Cascade - A sequential ensemble method for continuous control tasks

Anonymous authors
Paper under double-blind review

Keywords: Ensemble learning, reinforcement learning, continuous control, PPO.

Summary

Though reinforcement learning has been successfully applied to a variety of domains, there is still room left for improvement, in particular, in terms of the final performance. Ensemble Reinforcement Learning (ERL) tries to enhance reinforcement learning techniques by using multiple models or algorithms. We propose a novel ERL technique, called Cascade which in the context of continuous control tasks and with PPO as the base training algorithm clearly outperforms standard PPO in terms of the final performance. To shine light on the working mechanisms of Cascade, we conduct ablation studies, showing how the different components of Cascade contribute to its overall performance. Furthermore, we demonstrate that Cascade has a robust monotonicity as the ensemble's performance increases with each additional base agent even when weak base agents are added in large numbers.

Contribution(s)

1. The proposition of a novel Ensemble Reinforcement Learning (ERL) algorithm Cascade for continuous control tasks that outperforms its base learner.
   Context: To the best of our knowledge, there is no prior work where the ensemble policy uses a convex combination of its base learners and still gains a significant performance advantage.
2. By multiple ablation studies, we investigate the mechanisms contributing to Cascade's performance.
   Context: We show that Cascade relies on all base learners being trained at all stages of the training process as well as Cascade relying on sequentially adding base learners instead of starting with the final network. Lastly, we show that Cascade can chain an arbitrary number of base learners of arbitrary strengths without a loss in performance.

Cascade - A sequential ensemble method for continuous control tasks

Anonymous authors
Paper under double-blind review

Abstract

Though reinforcement learning has been successfully applied to a variety of domains, there is still room left for improvement, in particular, in terms of the final performance. Ensemble Reinforcement Learning (ERL) tries to enhance reinforcement learning techniques by using multiple models or algorithms. We propose a novel ERL technique, called Cascade which in the context of continuous control tasks and with PPO as the base training algorithm clearly outperforms standard PPO in terms of the final performance. To shine light on the working mechanisms of Cascade, we conduct ablation studies, showing how the different components of Cascade contribute to its overall performance. Furthermore, we demonstrate that Cascade has a robust monotonicity as the ensemble's performance increases with each additional base agent even when weak base agents are added in large numbers.

1 Introduction

Ensemble Reinforcement Learning (ERL) tries to improve reinforcement learning methods by using multiple models or algorithms. Advantages of ERL methods might stem from an increase in diversity, increased representational capabilities, or avoidance of the reliance on a single, possibly inaccurate predictor/decision-maker (Dietterich, 2000).

This paper's research question was the development of a novel ERL algorithm that could perform well on continuous control tasks. Hence our contribution is the proposal and experimental investigation of an ERL algorithm named Cascade which sequentially adds agents to an ensemble as opposed to starting with the full ensemble. The latest agent simultaneously learns a policy and how to integrate itself into the ensemble consisting of all previous agents. Additionally, the ensemble is designed in such a way that all ensemble members, once they are added, can refine their policy. Furthermore, Cascade will set the ensemble up so that it can be trained end-to-end, thereby requiring the selection of an arbitrary policy-based RL algorithm (such as PPO).

We show experimentally that Cascade, when using PPO to train the ensemble, outperforms PPO on a variety of continuous control tasks in terms of the final performance. Additionally, the individual components of Cascade were investigated, discovering that the sequential nature combined with allowing every ensemble member to be trainable at all times, are the biggest performance contributors. Furthermore, we show that Cascade is robust to the number of base agents as well as their expressivity by demonstrating that even for large ensembles consisting of weak base agents, adding a base agent almost always yields a performance improvement. In particular, we show that even agents, which on their own lack the capacity to solve the considered tasks, can be effectively combined by Cascade to solve these tasks. In this paper we focused mainly on PPO as it was the training algorithm that synergizes best with Cascade.

The paper is structured as follows: In Section 2 we give a short overview of ERL methods for continuous control tasks. Afterwards, in Section 3 we will define the Cascade algorithm and explain its intuition. The subsequent Section 4 presents experiments to answer structured research questions on Cascade's capabilities. In Section 5 we will give reasons for why Cascade works as well as it does and discuss avenues for future work. At last, in Section 6 we summarize our main findings.

2 Related work

A plethora of different approaches to continuous control tasks using ERL exists. Januszewski et al. (2021) use an ensemble of TD3-like (Fujimoto et al., 2018) agents to counteract the bias an individual Q function and consequently the actor π has by using the ensemble to average out the bias. They do this by simultaneously training an ensemble of differently initialized agents. At evaluation time, the ensemble's action is simply the average of the individual agents' actions.

Li et al. (2023) propose a modification, called TEEN , which also trains an ensemble of actor-critics but such that their state-action visit distributions are as diverse as possible to encourage exploration. Additionally, they directly decrease the bias of the Q-functions targets by using a shared target that contains the average of a random subset of the Q-function ensemble.

Saphal et al. (2021) developed SEERL which creates an ensemble of agents within just one training cycle by regularly saving the agent's current parameters. To promote diversity between the ensemble members, they use a cyclical learning rate which ensures convergence before parameters are saved and enough disturbance to get out of any local minimum. After training, they pick a subset of the saved agents and combine them without any extra training phase by averaging or binning the individual action outputs.

Chen & Huang (2023) introduce a hierarchical ensemble method named HED that trains an ensemble of actor-critics on two different levels. First, on a lower level, where the agents are trained independently of each other, and secondly on a higher level where the ensemble's policy, which is the average of the individual actor outputs, is updated in a fashion that promotes cooperation among the base learners.

Liu et al. (2023) took a slightly different approach that aims at increasing the stability of the ensemble algorithm. During training, they maintain an ensemble of actor-critics and compute their standard policy gradients which are used to construct a single so-called robust policy gradient which is then applied to all agents. The robust policy gradient is chosen in such a fashion that when applied to all agents at once, maximizes the average of the expected returns of all ensemble agents.

Yet another approach, SUNRISE, is demonstrated by Lee et al. (2021) who also use an actor-critic ensemble. They use the standard deviation of the ensemble's Q-functions as an uncertainty measure which is used to weigh samples and to construct an upper-confidence bound (UCB) during inference. More specifically, each ensemble agent samples an action during inference. The action that maximizes the UCB (which is a weighted sum of the Q-functions mean and standard deviation) is chosen as the ensemble's action.

Jaderberg et al. (2017) showed that an ensemble of agents can be used to optimize a hyperparameter schedule. They keep a fixed size population of agents which are trained individually in parallel. Regularly, the lowest scored agents are discarded and replaced by the better performing ones. To promote further hyperparameter exploration, once the training state and hyperparameters have been copied, the hyperparameters are slightly perturbed. Their method is applicable to continuous control tasks even though it was not specifically designed for it.

In contrast to these works, Cascade does not necessarily produce an ensemble of agents that would perform well on their own. The idea is that not every agent needs to know everything about the task. They just need to be able to account for the weaknesses of other agents. Hence, Cascade solely focuses on the performance of the entire ensemble and how the agents can optimally cooperate and complement each other.

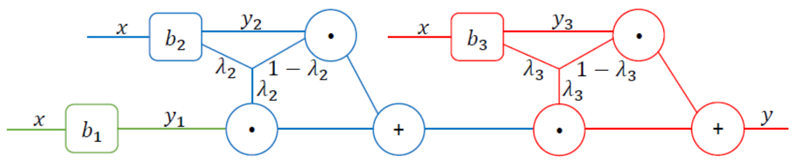


Figure 1: Cascade net of size 3. Except for the bottom net b_1, each net outputs a fallback action $\lambda_i \in [0, 1]$ which is used for the convex combination.

More generally, Song et al. (2024) published a comprehensive survey on ERL thereby giving an overview of the entire field. Their scope is broader than the restriction to continuous control as done here.

3 Method

The main idea behind Cascade is to sequentially train base agents in such a fashion that a new base agent always tries to optimally complement the current ensemble. This means that the base agent should only learn to perform well or learn to adjust the ensemble in a subspace where the ensemble performs poorly and otherwise delegate the task of choosing an action to the ensemble. The new base agent can then be integrated into the ensemble to obtain a new, improved ensemble.

A naive approach to this is to train a base agent on a surrogate environment that extends the action space of the original environment by an action we call the *fallback action*. The values of this fallback action lie in the range $[0, 1]$. With a probability equal to the fallback action, instead of executing the given action on the original environment, the action of the current ensemble is executed. The newly trained base agent can then be integrated into the ensemble. The action of the new ensemble is the action of the latest base agent with probability equal to its fallback action and otherwise the action of the previous ensemble. Since the base agent could learn to fall back with a probability that is arbitrarily close to 1 in all states, it is ensured that theoretically an arbitrary amount of base agents can be chained without a loss in performance.

However, by sampling and then taking either the action of the base agent or the ensemble at each step as well as treating the ensemble as a blackbox (surrogate environment), non-differentiability is introduced. Additionally, the weaknesses of older base agents cannot be corrected, as only the latest base agent is trained. To overcome this, a convex-combination of the current ensemble-policy π_e and the new base agent π_b is used to obtain a policy π:

$$\pi(x) = \lambda_w(x) \cdot \pi_e(x) + (1 - \lambda_w(x)) \cdot \pi_b(x) \tag{1}$$

where x is a state and λ_w is the fallback action of the base agent. This policy π is then used for training. Both the parameters of π_e and π_b remain trainable.

Any model that is used to represent base agents has to output the fallback action $\lambda(x) \in [0, 1]$ in addition to its usual action outputs. In the case of artificial neural networks (ANNs), the policy π can be approximated by taking a convex combination of the network's action outputs directly (instead of the distributions they represent. For example, the convex combination of two normal distributions is in general not the same as the distribution induced by the convex combination of their parameters). If several ANNs are stacked this way, then this results in an architecture called *Cascade net*. Fig. 1 shows how 3 base agents are chained which results in a Cascade net of size 3. For reference, the pseudocode of the proposed technique to sequentially train a Cascade net is presented in the supplementary materials. This algorithm will be referred to as *Cascade*.

4 Experiments

First, it will be demonstrated that Cascade is able to clearly outperform the baseline on most MuJoCo (Todorov et al., 2012) environments using PPO. The subsequent experiments will then shift their focus towards studying individual components of Cascade to gain insights into the reasons for this performance increase.

4.1 Experiment setup

Cascade as described in Section 3 was run on various MuJoCo tasks with a focus on the Ant-v4 and Walker2d-v4 environments because these two differ greatly in terms of dynamics and complexity which gives a good impression of how Cascade behaves in different settings. When not mentioned otherwise, PPO was used as the algorithm to train the Cascade, 6 million environment steps took place, and the Cascade net was extended every 1 million steps. For details and how performances (and other metrics) were measured see supplementary materials Section B. The code for our experiments is publicly available at [1]: `https://anonymous.4open.science/r/Cascade-6910`.

4.2 Standard Cascade

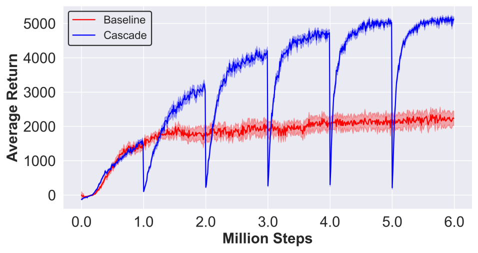
Fig. 2 shows the result when Cascade without any modifications as described in the experiment setup was run on Ant-v4, Walker2d-v4, Humanoid-v4, Hopper-v4, and HalfCheetah-v4. We refer to this setup as *standard Cascade*. For comparison, each graph also shows the PPO baseline (see supplementary materials Section B.4) trained for 6 million steps. However, as mentioned in the study design, different baseline performances emerge for different training durations since learning rate annealing is used (see supplementary materials Fig. 9). Except for the Hopper-v4 and Walker environment where Cascade performs slightly worse or only slightly better than the baseline, Cascade decisively outperforms the baseline in all other environments after 6M steps. While in Ant-v4, Humanoid-v4, and HalfCheetah-v4 the baseline performances are about equal when Cascade consists only of one base net, Cascade takes the lead in performance shortly after a second base net is added and grows the lead from there.

Next, we compared Cascade to a slightly different PPO agent that mimics a base agent of Cascade as closely as possible. Therefore, a single base net (see supplementary materials Section B) will be trained with PPO for 6 million steps. To mimic the conditions under which Cascade trained, the learning rate was cyclical, linearly decreasing from its initial value to 0, every 1 million timesteps. Other than changing the network architecture and learning rate schedule, everything else is identical to the baseline PPO. Fig. 3 shows the performance results compared with standard Cascade.

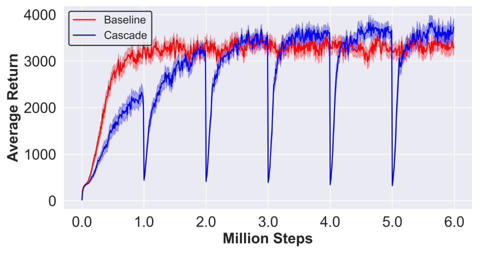
Surprisingly, in Ant-v4, a single base agent reached a significantly higher performance than the PPO baseline which just differs in the network size and learning rate schedule, which might suggest that the byproduct of having a cyclical learning rate (remember, a new PPO instance is used every time the Cascade net expands) adds to the performance of Cascade. This is not the case however, as Cascade is very robust to different learning rate schedules (for details, see supplementary materials F). Nonetheless, there is still a clear gap between the base agent and Cascade both in Ant-v4 and Walker2d-v4.

Additionally, we evaluated the Cascade algorithm with SAC and DDPG as the base training algorithm, as detailed in the supplementary materials in Section H. Using these training algorithms Cascade did not outperform PPO, therefore we chose PPO as the default training algorithm for subsequent experiments. It remains to be researched why Cascade synergizes so well with PPO in particular.

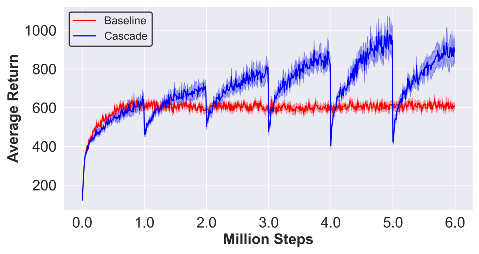
[1]The repository has been anonymized for review and will be replaced once the paper has been accepted.



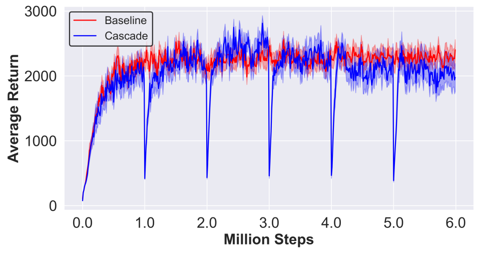
(a) Ant-v4



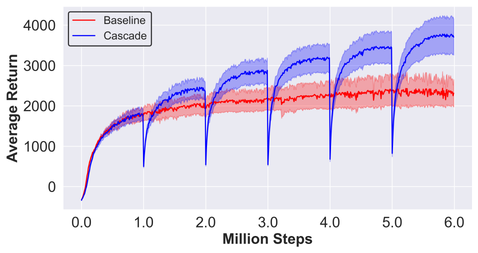
(b) Walker2d-v4



(c) Humanoid-v4



(d) Hopper-v4



(e) HalfCheetah-v4

Figure 2: Performance graphs of standard Cascade (blue) vs PPO (red) on Ant-v4 (top left), Walker2d-v4 (top right), Humanoid-v4 (bottom left), Hopper-v4 (bottom middle) and HalfCheetah-v4 (bottom right) for 6 million environment steps.

Question 1: Can Cascade outperform any RL-algorithm while using the same RL-algorithm to train its base agents?

Yes, Cascade using PPO as the training algorithm outperforms PPO in most of the considered environments. In the cases Cascade outperformed PPO it managed to do so rather early, requiring only two base agents. And in the environments Cascade does not or barely outperform the baseline, the peak performance throughout training at least draws even with peak PPO performance.

Though this paper's focus was continuous action space environments, we also applied a slightly modified version of Cascade to discretized versions of the MuJoCo environments (see supplementary materials Section I) and found that though Cascade also outperforms PPO in most of the discrete settings, the margins were very slim and Cascade took far longer to converge.

The final Cascade nets, obtained after running standard Cascade on Ant-v4 and Walker2d-v4, were investigated in more detail to see how many changes were applied to the initial base agent after its initial training phase. *Ant*: In this case, the entire Cascade mostly relies on the bottom base net (i.e. the net at the bottom of the Cascade net with no fallback action): While the entire Cascade net achieves an average performance of around 5.1k the bottom net alone reaches an average return

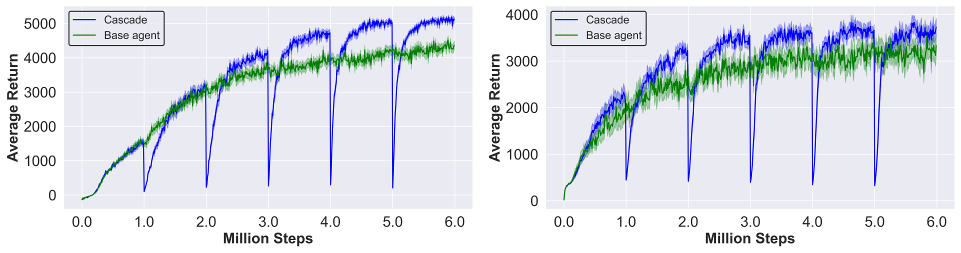


Figure 3: Performance graph of Cascade (blue) and PPO with cyclical learning rate on a base net of Cascade (green) for 6 million environment steps on Ant-v4 (left) and Walker2d-v4 (right).

of approximately 3.2k. All other policies further up the chain have a negative average return. As hypothesized in the algorithm-description this base policy improved even past 1 million steps when other agents were starting to be stacked on top. At 1 million steps the base policy performance was at around 1.5k while it was approximately 3.2k by the end of training. *Walker2d*: These characteristics differ for Walker2d-v4, as instead of improving past 1 million steps, the base policy degraded, going from 2.5k to approximately 600. However, all other policies further up the chain still perform poorly on their own, each with a performance of less than 100. Nonetheless, the entire chain still performs well, performing slightly better than the PPO baseline with a performance of roughly 3.9k at the end of training.

Question 2: Does Cascade still apply big changes to base agents once they are no longer at the top of the cascade?

Yes, judging by the performance of the bottom base agent. However, depending on the environment its performance may degrade or improve. Either way, the Cascade improves.

4.3 Keeping base nets frozen

Next, it was investigated how allowing every base net to be trainable at every stage of the training process impacted the performance. For this, only the parameters of the latest base agent were trainable, the rest were kept frozen. Fig. 4 shows the performance graphs compared to standard Cascade. It can be observed that even though each training iteration leads to a minor improvement in performance for the frozen version, the overall performance is drastically worse compared to the non-frozen version for Ant-v4 and slightly worse for Walker2d-v4. This suggests that it is in fact vital for all base agents to remain trainable to correct for imperfections, adapt to new normalization states (observation/reward normalization is used by default, for details see supplementary materials Section B.4) and in general make the entire network less rigid. The difference could in fact be partially explained by the ability to adapt to new normalization states as both versions only slightly differ in their final performance when no normalization is used (see supplementary materials Fig. 13). However, in the normed setting, older base agents are modified beyond just adapting to new normalization states as could be seen in the previous section where the policy of the oldest base agent changed drastically which was indicated by the drastic change in its average performance.

Question 3: What is the impact on the performance when keeping the parameters of the current ensemble frozen when training a new base agent?

Keeping the parameters frozen severely worsens performance.

4.4 Starting with the final Cascade net

Next, the effect of sequentially adding base agents to the Cascade was tested. For this, the Cascade net obtained at the end of training was used right from the start. To have the same training conditions, the learning rate was cyclical, linearly decreasing from its initial value to 0, every 1 million environment

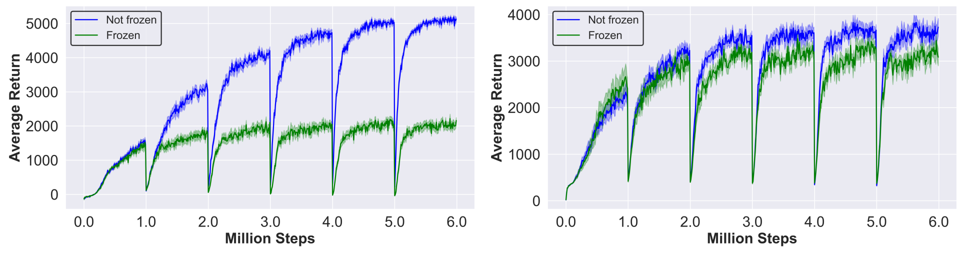


Figure 4: Performance graphs of standard Cascade (blue) vs Cascade with the weights of all but the latest base-net frozen (green) on Ant-v4 (top) and Walker2d-v4 for 6 million environment steps.

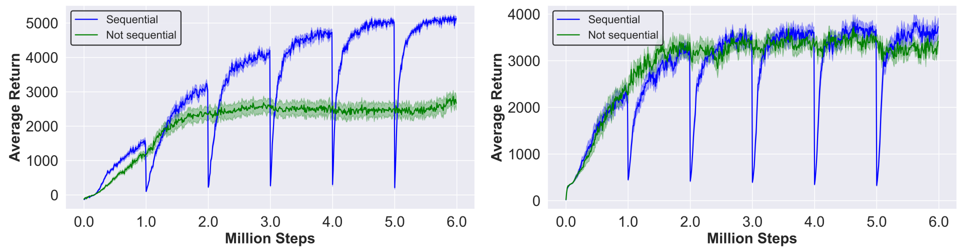


Figure 5: Performance graphs of standard Cascade (blue) versus Cascade where the final net of standard Cascade is trained right from the start (green) on Ant-v4 (top) and Walker2d-v4 (bottom) for 6 million environment steps.

steps (as PPO used to train the base agents in standard Cascade uses learning rate annealing. See supplementary materials Section B). Fig. 5 shows the performance results for this experiment on Ant-v4 and Walker2d-v4.

In both environments, the sequential variant clearly outperforms its non-sequential counterpart. While in Ant-v4, the sequential variant outperforms right from the start and ends with a final performance almost twice that of the non-sequential version, in Walker2d-v4 the sequential variant only takes the lead in performance during the third iteration and ends only with a small advantage.

This shows that the architecture itself is not the deciding factor for performance. In fact, there seems to be nothing inherently special to this architecture, as it performs worse than the standard feedforward architecture used for the baseline experiments.

Question 4: How does the performance of training the final Cascade net right from the start compare to adding the base agents sequentially?

Training the final Cascade net right from the start performs significantly worse than adding the base agents sequentially, thus proving that it is in fact vital to slowly build up the Cascade.

4.5 Fallback action characteristics

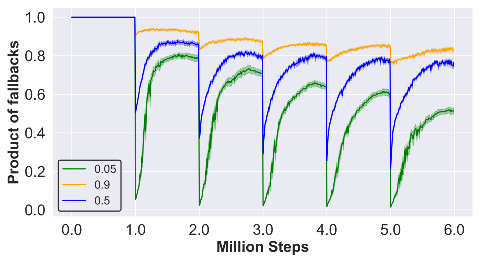
Now, the tendency of base agents to fallback will be looked into. This was done by measuring the average fallback actions of different base nets over the course of training. This was done for three different fallback action initializations, namely $\{0.05, 0.5\ \textit{(Default)}, 0.9\}$ (To see how this is done, see supplementary materials Section E).

First, simply the product of all fallback actions of all base agents within the Cascade net was measured. This is the contribution of the bottom base net to the final output. Fig. 6 shows this product over the course of training on Ant-v4 and Walker2d-v4. In all settings, the fallback product slowly decreases as the Cascade grows. However, during every iteration, the product mostly increases (except for the first two iterations with the 0.9 initialization). This shows that the base agents have a clear

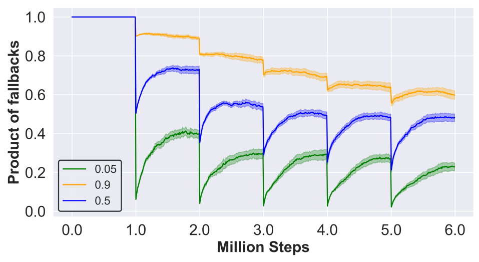
Env	**Init**	λ_2	λ_3	λ_4	λ_5	λ_6
Ant	0.05	0.97	0.99	0.99	0.97	**0.55**
Ant	0.5	0.96	0.96	0.96	0.96	**0.88**
Ant	0.9	**0.95**	0.97	0.97	0.97	0.96
Walker2d	0.05	0.66	0.82	0.79	0.80	**0.56**
Walker2d	0.5	0.82	0.85	0.92	0.90	**0.81**
Walker2d	0.9	**0.84**	0.89	0.91	0.95	0.94

Table 1: The final average fallback action of each base net in the Cascade net when standard Cascade was run on Ant-v4, Walker2d-v4 and with different fallback initializations for 6 million environment steps. λ_2 is the average fallback of the second oldest base net and λ_6 is the fallback of the one added last. The first base net isn't listed because it does not have a fallback action.

tendency towards fallbacking. This tendency is slightly stronger in Ant-v4 than in Walker2d-v4. For reference, if each base net would assign 0.5 to its fallback action then the bottom base net would only account for $0.5^5 = 0.03125$ of the final output. Since in each setting (except the 0.05 initialization in Walker2d-v4) the fallback product is ≥ 0.5 means that it is the bottom base net that contributes most to the final output.



(a) Ant-v4



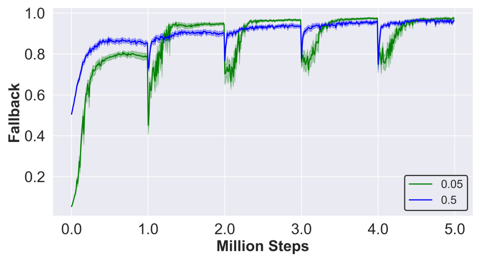
(b) Walker2d-v4

Figure 6: Product of all base agents' fallback actions over the course of training for 6 million environment steps with fallback initializations of 0.9 (yellow), 0.5 (blue) and (0.05) green.

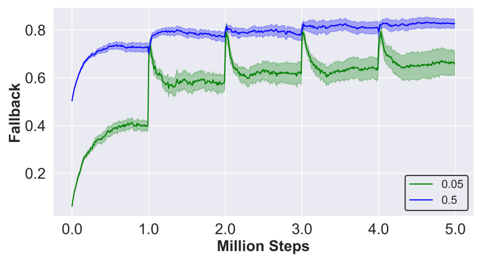
To see how the product is made up and how different base nets contribute to it, the fallback action of the individual base nets was tracked. Tab. 1 lists the final average fallback action of each base net for all settings. In the case of the 0.05 or 0.5 initialization, it is the 6th base net that contributes either the most or second most to the final output of the Cascade net. In the case of the 0.9 initialization, the base nets (except the bottom one) contribute roughly equally to the final output. Note that the contribution of the i-th net is $(1 - \lambda_i) \cdot \lambda_{i+1} \cdot \ldots \cdot \lambda_6$ for $i > 1$ and $\lambda_2 \cdot \ldots \cdot \lambda_6$ for $i = 1$. Still, in all cases (except for 0.9 initialization) and no matter the position in the Cascade, the final base agent's fallback is always significantly higher than what it started with, in most cases ending up well above 0.8.

Why for the cases 0.05 and 0.5 the 6th base net is the biggest or second biggest output-contributor can be best understood by looking at the fallback graph over the course of the training of an individual base agent. Fig. 7 shows this graph for the second to bottom base net for Ant-v4 and Walker2d-v4.

It becomes evident, that the base nets only reach their highest fallback value once they are no longer at the top of the Cascade. So naturally it is the last added base net (in this case the 6th) that has the lowest fallback value. On top of that, its final output contribution is directly given by $1 - \lambda_6$ (i.e. no other nets can lower the contribution other than itself). This explains why it is either the biggest or second biggest contributor to the final output.



(a) Ant-v4



(b) Walker2d-v4

Figure 7: The average fallback action value for the second oldest base agent (i.e. the first one that has a fallback action) over the course of 5 million environment steps after this base agent has been added. This value has been plotted for Cascade with 0.5 (blue) and 0.05 (green) as the fallback initialization. Note that the spiky fallback change that occurs whenever a base net is added to the Cascade comes from the sudden change in the state distribution that goes along with adding a new base net.

Question 5: Do the base agents have a tendency towards fallbacking or not and which base agents contribute most to the final output?

Given different fallback action initializations of different magnitudes, all base agents always learned to assign a large value to the fallback action. Hence there is a clear tendency to fallback. In general, it is the bottom base net that contributes the most and the last added base net that contributes the second most to the final output.

4.6 Tiny Cascade nets

To push the hypothesis to its limits, Cascade will be applied to extremely small base nets with only one hidden layer of size 1. A lower training duration of only half a million steps will also be considered to make it hard for Cascade to effectively chain base agents. 16 of these tiny base nets will be combined with Cascade. Fig. 8 shows the results of this experiment.

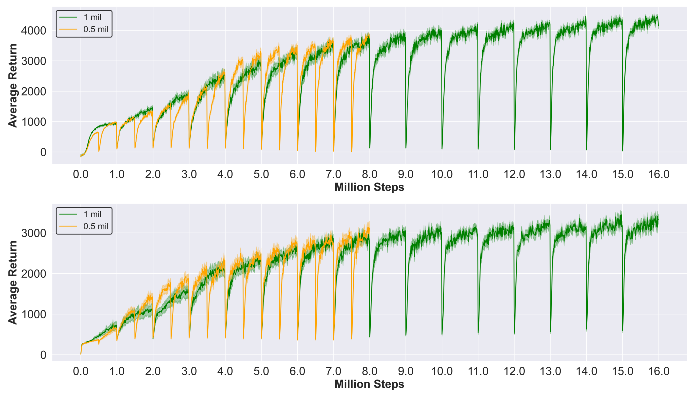


Figure 8: Performance graphs of Cascade with minimal base agents (one hidden layer of size 1) that are added every 1 million (green) or 0.5 million (yellow) steps for 16 million environment steps on Ant-v4 (top) and Walker2d-v4 (bottom).

As expected, the performances are much worse than those of standard Cascade. However, even though 16 base agents were chained, there was always a monotone performance increase from one iteration to another, even near the end when the Cascade consisted already of more than 10 base agents. At worst, the performance after a base agent was added only drew even to that of the previous iteration. This even held true for the case with only half a million training steps per iteration.

Question 6: Can a high number of base agents be chained without a loss in performance?

Yes. In all considered settings (Standard Cascade and Cascade with tiny base agents) adding additional agents to the Cascade has always been beneficial regardless of the training duration per iteration or the expressive capabilities of the added base agents.

5 Discussion

5.1 Reasons for the performance gain

Though the conducted experiments showed that the performance increase of Cascade over the standard PPO baseline is nontrivial and coming mainly from the fact that the base agents are added sequentially and that even older base nets are trainable at all times, the reasons why this is so effective remain unclear. Likely, there are many factors contributing to it. Two of these might be the following: First, regularly adding a net base agent helps with exploration: Directly after adding a base net the output of the entire cascade is greatly perturbed such that it initially becomes essentially random, as can be observed by the sharp drops in the performance graphs, thereby causing a huge shift in the state-distribution. Secondly, adding fresh networks prevents the Cascade net from losing its plasticity. Neural plasticity loss (Lyle et al., 2023) is a phenomenon where neural networks slowly lose their ability to train and adapt to new optimization targets which is present in reinforcement learning. In Section J in the supplementary materials, we measured the plasticity gain caused by adding a new ensemble member to the cascade.

5.2 Future work

First, pinpointing the exact causes of performance gain remains an open question that should be addressed in future work. In particular, it can be investigated why Cascade synergizes so well PPO yet yields only very little improvement in conjunction with other RL-algorithms. Even when considering only PPO, there are performance differences depending on the environment. It is worth investigating why in Hopper specifically, Cascade did not outperform PPO.

Future work might also include generalizing the strictly sequential nature of the resulting chain. It could be interesting to see what happens if the base nets choose between k nets (instead of $k = 1$) to fall back to. One possibility to implement this is to train groups of k nets (perhaps as diverse as possible), each of which has fallback access to each of the k nets of the previous group. Or more extremely, the entire output is directly determined by the latest base agent in the cascade, which assigns a fallback weight to all other base nets. In standard Cascade, all agents remain trainable and it showed that freezing all base nets except the top one mostly hampers performance. However, it could be interesting to see if this holds true if some sort of attention mechanism is used to dynamically control which base nets are frozen and which remain trainable. Perhaps at a certain size of the cascade, no new agents should be added and it is beneficial to train inner agents and leave the top agent frozen.

Application to different domains: The behavior of Cascade could be studied on a larger suite of environments. For example, surveying the Atari benchmark could lead to new insights. Additionally, multi-agent problems with a combinatorial action space might be worth investigating as they come with a natural way of delegating the task between the action nets (one action net for each agent).

Hyperparameter optimization: The RL-baselines used for comparison were highly optimized version of PPO, SAC and DDPG. However, little to no hyperparameter optimization took place for Cascade. And the hyperparameters that were tuned, were high-level ones such as the number of base agent's training steps in Cascade. However, Cascade might be sensitive to the hyperparameters of the employed training algorithm.

6 Summary

In this work, an ERL method, called Cascade has been proposed and experimentally verified on some MuJoCo environments. With some minor high-level hyperparameter tweaking, Cascade managed to decisively outperform the PPO baseline on 4 out of 5 MuJoCo environments. Also, standard Cascade proved to be an effective tool for combining a large number of base agents no matter their training time or expressive capabilities. The characteristics of Cascade were investigated and it was found that the base agents have a natural tendency to assign a large value to their fallback action which led to the bottom base agent contributing most to the Cascade's output. By observing the bottom base agent's performance, it was discovered that Cascade makes heavy use of the freedom to change older agents in the cascade. In fact, keeping older base agents frozen proved detrimental to performance in a normalized environment. It also proved vital that the Cascade is built sequentially since starting with the final Cascade net also hampered performance thus showing that the architecture is not special but rather its sequential buildup.

References

Gang Chen and Victoria Huang. Ensemble reinforcement learning in continuous spaces – a hierarchical multi-step approach for policy training. In Edith Elkind (ed.), *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, pp. 3514–3522. International Joint Conferences on Artificial Intelligence Organization, 8 2023. DOI: 10.24963/ijcai.2023/391. URL https://doi.org/10.24963/ijcai.2023/391. Main Track.

Thomas G. Dietterich. Ensemble Methods in Machine Learning. In *Multiple Classifier Systems*, pp. 1–15, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-45014-6.

Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1587–1596. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/fujimoto18a.html.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In Jennifer G. Dy and Andreas Krause (eds.), *International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1856–1865. PMLR, 2018.

Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022. URL http://jmlr.org/papers/v23/21-1342.html. Code last accessed on 9th July 2023.

Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks. *CoRR*, abs/1711.09846, 2017. URL http://arxiv.org/abs/1711.09846.

Piotr Januszewski, Mateusz Olko, Michał Królikowski, Jakub Swiatkowski, Marcin Andrychowicz, Łukasz Kuciński, and Piotr Miłoś. Continuous control with ensemble deep deterministic policy gradients. In *Deep RL Workshop NeurIPS 2021*, 2021. URL https://openreview.net/forum?id=TIUfoXsnxB.

Kimin Lee, Michael Laskin, Aravind Srinivas, and Pieter Abbeel. SUNRISE: A simple unified framework for ensemble learning in deep reinforcement learning. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pp. 6131–6141. PMLR, 2021. URL http://proceedings.mlr.press/v139/lee21g.html.

Chao Li, Chen Gong, Qiang He, and Xinwen Hou. Keep various trajectories: Promoting exploration of ensemble policies in continuous control. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

Guoqiang Liu, Gang Chen, and Victoria Huang. Policy ensemble gradient for continuous control problems in deep reinforcement learning. *Neurocomputing*, 548:126381, 2023. ISSN 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2023.126381. URL https://www.sciencedirect.com/science/article/pii/S0925231223005040.

Clare Lyle, Zeyu Zheng, Evgenii Nikishin, Bernardo Ávila Pires, Razvan Pascanu, and Will Dabney. Understanding plasticity in neural networks. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 23190–23211. PMLR, 2023. URL https://proceedings.mlr.press/v202/lyle23b.html.

Rohan Saphal, Balaraman Ravindran, Dheevatsa Mudigere, Sasikanth Avancha, and Bharat Kaul. SEERL: sample efficient ensemble reinforcement learning. In Frank Dignum, Alessio Lomuscio, Ulle Endriss, and Ann Nowé (eds.), *AAMAS '21: 20th International Conference on Autonomous Agents and Multiagent Systems, Virtual Event, United Kingdom, May 3-7, 2021*, pp. 1100–1108. ACM, 2021. DOI: 10.5555/3463952.3464080. URL https://www.ifaamas.org/Proceedings/aamas2021/pdfs/p1100.pdf.

Yanjie Song, Ponnuthurai Nagaratnam Suganthan, Witold Pedrycz, Junwei Ou, Yongming He, Ying-Wu Chen, and Yutong Wu. Ensemble reinforcement learning: A survey. *Appl. Soft Comput.*, 149 (Part A):110975, 2024. DOI: 10.1016/J.ASOC.2023.110975. URL https://doi.org/10.1016/j.asoc.2023.110975.

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*, pp. 5026–5033. IEEE, 2012. DOI: 10.1109/IROS.2012.6386109.

Supplementary materials

A Cascade pseudocode

Algorithm 1: Cascade

```
Input: Cascade size $n$, number of training steps per iteration $k$
$E \leftarrow \emptyset$        // Current ensemble
for $i = 1$ to $n$ do
    Initialize base net $b_i$
    $E \leftarrow E \cup \{b_i\}$
    Train $E$ for $k$ steps   // $E$ is interpreted
                              // as a Cascade net
end for
Return: $E$
```

B Experiment setup

If not explicitly mentioned otherwise, all experiments ran the Cascade-algorithm as described in Section 3. In the following, the default hyperparameter setup will be described. Modifications will always be explicitly mentioned.

B.1 Base nets

A base net as part of a Cascade net has the following architecture dependent on its position in the Cascade net:

- Bottom: The exact same architecture and initialization scheme was used for the standard PPO baseline (see supplementary materials Section B.4) except that the hidden size of the mean network is 16 instead of 64.
- Not Bottom: Same as bottom, except that only the output dimension of the mean network is increased by 1. The output of this extra dimension is the fallback action and followed by a sigmoid function to ensure its value lies in the range $[0, 1]$.

B.2 Cascade net

Initially, the Cascade net consists only of the bottom base net. After every k steps, the Cascade net C is extended by another base net b as follows: If $(m_b, \log_b)$ are the outputs (without fallback action) and $\lambda \in [0, 1]$ is the fallback action of b and $(m_C, \log_C)$ are the outputs of C, then the output of the extended Cascade net $\tilde{C}$ is $(\lambda m_C + (1 - \lambda)m_b, \lambda \log_C + (1 - \lambda) \log_b)$. As usual, these outputs are interpreted as the mean and logstd of a normal distribution. The set of base nets making up C is the set E in Algorithm 1 for Cascade.

B.3 Hyperparameters

If not mentioned otherwise, all experiments were conducted on Ant-v4 and Walker2d-v4 and as the long-term behavior was of interest, 6 million steps were performed for each experiment. The entire Cascade net is always trained with standard PPO as described in supplementary materials Section B.4 with the Cascade net used as the actor. The only exception to this is Section H in the supplementary materials where we deliberately vary the training algorithm. At each iteration (i.e. when a base net has been added) the Cascade net is always trained with a new instance of PPO (i.e. new value function and fresh learning rate schedule) but the observation/reward normalizations stay the same.

Different training times per iteration were tested (see supplementary materials Section D). Using 1 million steps (i.e. Cascade size $n = 6$ and $k = 1 \cdot 10^6$) proved to be the best choice and was taken as a default value for the subsequent experiments. Additionally, different initializations for the fallback action were tested (see supplementary materials Section E) and having an initial fallback action of 0.5 proved to be the best choice. Therefore, this was also chosen as the default value for the following studies. This default choice for running the Cascade algorithm will be referred to as standard/default Cascade.

B.4 Algorithm choice and hyperparameters

The exact version of PPO used to train the Cascade net is the one from CleanRL (Huang et al., 2022) (more specifically `ppo_continuous_action.py`). This is a highly optimized version of PPO as many optimizations such as observation/reward-normalization, learning-rate annealing, generalized advantage estimation, observation/reward-normalization, etc. have been implemented. When not explicitly mentioned otherwise, all hyperparameters are the default values of this implementation.

B.5 Measurements

For each experiment, every $10,000$ environment steps the exact policy that was used in training at that point was frozen and 10 evaluation runs were conducted. In particular, this evaluation policy was non-deterministic, as actions were sampled from a normal distribution with mean and standard deviation obtained from the network outputs. The average, undiscounted return from the 10 runs was used as the performance measure. Each experiment was run at least 10 times and no two run across all experiments used the same seed. Plots show the average of a measured metric (mostly the performance) over these 10 seeds along with the standard error for that experiment.

C Baselines

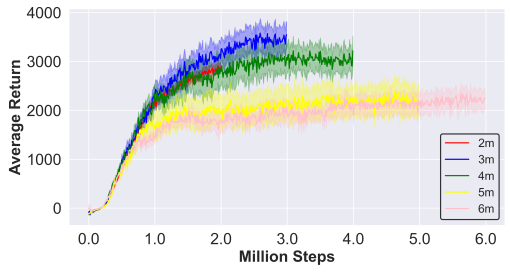
The chosen PPO version (see Section B.4) was run on MuJoCo environments to act as a baseline for the experiments. Fig. 9 shows the performance over the course of $k \in \{2, 3, 4, 5, 6\}$ million environment steps. Note, that for different k, different graphs emerge. This is because learning rate annealing is used in the PPO implementation.

D Varying base agents' training duration

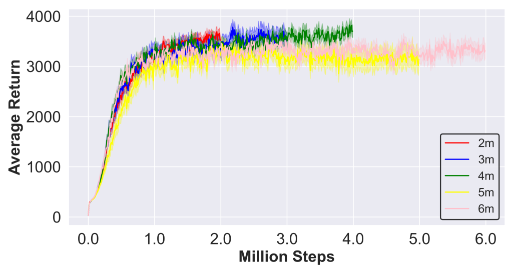
The impact of the rate at which new base agents are added was tested. To investigate this, the training duration per iteration was varied such that in 6 million timesteps a Cascade of size $n \in \{2, 3, 4, 5, 6, 7, 8\}$ could be built. For clarity, only $n \in \{2, 4, 6\}$ were visualized in Fig. 10. The others followed the general trend. A higher number of cycles led to a higher performance up to the peak of $n = 6$, after which the performance slightly degraded for $n \in \{7, 8\}$. Given these results, a training duration of 1 million steps per iteration was chosen as the default for the Cascade experiments which results in a Cascade of size 6.

E Fallback action initialization

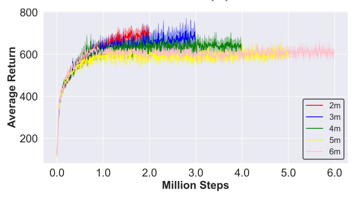
The impact of the fallback action initialization was tested in regard to performance. For this, the bias of the fallback-action output was initialized such that the fallback-action λ was initially equal to $\lambda \in \{0.05, 0.5, 0.95\}$ on average. This was possible because like any other neuron, the neuron responsible for the fallback action λ is a weighted sum of the activations of the prior layer $(l_i)_{i \leq n}$ followed by a sigmoid activation:



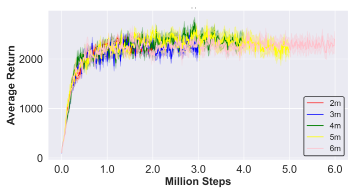
(a) Ant-v4



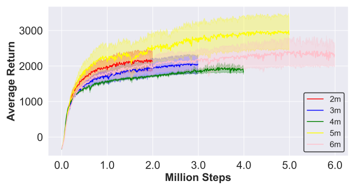
(b) Walker2d-v4



(c) Humanoid-v4

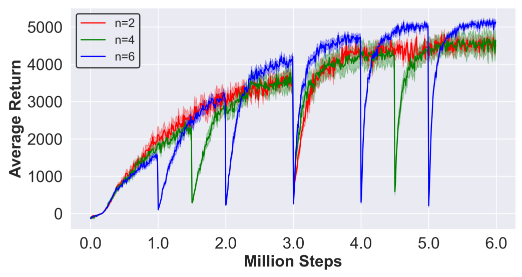


(d) Hopper-v4



(e) HalfCheetah-v4

Figure 9: Performance graphs of PPO for 2 million (red), 3 million (blue), 4 million (green), 5 million (yellow) and 6 million (beige) environment steps.



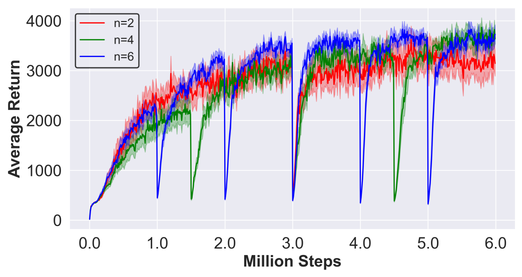


Figure 10: Performance graphs of Cascade when adding a new base agent every 3 million steps (red), every 1.5 million steps (green), and every 1 million steps (blue) on Ant-v4 (left) and Walker2d-v4 (right) for 6 million environment steps.

$$\lambda = \textit{sigmoid}(b_\lambda + \sum_{i=1}^{n} w_{\lambda,i} \cdot l_i) \tag{2}$$

But since the weights $(w_{\lambda,i})_{i\leq n}$ are initialized with mean 0 and low variance (see implementation CleanRL (Huang et al., 2022)), λ can initially be approximated by $\lambda \approx \textit{sigmoid}(b_\lambda)$. Therefore, by setting $b_\lambda = \textit{sigmoid}^{-1}(\lambda)$ the initial fallback action can be set.

Fig. 11 shows the results of this experiment with respect to performance for Ant-v4 and Walker2d-v4. The value of 0.5 appears to be the strongest choice for both environments. The poor performance of the 0.05 initialization could be explained by the natural tendency of the base agents to strive for a high fallback weight (shown in Section 4.5), hence the agent started in an unfavorable position. The 0.9 initialization was only slightly worse than 0.5. This could be explained by a lack of exploration as the newly initialized base nets are initially dominated by the base policy. Given these results, an initial fallback weight of 0.5 was picked as the default value for the Cascade experiments.

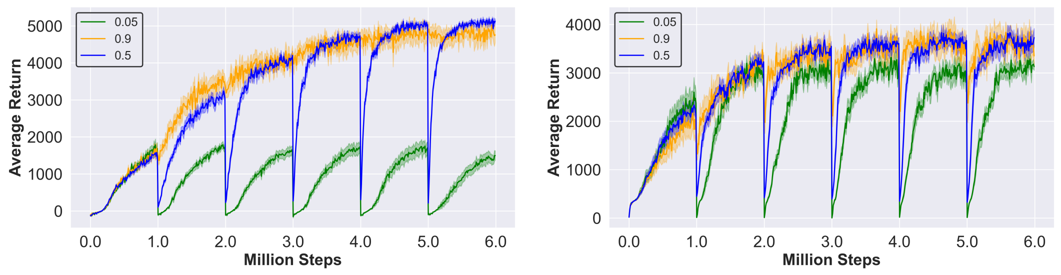


Figure 11: Performance graphs of Cascade on Ant-v4 (left) and Walker2d-v4 (right) for 6 million environment steps with 0.05 (green), 0.5 (blue) and 0.9 as the initial fallback value for new base agents.

F Learning rate schedules

Cascade will be investigated in terms of its sensitivity to different learning rate schedules. To do this, instead of having a cyclical learning rate that linearly decays to 0 every iteration, the learning rate will now linearly decrease from its initial value to 0 over the course of the entire training - exactly as the PPO baseline. Fig. 12 shows the performance result of this modification compared with standard Cascade on Ant-v4 and Walker2d-v4. Up until the last iteration, modifying the learning rate schedule

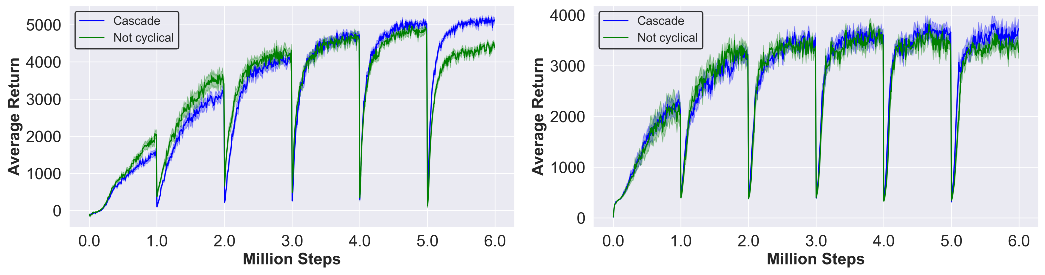


Figure 12: Performance graphs of standard Cascade (blue) and Cascade with a learning rate that linearly decays over the course of the entire training (green) as opposed to being cyclical for 6 million environment steps on Ant-v4 (left) and Walker2d-v4 (right).

had no significant impact on performance. The performance dip in the last iteration is explained by the learning rate having already decayed too much to guarantee convergence as usual. Therefore, when disregarding the last iteration it can be concluded that the cyclical learning rate that comes with standard Cascade has little to no impact on the performance. This shows that Cascade is more robust to different learning rate schedules than standard PPO where performance could vary drastically depending on the schedule (see C).

G Frozen base nets with no input/reward normalization

Keeping the base nets of Cascade frozen for the experiment in Section 4.3 naturally favors standard Cascade as it allows the adaptation to new normalization states of the environment. Therefore, this experiment was repeated for the non-normalized environments. Fig. 13 shows that indeed there is not much difference in terms of the final performance for both Cascade versions.

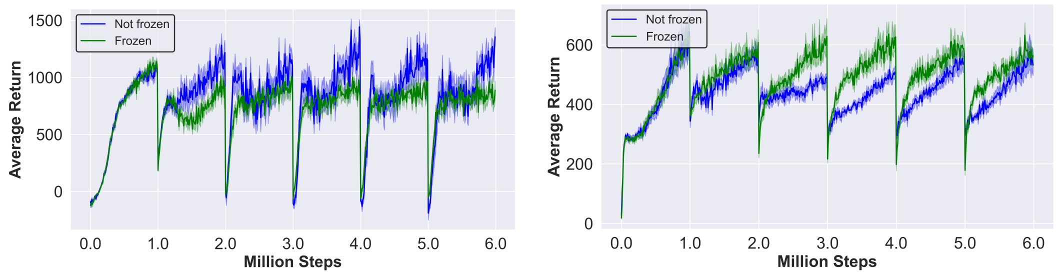


Figure 13: Performance graphs of standard Cascade (blue) vs Cascade with the weights of all but the latest base-net frozen (green) on Ant-v4 (left) and Walker2d-v4 (right) with no observation/reward normalization for 6 million environment steps.

H Using different base training algorithms

Besides PPO, we also evaluated Cascade with different base training algorithms, namely SAC (Haarnoja et al., 2018) and DDPG (Lillicrap et al., 2019), which required a slightly different experiment setup. In the following, we mention only the differences to our PPO experiment setup in Section B.

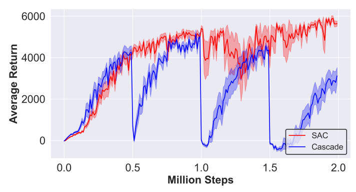
H.1 SAC

For SAC, we used the implementation of SAC along with its hyperparameters from CleanRL (Huang et al., 2022) (more specifically, `sac_continuous_action.py`). The only change we did apply was using orthogonal weight initialization for the networks' parameters (instead of Pytorch's default initialization). This is the setup we used as the baseline for SAC.

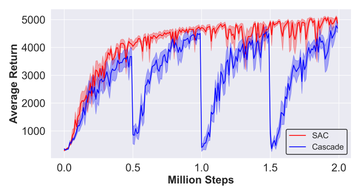
Next, we describe the changes made to the baseline for Cascade. Firstly, we set the actor's hidden dimension to 128 instead of 256 so that the parameter counts of the fully stacked Cascade net is about the same as the baseline SAC network. In contrast to PPO, SAC uses a replay buffer which we reset whenever a new base agent is added. Additionally, we discovered that SAC is much more sensitive to fully re-initializing the critic whenever a new base agent is added. Therefore, we simply kept the critic network whenever a new base agent is added by simply not resetting its parameters.

We ran the experiments using SAC only for 2 million total environment steps and updated added a new base agent every $5 \cdot 10^6$ steps as our version of SAC converged much faster than PPO.

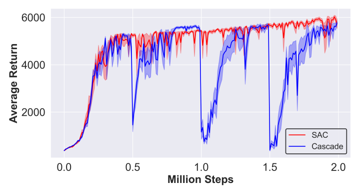
Fig. 14 shows the performance of the SAC baseline compared to Cascade with SAC. In contrast to the PPO experiments, using Cascade yields very little or no improvement at all.



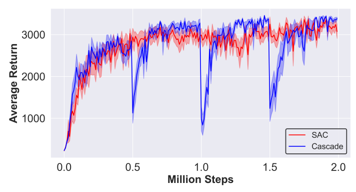
(a) Ant-v4



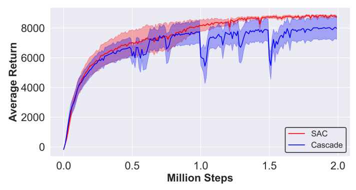
(b) Walker2d-v4



(c) Humanoid-v4



(d) Hopper-v4



(e) HalfCheetah-v4

Figure 14: Performance graphs of Cascade with SAC (blue) vs SAC (red) on Ant-v4 (top left), Walker2d-v4 (top right), Humanoid-v4 (bottom left), Hopper-v4 (bottom middle) and HalfCheetah-v4 (bottom right) for 2 million environment steps.

H.2 DDPG

For DDPG, we also used the DDPG implementation along with its hyperparameters from CleanRL (Huang et al., 2022) (more specifically, `ddpg_continuous_action.py`). This is the setup we used as the baseline for DDPG.

Like SAC, DDPG also uses a replay buffer which we reset every time a new base agent is added. Similarly, to SAC we found that a full re-initializing of the critic is detrimental to performance. Therefore, we also keep the critic network when we add a new base agent.

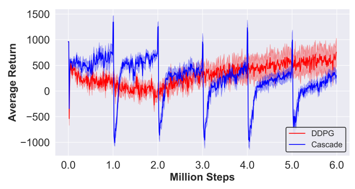
Fig. 15 shows the performance of the DDPG baseline compared to Cascade with DDPG. Like SAC, using Cascade yields very little to no improvement at all.

I Cascade for discrete action spaces

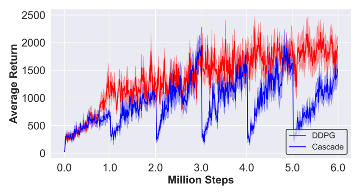
To test Cascade on a greater variety of tasks, Cascade was applied to environments with discrete action spaces. For this, the MuJoCo environment's action spaces were discretized to allow for a direct comparison to the performances on the continuous versions. The following discretization was used for a continuous action space $A_{cont} = [a_1, b_1] \times \cdots \times [a_n, b_n] \subseteq \mathbb{R}^n$: $A_{discr} = \{0, \ldots, 2^n - 1\}$ and action $0 \leq i \leq 2^n - 1$, is mapped to

$$i_1 \ldots i_n \mapsto (a_1 i_1 + b_1(1 - i_1), \ldots, a_n i_n + b_n(1 - i_n))$$

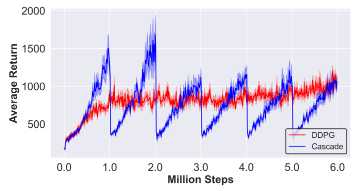
where $i_1 \ldots i_n$ ($i_j \in \{0, 1\}$) is the binary representation of i. This discretization allowed us to use the discrete version of PPO from CleanRL (Huang et al., 2022) (more specifically `ppo.py`) without any modifications. Though this simple discretization does not scale well in terms of dimensions, the discretized action spaces were still small enough to work well on all considered MuJoCo environments



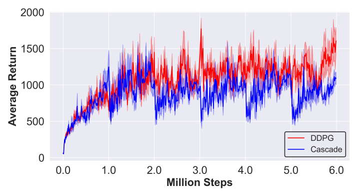
(a) Ant-v4



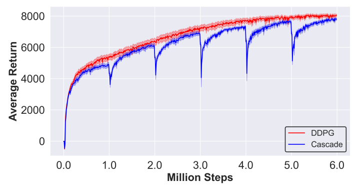
(b) Walker2d-v4



(c) Humanoid-v4



(d) Hopper-v4



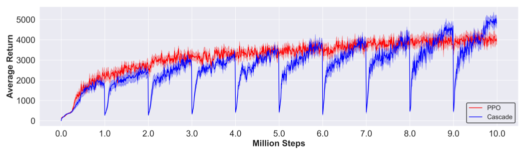
(e) HalfCheetah-v4

Figure 15: Performance graphs of Cascade with DDPG (blue) vs DDPG (red) on Ant-v4 (top left), Walker2d-v4 (top right), Humanoid-v4 (bottom left), Hopper-v4 (bottom middle) and HalfCheetah-v4 (bottom right) for 6 million environment steps.

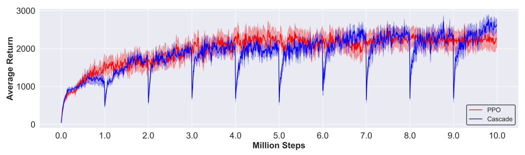
except Humanoid where this discretization would have 2^{17} actions. More precisely, discrete PPO on the discretized MuJoCo environments even managed to outperform its continuous counterpart on Hopper, HalfCheetah, and Walker2d. The performances on discrete Ant were poor even though Ant has an equally small action space which indicates that good performances on Ant require usage of the entire spectrum of its continuous action space which is not possible here since the discretization is too coarse. Therefore, we only compare Cascade to PPO on discrete Walker2d, Hopper, and HalfCheetah.

For Cascade to work on a discrete action space, the Cascade net is now trained with a discrete version of PPO from CleanRL (Huang et al., 2022) (more specifically `ppo.py`, in the following referred to as discrete PPO) along with their hyperparameter choice. The Cascade-specific hyperparameters like the fallback initialization and the frequency of base nets remained the same. The only change is that the hidden layer width of base agents was increased to 64 from 16 to deal with the large action spaces of the discretized environments (For example, in the discretized version of Ant, there are $256 = 2^8$ possible actions). The bigger hidden size always outperformed the smaller one in this discrete setting. The output of the Cascade net is interpreted as the logits of a Categorical distribution, therefore actions are chosen by applying softmax to Cascade's outputs and then sampling from that distribution.

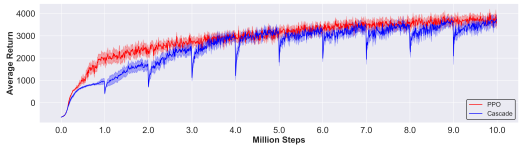
Fig. 16 compares the performances of the above-described discrete version of Cascade compared to discrete PPO applied to the discretized MuJoCo environments Walker2d, Hopper, and HalfCheetah for 10 million environment steps. Discrete PPO was also tested for shorter durations to account for the different learning rate schedules. However, the 10 million versions always performed best in terms of final performance. In contrast to previous experiments, 10 million steps were chosen since Cascade did not yet seem to converge for the 6 million version.



(a) Walker2d-v4



(b) Hopper-v4



(c) HalfCheetah-v4

Figure 16: Performance graphs of discrete PPO (red) compared to Cascade (blue) for 10 million steps on discretized versions of Walker2d, Hopper, and HalfCheetah.

Though not by much, Cascade manages to outperform PPO in terms of final performance on Walker2d and Hopper and draws even on HalfCheetah. Though Cascade on discrete action spaces takes longer to train, it is mostly still superior to PPO in these settings.

J Plasticity gain

In Section 5 we claimed that adding new base agents to the Cascade net improves its plasticity. In this subsection, we will formally introduce network plasticity and investigate the claim above. Lyle et al. (2023) describe network plasticity as "[...] the ability of a neural network to quickly change its predictions in response to new information [...]". Formally, they define it as the expected loss that is obtained after running an optimization algorithm to minimize a loss sampled from a distribution of losses. In the following, we describe the concrete optimization problem Lyle et al. used to measure their network's plasticity which is the one we employed here too. For details, we refer to their original paper (Lyle et al., 2023).

Assume f is an ANN architecture whose output layer has a dimension of one and $\Theta \subseteq \mathbb{R}^n$ is the set of parameters for f. Furthermore, assume that X is a distribution of inputs for f. For $\theta \in \Theta$ we can define a family of optimization problems

$$l_\omega := \mathbb{E}_{x\sim X}\left[(f(x,\theta) - (a + \sin(10^5 f(x,\omega))))^2\right] \text{ where } a = \mathbb{E}_{x\sim X}[f(x,\theta)] \in \mathbb{R},\ \omega \in \Theta. \quad (3)$$

If we use $\theta^*(\omega)$ to denote the final parameters obtained by running an optimization algorithm on l_ω, and W denotes a distribution over Θ, then the plasticity $\mathcal{P}(\theta)$ of θ is defined as

$$\mathcal{P}(\theta) = b - \mathbb{E}_{\omega\sim W}[l_\omega(\theta^*_\omega)] \text{ where } b = \mathbb{E}_{x\sim X,\omega\sim W}\left[\sin(10^5 f(x,\omega))\right]. \quad (4)$$

Intuitively, this definition of plasticity quantifies how well f can adapt to perturbations of its output.

We measured the plasticity of the Cascade net from the experiments in Section 4.2 right before the second base agent is added and directly afterwards for the Walker2d and Ant environment. The Cascade net has an output dimension greater than one as we output a mean and a logstd value for each dimension of the action space. To fix this, we simply averaged over the network's mean output and discarded the logstd output. We used a uniform distribution over 1000 samples from the observation distribution of the current Cascade net (after adding the new agent) for X and the parameter distribution W is the initial parameter distribution of our Cascade net. Furthermore, we used the same optimization as the Cascade net used during training and defined $\theta^*(\omega)$ as the result of

Env	**Parameters PPO**	**Parameters Cascade**	**Runtime PPO**	**Runtime Cascade**
Ant	12497	11286	5.5h	8.6h
Walker2d	11085	9470	5.1h	7.7h
Hopper	10119	8186	4.6h	7.4h
Humanoid	57763	68098	6.5h	8.7h
HalfCheetah	11085	9470	5.2h	8.1h

Table 2: Average runtime in hours of Cascade and vanilla PPO for 6 million steps along with the parameter count.

Env	σ_2	σ_3	σ_4	σ_5	σ_6
Ant	0.051	0.039	0.041	0.046	0.087
Walker2d	0.198	0.174	0.144	0.113	0.095

Table 3: The standard deviation of the fallback action within one episode for each base net in the Cascade net when standard Cascade was run on Ant-v4 and Walker2d-v4 for 6 million environment steps. We denote the fallback standard deviation of the second oldest base net by σ_2 and σ_6 is the fallback standard deviation of the one added last. The first base net isn't listed because it does not have a fallback action.

optimization after 10^4 parameter updates. The expectations in Equation 4 are approximated using 10 samples from W.

Taking the average of 5 runs, the Cascade net before adding the second base agent had the plasticity $\mathcal{P} = 0.014$ in Walker2d and $\mathcal{P} = 0.16$ in Ant. After the base agent has been added, the plasticity was $\mathcal{P} = 0.042$ in Walker2d and $\mathcal{P} = 0.30$ in Ant. The baseline b's average was 0.50 in Walker2d and 0.56 in Ant. This clearly shows that the addition of extra-base agents hugely improves the network's plasticity.

K Runtime measurements

In this section, we compare the runtime of Cascade to the baseline PPO agent that was measured in the experiments 4.2. Tab. 2 compares the runtimes of PPO and Cascade along with their parameter count for Ant, Walker2d, Hopper, Humanoid, and HalfCheetah for a training duration of 6 million steps which results in a Cascade net of size 6 for Cascade. The experiments have been run on Xeon Gold 5120 CPUs with 28 cores à 2.20GHz. Though by construction both PPO and Cascade have the same parameter count, Cascade is around 50% slower because its inference step is not parallelizable as each base agent has to wait for the output of the previous one.

L Fallback action distribution

In this section, we take a closer look at how the fallback actions reported in Tab. 1 are distributed over the state space. Firstly, just by looking at the standard deviation of the fallback action within one episode, we can directly see that the learned fallback behavior is non-trivial as the fallback action does not just assume a constant value. Tab. 3 lists these standard deviations for the final Cascade net for Ant and Walker2d.

We can go even one step further and look directly at the course of the fallback action within one episode for one concrete model. Fig. 17 visualizes the course of the fallback action over one episode for the second oldest and latest base agent on Ant-v4 and Walker2d-v4 for one randomly picked model (Though the individual graphs differ for each model, the main patterns are the same for each model).

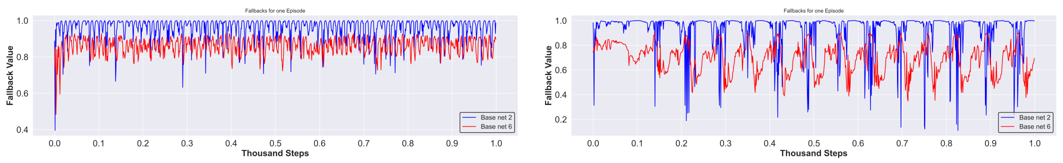


Figure 17: Course of the fallback action of a trained Cascade net over the course of one episode (capped at 1000 steps) for the second oldest base agent (blue) and the latest base agent (red) on Ant-v4 (left) and Walker2d-v4 (right)

It can be observed that base net 2 (second oldest base net) in general assumes values close to 1 meaning it mostly delegates its action to the original base net. However, there are some regularly spaced spikes (more so in Walker2d) where suddenly a much lower fallback value is assigned suggesting that it specialized itself to certain regions of the state space. The same can be said for base net 6 (lastest base net) on Walker2d where the fallback action regularly switches between roughly 0.8 to values as low as roughly 0.4. However, in Ant, base net 6 does not seem to have any spikes and oscillates only within a range of roughly 0.1 (We note though that in general this range is much larger and only coincidentally this small for the model we picked).