Hierarchical Orchestra of Policies

Thomas P Cannon *
Department of Computer Science
University of Bath
UK, Bath, BA2 7AY
tc2034@bath.ac.uk

Özgür Şimşek
Department of Computer Science
University of Bath
UK, Bath, BA2 7AY
os435@bath.ac.uk

Abstract

Continual reinforcement learning poses a major challenge due to the tendency of agents to experience catastrophic forgetting when learning sequential tasks. In this paper, we introduce a modularity-based approach, called Hierarchical Orchestra of Policies (HOP), designed to mitigate catastrophic forgetting in lifelong reinforcement learning. HOP dynamically forms a hierarchy of policies based on a similarity metric between the current observations and previously encountered observations in successful tasks. Unlike other state-of-the-art methods, HOP does not require task labelling, allowing for robust adaptation in environments where boundaries between tasks are ambiguous. Our experiments, conducted across multiple tasks in a procedurally generated suite of environments, demonstrate that HOP significantly outperforms baseline methods in retaining knowledge across tasks and performs comparably to state-of-the-art transfer methods that require task labelling. Moreover, HOP achieves this without compromising performance when tasks remain constant, highlighting its versatility.

1 Introduction

Neural networks are typically trained on data drawn independently and identically from a static distribution. While this approach works well in many cases, it becomes challenging in environments that are continuously changing or when new environments are introduced. In dynamic settings, such as reinforcement learning, robotics, or dialogue systems, models must adapt to new information while preserving knowledge from previous tasks (Parisi et al., 2019). However, neural networks often suffer from catastrophic forgetting, where learning new tasks leads to the rapid loss of previously acquired knowledge. The ability to learn new skills while maintaining existing knowledge is referred to as continual learning (Ring, 1994).

To address the challenge of catastrophic forgetting, researchers have developed three primary categories of methods. The first category, *regularization-based*, works by constraining updates to network parameters, thereby penalizing deviations from learned weight values that are critical for previous tasks. Notable examples of this approach include Elastic Weight Consolidation (EWC) and Synaptic Intelligence (SI) (Kirkpatrick et al., 2017; Zenke et al., 2017). The second category, *replay-based*, mitigates forgetting by periodically rehearsing past experiences, either through actual data or synthetic generations, ensuring that the network continues to perform well on earlier tasks (Rolnick et al., 2019; Shin et al., 2017). The third category, *modularity-based*, addresses the issue by structurally separating the network into modules, with each module dedicated to a specific task, thereby minimizing interference between tasks, prominent examples of this method are Progressive Neural Networks (PNN) by (Rusu et al., 2016) and adaptive multi-column stacked sparse denoising autoencoder (AMC-SSDA) by Agostinelli et al. (2013). Finally there are some methods which use a combination of these, for example, Schwarz et al. (2018) uses an active network and a knowledge-base

*https://tpcannon.com

Accepted at the Intrinsically Motivated Open-ended Learning workshop at NeurIPS 2024.

network similar to the modularity-based methods, however they periodically compresses knowledge from the active into the knowledge network using EWC regularisation.

Our method, Hierarchical Orchestra of Policies (HOP), is a modularity-based approach and is most similar to PNN. However, unlike PNN, HOP does not rely on a task identifier during training, which makes it more suitable for domain-incremental learning (Van de Ven and Tolias, 2019). Additionally, HOP differs in that it combines network probability outputs directly through hierarchical weightings, rather than using latent connections between networks. Finally, we demonstrate HOP at a significantly larger scale — 18 hierarchical policy levels compared to only three in PNN. We show that HOP performs comparably to PNN, even when PNN is provided with task labels while HOP is not. Furthermore, HOP mitigates catastrophic forgetting across several Procgen environments (Cobbe et al., 2020), achieving notable improvements over Proximal Policy Optimization (PPO), a reinforcement learning algorithm with inherent regularization properties (Schulman et al., 2017).

2 Hierarchical Orchestra of Policies

Hierarchical Orchestra of Policies (HOP) is a modularity-based deep learning framework designed to mitigate catastrophic forgetting when learning new tasks. In this framework, a *task* is defined as a specific Markov Decision Process (MDP), where distinct levels within a procedurally generated environment, or levels across different environments, are considered separate tasks (Puterman, 2014). Although HOP is task-agnostic, all tasks are treated as episodic.

HOP relies on reinforcement learning algorithms that output stochastic policies, represented as $\pi(a \mid s)$ (Sutton, 2018). In our work, PPO serves as the base algorithm for HOP. The framework introduces three key mechanisms to form and use a collection of policies:

1. *Checkpoints* to freeze and store policies at a certain stage of training.
2. *Orchestration* of policy activation based on state similarity.
3. *Hierarchical weightings* to balance the influence of previous and new policies.

These mechanisms enable the agent to recover and maintain performance across diverse tasks without significant interference, thereby promoting continual learning in complex environments.

Checkpoints. The agent initially learns a policy using a base algorithm. After $\mathcal{T}_{checkpoint}$ time-steps, HOP initializes a *checkpoint*, where the current learning policy π is frozen and evaluated in the currently available tasks. During checkpoint evaluation, if the episodic return R surpasses a predefined threshold $R_{threshold}$, all states encountered during that task episode are stored in a set of *trusted states* S_m which is linked with the policy checkpoint π_m, where m is the count of the checkpoint.

The Orchestra. When the agent resumes learning, it dynamically activates checkpoint policies π_m determined by the similarity between the current state s_t and any $s_m \in S_m$. If the current state s_t is *similar* to any state in S_m, then the corresponding frozen policy π_m is activated ($I_m = 1$). Similarity is determined by a threshold value ω, which, in all of our experiments, has been defined as any $s_m \in S_m$ with a cosine similarity greater than 0.98 with s_t.

Rather than selecting actions directly from the distribution of a single frozen policy ($a_t \sim \pi_m(s_t)$), which could lead to conflicts when multiple policy checkpoints are activated, HOP combines the distributions from activated policy checkpoints ($I_m \pi_m$) and the current learning policy π_n into a joined action policy, denoted as π_{n_a} (see equation 1). Here, n represents the current count of all policies, and the subscript a denotes the combined policy from which the action is sampled. This approach allows the agent to leverage past knowledge while adapting to new tasks, promoting continual learning.

To avoid significant and undesired output shifts caused by small changes in the state, frozen policies predict actions based on the most similar state, $s_m \in S_m$, to the current state s_t (Szegedy, 2013). This state is referred to as s^*_m, and actions are chosen as $\pi_m(a \mid s^*_m)$ rather than directly from s_t. This dynamic activation of multiple policies is called the *orchestra* of policies, a term borrowed from Jonckheere et al. (2023) but applied differently in this work.

Hierarchical Weightings. As the agent learns, it is expected to achieve higher task-specific rewards, which suggests that newer policies for the same tasks are likely to outperform older policy checkpoints

for the same task. Thus, simply averaging all policies, as represented by $\pi_{n_a} = \frac{1}{n}\sum_{m=0}^{n}\pi_m$, is impractical. Moreover, because the agent does not know the identities of tasks, multiple policy checkpoints may activate; therefore, simply sampling actions from newer policy checkpoints is not possible. To address this, HOP introduces a hierarchical discount factor, denoted as W, to determine the contributions of policy checkpoints. Each joined action policy at time of checkpoint is a combination of previous policy checkpoints, creating a hierarchical structure, as shown in Figure 2 and Equation 1. This hierarchical structure assigns a higher weight to more recent activated policies. For some examples of how this concept functions, please refer to Appendix A.4.

Policy Updates: The update process for the learning policy (π_n) follows the same procedure as that used by the base algorithm. Specifically, all necessary attributes associated with the policy are sampled from π_n, except for the action, which is instead sampled from the current joined action policy (π_{n_a}). We also allow gradients to propagate to the policy checkpoints π_m, but only for the states where they are activated. For more detail and pseudo code refer to Appendix A.1.

The HOP action policy is expressed as,

$$\pi_{n_a}(s_t) = \pi_n(s_t) + \sum_{m=1}^{n-1} W_m \pi_{m_a}(s_m^*), \tag{1}$$

$$W_m = \frac{I_m}{1 + \sum_{k=m}^{M} I_k}, \tag{2}$$

$$I_m = \begin{cases} 1 & \text{if } \frac{s_m^* \cdot s_t}{\|s_m^*\|\|s_t\|} > \omega. \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

Here, s_m^* is the most similar state from all $s \in S_m$, and π_{m_a} represents the logits of the m-th joined action policy checkpoint. M is the total number of checkpoints conducted. n is the current count of all policies. I_m is the activation related to each policy π_m. And ω is the similarity threshold of the current state with previous states in S_m.

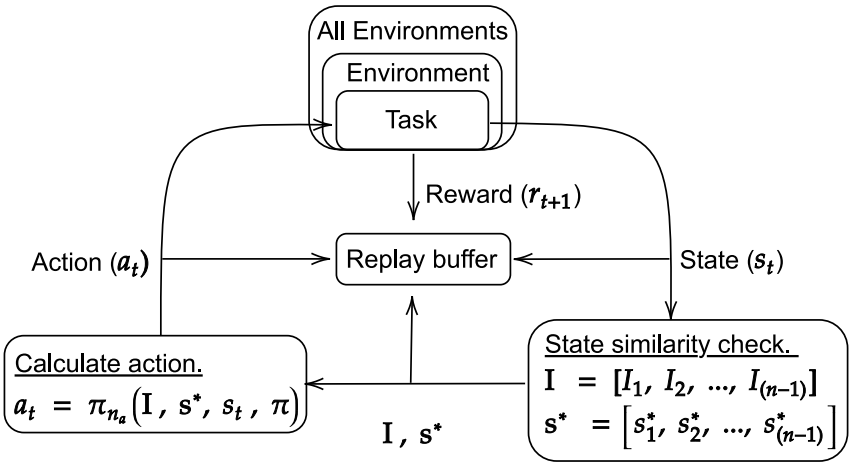

Figure 1: The flow of information as a HOP agent acts in a task.

Figure 1 depicts the flow of information within the HOP framework, illustrating how the agent evaluates observations, selects relevant policies, and takes actions to adapt continually across tasks. The agent begins by assessing the current state s_t against a similarity threshold for each checkpoint policy $\pi_m \in \boldsymbol{\pi}$, where $\boldsymbol{\pi}$ denotes all previously stored policies. For each policy, it identifies the most similar reference state $s_m^* \in S_m$ and calculates an activation I_m based on this similarity. If the similarity metric—calculated as the cosine similarity between s_t and s_m^*—exceeds the predefined threshold ω, the corresponding policy π_m is activated ($I_m = 1$), otherwise, it remains inactive ($I_m = 0$). The activated policies then contribute to the current joint action policy π_{n_a}, which

combines outputs from the activated checkpoint policies $I_m \pi_m$ and the current learning policy π_n, weighted hierarchically according to their relative recency and activations, as shown in Equation 1.

The agent samples actions from π_{n_a} rather than directly from the learning policy π_n, enabling it to leverage past knowledge while adapting to new tasks. The chosen action interacts with the environment, producing a new state and a corresponding reward. These states, actions, rewards, and activations are stored in the replay buffer to support continual learning and mitigate forgetting by allowing the agent to revisit past experiences. During training, the replay buffer's stored activations also guide the gradient computations, allowing only the activated policies to contribute to the policy update. This targeted update process refines gradient flow selectively based on activation, promoting modularity and stability in learning across diverse task environments.

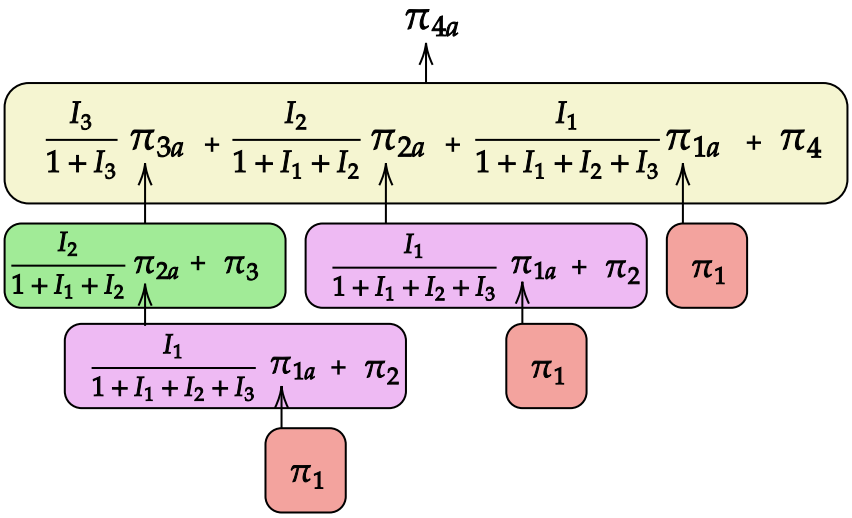

Figure 2: Hierarchical formation of the fourth level of a HOP action policy.

3 Results

We evaluated the performance of HOP using the Procgen suite of environments (Cobbe et al., 2020). The experimental setup consisted of three phases of training. In the first phase, the agent trains for three million time-steps on multiple levels of a selected Procgen environment to develop its ability to learn and generalize. In the second phase, the environment was switched to a different one, and the agent continued training for another three million time-steps, assessing its adaptability and ability to transfer learning. Finally, in the third phase, the agent returned to the original environment for an additional three million time-steps to evaluate retention of skills and re-adaptation. Throughout training, the agent's objective is to optimize the reward functions defined by the Procgen environments, which typically involve maximizing cumulative rewards for task-specific objectives such as reaching goals, collecting items, or avoiding obstacles. This is a simplified experimental set-up to that conducted by Schwarz et al. (2018) in their examination of P&C.

We conducted experiments with three different environment combinations: StarPilot and Climber, Ninja and StarPilot, and Ninja and CoinRun, repeating each with four random seeds. During training, periodic evaluation episodes were performed to measure performance, and *checkpoints* were saved every 500,000 time-steps.

HOP was compared with standard PPO and a modified version of Progressive Neural Networks (PNN) for use with PPO – see appendix A.3 for full details of the modifications. We allowed PNN to have task identifiers but not HOP. Results presented in Figure 3 indicate that HOP outperformed PPO in both the rate of performance recovery and the final averaged evaluation return after training. We found that HOP had comparable performance to PNN in all but the very beginning of the third phase of learning. Table 1 summarizes the total steps after the second phase of learning required for each method to recover to the performance level achieved at the end of the initial phase of training, and the

final averaged evaluation return. For a complete description of the experiments and environments please see appendix A.2.

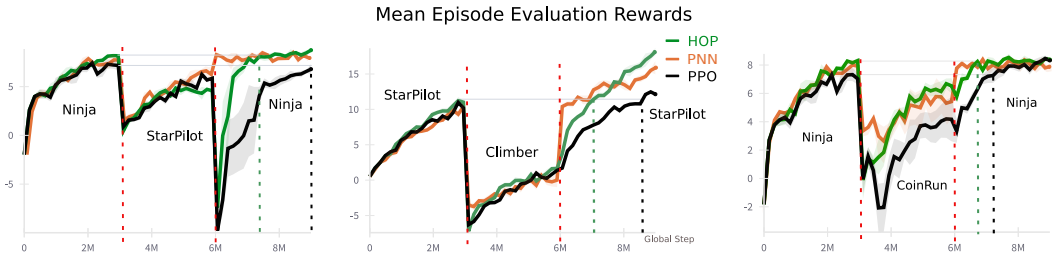

Figure 3: Training performance of HOP, PNN and PPO on three experiments where environments are periodically changed. The red dashed lines indicate the points when the environment are switched. The green dashed lines show when HOP returns to the highest average evaluation reward achieved in the first environment before the change. The black dashed lines represents this point for PPO. Shaded areas are the standard error. All experiments are conducted with the Procgen easy setting.

	Steps-to-return (10^6)		**Final Rewards**		
	PPO	HOP	PPO	HOP	PNN
StarPilot - Climber	2.68	**1.04** -61.2%	12.14	**18.15** 49.5%	15.98 31.6%
Ninja - StarPilot	3+	**1.70** -43+%	6.79	**8.73** 28.6%	**7.97** 17.37%
Ninja - Coinrun	1.37	**0.72** -47.7%	8.33	8.37 0.48%	7.83 -6.00%

Table 1: A comparison of PPO, HOP, and PNN. Steps-to-return represents the number of steps (in millions) to re-acquire the same average evaluation reward at the end of the first period of learning in that environment, PNN is not included in these comparisons as it uses a separate actor and critic network per task. Final rewards display the final average evaluation rewards at the end of all training. The percentages show the difference compared to the baseline PPO method.

4 Summary and Discussion

We present a novel modularity-based approach, the Hierarchical Orchestra of Policies, to address catastrophic forgetting in continual life-long reinforcement learning. In our empirical evaluation, HOP outperforms PPO in continual learning scenarios, achieving a faster recovery of performance and final performance. Both HOP and PNN demonstrate substantial transfer between environments with similar dynamics and state spaces such as Ninja and CoinRun. In these scenarios HOP can activate relevant frozen policies learned from Ninja while acting in CoinRun, similar to PNN's *adapter* networks connecting separate *columns*. However, unlike PNN, HOP does not require task labels, making it more versatile for real-world applications where task boundaries are not clearly defined.

However, the effectiveness of HOP depends on the careful tuning of some hyper-parameters, particularly the similarity threshold, w, and reward threshold $R_{threshold}$, which must be set appropriately for all expected tasks. See Appendix A.2

Future work could expand HOP's evaluation by testing transitions between highly diverse tasks and environments where task boundaries are ambiguous, a setting in which PNN and similar methods are less effective. Additionally, HOP could be adapted to continuous environments with fluid task transitions, further highlighting its robustness in real-world scenarios. To address performance drops immediately following task distribution changes, a learnable parameter could be introduced which could dynamically adjust the influence of previous checkpoints, enabling immediate adaptation while maintaining learning.

Acknowledgements. This work was supported by the UKRI Centre for Doctoral Training in Accountable, Responsible and Transparent AI (ART-AI) [EP/S023437/1] and the University of Bath.

References

Agostinelli, F., Anderson, M. R., and Lee, H. (2013). Adaptive multi-column deep neural networks with application to robust image denoising. *Advances in neural information processing systems*, 26.

Cobbe, K., Hesse, C., Hilton, J., and Schulman, J. (2020). Leveraging procedural generation to benchmark reinforcement learning. In *International conference on machine learning*, pages 2048–2056. PMLR.

Huang, S., Dossa, R. F. J., Ye, C., Braga, J., Chakraborty, D., Mehta, K., and AraÃšjo, J. G. (2022). Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18.

Jonckheere, M., Mignacco, C., and Stoltz, G. (2023). Symphony of experts: orchestration with adversarial insights in reinforcement learning. *arXiv preprint arXiv:2310.16473*.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526.

Mnih, V. (2016). Asynchronous methods for deep reinforcement learning. *arXiv preprint arXiv:1602.01783*.

Parisi, G. I., Kemker, R., Part, J. L., Kanan, C., and Wermter, S. (2019). Continual lifelong learning with neural networks: A review. *Neural networks*, 113:54–71.

Puterman, M. L. (2014). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

Ring, M. B. (1994). *Continual learning in reinforcement environments*. The University of Texas at Austin.

Rolnick, D., Ahuja, A., Schwarz, J., Lillicrap, T., and Wayne, G. (2019). Experience replay for continual learning. *Advances in neural information processing systems*, 32.

Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., Kirkpatrick, J., Kavukcuoglu, K., Pascanu, R., and Hadsell, R. (2016). Progressive neural networks. *arXiv preprint arXiv:1606.04671*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Schwarz, J., Czarnecki, W., Luketina, J., Grabska-Barwinska, A., Teh, Y. W., Pascanu, R., and Hadsell, R. (2018). Progress & compress: A scalable framework for continual learning. In *International conference on machine learning*, pages 4528–4537. PMLR.

Shin, H., Lee, J. K., Kim, J., and Kim, J. (2017). Continual learning with deep generative replay. *Advances in neural information processing systems*, 30.

Sutton, R. S. (2018). Reinforcement learning: An introduction. *A Bradford Book*.

Szegedy, C. (2013). Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*.

Van de Ven, G. M. and Tolias, A. S. (2019). Three scenarios for continual learning. *arXiv preprint arXiv:1904.07734*.

Zenke, F., Poole, B., and Ganguli, S. (2017). Continual learning through synaptic intelligence. In *International conference on machine learning*, pages 3987–3995. PMLR.

A Appendix

A.1 HOP Algorithm details

The logic provided in algorithm 1, is suitable for use with PPO (or any other actor critic style base function. It is expected that HOP would work with any method with a stochastic policy, however this has yet to be tested. Table 2 details all of the extra parameters that HOP requires.

Algorithm 1 Hierarchical Orchestra of Policies (HOP) with PPO

Initialize: Current hierarchy depth n = 1, Policy π_n == π_{n_a}, similarity threshold w, and reward threshold P, total steps D, step = 0, checkpoint interval C, state s_t, done = 0, value function V_θ, batch size T, buffers B, and other PPO parameters ϕ.

Training:
while step < D **do**
  while step < T **do**
    for each frozen policy π_m **do** — Activation logic
      if cosine similarity $\frac{s_{m_{\text{max-sim}}} \cdot s_t}{\|s_{m_{\text{max-sim}}}\| \|s_t\|} > w$ **then**
        Activate policy π_m, set $A_m = 1$
      else
        Deactivate policy π_m, set $A_m = 0$
      end if
    end for
    Sample action $a_t \sim \pi_{n_a}(s_t)$ as per equation 1
    s_t, reward, done = environment.step
    $B \leftarrow s_t$, reward, done, a_t
    if done **then**
      reset environment
    end if
  end while
  Update π_n and V_θ from B using PPO algorithm — Allow gradients to propagate to activated policies in those states only.
  if step == C **then** — Checkpoint logic
    Freeze policy π_n as π_m
    Evaluate π_{m_a} on all currently available tasks
    Evaluation can be from the most recent experiences or from running new evaluations. We use new evaluations in our experiments
    for each evaluation episode **do**
      if episodic return $R > P$ **then**
        Append all states in evaluation to S_m
      end if
    end for
  end if
end while

Symbol	Name	Default	Notes
P	Reward threshold	7.5	Can be generalized if rewards are normalized
w	Similarity threshold	0.98	Cosine similarity
C	Checkpoint interval	500,000	Time-steps. In our experiments forms 18 policies

Table 2: All additional parameters for the HOP algorithm.

A.2 Experiment details

Our experiments are conducted in the Procgen suite of environments introduced by Cobbe et al. (2020). Specifically, we use Ninja, StarPilot, Climber, and CoinRun as our environments. These can be viewed at https://github.com/openai/procgen, and we also provide a table with a snapshot of each environment in Figure 4. In Procgen, there are options that can reduce the complexity of the environments. We activate the following options: `use_backgrounds=False`, `restrict_themes=True`,

`distribution_mode=easy`, and `use_sequential_levels=True`. However, we do not activate `use_monochrome_assets`, as we found that it lacked proper indications for agent direction. In all of our experiments, the agent's goal was to maximize the cumulative reward provided by the environment. The state is represented as an 84x84 pixel image, and the agent has 15 possible actions.

We run the same experiment with different combinations of environments. The experiment is conducted in three phases, each evenly distributed over the total number of time steps (X):

1. The agent trains in environment 1 with T_1 tasks in distribution.
2. Learning is switched to environment 2 with T_2 tasks.
3. Learning is switched back to environment 1 for the same T_1 tasks.

In our experiments, $X = 9,000,000$, and $T_1 = T_2 = 30$. PNN is given a task identifier for each environment, enabling it to use the correct networks and adapters. HOP, on the other hand, does not require these and is not given them. Every 163,840 time steps, the agent is evaluated in the current distribution of tasks, which in this case consists of 30 Procgen levels in the current training environment of that phase, the reward in this evaluation phase is the cumulative reward - 0.01*total steps taken in the environment, which gives a better indication of efficiency. The results we report are based on these evaluated tasks. Conducting evaluation episodes at fixed intervals provides the clearest and most accurate representation of agent performance.

The three experiments shown in Figure 3 are conducted using the combinations of environments listed in Table 3. In the first experiment, the environments are completely different, with distinct dynamics and little to no shared understanding between them. For the second experiment, we believed there would be some overlap; while StarPilot scrolls from right to left, Climber scrolls vertically from bottom to top. The final experiment features considerable shared dynamics, as both Ninja and CoinRun are platforming games. We hypothesized that this setup would demonstrate the transfer and recovery of performance across different levels of difficulty. However, we observed that only the Ninja and CoinRun experiment exhibited meaningful transfer for both PNN and HOP.

We use PPO as a baseline algorithm and as the foundation for both HOP and PNN. Our PPO implementation is based on the version by Huang et al. (2022). The only modification we made was separating the actor and critic networks, which we found easier to work with and which outperformed the shared convolutional layer approach. Figure 5 illustrates our implementation. We kept the PPO-specific hyperparameters fixed for its use with HOP, PNN, and the base PPO. These hyperparameters were optimized for the base PPO, and while a small benefit might have been observed for HOP and PNN if a hyperparameter sweep had been conducted, both performed as expected, so we did not pursue this. The PPO-specific hyperparameters are shown in Table 4, and all other relevant parameters are shown in Table 5.

Experiment Number	**Phase 1**	**Phase 2**	**Phase 3**
1	Ninja	StarPilot	Ninja
2	StarPilot	Climber	StarPilot
3	Ninja	CoinRun	Ninja

Table 3: Experiment Phases for Different Environments

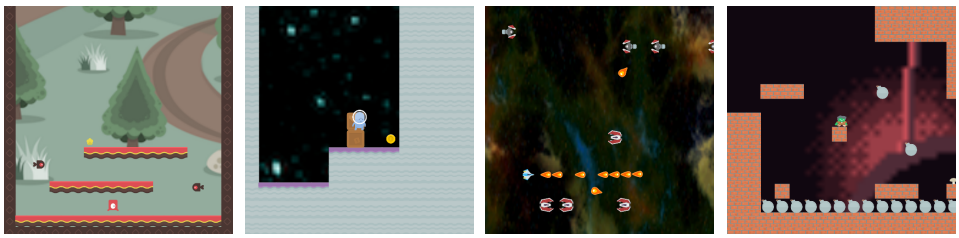

Figure 4: From left to right, Climber, CoinRun, StarPilot and, Ninja. In our experiments the backgrounds are all black (use_backgrounds=False).

PPO Hyperparameters	Description	Value
gamma	Discount factor for future rewards	0.999
vf_coef	Weight of value function loss	0.5
ent_coef	Weight of entropy bonus	0.01
norm_adv	Normalize advantages during optimization	true
num_steps	Number of steps per environment per update	256
clip_coef	Clipping factor for policy loss	0.2
gae_lambda	Generalized Advantage Estimation parameter	0.95
batch_size	Total number of samples per batch	16384
clip_vloss	Clip value function loss	false
target_kl	Target KL divergence	0.05
update_epochs	Number of optimization epochs per update	3
minibatch_size	Size of mini-batches used in optimization	2048
max_grad_norm	Maximum gradient norm for clipping	0.5
anneal_lr	Whether to anneal learning rate over time	false
num_envs	Number of parallel environments	64
num_minibatches	Number of mini-batches per optimization step	8

Table 4: PPO Hyperparameters

Other Parameters	Description	Value
cuda	Use CUDA for computation	true
easy	Procgen easy parameter	1
proc_start	Starting level in Procgen	1
reward_limit	Checkpoint minimum reward limit (P)	7.5
report_epoch	Number of steps between evaluation reports	163840
learning_rate	Learning rate for optimizer	0.0005
max_ep_length	Maximum number of steps per episode	1000
use_monochrome	Whether to use monochrome assets in environment	0
eval_batch_size	Number of episodes for evaluation	30
max_eval_ep_len	Maximum length of episodes during evaluation	1000
proc_num_levels	Number of levels in the Procgen environment	30
total_timesteps	Total number of timesteps for training	9000000
eval_specific_envs	Number of environments used for evaluation	30
torch_deterministic	Enable deterministic operations in PyTorch	true
min_similarity_score	Minimum cosine similarity for activation w	0.98
checkpoint_interval	Minimum cosine similarity for activation C	500,000

Table 5: Other Experiment Parameters

A.3 PNN with PPO Algorithm Details

Progressive Neural Networks (PNN) were introduced by Rusu et al. (2016). In their paper, they describe how separate policy networks (referred to as columns) and links between columns (adapters) are used to improve continuous learning. However, they report their results using the Asynchronous Advantage Actor-Critic (A3C) algorithm Mnih (2016). We could not find any evidence indicating whether they initialized separate columns and adapters for the value (critic) network as well as the policy (actor) network. Additionally, we could not locate any official implementation online, nor any implementation using actor-critic methods.

Intuitively, we expect that if PNN works for the policy, it should also work for the value function. Therefore, we implemented separate *columns* for both the critic and actor networks, along with adapters for each new task. We encountered another issue: PPO is generally expected to outperform A3C in ProcGen environments. Thus, comparing HOP-PPO or base PPO with PNN-A3C would be unfair to PNN. To address this, we modified the PNN implementation to use a PPO update that propagates through separate columns and adapters.

Since the inputs for ProcGen environments are image-based, we use a single ReLU-activated convolutional layer as each adapter network. The input to the adapter network is the final convolutional

output from the Critic or Actor of each column. The adapter outputs of the previous column are added to the original output and passed to the fully connected layers. All adapters are included in the gradient graphs to promote transfer, but the actor and critic columns are not included unless they are the active column.

[Upon publication, we will release our code for PNN-PPO].

A.4 Hierarchical Weighting Examples

Hierarchical weightings. This hierarchical structure implies that as the agent continues learning, the contributions of older policies diminish if more recent checkpoints are activated. Conversely, if more recent policies do not activate, the older policies will have a stronger influence. For example, consider the policy at the fourth checkpoint (π_4) as depicted in Figure 2. If activations A_{s3}, A_{s2}, and A_{s1} occur, the policy output for the current state s_t is given by:

$$\pi_{4a}(s_t) = \frac{9}{24}\pi_{1_{\text{max}}}(s_{1_{\text{max-sim}}}) + \frac{1}{2}\pi_{2_{\text{max}}}(s_{2_{\text{max-sim}}}) + \frac{1}{2}\pi_{3_{\text{max}}}(s_{3_{\text{max-sim}}}) + \pi_4(s_t)$$

In contrast, if only A_{s1} activates, the output becomes:

$$\pi_{4a}(s_t) = \frac{1}{2}\pi_{1_{\text{max}}}(s_{1_{\text{max-sim}}}) + \pi_4(s_t)$$

Here, the contribution of π_1 diminishes as more recent policies are activated. However, if only A_{s1} is activated, π_1 provides a significant contribution, which remains substantial regardless of how many other checkpoint policies exist but are inactive.

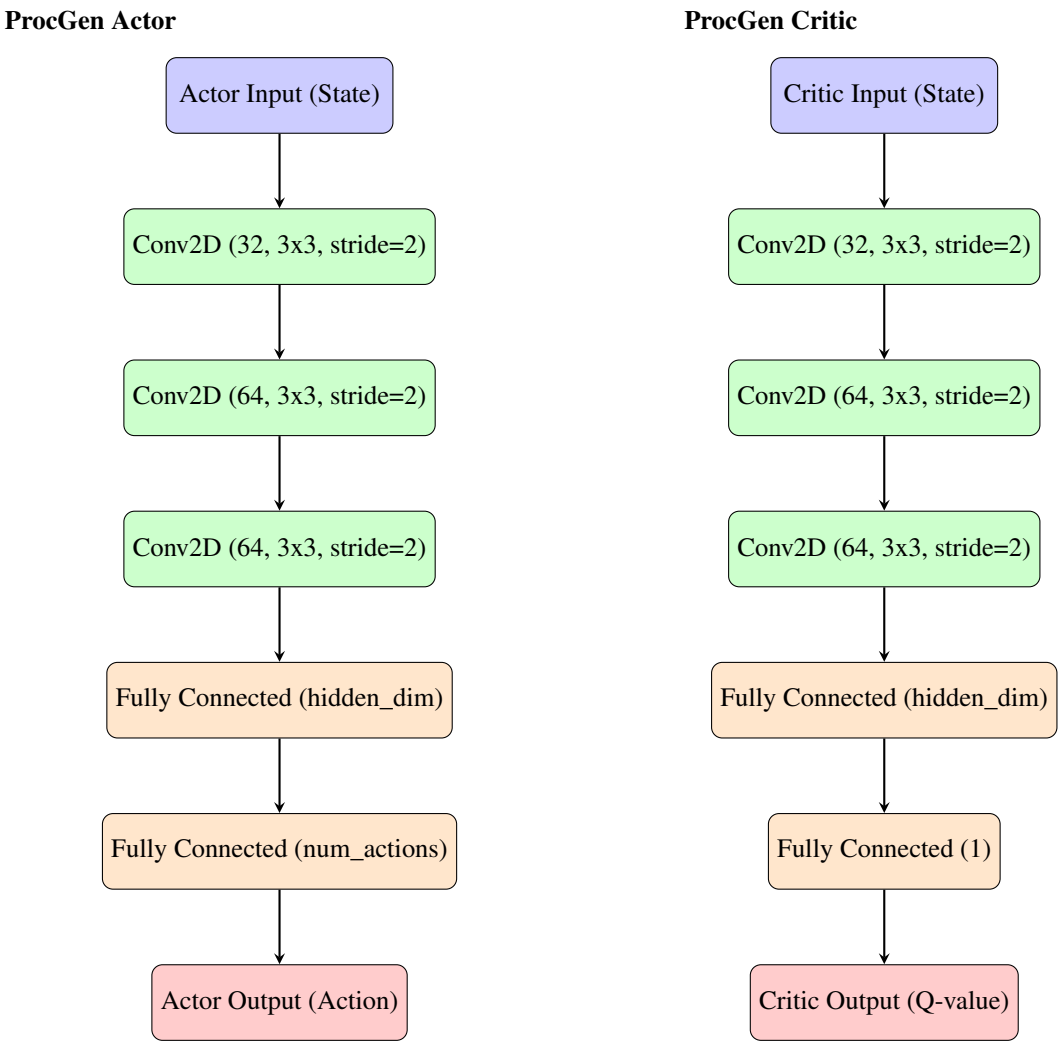

Figure 5: Separate Actor and Critic Networks for the ProcGen Architecture