

Study description. We investigate different policy regularizers (C32), which can have either the form of a penalty (C33, e.g. bonus for higher entropy) or a soft constraint (C34, e.g. entropy should not be lower than some threshold) which is enforced with a Lagrange multiplier. In particular, we consider the following regularization terms: entropy (C40, C46), the Kullback–Leibler divergence (KL) between a reference $\mathcal{N}(0, 1)$ action distribution and the current policy (C37, C43) and the KL divergence and reverse KL divergence between the current policy and the behavioral one (C35, C41, C36, C42), as well as the "decoupled" KL divergence from [18, 34] (C38, C39, C44, C45).

Interpretation. We do not find evidence that any of the investigated regularizers helps significantly on our environments with the exception of HalfCheetah on which all constraints (especially the entropy constraint) help (Fig. 76 and 77). However, the performance boost is largely independent on the constraint threshold (Fig. 83, 84, 87, 89, 90 and 91) which suggests that the effect is caused by the initial high strength of the penalty (before it gets adjusted) and not by the desired constraint. While it is a bit surprising that regularization does not help at all (apart from HalfCheetah), we conjecture that regularization might be less important in our experiments because: (1) the PPO policy loss already enforces the trust region which makes KL penalties or constraints redundant; and (2) the careful policy initialization (See Sec. 3.2) is enough to guarantee good exploration and makes the entropy bonus or constraint redundant.

4 Related Work

Islam et al. [15] and Henderson et al. [22] point out the reproducibility issues in RL including the performance differences between different code bases, the importance of hyperparameter tuning and the high level of stochasticity due to random seeds. Tucker et al. [26] showed that the gains, which had been attributed to one of the recently proposed policy gradients improvements, were, in fact, caused by the implementation details. The most closely related work to ours is probably Engstrom et al. [27] where the authors investigate code-level improvements in the PPO [17] code base and conclude that they are responsible for the most of the performance difference between PPO and TRPO [10]. Our work is also similar to other large-scale studies done in other fields of Deep Learning, e.g. model-based RL [31], GANs [24], NLP [35], disentangled representations [23] and convolution network architectures [36].

5 Conclusions

In this paper, we investigated the importance of a broad set of high- and low-level choices that need to be made when designing and implementing on-policy deep actor-critic RL algorithms. Based on more than 250'000 experiments in five continuous control environments, we evaluate the impact of different choices and provide practical recommendations. One of the surprising insights is that the initial action distribution plays an important role in agent performance. We expect this to be a fruitful avenue for future research.

[15] Importantly, switching from no momentum to momentum 0.9 increases the RMSprop step size by approximately $10\times$ and requires an appropriate adjustment to the learning rate (Fig. 74).

REFERENCES

[1] Peter J Huber. "Robust estimation of a location parameter". In: *Breakthroughs in statistics*. Springer, 1992, pp. 492–518.

[2] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[3] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*. Vol. 135. MIT press Cambridge, 1998.

[4] Brian D Ziebart. "Modeling purposeful adaptive behavior with the principle of maximum causal entropy". In: (2010).

[5] 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*. IEEE. 2012, pp. 5026–5033.

[6] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[7] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[8] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015. URL: `http://arxiv.org/abs/1412.6980`.

[9] John Schulman et al. "High-dimensional continuous control using generalized advantage estimation". In: *arXiv preprint arXiv:1506.02438* (2015).

[10] John Schulman et al. "Trust region policy optimization". In: *International conference on machine learning*. 2015, pp. 1889–1897.

[11] Martın Abadi et al. "Tensorflow: A system for large-scale machine learning". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 265–283.

[12] Greg Brockman et al. "Openai gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[13] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *International Conference on Learning Representations*. 2016.

[14] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International conference on machine learning*. 2016, pp. 1928–1937.

[15] Riashat Islam et al. "Reproducibility of benchmarked deep reinforcement learning tasks for continuous control". In: *arXiv preprint arXiv:1708.04133* (2017).

[16] Fabio Pardo et al. "Time limits in reinforcement learning". In: *arXiv preprint arXiv:1712.00378* (2017).

[17] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[18] Abbas Abdolmaleki et al. "Maximum a posteriori policy optimisation". In: *arXiv preprint arXiv:1806.06920* (2018).

[19] Lasse Espeholt et al. "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures". In: *International Conference on Machine Learning*. 2018, pp. 1406–1415.

[20] Tuomas Haarnoja et al. "Soft actor-critic algorithms and applications". In: *arXiv preprint arXiv:1812.05905* (2018).

[21] Tuomas Haarnoja et al. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". In: *International Conference on Machine Learning*. 2018, pp. 1861–1870.

[22] Peter Henderson et al. "Deep reinforcement learning that matters". In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[23] Francesco Locatello et al. "Challenging common assumptions in the unsupervised learning of disentangled representations". In: *arXiv preprint arXiv:1811.12359* (2018).

[24] Mario Lucic et al. "Are gans created equal? a large-scale study". In: *Advances in neural information processing systems*. 2018, pp. 700–709.

[25] M Andrychowicz OpenAI et al. "Learning dexterous in-hand manipulation". In: *arXiv preprint arXiv:1808.00177* (2018).

[26] George Tucker et al. "The mirage of action-dependent baselines in reinforcement learning". In: *arXiv preprint arXiv:1802.10031* (2018).

[27] Logan Engstrom et al. "Implementation Matters in Deep RL: A Case Study on PPO and TRPO". In: *International Conference on Learning Representations*. 2019.

[28] Lasse Espeholt et al. "SEED RL: Scalable and Efficient Deep-RL with Accelerated Central Inference". In: *arXiv preprint arXiv:1910.06591* (2019).

[29] Michael Janner et al. "When to trust your model: Model-based policy optimization". In: *Advances in Neural Information Processing Systems*. 2019, pp. 12498–12509.

[30] Alex Kendall et al. "Learning to drive in a day". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 8248–8254.

[31] Eric Langlois et al. "Benchmarking model-based reinforcement learning". In: *arXiv preprint arXiv:1907.02057* (2019).

[32] Ilge OpenAI et al. "Solving Rubik's Cube with a Robot Hand". In: *arXiv preprint arXiv:1910.07113* (2019).

[33] Xue Bin Peng et al. "Advantage-Weighted Regression: Simple and Scalable Off-Policy Reinforcement Learning". In: *arXiv preprint arXiv:1910.00177* (2019).

[34] H Francis Song et al. "V-MPO: On-Policy Maximum a Posteriori Policy Optimization for Discrete and Continuous Control". In: *arXiv preprint arXiv:1909.12238* (2019).

[35] Jared Kaplan et al. "Scaling laws for neural language models". In: *arXiv preprint arXiv:2001.08361* (2020).

[36] Ilija Radosavovic et al. "Designing Network Design Spaces". In: *arXiv preprint arXiv:2003.13678* (2020).

A REINFORCEMENT LEARNING BACKGROUND

We consider the standard reinforcement learning formalism consisting of an agent interacting with an environment. To simplify the exposition we assume in this section that the environment is fully observable. An environment is described by a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, a distribution of initial states $p(s_0)$, a reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, transition probabilities $p(s_{t+1}|s_t, a_t)$ (t is a timestep index explained later), termination probabilities $T(s_t, a_t)$ and a discount factor $\gamma \in [0, 1]$.

A policy π is a mapping from state to a distribution over actions. Every episode starts by sampling an initial state s_0. At every timestep t the agent produces an action based on the current state: $a_t \sim \pi(\cdot|s_t)$. In turn, the agent receives a reward $r_t = r(s_t, a_t)$ and the environment's state is updated. With probability $T(s_t, a_t)$ the episode is terminated, and otherwise the new environments state s_{t+1} is sampled from $p(\cdot|s_t, a_t)$. The discounted sum of future rewards, also referred to as the *return*, is defined as $R_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_i$. The agent's goal is to find the policy π which maximizes the expected return $\mathbb{E}_\pi[R_0|s_0]$, where the expectation is taken over the initial state distribution, the policy, and environment transitions accordingly to the dynamics specified above. The *Q-function* or *action-value* function of a given policy π is defined as $Q^\pi(s_t, a_t) = \mathbb{E}_\pi[R_t|s_t, a_t]$, while the *V-function* or *state-value* function is defined as $V^\pi(s_t) = \mathbb{E}_\pi[R_t|s_t]$. The value $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$ is called the *advantage* and tells whether the action a_t is better or worse than an average action the policy π takes in the state s_t.

In practice, the policies and value functions are going to be represented as neural networks. In particular, RL algorithms we consider maintain two neural networks: one representing the current policy π and a value network which approximates the value function of the current policy $V \approx V^\pi$.

B LIST OF INVESTIGATED CHOICES

In this section we list all algorithmic choices which we consider in our experiments. See Sec. A for a very brief introduction to RL and the notation we use in this section.

B.1 DATA COLLECTION AND OPTIMIZATION LOOP

RL algorithms interleave running the current policy in the environment with policy and value function networks optimization. In particular, we create `num_envs (C1)` environments [14]. In each iteration, we run all environments synchronously sampling actions from the current policy until we have gathered `iteration_size (C2)` transitions total (this means that we have `num_envs (C1)` experience fragments, each consisting of `iteration_size (C2)` / `num_envs (C1)` transitions). Then, we perform `num_epochs (C3)` epochs of minibatch updates where in each epoch we split the data into minibatches of size `batch_size (C4)`, and performing gradient-based optimization [17]. Going over collected experience multiple times means that it is not strictly an on-policy RL algorithm but it may increase the sample complexity of the algorithm at the cost of more computationally expensive optimization step.

We consider four different variants of the above scheme (choice `C5`):

- `Fixed trajectories`: Each minibatch consists of full experience fragments and in each epoch we go over exactly the same minibatches in the same order.
- `Shuffle trajectories`: Like `Fixed trajectories` but we randomly assign full experience fragments to minibatches in each epoch.
- `Shuffle transitions`: We break experience fragments into individual transitions and assign them randomly to minibatches in each epoch. This makes the estimation of advantages impossible in each minibatch (most of the advantage estimators use future states, See App. B.2) so we precompute all advantages at the beginning of *each iteration* using full experience fragments. This approach leads to higher diversity of data in each minibatch at the price of somewhat stale advantage estimations. The original PPO implementation from OpenAI Baselines[16] works this way but this is not mentioned in the PPO paper [17].
- `Shuffle transitions (recompute advantages)`: Like `Shuffle transitions` but we recompute advantages at the beginning of *each epoch*.

B.2 ADVANTAGE ESTIMATION

Let V be an approximator of the value function of some policy, i.e. $V \approx V^\pi$. We experimented with the three most commonly used advantage estimators in on-policy deep actor-critic RL (choice `C6`):

[16] `https://github.com/openai/baselines/tree/master/baselines/ppo2`

- **N-step** return [3] is defined as

$$\hat{V}_t^{(N)} = \sum_{i=t}^{t+N-1} \gamma^{i-t} r_i + \gamma^N V(s_{t+N}) \approx V^\pi(s_t).$$

  The parameter N (choice `C7`) controls the bias–variance tradeoff of the estimator with bigger values resulting in an estimator closer to empirical returns and having less bias and more variance. Given N-step returns we can estimate advantages as follows:

$$\hat{A}_t^{(\mathrm{N})} = \hat{V}_t^{(\mathrm{N})} - V(s_t) \approx A^\pi(s_t, a_t).$$

- **Generalized Advantage Estimator, GAE(λ)** [9] is a method that combines multi-step returns in the following way:

$$\hat{V}_t^{\mathrm{GAE}} = (1-\lambda) \sum_{N>0} \lambda^{N-1} \hat{V}_t^{(N)} \approx V^\pi(s_t),$$

  where $0 < \lambda < 1$ is a hyperparameter (choice `C8`) controlling the bias–variance trade-off. Using this, we can estimate advantages with:

$$\hat{A}_t^{\mathrm{GAE}} = \hat{V}_t^{\mathrm{GAE}} - V(s_t) \approx A^\pi(s_t, a_t).$$

  It is possible to compute the values of this estimator for all states encountered in an episode in linear time [9].

- **V-trace($\lambda, \bar{c}, \bar{\rho}$)** [19] is an extension of GAE which introduces truncated importance sampling weights to account for the fact that the current policy might be slightly different from the policy which generated the experience. It is parameterized by λ (choice `C9`) which serves the same role as in GAE and two parameters $\bar{c}$ and $\bar{\rho}$ which are truncation thresholds for two different types of importance weights. See [19] for the detailed description of the V-trace estimator. All experiments in the original paper [19] use $\bar{c} = \bar{\rho} = 1$. Similarly, we only consider the case $\bar{c} = \bar{\rho}$, i.e., we consider a single choice `V-Trace advantage` c, ρ `(C10)`.

The value network is trained by fitting one of the returns described above with an MSE (quadratic) or a Huber [1] loss (choice `C11`). Huber loss is a quadratic around zero up to some threshold (choice `C12`) at which point it becomes a linear function.

The original PPO implementation [17] uses an additional pessimistic clipping in the value loss function. See [27] for the description of this technique. It is parameterized by a clipping threshold (choice `C13`).

B.3 Policy losses

Let π denote the policy being optimized, and μ the behavioral policy, i.e. the policy which generated the experience. Moreover, let $\hat{A}_t^\pi$ and $\hat{A}_t^\mu$ be some estimators of the advantage at timestep t for the policies π and μ.

We consider optimizing the policy with the following policy losses (choice `C14`):

- **Policy Gradients (PG)** [2] with advantages: $\mathcal{L}_{\mathtt{PG}} = -\log \pi(a_t|s_t)\hat{A}_t^\pi$. It can be shown that if $\hat{A}_t^\pi$ estimators are unbiased, then $\nabla_\theta \mathcal{L}_{\mathtt{PG}}$ is an unbiased estimator of the gradient of the policy performance assuming that experience was generated by the current policy π.
- **V-trace [19]**: $\mathcal{L}^{\bar{\rho}}_{\mathtt{V\text{-}trace}} = \mathbf{sg}(\rho_t)\mathcal{L}_{\mathtt{PG}}$, where $\rho_t = \min(\frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}, \bar{\rho})$ is a truncated importance weight, **sg** is the `stop_gradient` operator[17] and $\bar{\rho}$ is a hyperparameter (choice `C15`). $\nabla_\theta L^{\bar{\rho}}_{\mathtt{V\text{-}trace}}$ is an unbiased estimator of the gradient of the policy performance if $\bar{\rho} = \infty$ regardless of the behavioural policyμ[18].
- **Proximal Policy Optimization (PPO)** [17]:

$$\mathcal{L}^{\epsilon}_{\mathrm{PPO}} = -\min\left[\frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}\hat{A}_t^\pi,\ \mathrm{clip}\left(\frac{\pi(a_t|s_t)}{\mu(a_t|s_t)},\ \frac{1}{1+\epsilon},\ 1+\epsilon\right)\hat{A}_t^\pi\right],$$

  where ϵ is a hyperparameter[19] `C16`. This loss encourages the policy to take actions which are better than average (have positive advantage) while clipping discourages bigger changes to the policy by limiting how much can be gained by changing the policy on a particular data point.

[17] Identity function with gradient zero.

[18] Assuming that advantage estimators are unbiased and $\mu(a_t|s_t) > 0$ for all pairs (s_t, a_t) for which $\pi(a_t|s_t) > 0$.

[19] The original PPO paper used $1-\epsilon$ instead $1/(1+\epsilon)$ as the lower bound for the clipping. Both variants are used in practice and we have decided to use $1/(1+\epsilon)$ as it is more symmetric.

- **Advantage-Weighted Regression (AWR)** [33]:

$$\mathcal{L}_{\texttt{AWR}}^{\beta,\,\omega_{\texttt{max}}} = -\log \pi(a_t|s_t) \min\left(\exp(A_t^{\mu}/\beta),\, \omega_{\texttt{max}}\right).$$

  It can be shown that for $\omega_{\texttt{max}} = \infty$ (choice `C17`) it corresponds to an approximate optimization of the policy π under a constraint of the form $\texttt{KL}(\pi||\mu) < \epsilon$ where the KL bound ϵ depends on the exponentiation temperature β (choice `C18`). Notice that in contrast to previous policy losses, AWR uses estimates of the advantages for the behavioral policy (A_t^{μ}) and not the current one (A_t^{π}). AWR was proposed mostly as an off-policy RL algorithm.

- **On-Policy Maximum a Posteriori Policy Optimization (V-MPO)** [34]: This policy loss is the same as AWR with the following differences: (1) exponentiation is replaced with the `softmax` operator and there is no clipping with $\omega_{\texttt{max}}$; (2) only samples with the top half advantages in each batch are used; (3) the temperature β is treated as a Lagrange multiplier and adjusted automatically to keep a constraint on how much the weights (i.e. softmax outputs) diverge from a uniform distribution with the constraint threshold ϵ being a hyperparameter (choice `C19`). (4) A soft constraint on $\texttt{KL}(\mu||\pi)$ is added. In our experiments, we did not treat this constraint as a part of the V-MPO policy loss as policy regularization is considered separately (See Sec. B.6).

- **Repeat Positive Advantages (RPA)**: $\mathcal{L}_{\texttt{RPA}} = -\log \pi(a_t|s_t)[A_t > 0]$[20]. This is a new loss we introduce in this paper. We choose this loss because it is the limiting case of AWR and V-MPO. In particular, $\mathcal{L}_{\texttt{AWR}}^{\beta,\,\omega_{\texttt{max}}}$ converges to $\omega_{\texttt{max}}\mathcal{L}_{\texttt{RPA}}$ for $\beta \to 0$ and for $\epsilon \to 0$ V-MPO converges to RPA with $[A_t > 0]$ replaced by only taking the top half advantages in each batch[21] (the two conditions become even more similar if advantage normalization is used, See Sec. B.9).

B.4 Handling of timesteps

The most important hyperparameter controlling how timesteps are handled is the discount factor γ (choice `C20`). Moreover, we consider the so-called *frame skip*[22] (choice `C21`). Frame skip equal to n means that we modify the environment by repeating each action outputted by the policy n times (unless the episode has terminated in the meantime) and sum the received rewards. When using frame skip, we also adjust the discount factor appropriately, i.e. we discount with γ^n instead of γ.

Many reinforcement learning environments (including the ones we use in our experiments) have *step limits* which means that an episode is terminated after some fixed number of steps (assuming it was not terminated earlier for some other reason). Moreover, the number of remaining environment steps is not included in policy observations which makes the environments non-Markovian and can potentially make learning harder [16]. We consider two ways to handle such *abandoned* episodes. We either treat the final transition as any other terminal transition, e.g. the value target for the last state is equal to the final reward, or we take the fact that we do not know what would happen if the episode was not terminated into account. In the latter case, we set the advantage for the final state to zero and its value target to the current value function. This also influences the value targets for prior states as the value targets are computed recursively [9, 19]. We denote this choice by `Handle abandoned? (C22)`.

B.5 Optimizers

We experiment with two most commonly used gradient-based optimizers in RL (choice `C23`): Adam [8] and RMSProp.[23] You can find the description of the optimizers and their hyperparameters in the original publications. For both optimizers, we sweep the learning rate (choices `C24` and `C25`), momentum (choices `C26` and `C27`) and the ϵ parameters added for numerical stability (choice `C28` and `C29`). Moreover, for RMSProp we consider both centered and uncentered versions (choice `C30`). For their remaining hyperparameters, we use the default values from TensorFlow [11], i.e. $\beta_2 = 0.999$ for Adam and $\rho = 0.1$ for RMSProp. Finally, we allow a linear learning rate schedule via the hyperparameter `Learning rate decay (C31)` which defines the terminal learning rate as a fraction of the initial learning rate (i.e., 0.0 correspond to a decay to zero whereas 1.0 corresponds to no decay).

B.6 Policy regularization

We consider three different modes for regularization (choice `C32`):

[20] $[P]$ denotes the Iverson bracket, i.e. $[P] = 1$ if P is true and $[P] = 0$ otherwise.

[21] For $\epsilon \to 0$, we have $\beta \to \infty$ which results in $\texttt{softmax}(A_t^{\mu}/\beta) \to 1$.

[22] While not too common in continuous control, this technique is standard in RL for Atari [6].

[23] RMSProp was proposed by Geoffrey Hinton in one of his Coursera lectures: `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

- `No regularization`: We apply no regularization.
- `Penalty`: we apply a regularizer R with fixed strength, i.e., we add to the loss the term αR for some fixed coefficient α which is a hyperparameter.
- `Constraint`: We impose a soft constraint of the form $R < \epsilon$ on the value of the regularizer where ϵ is a hyperparameter. This can be achieved by treating α as a Lagrange multiplier. In this case, we optimize the value of $\alpha > 0$ together with the networks parameter by adding to the loss the term $\alpha \cdot \mathbf{sg}(\epsilon - R)$, where **sg** in the `stop_gradient` operator. In practice, we use the following parametrization $\alpha = \exp(c \cdot p)$ where p is a trainable parameter (initialized with 0) and c is a coefficient controlling how fast α is adjusted (we use $c = 10$ in all experiments). After each gradient step, we clip the value of p to the range $[\log(10^{-6})/10,\ \log(10^{6})/10]$ to avoid extremely small and large values.

We consider a number of different policy regularizers both for penalty (choice `C33`) and constraint regularization (choice `C34`):

- Entropy $\mathtt{H}(\pi(\cdot|s))$ — it encourages the policy to try diverse actions [4].
- $\mathtt{KL}(\mu(\cdot|s)||\pi(\cdot|s))$ — the Kullback–Leibler divergence between the behavioral and the current policy [10] prevents the probability of taking a given action from *decreasing* too rapidly.
- $\mathtt{KL}(\pi(\cdot|s)||\mu(\cdot|s))$ — similar to the previous one but prevents too rapid *increase* of probabilities.
- $\mathtt{KL}(\mathtt{ref}(\cdot|s)||\pi(\cdot|s))$ where `ref` is some reference distribution. We use $\mathtt{ref} = \mathcal{N}(0, 1)$ in all experiments. This kind of regularization encourages the policy to try all possible actions.
- Decoupled $\mathtt{KL}(\mu(\cdot|s)||\pi(\cdot|s))$. For Gaussian distributions we can split $\mathtt{KL}(\mu(\cdot|s)||\pi(\cdot|s))$ into a term which depends on the change in the mean of the distribution and another one which depends on the change in the standard deviation: $\mathtt{KL}(\mu(\cdot|s)||\pi(\cdot|s)) = \mathtt{KL}(\mu(\cdot|s)||\zeta(\cdot|s)) + \mathtt{KL}(\zeta(\cdot|s)||\pi(\cdot|s))$ where $\zeta(\cdot|s)$ is a Gaussian distribution with same mean as $\mu(\cdot|s)$ and the same standard deviation as $\pi(\cdot|s)$. Therefore, instead of using $\mathtt{KL}(\mu(\cdot|s)||\pi(\cdot|s))$ directly, we can use two separate regularizers, $\mathtt{KL}(\mu(\cdot|s)||\zeta(\cdot|s))$ and $\mathtt{KL}(\zeta(\cdot|s)||\pi(\cdot|s))$, with different strengths. The soft constraint version of this regularizer in used in V-MPO[24] [34] with the threshold on $\mathtt{KL}(\mu(\cdot|s)||\zeta(\cdot|s))$ being orders of magnitude lower than the one on $\mathtt{KL}(\zeta(\cdot|s)||\pi(\cdot|s))$.

While one could add any linear combination of the above terms to the loss, we have decided to only use a single regularizer in each experiment. Overall, all these combinations of regularization modes and different hyperparameters lead to the choices detailed in Table 1.

Table 1: Choices pertaining to regularization.

Choice	Name
`C32`	`Regularization type`
`C33`	`Regularizer (in case of penalty)`
`C34`	`Regularizer (in case of constraint)`
`C35`	`Threshold for` $\mathtt{KL}(\mu\|\|\pi)$
`C36`	`Threshold for` $\mathtt{KL}(\pi\|\|\mu)$
`C37`	`Threshold for` $\mathtt{KL}(\mathtt{ref}\|\|\pi)$
`C38`	`Threshold for mean in decoupled` $\mathtt{KL}(\mu\|\|\pi)$
`C39`	`Threshold for std in decoupled` $\mathtt{KL}(\mu\|\|\pi)$
`C40`	`Threshold for entropy` $H(\pi)$
`C41`	`Regularizer coefficient for` $\mathtt{KL}(\mu\|\|\pi)$
`C42`	`Regularizer coefficient for` $\mathtt{KL}(\pi\|\|\mu)$
`C43`	`Regularizer coefficient for` $\mathtt{KL}(\mathtt{ref}\|\|\pi)$
`C44`	`Regularizer coefficient for mean in decoupled` $\mathtt{KL}(\mu\|\|\pi)$
`C45`	`Regularizer coefficient for std in decoupled` $\mathtt{KL}(\mu\|\|\pi)$
`C46`	`Regularizer coefficient for entropy`

B.7 Neural network architecture

We use multilayer perceptrons (MLPs) to represent policies and value functions. We either use separate networks for the policy and value function, or use a single network with two linear heads, one for the policy and one for

[24]The current arXiv version of the V-MPO paper [34] incorrectly uses the standard deviation of the old policy instead of the new one in the definition of $\mathtt{KL}(\mu(\cdot|s)||\pi(\cdot|s))$ which leads to a slightly different decomposition. We do not expect this to make any difference in practice.

the value function (choice `C47`). We consider different widths for the shared MLP (choice `C48`), the policy MLP (choice `C49`) and the value MLP (choice `C50`) as well as different depths for the shared MLP (choice `C51`), the policy MLP (choice `C52`) and the value MLP (choice `C53`). If we use the shared MLP, we further add a hyperparameter `Baseline cost (shared) (C54)` that rescales the contribution of the value loss to the full objective function. This is important in this case as the shared layers of the MLP affect the loss terms related to both the policy and the value function. We further consider different activation functions (choice `C55`) and different neural network initializers (choice `C56`). For the initialization of both the last layer in the policy MLP / the policy head (choice `C57`) and the last layer in the value MLP / the value head (choice `C58`), we further consider a hyperparameter that rescales the network weights of these layers after initialization.

B.8 Action distribution parameterization

A policy is a mapping from states to distributions of actions. In practice, a parametric distribution is chosen and the policy output is treated as the distribution parameters. The vast majority of RL applications in continuous control use a Gaussian distribution to represent the action distribution and this is also the approach we take.

This, however, still leaves a few decisions which need to be make in the implementation:

- Should the standard deviation of actions be a part of the network output (used e.g. in [21]) or should it be independent of inputs like in [17] (choice `C59`)? In the latter case, the standard deviation is still learnable but it is the same for each state.
- Gaussian distributions are parameterized with a mean and a standard deviation which has to be non-negative. What function should be used to transform network outputs which can be negative into the standard deviation (choice `C60`)? We consider exponentiation[25] (used e.g. in [17]) and the `softplus`[26] function (used e.g. in [29]).
- What should be the initial standard deviation of the action distribution (choice `C61`)? We can control it by adding some fixed value to the input to the function computing the standard deviation (e.g. `softplus`).
- Should we add a small value to the standard deviation to avoid very low values (choice `C62`)?
- Most continuous control environments expect actions from a bounded range (often $[-1,\ 1]$) but the commonly used Gaussian distribution can produce values of an arbitrary magnitude. We consider two approaches to handle this (choice `C63`): The easiest solution is to just clip the action to the allowed range when sending it to the environment (used e.g. in [17]). Another approach is to apply the `tanh` function to the distribution to bound the range of actions (used e.g. in [25]). This additional transformation changes the density of actions — if action u is parameterized as $u = \texttt{tanh}(x)$, where x is a sample from a Gaussian distribution with probability density function p_θ, than the density of u is $\log p_u(u) = \log p_\theta(x) - \log \tanh'(x)$, where $x = \tanh^{-1}(u)$. This additional $\log \tanh'(x)$ term does not affect policy losses because they only use $\nabla_\theta \log p_u(u) = \nabla_\theta \log p_\theta(x)$. Similarly, this term does not affect the KL divergences which may be used for regularization (See Sec. B.6) because the KL divergence has a form of the difference of two log-probabilities on the same sample and the two $\log \tanh'(x)$ terms cancel out.[27] The only place where the $\log \tanh'(x)$ term affects the policy gradient computation and should be included is the entropy regularization as $H(U) = -\mathbb{E}_u \log p_u(u) = \mathbb{E}_x[-\log p_\theta(x) + \log \tanh'(x)]$. This additional $\log \tanh'(x)$ term penalizies the policy for taking extreme actions which prevents $\tanh$ saturation and the loss of the gradient. Moreover, it prevents the action entropy from becoming unbounded.

To sum up, we parameterize the actions distribution as

$$T_u(\mathcal{N}(x_\mu,\, T_\rho(x_\rho + c_\rho) + \epsilon_\rho)),$$

where

- x_μ is a part of the policy network output,
- x_ρ is either a part of the policy network output or a separate learnable parameter (one per action dimension),
- ϵ_ρ (`C62`) is a hyperparameter controlling minimal standard deviation,
- T_ρ (`C60`) is a standard deviation transformation ($\mathbb{R} \to \mathbb{R}_{\geq 0}$),
- T_u (`C63`) is an action transformation ($\mathbb{R} \to [-1,\ 1]$),
- c_ρ is a constant controlling the initial standard deviation and computed as $c_\rho = T_\rho^{-1}(i_\rho - \epsilon_\rho)$ where i_ρ is the desired initial standard deviation (`C61`).

[25] For numerical stability, we clip the exponent to the range $[-15, 15]$. Notice that due to clipping this function has zero derivative outside of the range $[-15, 15]$ which is undesirable. Therefore, we use a "custom gradient" for the clipping function, namely we assume that it has derivative equal 1 everywhere.

[26] $\texttt{softplus}(x) = \log(e^x + 1)$

[27] $\texttt{KL}(U_1, U_2) = \mathbb{E}_{u \leftarrow U_1} \log U_1(u) - \log U_2(u) = \mathbb{E}_{x \leftarrow X_1}(\log X_1(x) - \log \tanh'(x)) - (\log X_2(x) - \log \tanh'(x)) = \mathbb{E}_{x \leftarrow X_1} \log X_1(x) - \log X_2(x) = \texttt{KL}(X_1 || X_2)$.

B.9 DATA NORMALIZATION AND CLIPPING

While it is not always mentioned in RL publications, many RL implementations perform different types of data normalization. In particular, we consider the following:

- Observation normalization (choice `C64`). If enabled, we keep the empirical mean o_μ and standard deviation o_ρ of each observation coordinate (based on all observations seen so far) and normalize observations by subtracting the empirical mean and dividing by $\max(o_\rho,\, 10^{-6})$. This results in all neural networks inputs having approximately zero mean and standard deviation equal to one. Moreover, we optionally clip the normalized observations to the range $[-o_{\texttt{max}},\, o_{\texttt{max}}]$ where $o_{\texttt{max}}$ is a hyperparameter (choice `C65`).
- Value function normalization (choice `C66`). Similarly to observations, we also maintain the empirical mean v_μ and standard deviation v_ρ of value function targets (See Sec. B.2). The value function network predicts normalized targets $(\hat{V} - v_\mu)/\max(v_\rho,\, 10^{-6})$ and its outputs are denormalized accordingly to obtain predicted values: $\hat{V} = v_\mu + V_{\texttt{out}} \max(v_\rho,\, 10^{-6})$ where $V_{\texttt{out}}$ is the value network output.
- Per minibatch advantage normalization (choice `C67`). We normalize advantages in each minibatch by subtracting their mean and dividing by their standard deviation for the policy loss.
- Gradient clipping (choice `C68`). We rescale the gradient before feeding it to the optimizer so that its L2 norm does not exceed the desired threshold.

C DEFAULT SETTINGS FOR EXPERIMENTS

Table 2 shows the default configuration used for all the experiments in this paper. We only list sub-choices that are active (e.g. we use the PPO loss so we do not list hyperparameters associated with different policy losses).

Table 2: Default settings used in experiments.

Choice	Name	Default value
C1	num_envs	256
C2	iteration_size	2048
C3	num_epochs	10
C4	batch_size	64
C5	batch_mode	Shuffle transitions
C6	advantage_estimator	GAE
C8	GAE λ	0.95
C11	Value function loss	MSE
C13	PPO-style value clipping ϵ	0.2
C14	Policy loss	PPO
C16	PPO ϵ	0.2
C20	Discount factor γ	0.99
C21	Frame skip	1
C22	Handle abandoned?	False
C23	Optimizer	Adam
C24	Adam learning rate	3e-4
C26	Adam momentum	0.9
C28	Adam ϵ	1e-7
C31	Learning rate decay	0.0
C32	Regularization type	None
C47	Shared MLPs?	Shared
C49	Policy MLP width	64
C50	Value MLP width	64
C52	Policy MLP depth	2
C53	Value MLP depth	2
C55	Activation	tanh
C56	Initializer	Orthogonal with gain 1.41
C57	Last policy layer scaling	0.01
C58	Last value layer scaling	1.0
C59	Global standard deviation?	True
C60	Standard deviation transformation T_ρ	safe_exp
C61	Initial standard deviation i_ρ	1.0
C63	Action transformation T_u	clip
C62	Minimum standard deviation ϵ_ρ	1e-3
C64	Input normalization	Average
C65	Input clipping	10.0
C66	Value function normalization	Average
C67	Per minibatch advantage normalization	False
C68	Gradient clipping	0.5

D EXPERIMENT POLICY LOSSES

D.1 DESIGN

For each of the 5 environments, we sampled 2000 choice configurations where we sampled the following choices independently and uniformly from the following ranges:

- `num_epochs (C3)`: {1, 3, 10}
- `Policy loss (C14)`: {AWR, PG, PPO, RPA, V-MPO, V-Trace}
  - For the case "`Policy loss (C14)` = AWR", we further sampled the sub-choices:
    - `AWR` β `(C18)`: {0.0001, 0.0003, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3}
    - `AWR` $\omega_{\texttt{max}}$ `(C17)`: {1.1, 1.2, 1.3, 1.5}
  - For the case "`Policy loss (C14)` = PPO", we further sampled the sub-choices:
    - `PPO` ϵ `(C16)`: {0.1, 0.2, 0.3, 0.5}
  - For the case "`Policy loss (C14)` = V-MPO", we further sampled the sub-choices:
    - `V-MPO` ϵ_n `(C19)`: {0.0001, 0.0003, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1.0}
  - For the case "`Policy loss (C14)` = V-Trace", we further sampled the sub-choices:
    - `V-Trace loss` ρ `(C15)`: {1.0, 1.2, 1.5, 2.0}
- `Adam learning rate (C24)`: {3e-05, 0.0001, 0.0003, 0.001, 0.003}

All the other choices were set to the default values as described in Appendix C.

For each of the sampled choice configurations, we train 3 agents with different random seeds and compute the performance metric as described in Section 2.

D.2 RESULTS

We report aggregate statistics of the experiment in Table 3 as well as training curves in Figure 3. For each of the investigated choices in this experiment, we further provide a per-choice analysis in Figures 5-13.

Table 3: Performance quantiles across choice configurations.

	Ant-v1	HalfCheetah-v1	Hopper-v1	Humanoid-v1	Walker2d-v1
90th percentile	1490	994	1103	1224	459
95th percentile	1727	1080	1297	1630	565
99th percentile	2290	1363	1621	2611	869
Max	2862	2048	1901	3435	1351

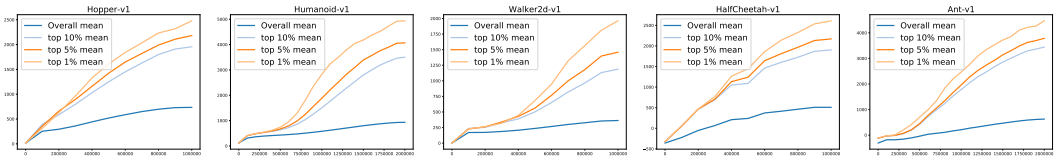

Figure 3: Training curves.

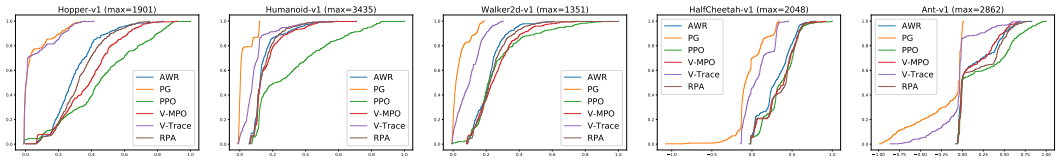

Figure 4: Empirical cumulative density functions of agent performance conditioned on different values of `Policy loss (C14)`. The x axis denotes performance rescaled so that 0 corresponds to a random policy and 1 to the best found configuration, and the y axis denotes the quantile.

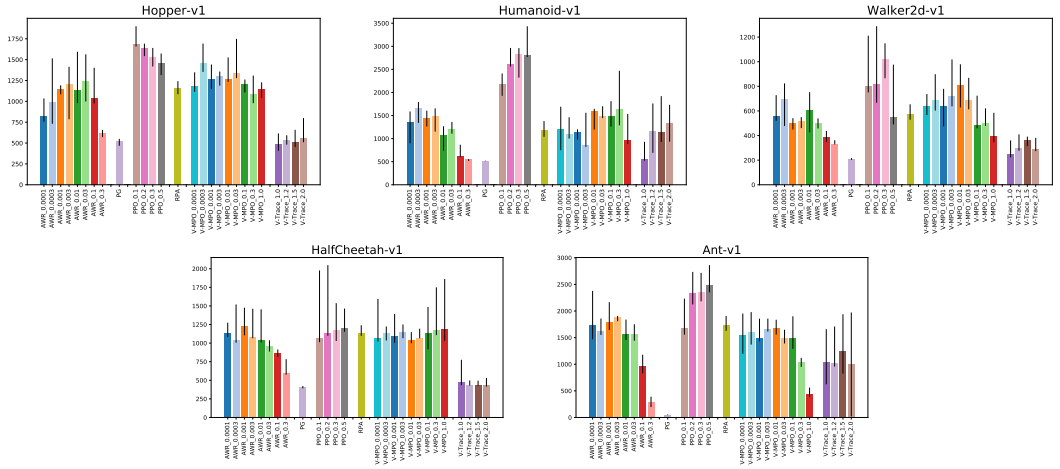

Figure 5: Comparison of 95th percentile of the performance of different policy losses conditioned on their hyperparameters.

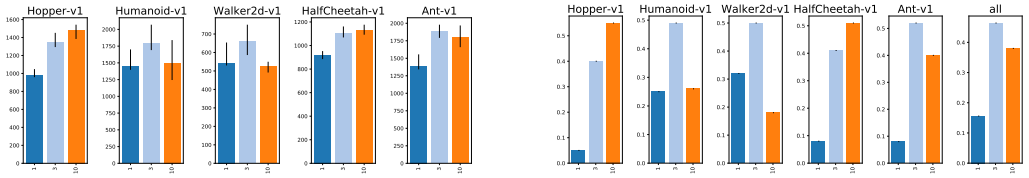

Figure 6: Analysis of choice `num_epochs (C3)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

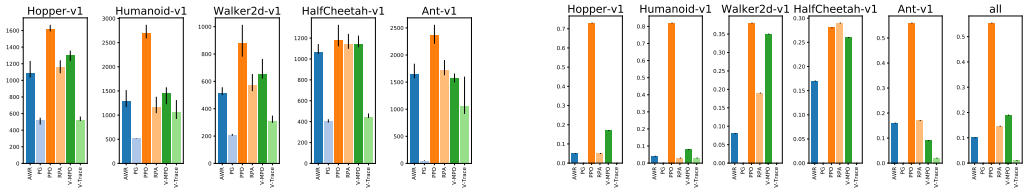

Figure 7: Analysis of choice `Policy loss (C14)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

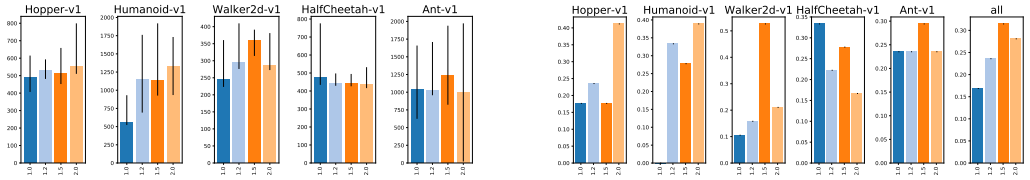

Figure 8: Analysis of choice `V-Trace loss` ρ `(C15)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

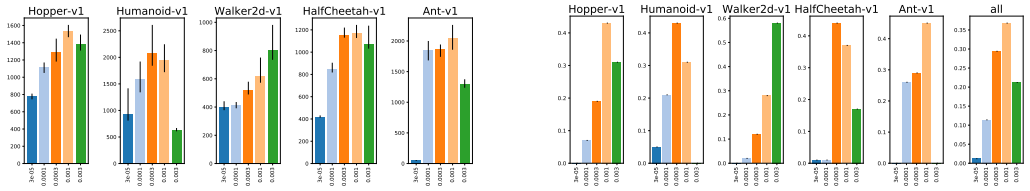

Figure 9: Analysis of choice `Adam learning rate (C24)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

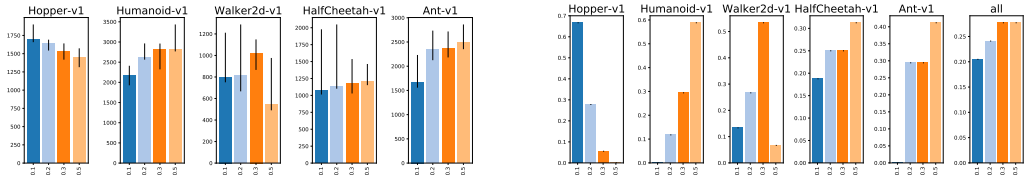

Figure 10: Analysis of choice `PPO` ϵ `(C16)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

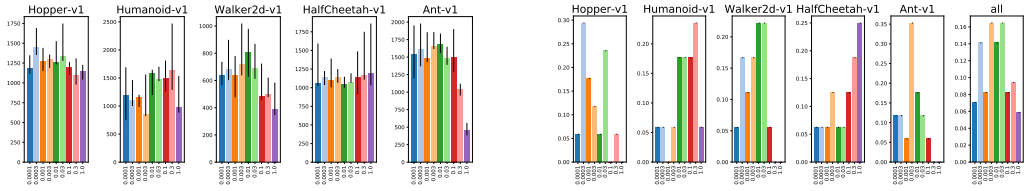

Figure 11: Analysis of choice `V-MPO` ϵ_n `(C19)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

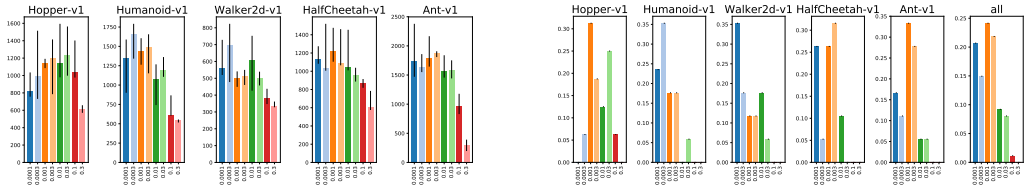

Figure 12: Analysis of choice `AWR` β `(C18)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

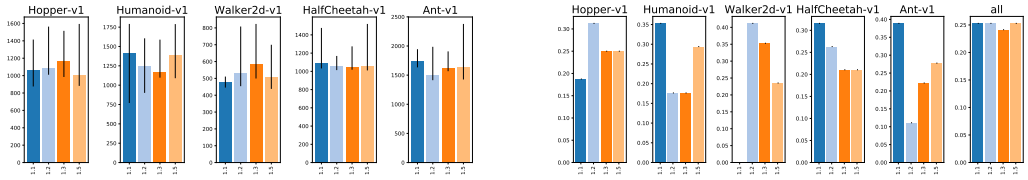

Figure 13: Analysis of choice `AWR` $\omega_{\max}$ `(C17)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

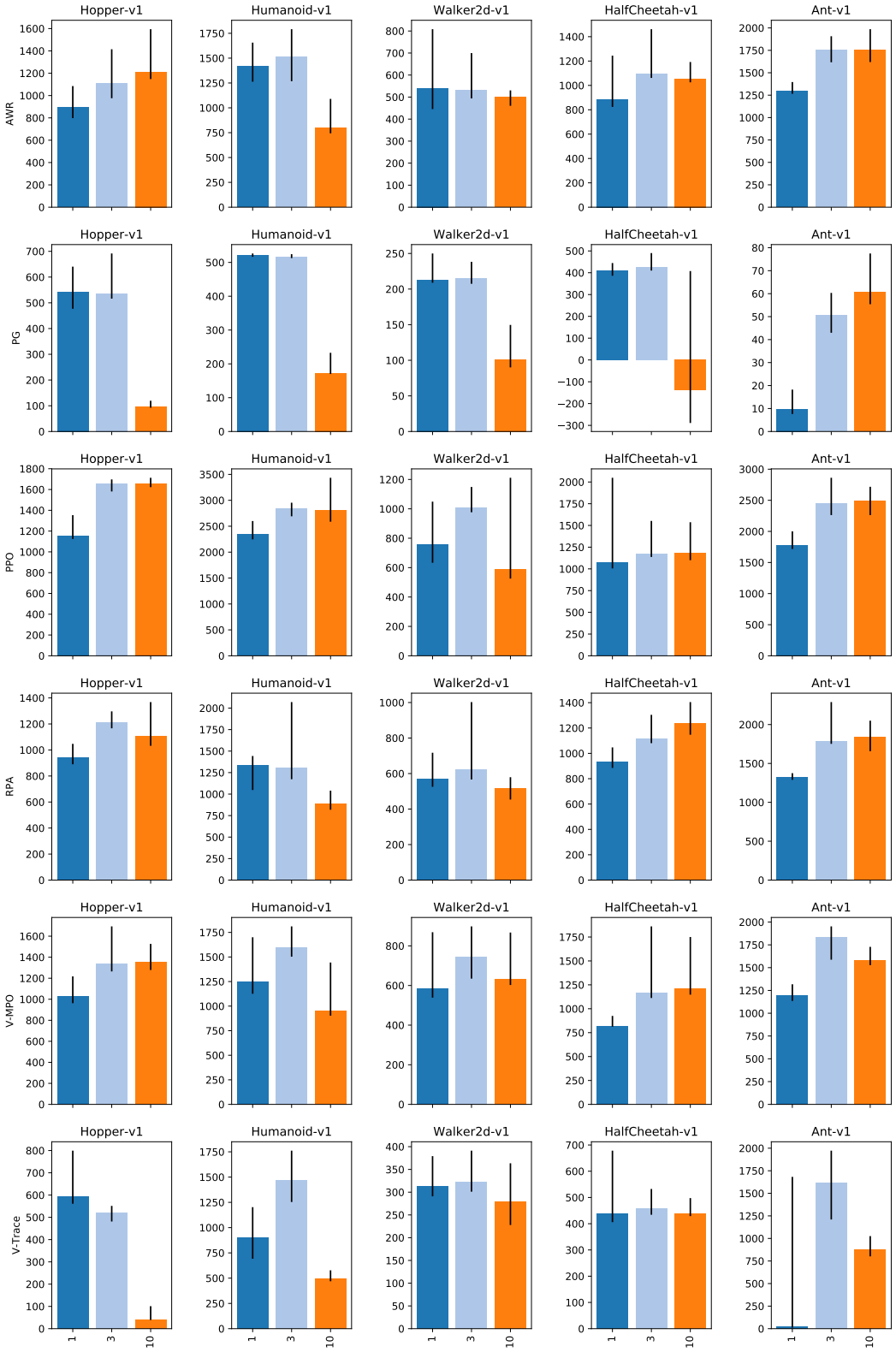

Figure 14: 95th percentile of performance scores conditioned on `Policy loss (C14)` (rows) and `num_epochs (C3)` (bars).

E EXPERIMENT NETWORKS ARCHITECTURE

E.1 DESIGN

For each of the 5 environments, we sampled 4000 choice configurations where we sampled the following choices independently and uniformly from the following ranges:

- `Action transformation` T_u `(C63)`: {clip, tanh}
- `Last value layer scaling (C58)`: {0.001, 0.01, 0.1, 1.0}
- `Global standard deviation? (C59)`: {False, True}
- `Last policy layer scaling (C57)`: {0.001, 0.01, 0.1, 1.0}
- `Standard deviation transformation` T_ρ `(C60)`: {exp, softplus}
- `Initial standard deviation` i_ρ `(C61)`: {0.1, 0.5, 1.0, 2.0}
- `Initializer (C56)`: {Glorot normal, Glorot uniform, He normal, He uniform, LeCun normal, LeCun uniform, Orthogonal, Orthogonal(gain=1.41)}
- `Shared MLPs? (C47)`: {separate, shared}
  - For the case "`Shared MLPs? (C47)` = separate", we further sampled the sub-choices:
    - `Policy MLP width (C49)`: {16, 32, 64, 128, 256, 512}
    - `Policy MLP depth (C52)`: {1, 2, 4, 8}
    - `Value MLP width (C50)`: {16, 32, 64, 128, 256, 512}
    - `Value MLP depth (C53)`: {1, 2, 4, 8}
  - For the case "`Shared MLPs? (C47)` = shared", we further sampled the sub-choices:
    - `Shared MLP width (C48)`: {16, 32, 64, 128, 256, 512}
    - `Shared MLP depth (C51)`: {1, 2, 4, 8}
    - `Baseline cost (shared) (C54)`: {0.001, 0.1, 1.0, 10.0, 100.0}
- `Minimum standard deviation` ϵ_ρ `(C62)`: {0.0, 0.01, 0.1}
- `Adam learning rate (C24)`: {3e-05, 0.0001, 0.0003, 0.001}
- `Activation (C55)`: {ELU, Leaky ReLU, ReLU, Sigmoid, Swish, Tanh}

All the other choices were set to the default values as described in Appendix C.

For each of the sampled choice configurations, we train 3 agents with different random seeds and compute the performance metric as described in Section 2.

After running the experiment described above we noticed (Fig. 15) that separate policy and value function networks (`C47`) perform better and we have rerun the experiment with only this variant present.

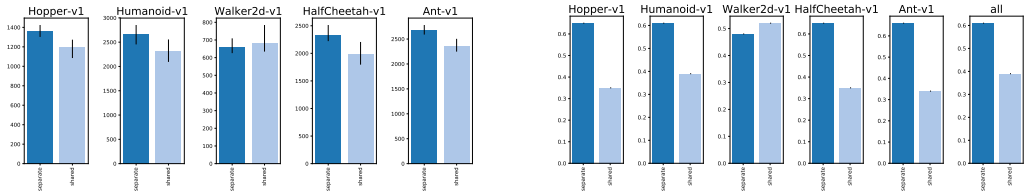

Figure 15: Analysis of choice `Shared MLPs? (C47)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

E.2 RESULTS

We report aggregate statistics of the experiment in Table 4 as well as training curves in Figure 16. For each of the investigated choices in this experiment, we further provide a per-choice analysis in Figures 17-30.

Table 4: Performance quantiles across choice configurations.

	Ant-v1	HalfCheetah-v1	Hopper-v1	Humanoid-v1	Walker2d-v1
90th percentile	2098	1513	1133	1817	528
95th percentile	2494	2120	1349	2382	637
99th percentile	3138	3031	1582	3202	934
Max	4112	4358	1875	3987	1265

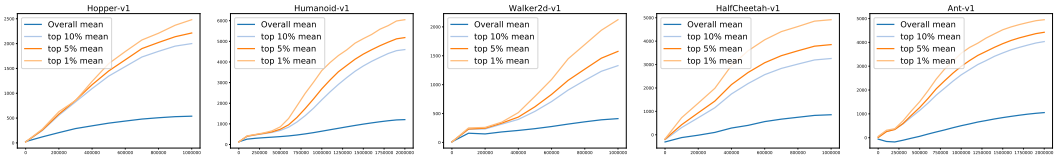

Figure 16: Training curves.

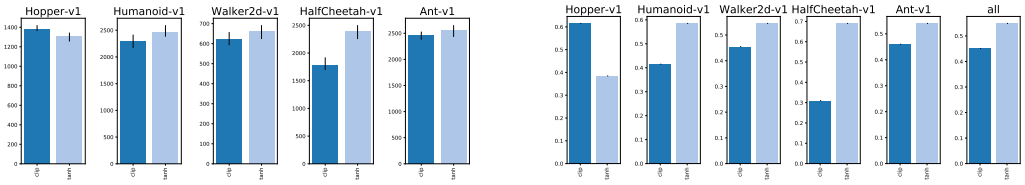

Figure 17: Analysis of choice `Action transformation` T_u `(C63)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

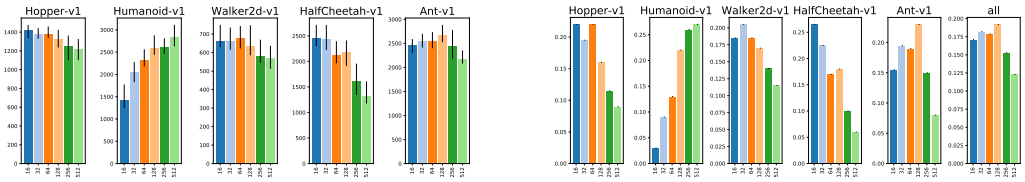

Figure 18: Analysis of choice `Policy MLP width (C49)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

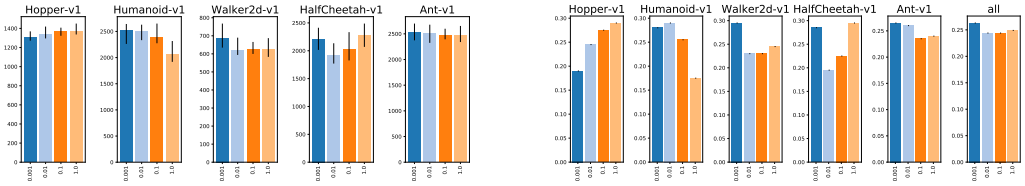

Figure 19: Analysis of choice `Last value layer scaling (C58)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

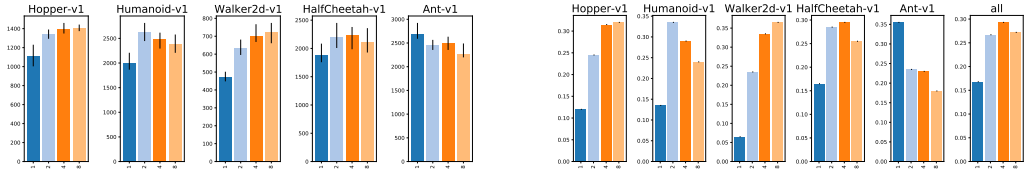

Figure 20: Analysis of choice `Value MLP depth (C53)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

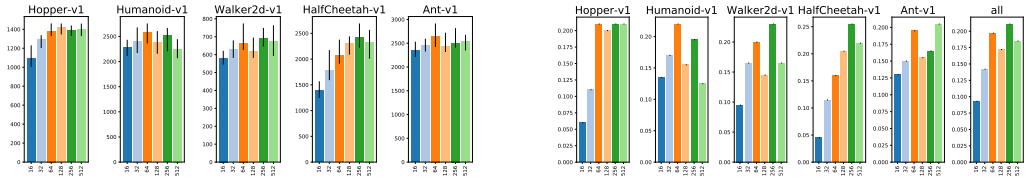

Figure 21: Analysis of choice `Value MLP width (C50)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

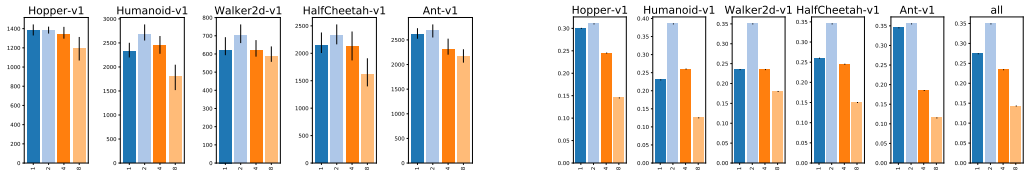

Figure 22: Analysis of choice `Policy MLP depth (C52)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

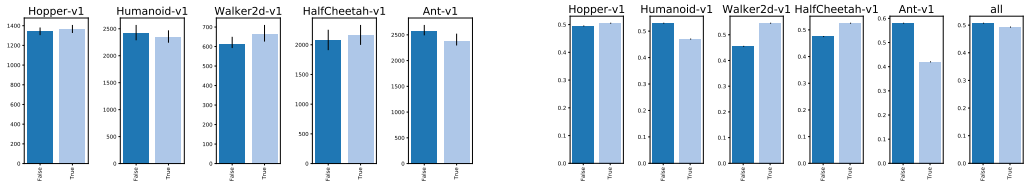

Figure 23: Analysis of choice `Global standard deviation? (C59)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

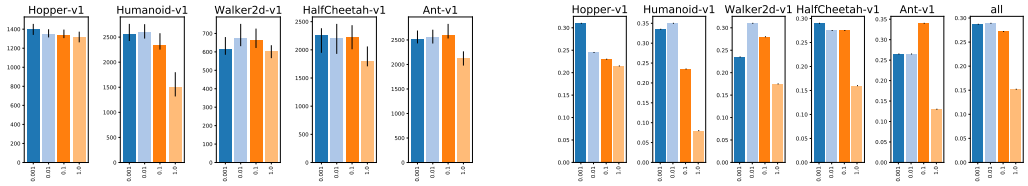

Figure 24: Analysis of choice `Last policy layer scaling (C57)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

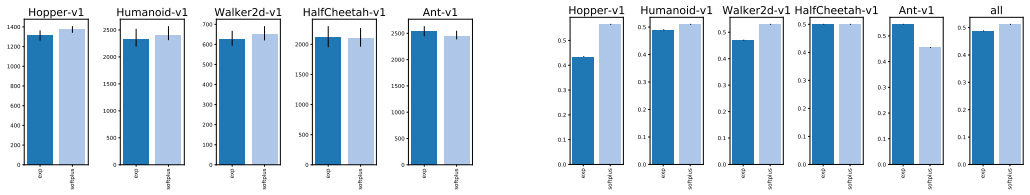

Figure 25: Analysis of choice `Standard deviation transformation` T_ρ `(C60)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

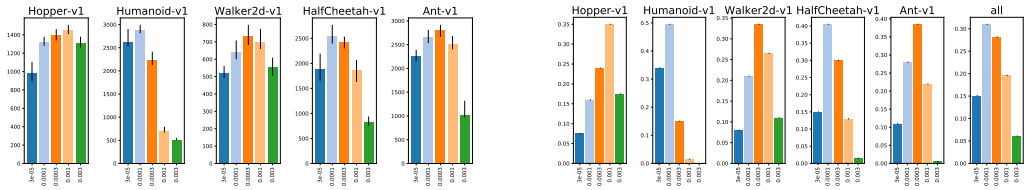

Figure 26: Analysis of choice `Adam learning rate (C24)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

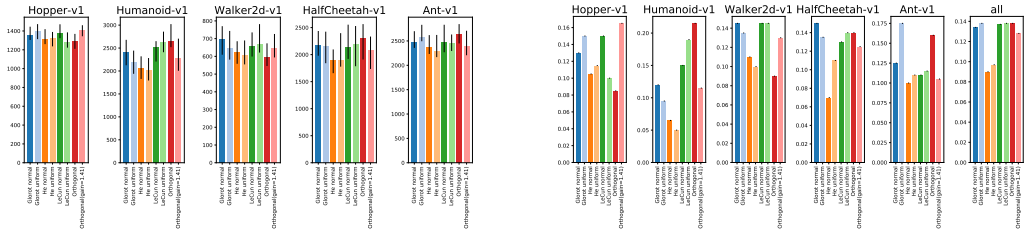

Figure 27: Analysis of choice `Initializer (C56)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

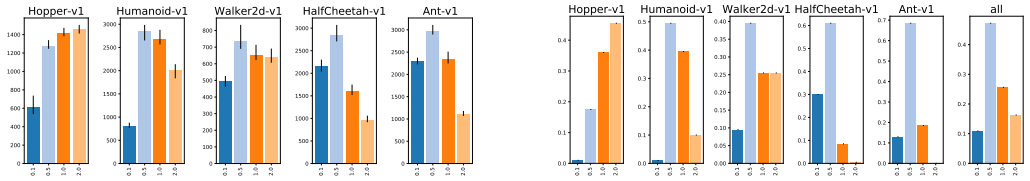

Figure 28: Analysis of choice `Initial standard deviation` i_ρ `(C61)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

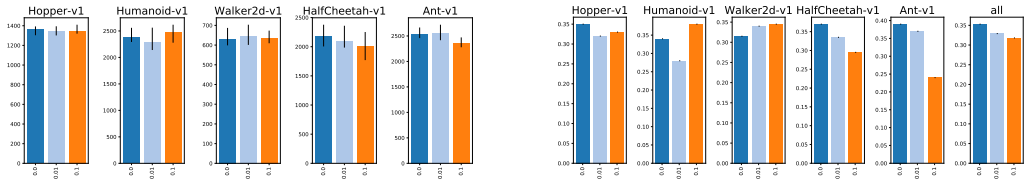

Figure 29: Analysis of choice `Minimum standard deviation` ϵ_ρ `(C62)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

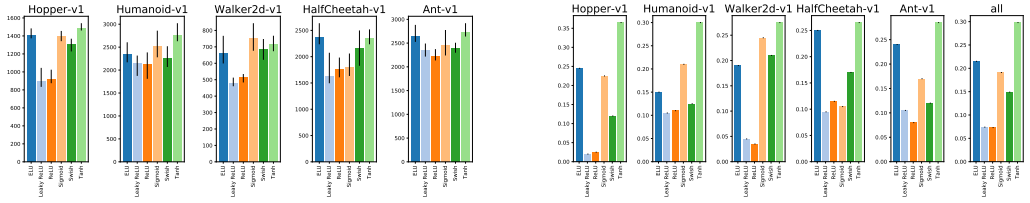

Figure 30: Analysis of choice `Activation (C55)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

F EXPERIMENT NORMALIZATION AND CLIPPING

F.1 DESIGN

For each of the 5 environments, we sampled 2000 choice configurations where we sampled the following choices independently and uniformly from the following ranges:

- `PPO ϵ (C16)`: {0.1, 0.2, 0.3, 0.5}
- `Input normalization (C64)`: {Average, None}
  - For the case "`Input normalization (C64)` = Average", we further sampled the sub-choices:
    - `Input clipping (C65)`: {1.0, 2.0, 5.0, 10.0, None}
- `Gradient clipping (C68)`: {0.5, 1.0, 2.0, 5.0, None}
- `Per minibatch advantage normalization (C67)`: {False, True}
- `Adam learning rate (C24)`: {3e-05, 0.0001, 0.0003, 0.001}
- `Value function normalization (C66)`: {Average, None}

All the other choices were set to the default values as described in Appendix C.

For each of the sampled choice configurations, we train 3 agents with different random seeds and compute the performance metric as described in Section 2.

F.2 RESULTS

We report aggregate statistics of the experiment in Table 5 as well as training curves in Figure 31. For each of the investigated choices in this experiment, we further provide a per-choice analysis in Figures 32-38.

Table 5: Performance quantiles across choice configurations.

	Ant-v1	HalfCheetah-v1	Hopper-v1	Humanoid-v1	Walker2d-v1
90th percentile	2058	1265	1533	1649	1143
95th percentile	2287	1716	1662	2165	1564
99th percentile	2662	2465	1809	3100	2031
Max	3333	3515	2074	3482	2371

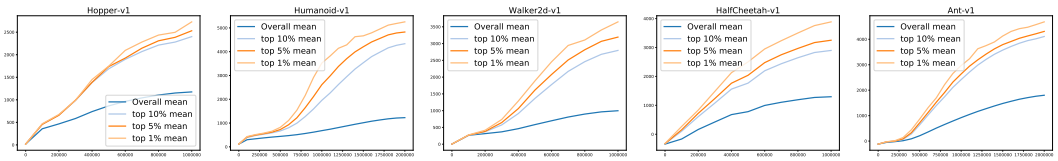

Figure 31: Training curves.

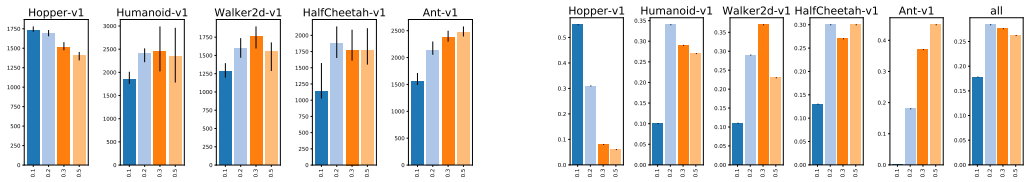

Figure 32: Analysis of choice `PPO ϵ (C16)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

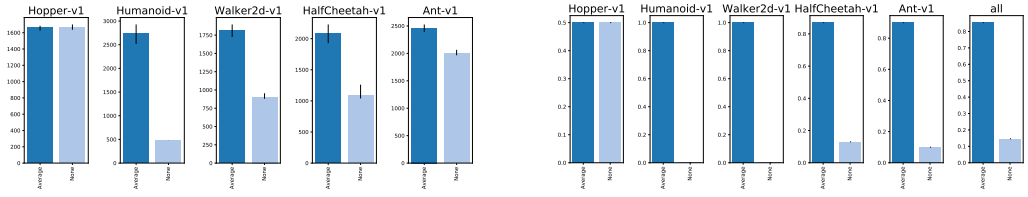

Figure 33: Analysis of choice `Input normalization (C64)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

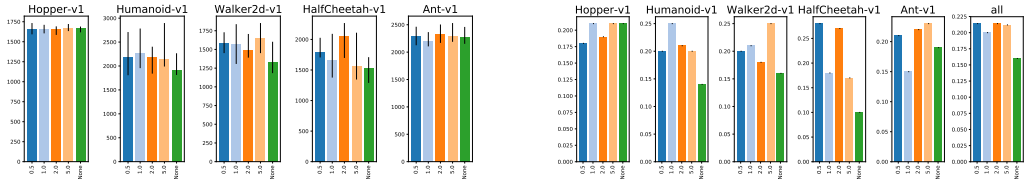

Figure 34: Analysis of choice `Gradient clipping (C68)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

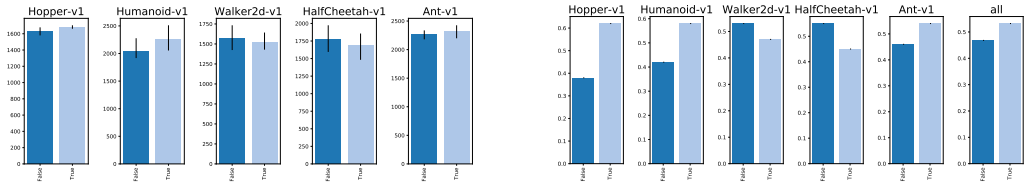

Figure 35: Analysis of choice `Per minibatch advantage normalization (C67)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

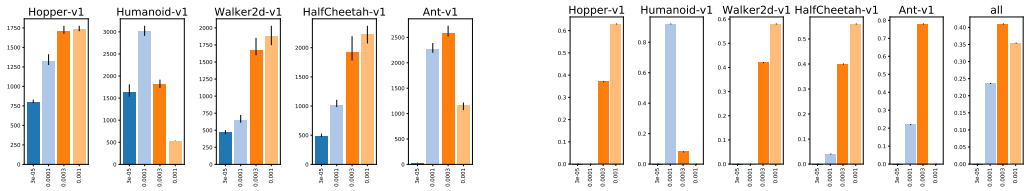

Figure 36: Analysis of choice `Adam learning rate (C24)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

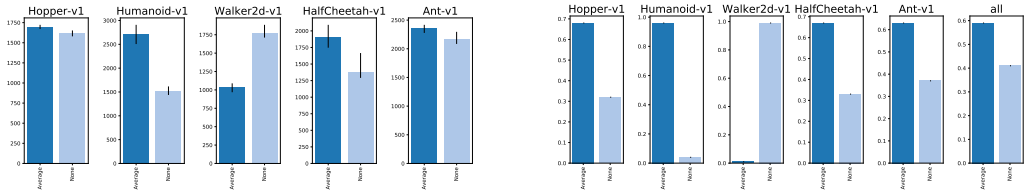

Figure 37: Analysis of choice `Value function normalization (C66)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

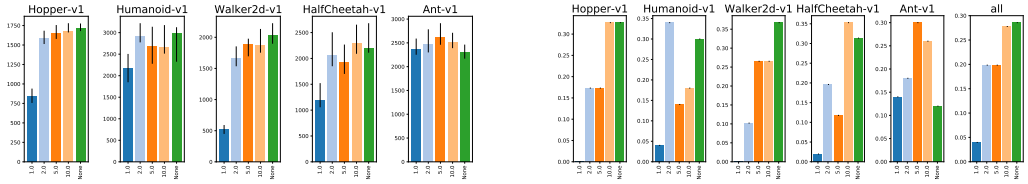

Figure 38: Analysis of choice `Input clipping (C65)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

G EXPERIMENT ADVANTAGE ESTIMATION

G.1 DESIGN

For each of the 5 environments, we sampled 4000 choice configurations where we sampled the following choices independently and uniformly from the following ranges:

- `num_envs (C1)`: {64, 128, 256}
- `Value function loss (C11)`: {Huber, MSE}
  - For the case "`Value function loss (C11)` = Huber", we further sampled the sub-choices:
    - `Huber delta (C12)`: {0.001, 0.01, 0.1, 1.0}
- `PPO-style value clipping` ϵ `(C13)`: {0.001, 0.01, 0.1, 1.0, None}
- `advantage_estimator (C6)`: {GAE, N-step, V-Trace}
  - For the case "`advantage_estimator (C6)` = GAE", we further sampled the sub-choices:
    - `GAE` λ `(C8)`: {0.8, 0.9, 0.95, 0.99}
  - For the case "`advantage_estimator (C6)` = N-step", we further sampled the sub-choices:
    - `N-step` N `(C7)`: {1, 3, 10, 1000000}
  - For the case "`advantage_estimator (C6)` = V-Trace", we further sampled the sub-choices:
    - `V-Trace advantage` λ `(C9)`: {0.8, 0.9, 0.95, 0.99, 1.0}
    - `V-Trace advantage` c, ρ `(C10)`: {1.0, 1.2, 1.5, 2.0}
- `Adam learning rate (C24)`: {3e-05, 0.0001, 0.0003, 0.001, 0.003}

All the other choices were set to the default values as described in Appendix C.

For each of the sampled choice configurations, we train 3 agents with different random seeds and compute the performance metric as described in Section 2.

G.2 RESULTS

We report aggregate statistics of the experiment in Table 6 as well as training curves in Figure 39. For each of the investigated choices in this experiment, we further provide a per-choice analysis in Figures 40-50.

Table 6: Performance quantiles across choice configurations.

	Ant-v1	HalfCheetah-v1	Hopper-v1	Humanoid-v1	Walker2d-v1
90th percentile	1705	1128	1626	1922	947
95th percentile	2114	1535	1777	2374	1185
99th percentile	2781	2631	2001	3013	1697
Max	3775	3613	2215	3564	2309

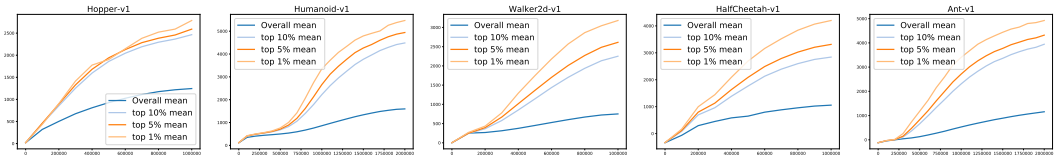

Figure 39: Training curves.

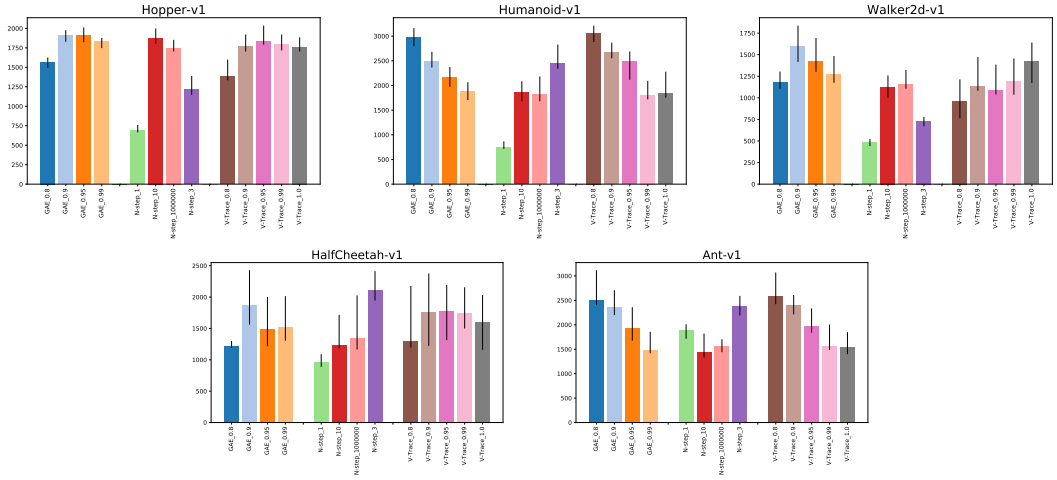

Figure 40: Comparison of 95th percentile of the performance of different advantage estimators conditioned on their hyperparameters.

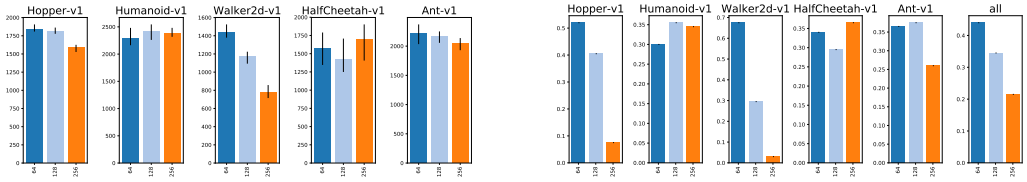

Figure 41: Analysis of choice `num_envs (C1)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

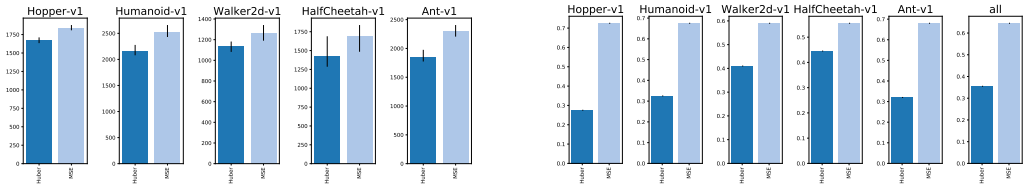

Figure 42: Analysis of choice `Value function loss (C11)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

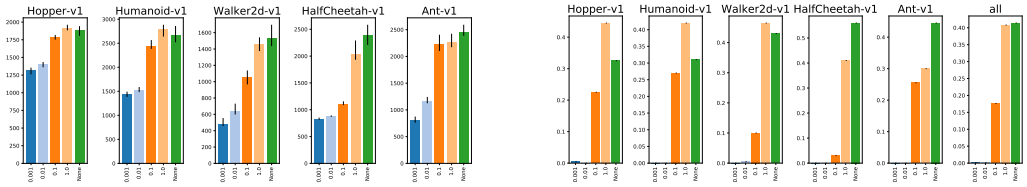

Figure 43: Analysis of choice `PPO-style value clipping` ϵ `(C13)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

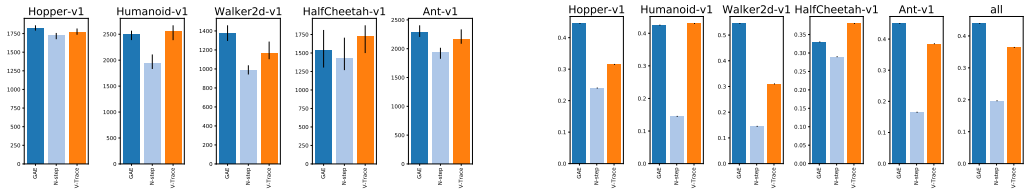

Figure 44: Analysis of choice `advantage_estimator (C6)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

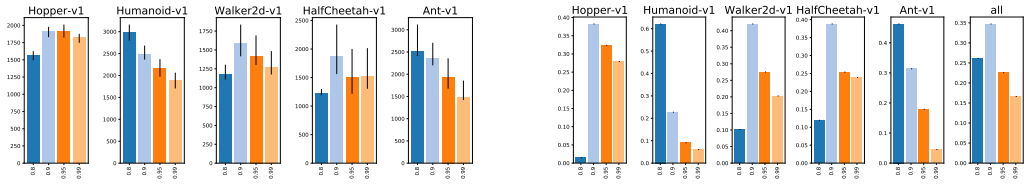

Figure 45: Analysis of choice `GAE` λ `(C8)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

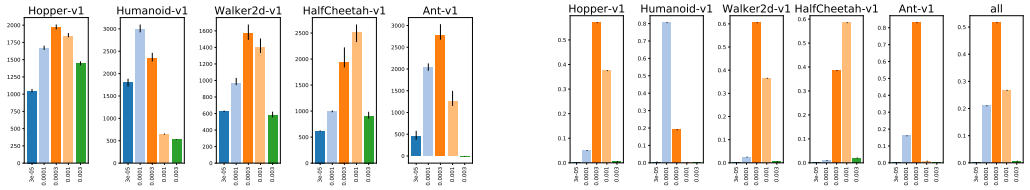

Figure 46: Analysis of choice `Adam learning rate (C24)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

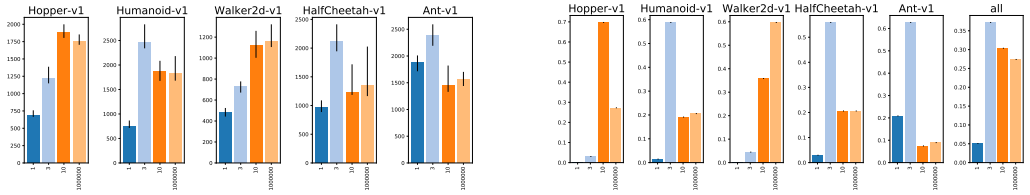

Figure 47: Analysis of choice `N-step` N `(C7)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

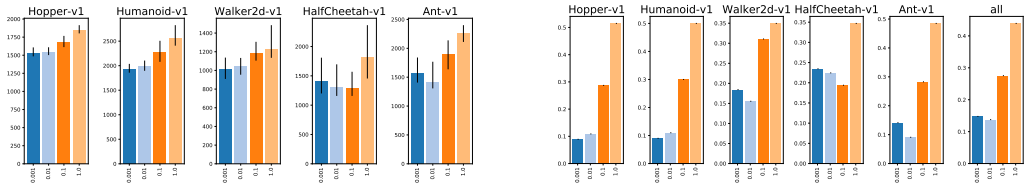

Figure 48: Analysis of choice `Huber delta (C12)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

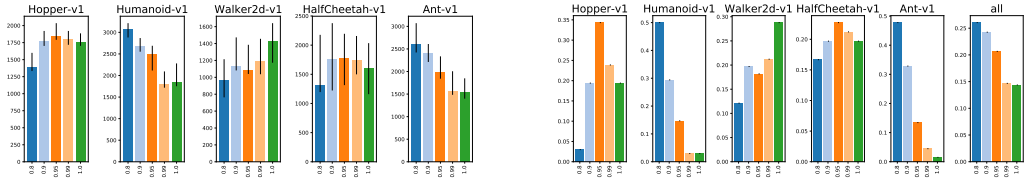

Figure 49: Analysis of choice `V-Trace advantage` λ `(C9)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

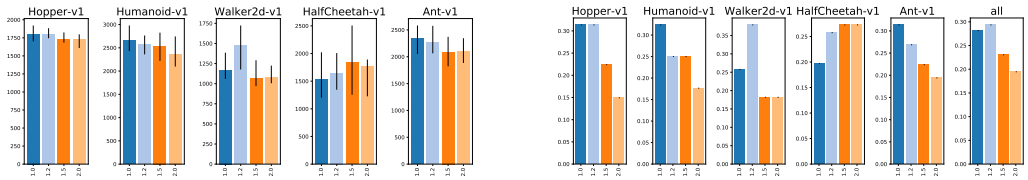

Figure 50: Analysis of choice `V-Trace advantage` c, ρ `(C10)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

H EXPERIMENT TRAINING SETUP

H.1 DESIGN

For each of the 5 environments, we sampled 2000 choice configurations where we sampled the following choices independently and uniformly from the following ranges:

- `iteration_size (C2)`: {512, 1024, 2048, 4096}
- `batch_mode (C5)`: {`Fixed trajectories`, `Shuffle trajectories`, `Shuffle transitions`, `Shuffle transitions (recompute advantages)`}
- `num_epochs (C3)`: {1, 3, 10}
- `num_envs (C1)`: {64, 128, 256}
- `Adam learning rate (C24)`: {3e-05, 0.0001, 0.0003, 0.001, 0.003}
- `batch_size (C4)`: {64, 128, 256}

All the other choices were set to the default values as described in Appendix C.

For each of the sampled choice configurations, we train 3 agents with different random seeds and compute the performance metric as described in Section 2.

H.2 RESULTS

We report aggregate statistics of the experiment in Table 7 as well as training curves in Figure 51. For each of the investigated choices in this experiment, we further provide a per-choice analysis in Figures 52-58.

Table 7: Performance quantiles across choice configurations.

	Ant-v1	HalfCheetah-v1	Hopper-v1	Humanoid-v1	Walker2d-v1
90th percentile	2203	1316	1695	2310	1190
95th percentile	2484	1673	1853	2655	1431
99th percentile	2907	2665	2060	3014	1844
Max	3563	3693	2434	3502	2426

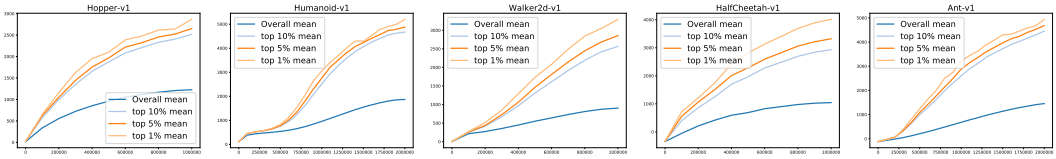

Figure 51: Training curves.

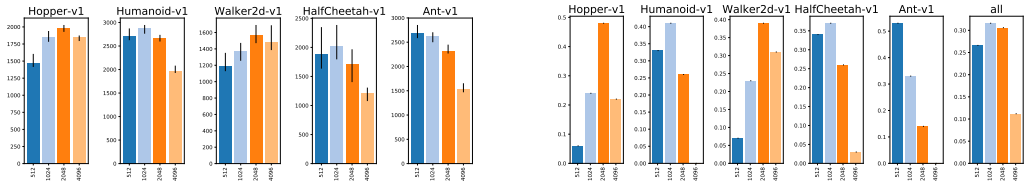

Figure 52: Analysis of choice `iteration_size (C2)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

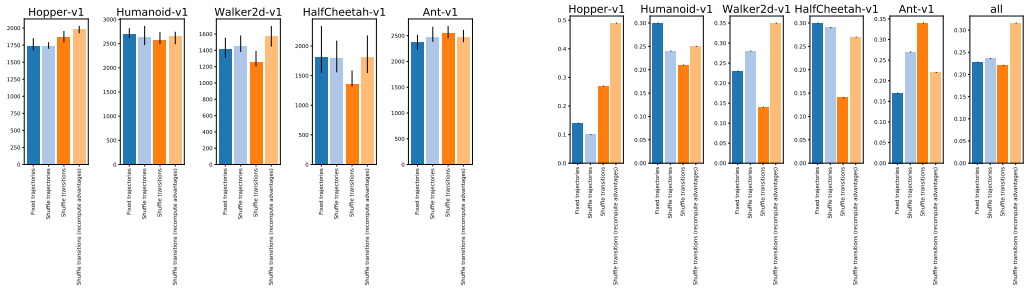

Figure 53: Analysis of choice `batch_mode (C5)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

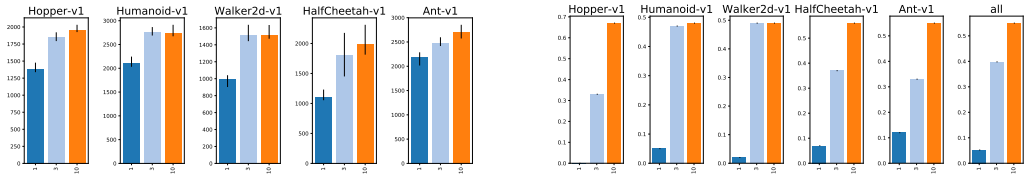

Figure 54: Analysis of choice `num_epochs (C3)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

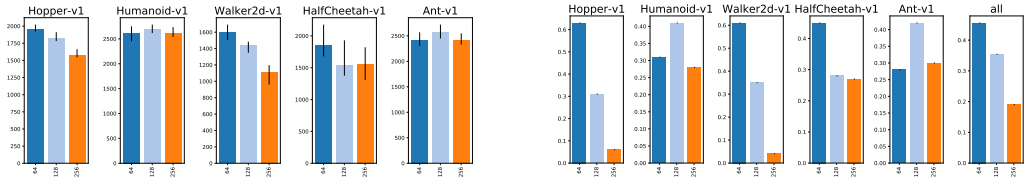

Figure 55: Analysis of choice `num_envs (C1)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

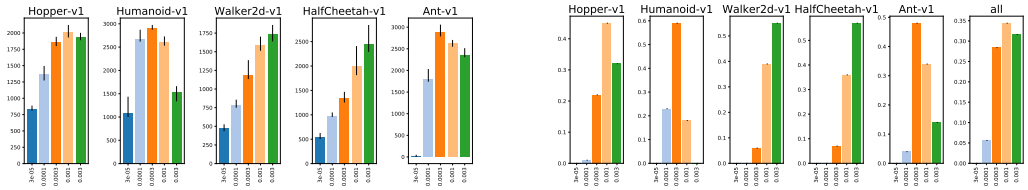

Figure 56: Analysis of choice `Adam learning rate (C24)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

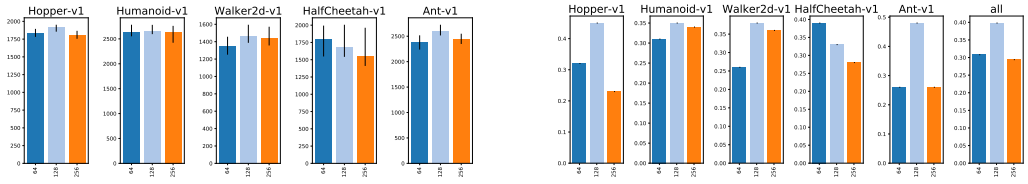

Figure 57: Analysis of choice `batch_size (C4)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

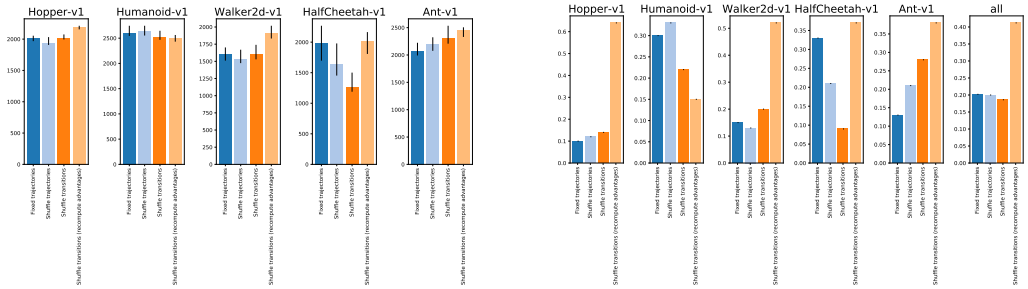

Figure 58: Analysis of choice `batch_mode (C5)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations(right). In order to obtain narrower confidence intervals in this experiment we only sweep `batch_mode (C5)`, `num_envs (C1)`,`Adam learning rate (C24)`.

I EXPERIMENT TIME

I.1 DESIGN

For each of the 5 environments, we sampled 2000 choice configurations where we sampled the following choices independently and uniformly from the following ranges:

- `Discount factor` γ `(C20)`: {0.95, 0.97, 0.99, 0.999}
- `Frame skip (C21)`: {1, 2, 5}
- `Handle abandoned? (C22)`: {False, True}
- `Adam learning rate (C24)`: {3e-05, 0.0001, 0.0003, 0.001}

All the other choices were set to the default values as described in Appendix C.

For each of the sampled choice configurations, we train 3 agents with different random seeds and compute the performance metric as described in Section 2.

I.2 RESULTS

We report aggregate statistics of the experiment in Table 8 as well as training curves in Figure 59. For each of the investigated choices in this experiment, we further provide a per-choice analysis in Figures 60-63.

Table 8: Performance quantiles across choice configurations.

	Ant-v1	HalfCheetah-v1	Hopper-v1	Humanoid-v1	Walker2d-v1
90th percentile	1462	1063	1243	1431	761
95th percentile	1654	1235	1675	2158	810
99th percentile	2220	1423	2204	2769	974
Max	2833	1918	2434	3106	1431

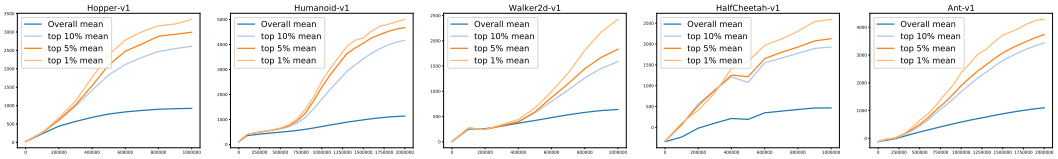

Figure 59: Training curves.

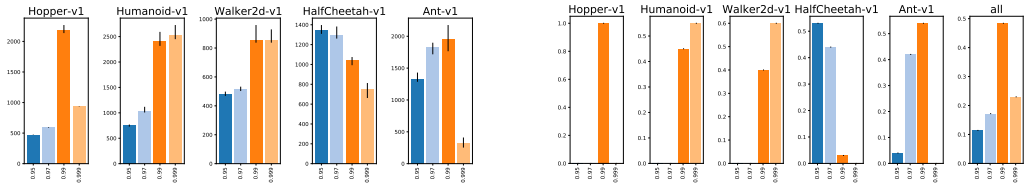

Figure 60: Analysis of choice `Discount factor` γ `(C20)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

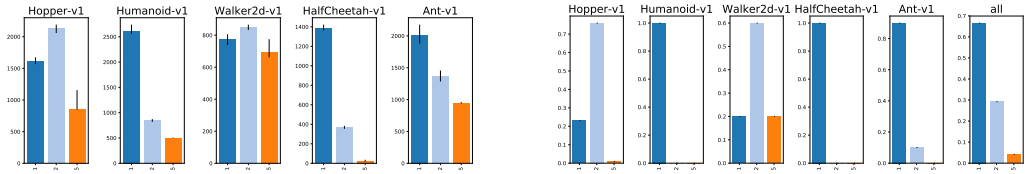

Figure 61: Analysis of choice `Frame skip (C21)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

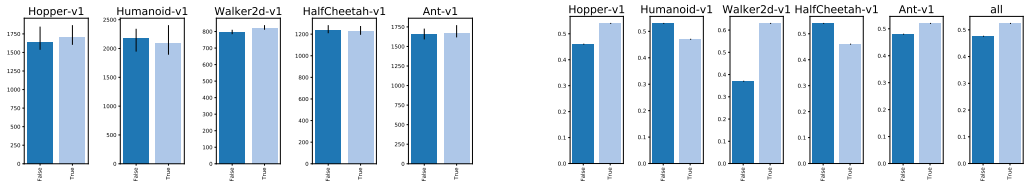

Figure 62: Analysis of choice `Handle abandoned? (C22)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

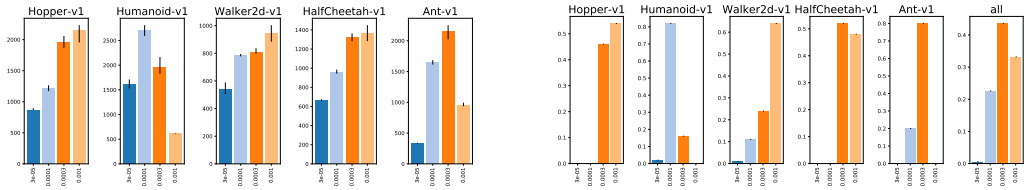

Figure 63: Analysis of choice `Adam learning rate (C24)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

J EXPERIMENT OPTIMIZERS

J.1 DESIGN

For each of the 5 environments, we sampled 2000 choice configurations where we sampled the following choices independently and uniformly from the following ranges:

- `Learning rate decay (C31)`: {0.0, 1.0}
- `Optimizer (C23)`: {Adam, RMSProp}
  - For the case "`Optimizer (C23)` = Adam", we further sampled the sub-choices:
    - `Adam momentum (C26)`: {0.0, 0.9}
    - `Adam` ϵ `(C28)`: {1e-09, 1e-08, 1e-07, 1e-06, 1e-05, 0.0001}
    - `Adam learning rate (C24)`: {3e-05, 0.0001, 0.0003, 0.001}
  - For the case "`Optimizer (C23)` = RMSProp", we further sampled the sub-choices:
    - `RMSProp centered? (C30)`: {False, True}
    - `RMSProp momentum (C27)`: {0.0, 0.9}
    - `RMSProp` ϵ `(C29)`: {1e-09, 1e-08, 1e-07, 1e-06, 1e-05, 0.0001}
    - `RMSProp learning rate (C25)`: {3e-05, 0.0001, 0.0003, 0.001}

All the other choices were set to the default values as described in Appendix C.

For each of the sampled choice configurations, we train 3 agents with different random seeds and compute the performance metric as described in Section 2.

J.2 RESULTS

We report aggregate statistics of the experiment in Table 9 as well as training curves in Figure 64. For each of the investigated choices in this experiment, we further provide a per-choice analysis in Figures 65-73.

Table 9: Performance quantiles across choice configurations.

	Ant-v1	HalfCheetah-v1	Hopper-v1	Humanoid-v1	Walker2d-v1
90th percentile	2180	1085	1675	2549	712
95th percentile	2388	1124	1728	2726	797
99th percentile	2699	1520	1826	2976	1079
Max	2953	2532	1959	3332	1453

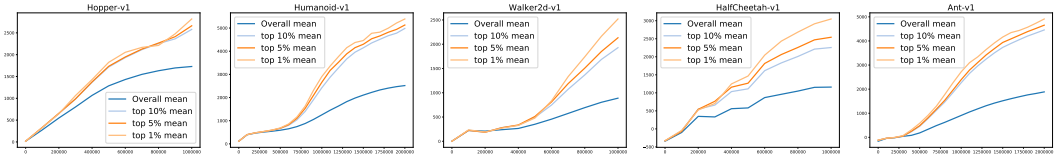

Figure 64: Training curves.

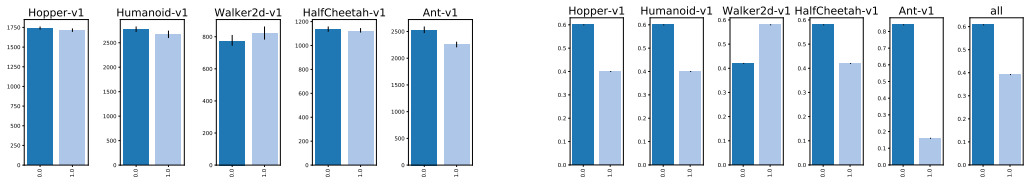

Figure 65: Analysis of choice `Learning rate decay (C31)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

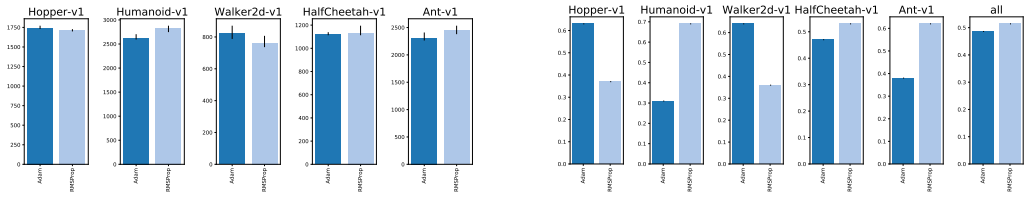

Figure 66: Analysis of choice `Optimizer (C23)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

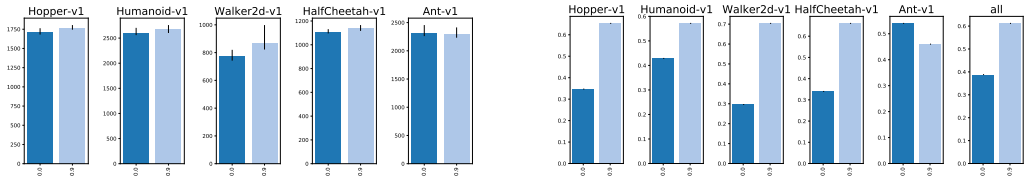

Figure 67: Analysis of choice `Adam momentum (C26)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

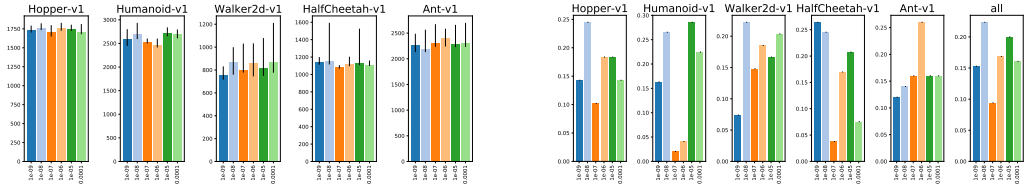

Figure 68: Analysis of choice `Adam` ϵ `(C28)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

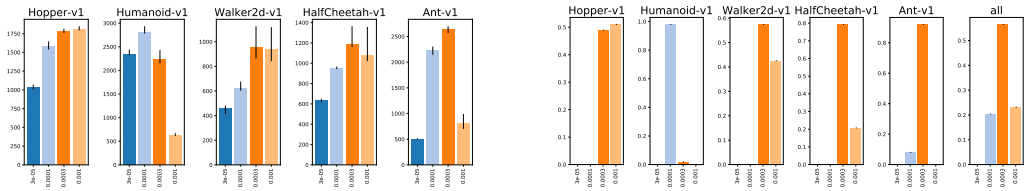

Figure 69: Analysis of choice `Adam learning rate (C24)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

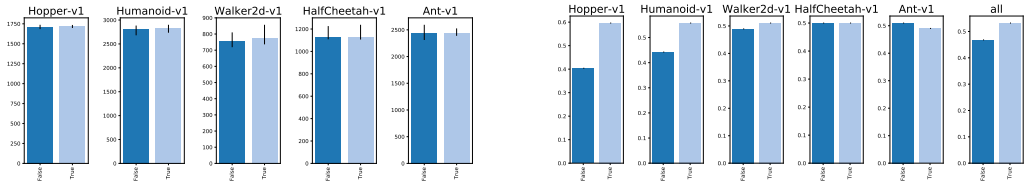

Figure 70: Analysis of choice `RMSProp centered? (C30)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

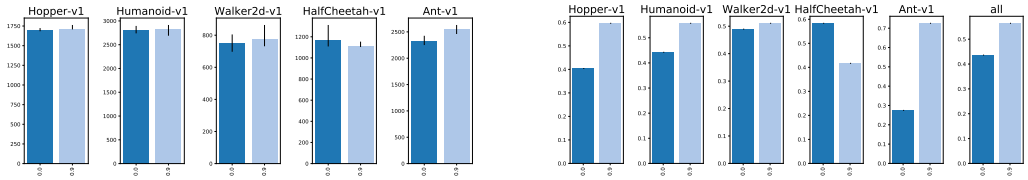

Figure 71: Analysis of choice `RMSProp momentum (C27)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

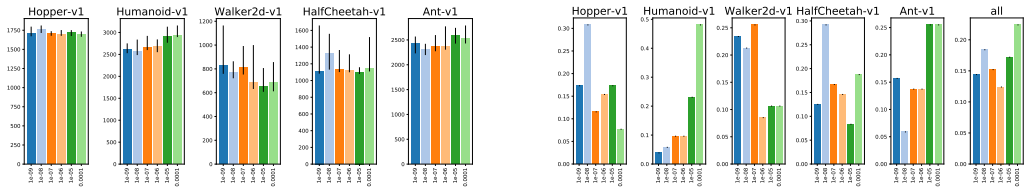

Figure 72: Analysis of choice `RMSProp` ϵ `(C29)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

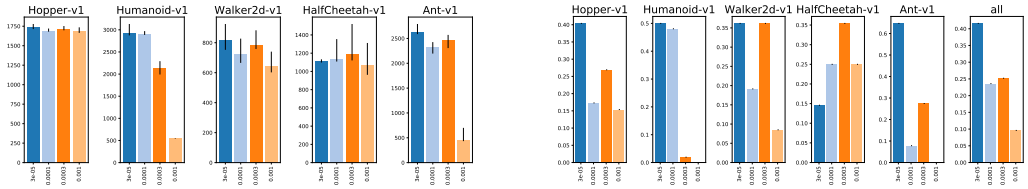

Figure 73: Analysis of choice `RMSProp learning rate (C25)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

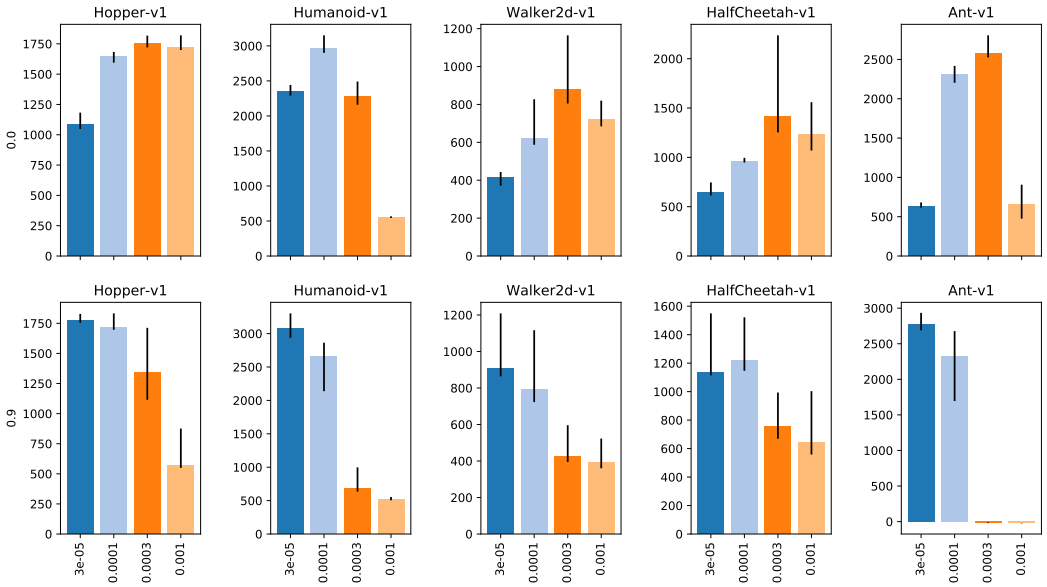

Figure 74: 95th percentile of performance scores conditioned on `RMSProp momentum (C27)` (rows) and `RMSProp learning rate (C25)` (bars).

K EXPERIMENT REGULARIZERS

K.1 DESIGN

For each of the 5 environments, we sampled 4000 choice configurations where we sampled the following choices independently and uniformly from the following ranges:

- `Regularization type (C32)`: {Constraint, No regularization, Penalty}
  - For the case "`Regularization type (C32)` = Constraint", we further sampled the sub-choices:
    - `Regularizer (in case of constraint) (C34)`: {$\mathtt{KL}(\mu||\pi)$, $\mathtt{KL}(\pi||\mu)$, $\mathtt{KL}(\mathtt{ref}||\pi)$, decoupled $\mathtt{KL}(\mu||\pi)$, entropy}
      - For the case "`Regularizer (in case of constraint) (C34)` = $\mathtt{KL}(\mu||\pi)$", we further sampled the sub-choices:
        - `Threshold for` $\mathtt{KL}(\mu||\pi)$ `(C35)`: {0.005, 0.01, 0.02, 0.04, 0.08}
      - For the case "`Regularizer (in case of constraint) (C34)` = $\mathtt{KL}(\pi||\mu)$", we further sampled the sub-choices:
        - `Threshold for` $\mathtt{KL}(\pi||\mu)$ `(C36)`: {0.005, 0.01, 0.02, 0.04, 0.08}
      - For the case "`Regularizer (in case of constraint) (C34)` = $\mathtt{KL}(\mathtt{ref}||\pi)$", we further sampled the sub-choices:
        - `Threshold for` $\mathtt{KL}(\mathtt{ref}||\pi)$ `(C37)`: {10.0, 20.0, 40.0, 80.0, 160.0}
      - For the case "`Regularizer (in case of constraint) (C34)` = decoupled $\mathtt{KL}(\mu||\pi)$", we further sampled the sub-choices:
        - `Threshold for mean in decoupled` $\mathtt{KL}(\mu||\pi)$ `(C38)`: {0.005, 0.01, 0.02, 0.04, 0.08}
        - `Threshold for std in decoupled` $\mathtt{KL}(\mu||\pi)$ `(C39)`: {5e-05, 0.000125, 0.00025, 0.0005, 0.001, 0.002, 0.004}
      - For the case "`Regularizer (in case of constraint) (C34)` = entropy", we further sampled the sub-choices:
        - `Threshold for entropy` $H(\pi)$ `(C40)`: {0.0, -5.0, -10.0, -15.0}
  - For the case "`Regularization type (C32)` = Penalty", we further sampled the sub-choices:
    - `Regularizer (in case of penalty) (C33)`: {$\mathtt{KL}(\mu||\pi)$, $\mathtt{KL}(\pi||\mu)$, $\mathtt{KL}(\mathtt{ref}||\pi)$, decoupled $\mathtt{KL}(\mu||\pi)$, entropy}
      - For the case "`Regularizer (in case of penalty) (C33)` = $\mathtt{KL}(\mu||\pi)$", we further sampled the sub-choices:
        - `Regularizer coefficient for` $\mathtt{KL}(\mu||\pi)$ `(C41)`: {0.003, 0.01, 0.03, 0.1, 0.3, 1.0}
      - For the case "`Regularizer (in case of penalty) (C33)` = $\mathtt{KL}(\pi||\mu)$", we further sampled the sub-choices:
        - `Regularizer coefficient for` $\mathtt{KL}(\pi||\mu)$ `(C42)`: {0.003, 0.01, 0.03, 0.1, 0.3, 1.0}
      - For the case "`Regularizer (in case of penalty) (C33)` = $\mathtt{KL}(\mathtt{ref}||\pi)$", we further sampled the sub-choices:
        - `Regularizer coefficient for` $\mathtt{KL}(\mathtt{ref}||\pi)$ `(C43)`: {3e-06, 1e-05, 3e-05, 0.0001, 0.0003, 0.001}
      - For the case "`Regularizer (in case of penalty) (C33)` = decoupled $\mathtt{KL}(\mu||\pi)$", we further sampled the sub-choices:
        - `Regularizer coefficient for mean in decoupled` $\mathtt{KL}(\mu||\pi)$ `(C44)`: {0.003, 0.01, 0.03, 0.1, 0.3, 1.0}
        - `Regularizer coefficient for std in decoupled` $\mathtt{KL}(\mu||\pi)$ `(C45)`: {0.1, 0.3, 1.0, 3.0, 10.0, 30.0, 100.0, 300.0}
      - For the case "`Regularizer (in case of penalty) (C33)` = entropy", we further sampled the sub-choices:
        - `Regularizer coefficient for entropy (C46)`: {1e-05, 3e-05, 0.0001, 0.0003, 0.001, 0.003}
- `Adam learning rate (C24)`: {3e-05, 0.0001, 0.0003, 0.001, 0.003}

All the other choices were set to the default values as described in Appendix C.

For each of the sampled choice configurations, we train 3 agents with different random seeds and compute the performance metric as described in Section 2.

K.2 RESULTS

We report aggregate statistics of the experiment in Table 10 as well as training curves in Figure 75. For each of the investigated choices in this experiment, we further provide a per-choice analysis in Figures 76-92.

Table 10: Performance quantiles across choice configurations.

	Ant-v1	HalfCheetah-v1	Hopper-v1	Humanoid-v1	Walker2d-v1
90th percentile	2158	1477	1639	2624	705
95th percentile	2600	1870	1707	2832	814
99th percentile	2956	2413	1812	3071	1016
Max	3202	3156	1979	3348	1597

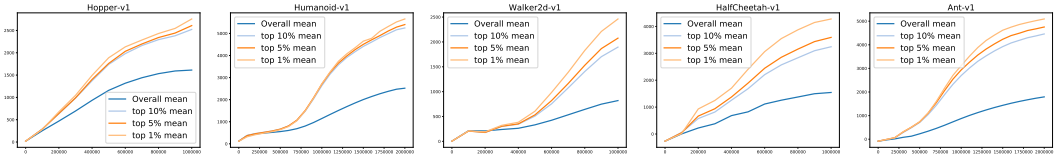

Figure 75: Training curves.

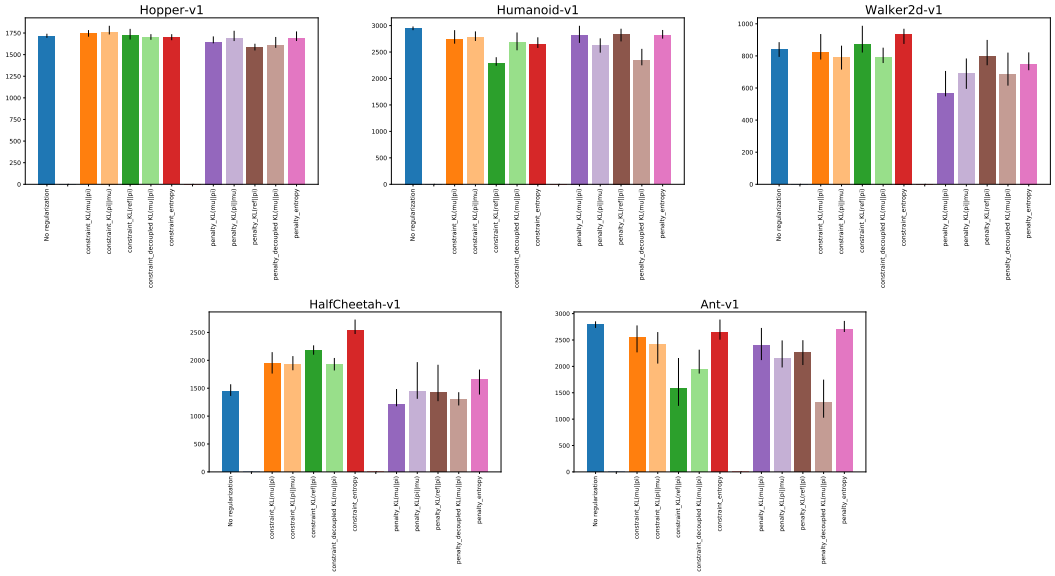

Figure 76: Comparison of 95th percentile of the performance of different regularization approaches conditioned on their type.

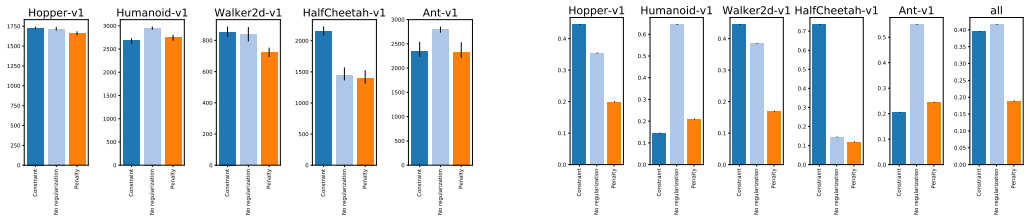

Figure 77: Analysis of choice `Regularization type (C32)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

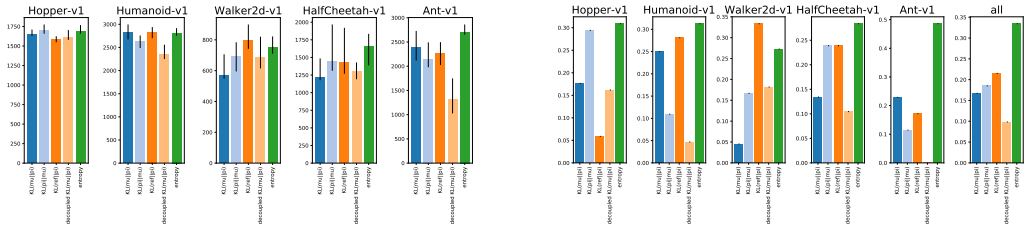

Figure 78: Analysis of choice `Regularizer (in case of penalty) (C33)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

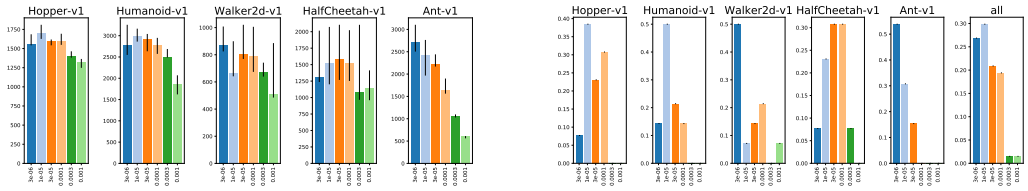

Figure 79: Analysis of choice `Regularizer coefficient for KL(ref||π) (C43)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

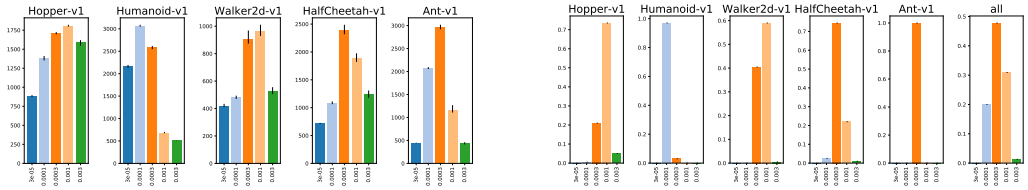

Figure 80: Analysis of choice `Adam learning rate (C24)`: 95th percentile of performance scores conditioned on choice (left) and distribution of choices in top 5% of configurations (right).

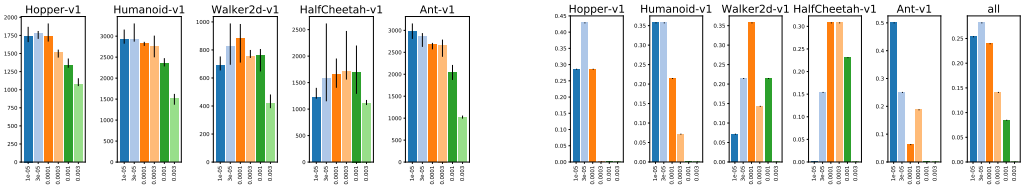

Figure 81: Analysis of choice `Regularizer coefficient for entropy (C46)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

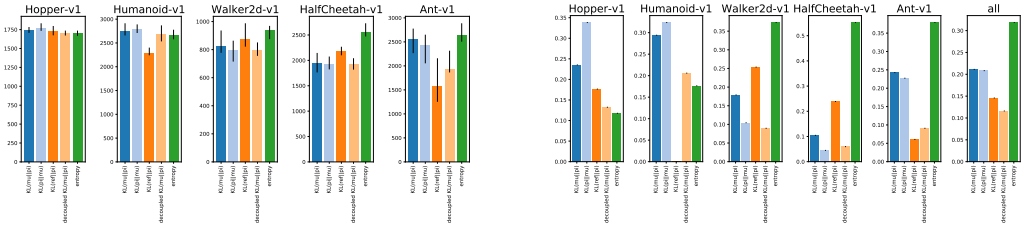

Figure 82: Analysis of choice `Regularizer (in case of constraint) (C34)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

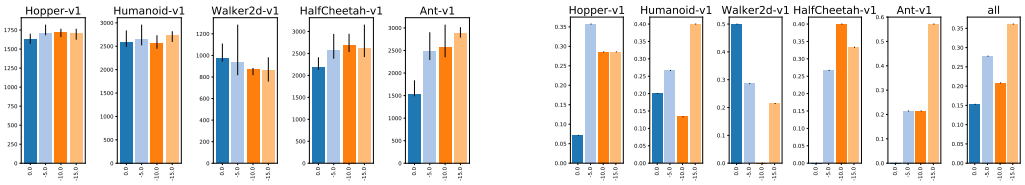

Figure 83: Analysis of choice `Threshold for entropy` $H(\pi)$ `(C40)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

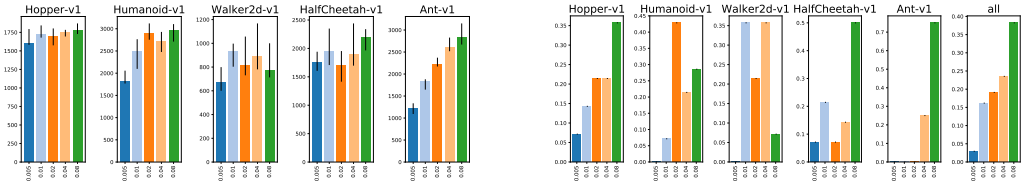

Figure 84: Analysis of choice `Threshold for KL`$(\mu||\pi)$ `(C35)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

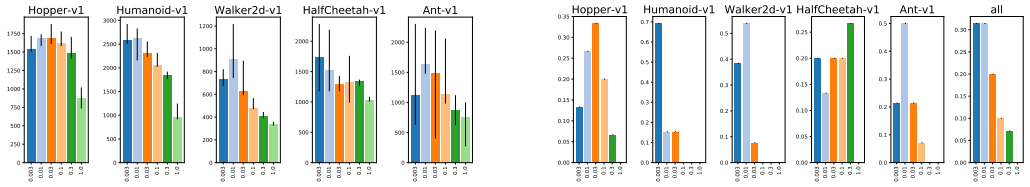

Figure 85: Analysis of choice `Regularizer coefficient for mean in decoupled` $\mathrm{KL}(\mu||\pi)$ `(C44)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

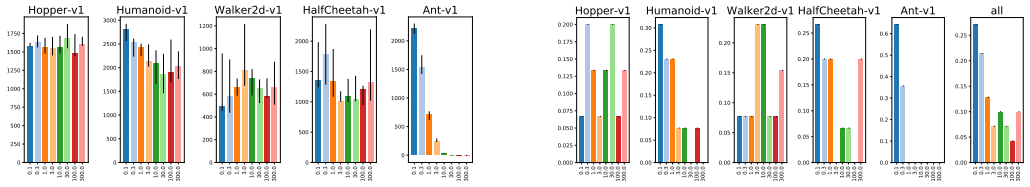

Figure 86: Analysis of choice `Regularizer coefficient for std in decoupled` $\mathrm{KL}(\mu||\pi)$ `(C45)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

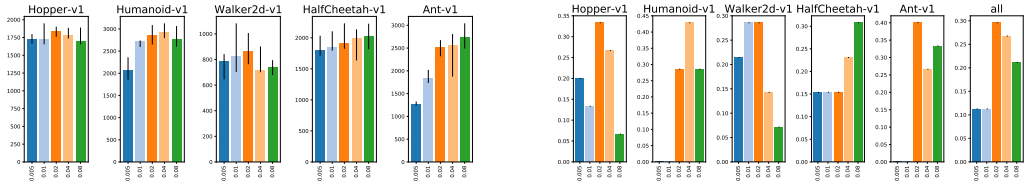

Figure 87: Analysis of choice `Threshold for` $\mathrm{KL}(\pi||\mu)$ `(C36)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

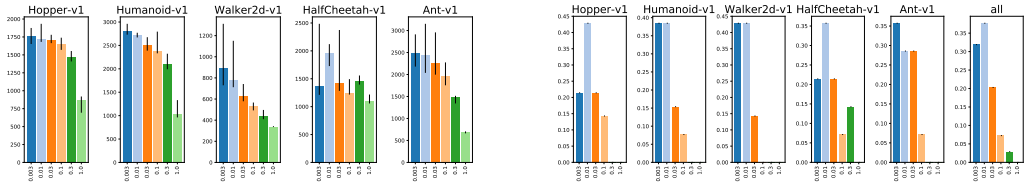

Figure 88: Analysis of choice `Regularizer coefficient for` $\mathrm{KL}(\pi||\mu)$ `(C42)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

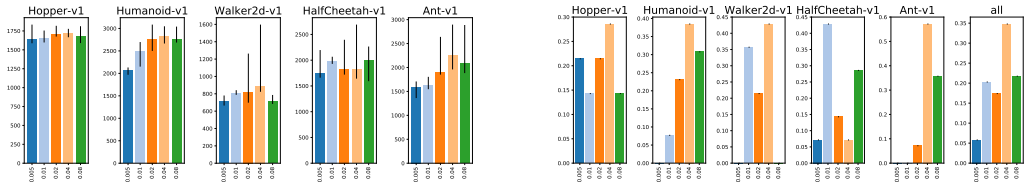

Figure 89: Analysis of choice `Threshold for mean in decoupled KL`$(\mu||\pi)$ `(C38)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

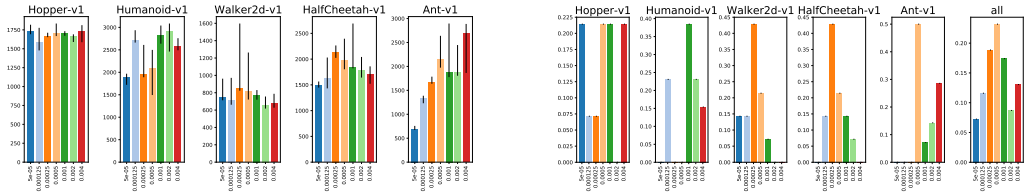

Figure 90: Analysis of choice `Threshold for std in decoupled KL`$(\mu||\pi)$ `(C39)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

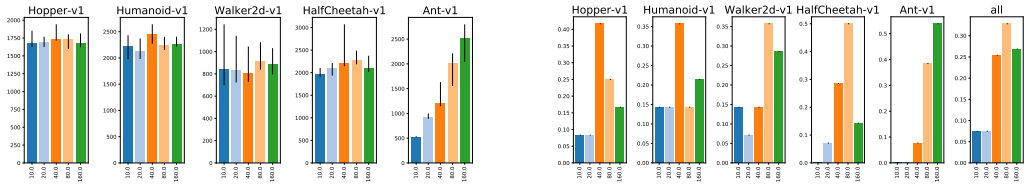

Figure 91: Analysis of choice `Threshold for KL`$(\text{ref}||\pi)$ `(C37)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).

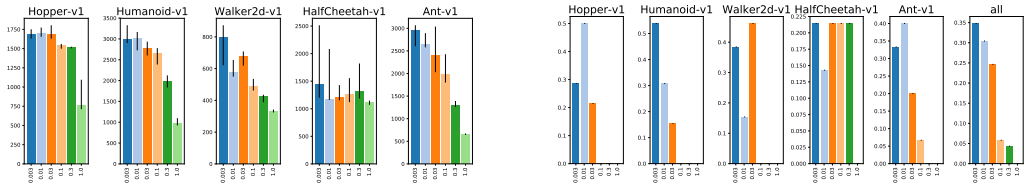

Figure 92: Analysis of choice `Regularizer coefficient for KL`$(\mu||\pi)$ `(C41)`: 95th percentile of performance scores conditioned on sub-choice (left) and distribution of sub-choices in top 5% of configurations (right).