Supplementary for Neuro-algorithmic Policies Enable Fast Combinatorial Generalization

Marin Vlastelica [1] , **Michal Rolínek** [1] **Georg Martius** [1]

A. Data Generation

To do imitation learning, we require expert data. For CRASH JEWEL HUNT (CRASH 5×5 and 5×10), LEAPER(GRID) and MAZE we can determine the exact ground truth costs leading to optimal behavior. As an example, CRASH 5×5 contains moving boxes that when encountered lead to instant death, meaning infinite costs and otherwise the fixed cost of moving around in the environment.

Since the environments become deterministic for a fixed random seed, we first unrolled their dynamics for each level. After obtaining the underlying grid structure and entities, we labeled them with costs and constructed a graph that reflects the grid structure. An expert trajectory is constructed by applying Dijkstra's algorithm on this graph and the human-labeled costs and then executing in simulation.

For the CRASH JEWEL HUNT experiments, we randomly sampled 2000 solvable levels by varying number of boxes per column, their speed, the agent start position and the jewel position. The training levels were taken from the first half and the second half of levels was used for testing. For the PROCGEN environments LEAPER(GRID) and MAZE we have taken the levels determined by seeds 0-1000.

For CHASER, we applied a similar procedure but additionally, we recorded two sets of human trajectories, as we observed benefits in performance by incorporating more different expert trajectories for the same level. Since both the search procedure and human labeling are time consuming for this environment, we collected fewer expert trajectories for the CHASER than for the other environments, 3×100, two-thirds of which are from human players.

Level seeds 1000000-1001000 were taken for testing in the PROCGEN experiments.

B. Environments

Our method is applicable in discrete environments, therefore we evaluated on environments from the PROCGEN benchmark and the CRASH JEWEL HUNT environment.

We created the CRASH JEWEL HUNT environment to evaluate our method, where the goal is for the fox (Crash) to reach the jewel. We found this environment convenient since we can influence the combinatorial difficulty directly, which is not true for the PROCGEN benchmark where we are limited to the random seeds used in the OpenAI implementation. The sources of variation in the CRASH JEWEL HUNT are the box velocities, initial positions, sizes, as well as the agent initial position and the jewel position.

We modified the LEAPER environment to make grid steps for our method to be applicable. This involved making the logs on the river move in discrete steps as well as the agent. Moreover, in our version, the agent is not transported by the logs as they move, but has to move actively with them. For an additional description of the PROCGEN environmnets, we refer the reader to Cobbe et al. (2019).

C. Network Architecture and Input

For all of our experiments, we use the PyTorch implementation of the ResNet18 architecture as the base of our model. All approaches receive two stacked frames of the two previous time steps as input to make dynamics prediction possible. For the PPO baseline, we did not observe any benefit in adding the stacked frames as input and we used stable-baselines implementation from OpenAI to train it on the PROCGEN environments.

In the case of the behavior cloning baseline, the problem is a multi-class classification problem with the output being a multinomial distribution over actions.

For the variant NAP*, we train a cost prediction network

*Equal contribution [1]Max Planck Institute for Intelligent Systems , T ubingen, Germany. Correspondence to: Marin Vlastelica <mvlastelica@tue.mpg.de>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

	CRASH 5×5	CRASH 5×10	LEAPER(GRID)	MAZE
learning rate	10^{-3}	10^{-3}	10^{-3}	10^{-3}
α	0.2	0.2	0.15	0.15
λ	20	20	20	20
resnet layers	4	4	4	4
kernel size	4	4	6	6
batch size	32	32	16	16

Table S1: Training hyperparameters, where $alpha$ denotes the margin that was used on the vertex costs and λ the interpolation parameter for blackbox differentiation of Dijkstra's algorithm. We vary the kernel size of the initial convolution for ResNet18.

on top of which we run Dijkstra's algorithm on the output costs of the planning graph. This requires modifications to the original ResNet18 architecture. We remove the linear readout of the original ResNet18 architecture and replace it with a convolutional layer of filter size 1 and adaptive max pooling layer to obtain the desired dimensions of the underlying latent planning graph. More concretely, the output x of the last ResNet18 block is followed by the following operation (as output by PyTorch) to obtain the graph costs:

```
Sequential(
    Conv2d(256, 2, kernel_size=(1, 1), stride=(1, 1))
    Abs()
    AdaptiveMaxPool2d(output_size=(grid_height, grid_width))
)
```

Where grid_{height,width} denotes the height and width of the planning grid. For the full variant of NAP with goal prediction and agent position prediction we have a separate position classifier that has the same base architecture as the cost prediction network with 2 additional linear readouts for the likelihoods of the latent graph vertices, more concretely (as output by PyTorch):

```
Sequential(
    Conv2d(256, 2, kernel_size=(1, 1), stride=(1, 1))
    Abs()
    AdaptiveMaxPool2d(output_size=(grid_height, grid_width))
    Flatten()
    Linear(grid_height × grid_width, grid_height × grid_width)
)
```

For training the position classifier, we use a standard cross-entropy loss on the likelihoods. For NAP with position classification, we use the ground-truth expert start and goal positions to calculate the Hamming loss of the predicted path by the solver. At evaluation time, NAP uses the position classifier to determine the start and end vertices in the latent planning graph.

D. Training Procedure

For CRASH 5×5, CRASH 5×10, LEAPER(GRID) and MAZE we train the models on the same #levels, namely 1, 2, 5, 10, 20, 50, 100, 200, 500 and 1000. We evaluate on unseen 1000 levels in order to show that NAP exhibits superior generalization. The levels are generated as per description in section A. Each dataset is normalize to be zero mean and unit variancesize. For each dataset size (#levels) we run experiments with 3 random restarts (seeds for network initialization). For all experiments, we make use of the ADAM optimizer.

We determine the number of epochs for training depending on each dataset size as $\min(150000/\#\text{levels}, 15000)$ to have roughly the same number of gradient updates in each experiment. We take the minimum over the 2 values because for smaller number of levels a large number iterations is not necessary to achieve good performance, but for a larger number of levels it is necessary. If we observe no error on the training set, we stop the training.

For the CHASER, the training conditions were analogous to the other environments, only of slightly smaller scale due to its higher complexity. Models were trained on 10, 20, 50, 100 and 200 levels and evaluated on 200 unseen levels.

	CHASER
learning rate	$1e^{-3}$
α	0.2
λ	40
resnet layers	3
kernel size	4
batch size	16

Table S2: Training hyperparameters for the CHASER experiment, where α denotes the margin that was used on the vertex costs and λ the interpolation parameter for blackbox differentiation od Dijkstra.

D.1. PPO Training Procedure

The training of the PPO baseline is exactly the same as described in Cobbe et al. (2019) using the official code from https://github.com/openai/train-procgen, see Table S3 for the used parameters. The network architecture is the IMPALA-CNN. The algorithm is trained on the specified number of levels for 200 million environments interactions gathered from 256 (instead of 64 as in Cobbe et al. (2019)) to compensate for not having access to 4 parallel workers. We report numbers for 3 independent restarts.

learning rate	$5e^{-4}$
α	0.2
discount γ	0.999
entropy coefficient	0.01
steps per update	2^{16}

Table S3: PPO hyperparameters, as used in Cobbe et al. (2019).

D.2. DrAC Training Procedure

For the DrAC algorithm, we run all versions introduced by Raileanu et al. (2020) (Meta-DrAC, RL2-DrAC, UCB-DrAC and DrAC-Crop) and choose the best one in the main plots, denoted as DrAC*. We used the original hyperparameters from Raileanu et al. (2020) and the implementation from https://github.com/rraileanu/auto-drac. As with the other experiments, we report numbers from 3 different random seeds.

E. On Comparing Imitation Learning to Reinforcement Learning

We compare our method to Data Augmented Actor Critic, PPO and a behavior cloning baseline. Arguably, since NAP is used in an imitation learning setting, it reaps benefits from having access to expert trajectories. Nevertheless, it is not straight forward that embedding a solver in a neural architecture leads to better generalization in comparison to reinforcement learning methods.

Behavior cloning with a standard neural architecture has access to the same amount of data, whereas reinforcement learning agents have access to orders of magnitude more data for inference ($2 \cdot 10^8$ transitions in comparison to $\sim 10^5$ max). This would lead us to believe that reinforcement learning agents are able to generalize well because of the sheer amount of data that they have at their disposal, but we show that nevertheless it is possible to extract meaningful policies even with a small number of training levels seen with expert trajectories, that are more optimal.

In addition, NAP is a general architecture paradigm that may be composed with various different objective functions, including a reinforcement learning formulation. It would be interesting to see how NAP behaves when used in such a formulation and if this would lead to even better generalization properties with more data. We provide training performance curves in Fig. S1 and density plots for different numbers of training levels in Fig. S2.

F. Data Regularized Actor-Critic

The DrAC algorithm (Raileanu et al., 2020) attacks the problem of generalization in reinforcement learning from a different angle, namely applying (in some versions optimized) data augmentations to the PPO algorithm. The main insight is that naively applied data augmentations result in faulty policy gradient estimates because the resulting policy after applying the augmentation is too far from the behavior policy.

To alleviate this, a policy regularization term G_π and value function regularization term G_V are added to the PPO objective:

$$J_{\text{DrAC}} = J_{\text{PPO}} - \alpha_r(G_\pi + G_V). \tag{1}$$

Furthermore, various augmentations and augmentation selection schemes were proposed. We ran all of the proposed selectin schemes on our environments, Meta-DrAC, RL2-DrAC, UCB-DrAC and DrAC. Meta-DrAC meta-learns the weights of a convolutional neural network used for data augmentation. RL2-DrAC meta-learns a policy that selects an augmentation from a pre-defined set of augmentations. UCB-DrAC is a bandit formulation of the augmentation selection problem with application of a upper confidence bound algorithm for selection strategy. DrAC denotes the version with the crop augmentation, which has been shown to work well with more than half of the environments in the ProcGen benchmark. For more details, we refer the reader to Raileanu et al. (2020).

DrAC's approach to improving generalization is orthogonal to NAP and the approaches may be composed in order to achieve even better generalization capabilities.

G. Additional Related Work

Generalization in reinforcement learning In addition to the work of Raileanu et al. (2020), there is a plethora of approaches that attempt to improve generalization in reinforcement learning by considering various data augmentation techniques while mainly drawing motivation from supervised learning approaches (Kostrikov et al., 2020; Laskin et al., 2020). Other approaches combine unsupervised learning with data augmentation (Srinivas et al., 2020; Zhan et al., 2020).

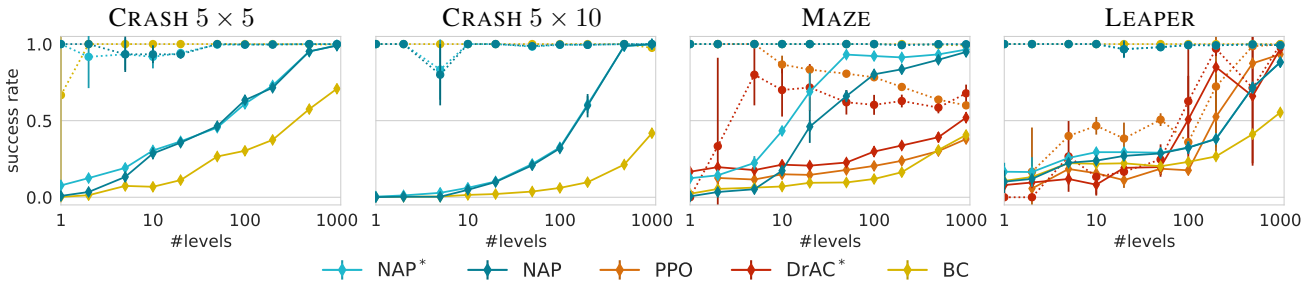

Figure S1: The dotted lines denote the training performance of the methods. We observe that the behavior cloning baseline and NAP have fitted the training set almost perfectly.

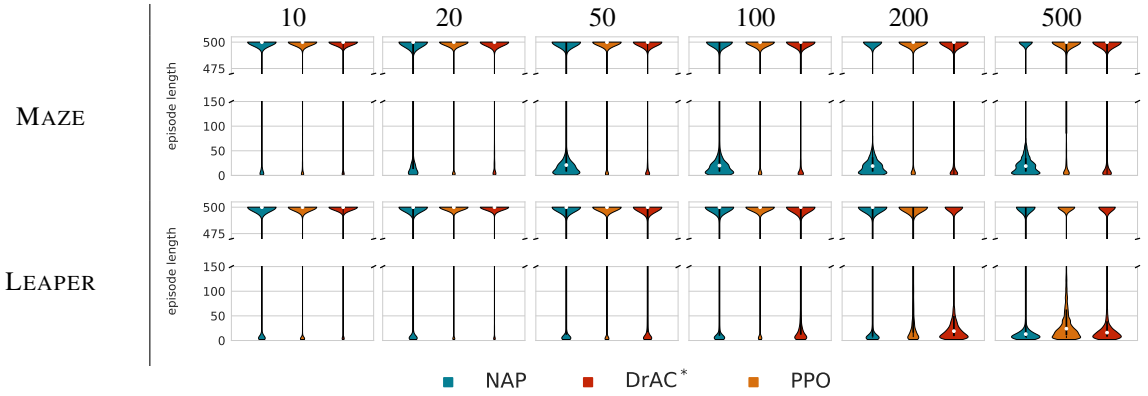

Figure S2: Density plots of performance on the test set (1000 unseen levels) after different number of training levels for the MAZE and LEAPER environments, the white point denotes the median performance on the test set.

Notably, the problem of sim-to-real transfer can be seen as a problem of generalization to different system dynamics. Domain randomization (Tobin et al., 2017), i.e. augmenting system dynamics in a structured way, has emerged as one of the main techniques for tackling this problem.

H. Cost Margin Ablation

Here we show the results for the Maze environment with 200, 500 and 1000 train levels evaluated on 1000 unseen test levels, with and without margin. The results indicate that using the margin on the costs improves generalization.

	200	500	1000
w	0.839 ± 0.014	0.895 ± 0.007	0.948 ± 0.006
w/o	0.834 ± 0.015	0.883 ± 0.008	0.855 ± 0.241

References

Cobbe, K., Hesse, C., Hilton, J., and Schulman, J. Leveraging procedural generation to benchmark reinforcement learning. *arXiv preprint:1912.01588*, 2019.

Kostrikov, I., Yarats, D., and Fergus, R. Image augmentation is all you need: Regularizing deep reinforcement learning from pixels. *arXiv preprint arXiv:2004.13649*, 2020.

Laskin, M., Lee, K., Stooke, A., Pinto, L., Abbeel, P., and Srinivas, A. Reinforcement learning with augmented data. *arXiv preprint arXiv:2004.14990*, 2020.

Raileanu, R., Goldstein, M., Yarats, D., Kostrikov, I., and Fergus, R. Automatic data augmentation for generalization in deep reinforcement learning. *arXiv preprint arXiv:2006.12862*, 2020.

Srinivas, A., Laskin, M., and Abbeel, P. Curl: Contrastive unsupervised representations for reinforcement learning. *arXiv preprint arXiv:2004.04136*, 2020.

Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., and Abbeel, P. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pp. 23–30. IEEE, 2017.

Zhan, A., Zhao, P., Pinto, L., Abbeel, P., and Laskin, M. A framework for efficient robotic manipulation. *arXiv preprint arXiv:2012.07975*, 2020.