Learning Compositional Neural Programs for Continuous Control

Thomas Pierrot
InstaDeep
t.pierrot@instadeep.com

Nicolas Perrin
CNRS, Sorbonne Université
perrin@isir.upmc.fr

Feryal Behbahani
DeepMind
feryal@google.com

Alexandre Laterre
InstaDeep
a.laterre@instadeep.com

Olivier Sigaud
Sorbonne Université
olivier.sigaud@upmc.fr

Karim Beguir
InstaDeep
kb@instadeep.com

Nando de Freitas
DeepMind
nandodefreitas@google.com

Abstract

We propose a novel solution to challenging sparse-reward, continuous control problems that require hierarchical planning at multiple levels of abstraction. Our solution, dubbed AlphaNPI-X, involves three separate stages of learning. First, we use off-policy reinforcement learning algorithms with experience replay to learn a set of atomic goal-conditioned policies, which can be easily repurposed for many tasks. Second, we learn self-models describing the effect of the atomic policies on the environment. Third, the self-models are harnessed to learn recursive compositional programs with multiple levels of abstraction. The key insight is that the self-models enable planning by imagination, obviating the need for interaction with the world when learning higher-level compositional programs. We empirically show that AlphaNPI-X can effectively learn to tackle challenging sparse manipulation tasks, such as stacking multiple blocks, where powerful model-free baselines fail.

1 Introduction

Many real-world tasks are naturally decomposed into hierarchical structures. We hypothesize that learning a variety of skills which can be reused and composed to learn more complex skills is key to tackling long-horizon sparse reward tasks in a sample efficient manner. Such compositionality, formalised by hierarchical RL (HRL), enables agents to explore in a temporally correlated manner, improving sample efficiency by reusing previously trained lower level skills. Unfortunately, prior studies in HRL typically assume that the hierarchy is given, or learn very simple forms of hierarchy in a model-free manner. We propose a novel method, *AlphaNPI-X*, to learn programmatic policies which can perform hierarchical planning at multiple levels of abstraction in sparse reward continuous control problems. We first train low-level atomic policies that can be recomposed and re-purposed, represented by a *single* goal-conditioned neural network. We leverage off-policy reinforcement learning with hindsight experience replay [1] to train these efficiently. Next, we learn a transition model over the effects of these atomic policies, to imagine likely future scenarios, removing the need to interact with the real environment. Lastly, we learn recursive compositional programs, which combine low-level atomic policies at multiple levels of hierarchy, by planning over the learnt transition models, alleviating the need to interact with the environment. This is made possible by

1st NeurIPS workshop on Interpretable Inductive Biases and Physically Structured Learning (2020), virtual.

extending the AlphaNPI algorithm [2] which applies AlphaZero-style planning [3] in a recursive manner to learn recombinable libraries of symbolic programs.

We show that our agent can learn to successfully combine skills hierarchically to solve challenging robotic manipulation tasks through look-ahead planning, even in the absence of any further interactions with the environment and where powerful model-free baselines struggle to get off the ground.[1]

2 Problem Definition

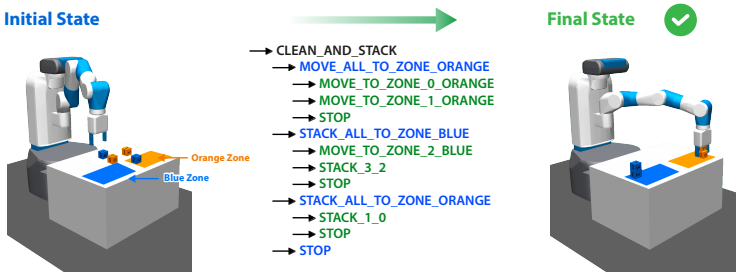

Figure 1: Illustrative example of an execution trace for the CLEAN_AND_STACK program. *Atomic* program calls are shown in green and *non-atomic* program calls are shown in blue.

In this paper, we aim to learn libraries of skills to solve a variety of tasks in continuous action domains with sparse rewards. Consider the task shown in Figure 1 where the agent's goal is to take the environment from its initial state where four blocks are randomly placed to the desired final state where blocks are in their corresponding coloured zones and stacked on top of each other. We formalize skills and their combinations as *programs*. An example programmatic trace for solving this task is shown where the sequence of programs are called to take the environment from the initial state to the final rewarding state. We specify two distinct types of programs: *Atomic* programs (shown in green) are low-level goal-conditioned policies which take actions in the environment for a fixed number of steps T. *Non-atomic* programs (shown in blue) are a combination of atomic and/or other non-atomic programs, allowing multiple possible levels of hierarchies in behaviour.

We base our experiments on a set of robotic tasks with continuous action space. Due to the lack of any long-horizon hierarchical multi-task benchmarks, we extended the OpenAI Gym Fetch environment [4] with tasks exhibiting such requirements. We consider a target set of tasks represented by a hierarchical library of programs, see Table 1. These tasks involve controlling a robotic arm with 7-DOF to manipulate 4 coloured blocks in the environment. Tasks vary from simple block stacking to arranging all blocks into different areas depending on their colour. Initial block positions on the table and arm positions are randomized in all tasks. We consider 20 atomic programs that correspond to operating on one block at a time as well as 8 non-atomic programs that require interacting with 2 to 4 blocks.

PROGRAM	ARGUMENTS	DESCRIPTION
STACK	TWO BLOCKS ID	STACK A BLOCK ON ANOTHER BLOCK.
MOVE_TO_ZONE	BLOCK ID & COLOUR	MOVE A BLOCK TO A COLOUR ZONE.
STACK_ALL_ALTERNATE	NO ARGUMENTS	STACK ALL BLOCKS WITH COLOR ALTERNATING ORDER.
STACK_ALL_CONSECUTIVE	NO ARGUMENTS	STACK ALL BLOCKS WITH COLOR CONSECUTIVE ORDER.
STACK_ALL_TO_ZONE_BLUE	COLOUR	STACK BLUE BLOCKS IN THE BLUE ZONE.
STACK_ALL_TO_ZONE_ORANGE	COLOUR	STACK ORANGE BLOCKS IN THE ORANGE ZONE.
MOVE_ALL_TO_ZONE_BLUE	COLOUR	MOVE BLUE BLOCKS TO THE BLUE ZONE.
MOVE_ALL_TO_ZONE_ORANGE	COLOUR	MOVE ORANGE BLOCKS TO THE ORANGE ZONE.
CLEAN_TABLE	NO ARGUMENTS	MOVE ALL BLOCKS TO THEIR COLOUR ZONE.
CLEAN_AND_STACK	NO ARGUMENTS	STACK BLOCKS OF THE SAME COLOUR IN ZONES.

Table 1: Programs library for the fetch arm environment. We obtain 8 non-atomic programs and 20 atomic programs when considering all possible combinations when expending their arguments.

[1] Videos of agent behaviour are available at: `https://sites.google.com/view/alphanpix`

We consider a continuous action space $\mathcal{A}$, a continuous state space $\mathcal{S}$, an initial state distribution ρ and a transition function $\mathcal{T} : \mathcal{A} \times \mathcal{S} \to \mathcal{S}$. The state vector contains the positions, rotations, linear and angular velocities of the gripper and all blocks.

We aim to learn a set of n programs $p_i,\ i \in \{1, \dots, n\}$. A program p_i is defined by its pre-condition $\phi_i : \mathcal{S} \to \{0, 1\}$ which assesses whether the program can start and its post-condition $\psi_i : \mathcal{S} \to \{0, 1\}$ which corresponds to the reward function here. Each program is associated to an MDP $(\mathcal{S}, \mathcal{A}, \mathcal{T}, R_i)$ which can start only in states such that the pre-condition ϕ_i is satisfied, and where R_i is a reward function that outputs 1 when the post-condition ψ_i is satisfied and 0 otherwise. Atomic programs are represented by a goal-conditioned neural network with continuous action space. Non-atomic programs use a modified action space: we replace the original continuous action space by a discrete action space where actions correspond to call programs p_i or a STOP action that enables the current program to terminate and return the execution to its calling program. Atomic programs don't have a STOP action, they terminate after T timesteps.

3 AlphaNPI-X

AlphaNPI-X learns to solve multiple tasks by composing programs at different levels of hierarchy. Given a one-hot encoding of a program and states from the environment, the meta-controller calls either an atomic program, a non-atomic program or the STOP action. Learning in our system operates in three stages: first we learn the atomic programs, then we learn a transition model over their effect and finally we train the meta-controller to combine them. We provide detailed explanation of how these modules are learned below.

Learning Atomic Programs

Atomic programs are executed by an atomic policy conditioned on the atomic program's index and produces continuous actions. To execute an atomic program p_i from an initial state s satisfying the pre-condition ϕ_i, we condition the atomic policy on the program's one-hot encoding and take actions for T timesteps in the environment to satisfy the atomic program's post-condition.

We parametrize our atomic policy as a shared goal-conditioned policy using Universal Value Function Approximators (UVFAs) [5]. UVFAs estimate a value function that does not only generalise over states but also goals. To accelerate training of this goal-conditioned UVFA, we leverage the "final" goal relabeling strategy introduced in HER [1]. Past episodes of experience are relabelled retroactively with goals that are different from the goal aimed for during data collection and instead correspond to the goal achieved in the final state of the episode. The mappings from state vector to goals is simply done via extracting the programs whose post-conditions are satisfied in that state. If several programs post-conditions are satisfied, the program used for goal relabeling is sampled uniformly among these. In contrary, if no program post-condition is verified, a full-zero encoding is used. We use DDPG [6] to train the goal-conditioned policy.

The atomic policy is trained from a sparse reward signal where the agent receives 1 if the goal is satisfied and 0 otherwise. We train the policy with HER and learn to execute atomic programs sampled uniformly from any state $s \sim \rho$. However, the distribution of initial states encountered by each atomic program may be very different when executing programs sequentially (as will happen when non-atomic programs are introduced). Thus, we do not reset the environment between episodes with a probability 0.5, to approximate the real distribution of initial states when atomic programs are chained together. Hence, the initial state s is either sampled randomly or kept as the last state observed in the previous episode.

Learning Self-Behavioural Model

After learning a set of atomic programs we learn a transition model over their effects: $\Omega_w : \mathcal{S} \times \{1 \dots k\} \to \mathcal{S}$, parameterized with a neural network. This module takes as input an initial state and an atomic program index and the output is the prediction of the environment state obtained when rolling the policy associated to this program for T timesteps from the initial state. As in [7], we use an ensemble of fully connected MLPs initialised randomly and train them by minimizing the mean-squared error to the ground-truth final states. Learning this model enables to make jumpy

predictions over the effect of executing an atomic program during search, hence avoiding any further calls to atomic programs that would each have to perform many actions in the environment.

Learning the Meta-Controller

In order to compose atomic programs together into hierarchical stacks of non-atomic programs, we use a meta-controller inspired by AlphaNPI [2]. The meta-controller interprets and selects the next program to execute using neural-network guided Monte Carlo Tree Search (MCTS) [3, 2], conditioned on the current program index and states from the environment. We train the meta-controller using the recursive MCTS strategy introduced in AlphaNPI [2]: during search, if the selected action is non-atomic, we recursively build a new Monte Carlo tree for that program. In AlphaNPI [2], similar to AlphaZero, during the tree search, future scenarios were evaluated by leveraging the ground-truth environment, without any temporal abstraction. Instead in this work, we do not use the environment directly during planning, but replace it by our learnt transition model over the effects of the atomic programs, the self-behavioural model described in Section 3, resulting in a far more sample efficient algorithm.

4 Experiments and Results

We first train the atomic policies using DDPG with HER relabelling described in Section 3. Initial block positions on the table as well as gripper position are randomized. We then train the self-behavioural model to predict the effect of atomic programs, using a dataset made of 100k episodes collected with the goal-conditioned policy. We did not reset the environment between two episodes with a probability 0.5 similar to the atomic program training regime.

To train the meta-controller, we randomly sample from the set of non-atomic programs during training. We compare two different inferences strategies for evaluation: 1) Rely only on the policy network, without planning. 2) Plan a full trajectory using the learned transition model (i.e. self-behavioural model) and commit to it during the full episode (i.e. open-loop planning). Our results indicate that removing planning significantly reduces performance, particularly for the tasks that require precise execution of multiple atomic programs sequentially.

PROGRAM	NO PLAN	PLANNING	HIERARCHICAL PPO	MULTITASK DDPG	MULTITASK DDPG + HER
CLEAN_TABLE	1 ± 1	50 ± 21	0 ± 0	0.0	0.0
CLEAN_AND_STACK	3 ± 2	17 ± 11	1 ± 1	0.0	0.0
STACK_ALL_ALTERNATE	3 ± 1	38 ± 1	0 ± 0	0.0	0.0
STACK_ALL_CONSECUTIVE	3 ± 4	32 ± 5	0 ± 0	0.0	0.0
STACK_ALL_TO_ZONE_BLUE	49 ± 23	90 ± 1	46 ± 30	0.0	0.0
STACK_ALL_TO_ZONE_ORANGE	59 ± 8	90 ± 1	42 ± 25	0.0	0.0
MOVE_ALL_TO_ZONE_BLUE	61 ± 8	93 ± 1	72 ± 5	0.0	0.0
MOVE_ALL_TO_ZONE_ORANGE	58 ± 13	92 ± 1	64 ± 5	0.0	0.0

Table 2: We compare the success rate (in percentage) of AlphaNPI-X in different inference settings (No Plan and Planning) as well as the performance of 3 model-free baselines. Each program is evaluated by executing 100 episodes with randomized environment configuration.

We compare our method to three baselines to illustrate the difficulty for RL methods to solve tasks with sparse reward signals. First, we implemented a multitask DDPG (M-DDPG) that takes as input the environment state and a one-hot encoding of a non-atomic program and has continuous action space. This is similar to our goal-conditioned atomic policy. During training, for each episode a random program index is selected which defines the reward function. Second, we implemented a M-DDPG + HER agent which leverages richer goals for non-atomic tasks. While M-DDPG gets conditioned on the program index, M-DDPG + HER receives the goal in the form of the desired (x, y, z) coordinates for all objects in the environment as in [8]. This knowledge is not available to AlphaNPI-X. As in HER, goals are relabelled during training.

We observe that M-DDPG is unable to learn to execute any non-atomic program. It is also the case for M-DDPG + HER despite having access to the additional privileged goal representations. This shows that standard exploration mechanisms in model-free agents such as in DDPG, where Gaussian noise is added to the actions, is very unlikely to lead to rewarding sequences and hence learning is hindered.

Finally, we implemented a hierarchical PPO (H-PPO) agent which leverages the pre-trained atomic policy and its behavioral model. We observe that H-PPO manages to obtain non-zero performance on the simpler tasks, however despite the access to learned skills and the behaviour model it struggles to get off the ground for the more complex ones. These results suggest that the combination of pre-trained atomic policies and the recursive AlphaZero-style planning with multiple levels of hierarchy leveraged in AlphaNPI-X is required to get off the ground when the reward is sparse and additionally only a few action sequences can result in successful task completion.

5 Conclusion

In this paper, we proposed AlphaNPI-X, a novel method for constructing a hierarchy of abstract actions in a rich object manipulation domain with sparse rewards and long horizons. By learning a self-behavioural model, we leveraged the power of recursive AlphaZero-style look-ahead planning across multiple levels of hierarchy. Experimental results demonstrated that AlphaNPI-X, using an abstract imagination-based reasoning, can simultaneously solve multiple complex tasks involving dexterous object manipulation beyond the reach of model-free methods.

References

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017.

[2] Thomas Pierrot, Guillaume Ligner, Scott E Reed, Olivier Sigaud, Nicolas Perrin, Alexandre Laterre, David Kas, Karim Beguir, and Nando de Freitas. Learning compositional neural programs with recursive tree search and planning. In *Advances in Neural Information Processing Systems 32*, pages 14646–14656, 2019.

[3] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[5] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1312–1320, Lille, France, 07–09 Jul 2015. PMLR.

[6] 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. *arXiv preprint arXiv:1509.02971*, 2015.

[7] Ramanan Sekar, Oleh Rybkin, Kostas Daniilidis, Pieter Abbeel, Danijar Hafner, and Deepak Pathak. Planning to explore via self-supervised world models. *arXiv preprint arXiv:2005.05960*, 2020.

[8] John B Lanier, Stephen McAleer, and Pierre Baldi. Curiosity-driven multi-criteria hindsight experience replay. *arXiv preprint arXiv:1906.03710*, 2019.