

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:

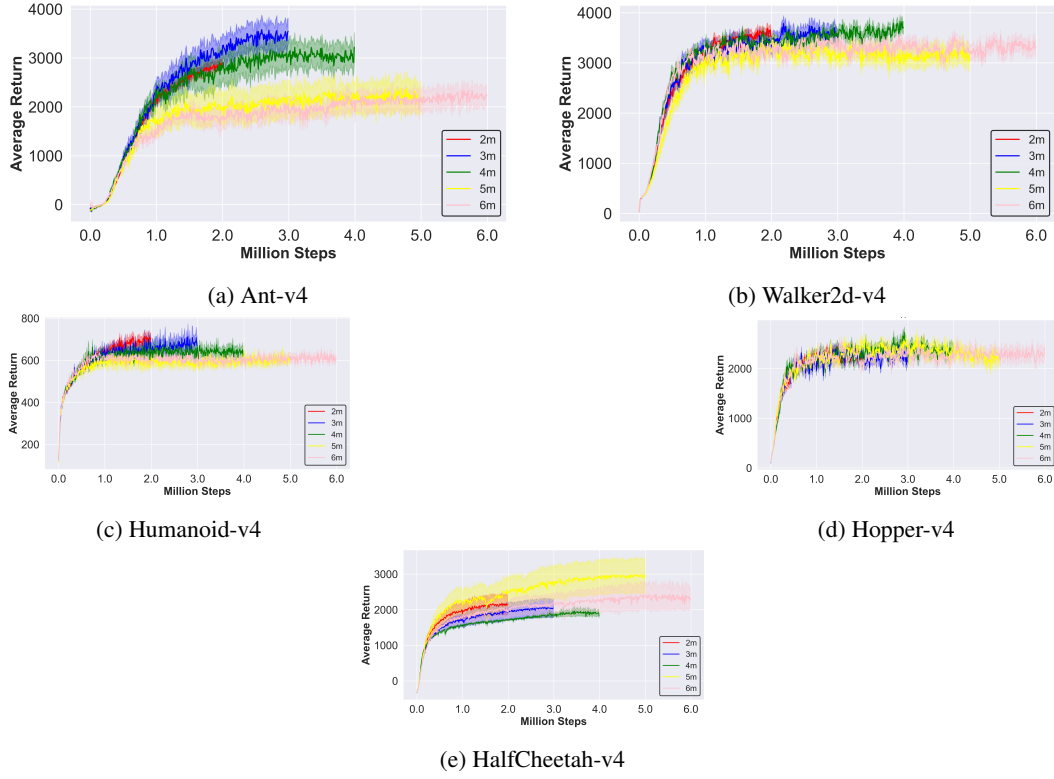


Figure 9: Performance graphs of PPO for 2 million (red), 3 million (blue), 4 million (green), 5 million (yellow) and 6 million (beige) environment steps.

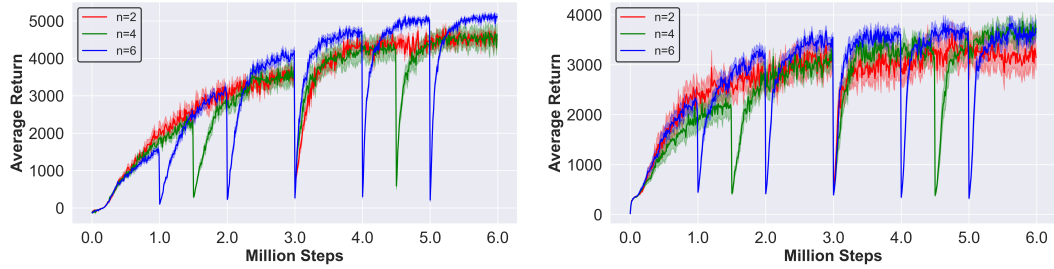


Figure 10: Performance graphs of Cascade when adding a new base agent every 3 million steps (red), every 1.5 million steps (green), and every 1 million steps (blue) on Ant-v4 (left) and Walker2d-v4 (right) for 6 million environment steps.

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

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 $\lambda \approx \text{sigmoid}(b_\lambda)$. Therefore, by setting $b_\lambda = \text{sigmoid}^{-1}(\lambda)$ 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.

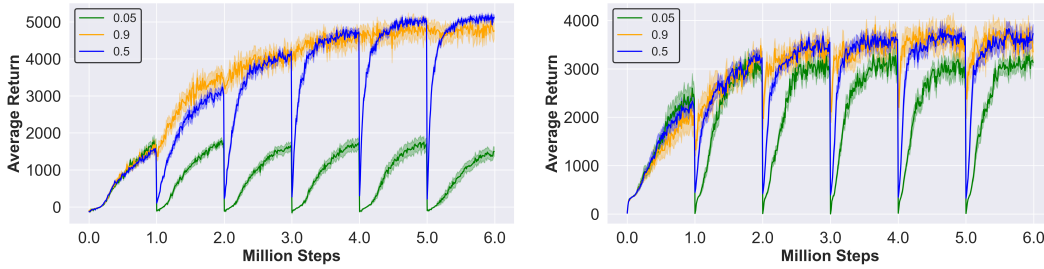


Figure 11: Performance graphs of Cascade on Ant-v4 (left) and Walker2d-v4 (right) for 6 million environment steps with 0.05 (green), 0.5 (blue) and 0.9 as the initial fallback value for new base agents.

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

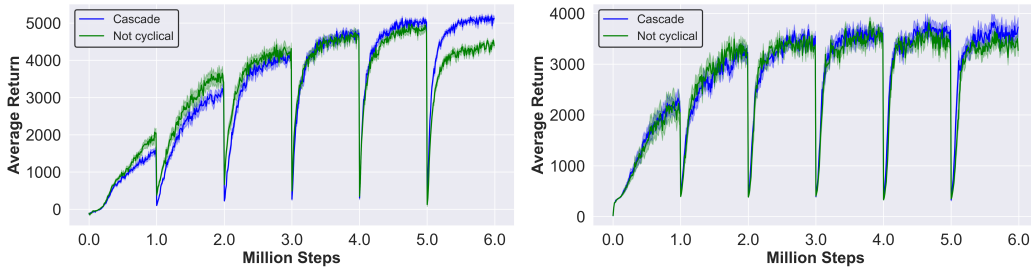


Figure 12: Performance graphs of standard Cascade (blue) and Cascade with a learning rate that linearly decays over the course of the entire training (green) as opposed to being cyclical for 6 million environment steps on Ant-v4 (left) and Walker2d-v4 (right).

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.

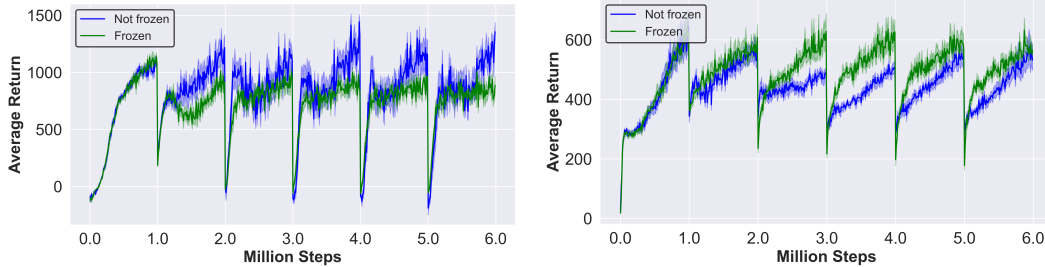


Figure 13: Performance graphs of standard Cascade (blue) vs Cascade with the weights of all but the latest base-net frozen (green) on Ant-v4 (left) and Walker2d-v4 (right) with no observation/reward normalization for 6 million environment steps.

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.

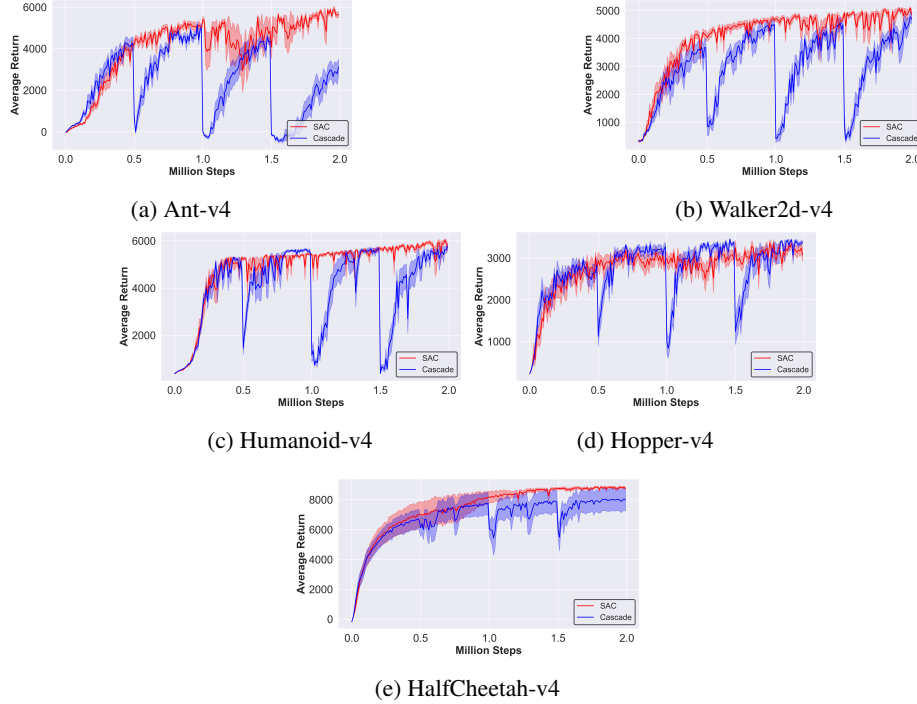


Figure 14: Performance graphs of Cascade with SAC (blue) vs SAC (red) on Ant-v4 (top left), Walker2d-v4 (top right), Humanoid-v4 (bottom left), Hopper-v4 (bottom middle) and HalfCheetah-v4 (bottom right) for 2 million environment steps.

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 $A_{cont} = [a_1, b_1] \times \dots \times [a_n, b_n] \subseteq \mathbb{R}^n$: $A_{discr} = \{0, \dots, 2^n - 1\}$ and action $0 \leq i \leq 2^n - 1$, is mapped to

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

where $i_1 \dots i_n$ ($i_j \in \{0, 1\}$) 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

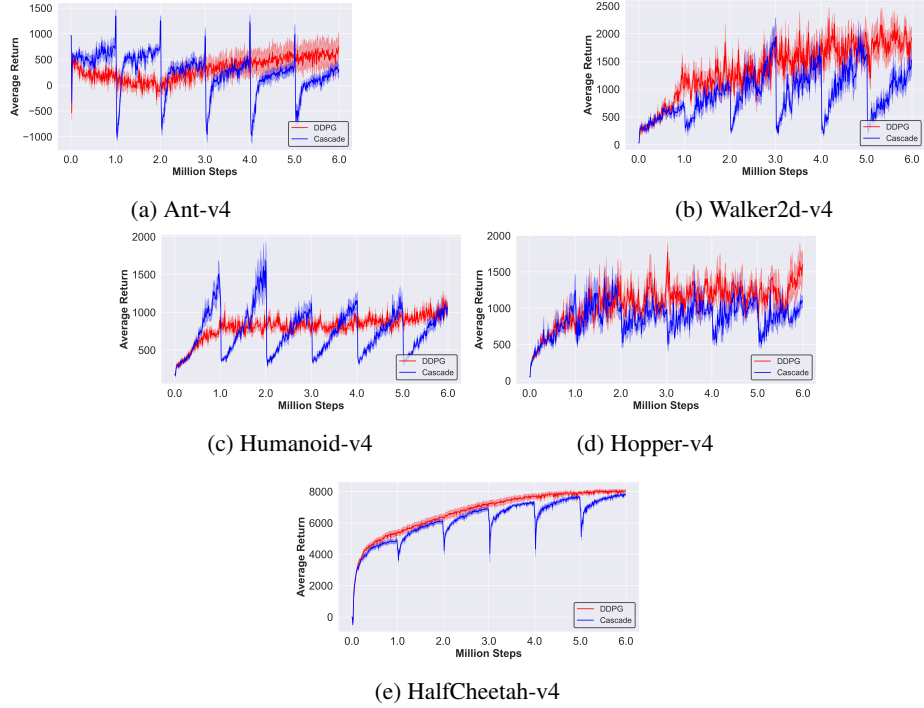


Figure 15: Performance graphs of Cascade with DDPG (blue) vs DDPG (red) on Ant-v4 (top left), Walker2d-v4 (top right), Humanoid-v4 (bottom left), Hopper-v4 (bottom middle) and HalfCheetah-v4 (bottom right) for 6 million environment steps.

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.

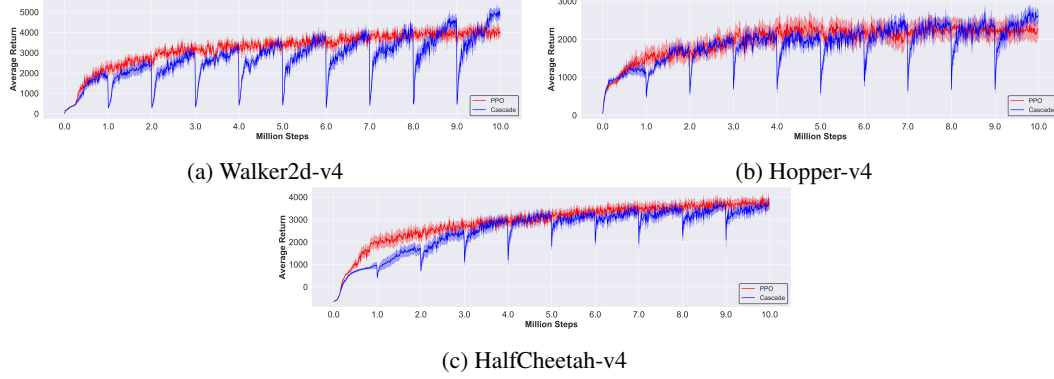


Figure 16: Performance graphs of discrete PPO (red) compared to Cascade (blue) for 10 million steps on discretized versions of Walker2d, Hopper, and HalfCheetah.

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

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

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

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

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).

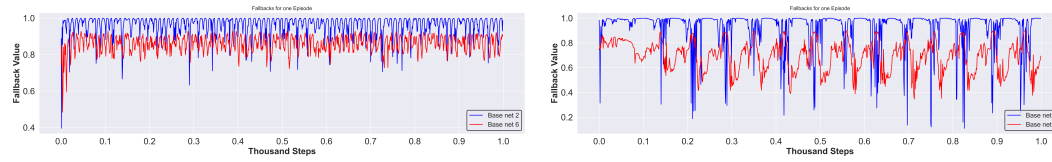


Figure 17: Course of the fallback action of a trained Cascade net over the course of one episode (capped at 1000 steps) for the second oldest base agent (blue) and the latest base agent (red) on Ant-v4 (left) and Walker2d-v4 (right)

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 (latest 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).