Effective, interpretable algorithms for curiosity automatically discovered by evolutionary search

Martin Schneider*[1], Ferran Alet*[1], Tomás Lozano-Pérez[1], Leslie Pack Kaelbling[1]

***Abstract*— We take the hypothesis that curiosity is a mechanism found by evolution that encourages meaningful exploration early in an agent's life in order to expose it to experiences that enable it to obtain high rewards over the course of its lifetime. We formulate the problem of generating curious behavior as one of meta-learning: an outer loop will search over a space of curiosity algorithms that dynamically adapt the agent's reward signal, and an inner loop will perform standard reinforcement learning using the adapted reward signal. These meta-learned algorithms are pieces of code similar to those designed by humans in ML papers. Our rich language of programs combines neural networks with other building blocks such as buffers, nearest-neighbor modules and custom loss functions. We find two novel curiosity algorithms that perform on par or better than human-designed published curiosity algorithms in domains as disparate as grid navigation with image input, acrobot, lunar lander, MuJoCo ant and MuJoCo hopper. Interestingly both algorithms, which we call** FAST **(Fast Action Space Transition) and *Cycle-consistency* intrinsic motivation, have interpretable functionalities and, to the best of our knowledge, had not been proposed before. Finally, altohugh most of the intrinsic motivation literature focuses on purely intrinsic reward, we show that this needs to be combined with extrinsic reward for meaningful performance in many environments and describe a simple way to combine them automatically discovered by our search.**

I. Introduction

Note: this work complements [1], where we showed how to meta-learn curiosity algorithms and demonstrated why this leads to greater generalization capabilities than meta-learning neural representations. In this work we focus instead on analyzing the product of our search: we explain in detail the algorithms discovered by our search, in a similar way a human would introduce their own algorithms.

When a reinforcement learning agent is learning to behave, it is critical that it both explores its domain and exploits its rewards effectively. In very simple problems, it is possible to solve the problem optimally, using techniques of Bayesian decision theory [2]. However, these techniques do not scale well and are not effectively applicable to the problems addressable by modern deep RL, with large state and action spaces and sparse rewards. This difficulty has left researchers the task of designing good exploration strategies for RL systems in complex environments. One way to think of this problem is in terms of *curiosity* or *intrisic motivation*: constructing reward signals that augment or even replace the extrinsic reward from the domain, which induce the RL agent to explore their domain in a way that results in effective longer-term learning and behavior [3], [4], [5]. The primary difficulty with this approach is that researchers are hand-designing these strategies: it is difficult for humans to systematically consider the space of strategies or to tailor strategies for the distribution of environments an agent might be expected to face.

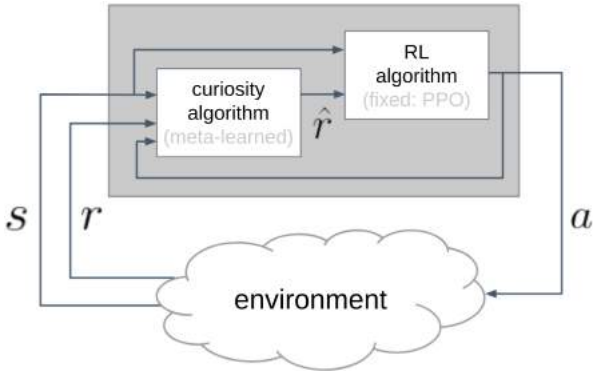

Fig. 1. Our RL agent is augmented with a *curiosity module*, obtained by meta-learning over a complex space of programs, which computes a pseudo-reward $\hat{r}$ at every time step.

We take inspiration from the curious behavior observed in young humans and other animals and hypothesize that curiosity is a mechanism found by evolution that encourages meaningful exploration early in agent's life. This exploration exposes the agent to experiences that enable it to learn to obtain high rewards over the course of its lifetime. We propose to formulate the problem of generating curious behavior as one of meta-learning: an outer loop, operating at "evolutionary" scale will search over a space of algorithms for generating curious behavior by dynamically adapting the agent's reward signal, and an inner loop will perform standard reinforcement learning using the adapted reward signal. This process is illustrated in figure 1; note that the aggregate agent, outlined in gray, has the standard interface of an RL agent. The inner RL algorithm is continually adapting to its input stream of states and rewards, attempting to learn a policy that optimizes the discounted sum of proxy rewards $\sum_{k\geq 0}\gamma^k\hat{r}_{t+k}$. The outer "evolutionary" search is attempting to find a program for the curiosity module, so as to optimize the agent's lifetime return $\sum_{t=0}^{T} r_t$, or another global objective like the mean performance on the last few trials.

In this meta-learning setting, our objective is to find a curiosity module that works well given a distribution of

* equal contribution

[1] Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology {martinfs,alet,tlp,lpk}@mit.edu

environments from which we can sample at meta-learning time. Meta-RL has been widely explored recently, in some cases with a focus on reducing the amount of experience needed by initializing the RL algorithm well [6], [7] and, in others, for efficient exploration [8], [9]. The environment distributions in these cases have still been relatively low-diversity, mostly limited to variations of the same task, such as exploring different mazes or navigating terrains of different slopes. We would like to discover curiosity mechanisms that can generalize across a much broader distribution of environments, even those with different state and action spaces: from image-based games, to joint-based robotic control tasks. To do that, we perform meta-learning in a rich, combinatorial, open-ended space of programs.

II. Meta-learning curiosity algorithms

A. Meta-learning problem formulation

Let us assume we have an agent equipped with an RL algorithm $\mathscr{A}$ (such as DQN or PPO, with all hyperparameters specified), which receives states and rewards from and outputs actions to an environment $\mathscr{E}$, generating a stream of experienced transitions $e(\mathscr{A};\mathscr{E})_t = (s_t, a_t, r_t, s_{t+1})$. The agent continually learns a policy $\pi(t) : s_t \to a_t$, which will change in time as described by algorithm $\mathscr{A}$; so $\pi(t) = \mathscr{A}(e_{1:t-1})$ and thus $a_t \sim \mathscr{A}(e_{1:t-1})(s_t)$. Although this need not be the case, we can think of $\mathscr{A}$ as an algorithm that tries to maximize the discounted reward $\sum_i \gamma^i r_{t+i}, \gamma < 1$ and that, at any time-step t, always takes the greedy action that maximizes its estimated expected discounted reward.

To add exploration to this policy, we include a *curiosity module* $\mathscr{C}$ that has access to the stream of state transitions e_t experienced by the agent and that, at every time-step t, outputs a proxy reward $\widehat{r}_t$. We connect this module so that the original RL agent receives these modified rewards, thus observing $e(\mathscr{A},\mathscr{C};\mathscr{E})_t = (s_t, a_t, \widehat{r}_t = \mathscr{C}(e_{1:t-1}), s_{t+1})$, without having access to the original r_t. Now, even though the inner RL algorithm acts in a purely exploitative manner with respect to $\widehat{r}$, it may efficiently explore in the outer environment.

Our overall goal is to design a curiosity module $\mathscr{C}$ that induces the agent to maximize $\sum_{t=0}^{T} r_t$, for some number of total time-steps T or some other global goal, like final episode performance. In an episodic problem, T will span many episodes. More formally, given a single environment $\mathscr{E}$, RL algorithm $\mathscr{A}$, and curiosity module $\mathscr{C}$, we can see the triplet (environment, curiosity module, agent) as a dynamical system that induces state transitions for the environment, and learning updates for the curiosity module and the agent. Our objective is to find $\mathscr{C}$ that maximizes the expected original reward obtained by the composite system in the environment. Note that the expectation is over two different distributions at different time scales: there is an "outer" expectation over environments $\mathscr{E}$, and in "inner" expectation over the rewards received by the composite system in that environment, so our final objective is:

$$\max_{\mathscr{C}} \left[\mathbb{E}_{\mathscr{E}} \left[\mathbb{E}_{r_t \sim e(\mathscr{A},\mathscr{C};\mathscr{E})} \left[\sum_{t=0}^{T} r_t \right] \right] \right] .$$

B. Programs for curiosity

In science and computing, mathematical language has been very successful in describing varied phenomena and powerful algorithms with short descriptions. Therefore, in order to obtain curiosity modules that can generalize over a very broad range of tasks and that are sophisticated enough to provide exploration guidance over very long horizons, we describe them in terms of general programs in a domain-specific language. Algorithms in this language will map a history of (s_t, s_{t+1}, a_t, r_t) tuples into a proxy reward $\widehat{r}_t$. Inspired by human-designed systems that compute and use intrinsic rewards, and to simplify the search, we decompose the curiosity module into two components: the first, I, outputs an intrinsic reward value i_t based on the current experienced transition (s_t, a_t, s_{t+1}) (and past transitions $(s_{1:t-1}, a_{1:t-1})$ indirectly through its memory); the second, χ, takes the current time-step t, the actual reward r_t, and the intrinsic reward i_t (and, if it chooses to store them, their histories) and combines them to yield the proxy reward $\widehat{r}_t$. To ease generalization across different timescales, in practice, before feeding t into χ we normalize it by the total length of the agent's lifetime, T.

Both programs consist of a directed acyclic graph (DAG) of modules with polymorphically typed inputs and outputs. As shown in figure 2, there are four classes of modules: **Input** modules (shown in blue), drawn from the set $\{s_t, a_t, s_{t+1}\}$ for the I component and from the set $\{i_t, r_t\}$ for the χ component. They have no inputs, and their outputs have the type corresponding to the types of states and actions in whatever domain they are applied to, or the reals numbers for rewards. **Buffer and parameter** modules (shown in gray) of two kinds: FIFO queues that provide as output a finite list of the k most recent inputs, and neural network weights initialized at random at the start of the program and which may (pink border) or may not get updated via back-propagation depending on the computation graph. **Functional** modules (shown in white), which compute output values given the inputs.

A single node in the DAG is designated as the *output node* (shown in green): the output of this node is considered to be the output of the entire program, but it need not be a leaf node of the DAG. On each call to a program (corresponding to one time-step of the system) the current input values and parameter values are propagated through the functional modules, and the output node's output is given to the RL algorithm. Before the call terminates, the FIFO buffers are updated and the adjustable parameters are updated via gradient descent using the Adam optimizer [10]. Most operations are differentiable and thus able to propagate gradient backwards. Some operations are not differentiable such as buffers (to avoid backpropagating through time) and "Detach" whose purpose is stopping the gradient from flowing back. In practice, we have multiple copies of the same agent running at the same time, with both a shared policy and shared curiosity module. Thus, we execute multiple reward predictions on a batch and then update on a batch.

Programs representing several published designs for cu-

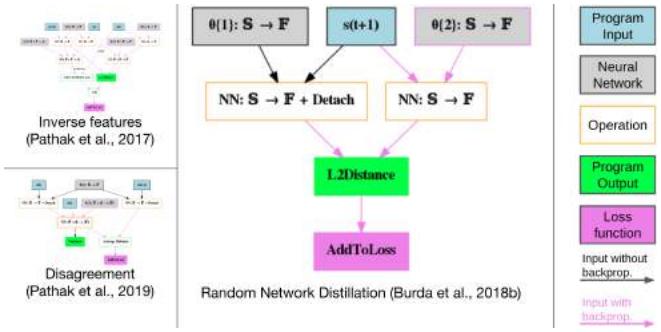

Fig. 2. Example diagrams of published algorithms covered by our language (larger figures in the appendix). The green box represents the output of the intrinsic curiosity function, the pink box is the loss to be minimized. Pink arcs represent paths and networks along which gradients flow back from the minimizer to update parameters.

riosity modules that perform internal gradient descent, including inverse features [3], random network distillation (RND) [4], and self-supervised exploration via disagreement [11], are shown in figure 2 (and bigger versions can be found in appendix B). We can also represent algorithms similar to novelty search [12] and EX^2 [13], which include buffers and nearest neighbor regression modules. Details on the data types and module library are given in appendix A.

A crucial, and possibly somewhat counter-intuitive, aspect of these programs is their use of neural network weight updates via gradient descent as a form of memory. In the parameter update step, all adjustable parameters are decremented by the gradient of the sum of the outputs of the loss modules, with respect to the parameters. This type of update allows the program to, for example, learn to make some types of predictions, online, and use the quality of those predictions in a state to modulate the proxy reward for visiting that state (as is done, for example, in RND).

Key to our program search are *polymorphic data types*: the inputs and outputs to each module are typed, but the instantiation of some types, and thus of some operations, depends on the environment. We have four types: reals $\mathbb{R}$, state space of the given environment $\mathbb{S}$, action space of the given environment $\mathbb{A}$ and feature space $\mathbb{F}$, used for intermediate computations and always set to $\mathbb{R}^{32}$ in our current implementation. For example, a neural network module going from $\mathbb{S}$ to $\mathbb{F}$ will be instantiated as a convolutional neural network if $\mathbb{S}$ is an image and as a fully connected neural network of the appropriate dimension if $\mathbb{S}$ is a vector. This facility means that the same curiosity program can be applied, independent of whether states are represented as images or vectors, or whether the actions are discrete or continuous, or the dimensionality of either. This type of abstraction enables our meta-learning approach to discover curiosity modules that generalize *radically*, applying not just to new tasks, but to tasks with substantially different input and output spaces than the tasks they were trained on.

C. Summary of the experiments

In this section we review how we searched over curiosity algorithms and how we sped up the search. For more information, please refer to appendix C on how we search in this large space and appendix D about the details of the experiments. Our code to search and execute the algorithms, which can take in any OpenAI gym environment [14], can be found at https://github.com/mfranzs/meta-learning-curiosity-algorithms.

We first generated all valid intrinsic curiosity programs with at most 7 operations, of which there are 52,000. Then, we evaluated the most promising 26,000, according to a learned predictor, on a task of exploring a grid, where we evaluated the total number of distinct states covered by each algorithm. We found that almost all programs perform about the same, but a small fraction of 0.5% of programs perform statistically significantly better, see fig. 10 in the appendix. We then picked approximately the top 10% of programs on grid world and evaluated them on two much more complex tasks (*acrobot* and *lunar lander*) involving an order of magnitude more learning steps and vector inputs instead of image inputs. Surprisingly, we found (fig 11) that almost all programs that had statistically significant performance on grid world also had good performance on both of these "test" domains. In other words, programs found on a single task generalized *radically* to very different tasks.

Finally, we compared the performance of the top 16 algorithms against simple baselines (constant intrinsic rewards of 0,1,-1 and gaussian noise) and the 3 published algorithms of figure B on two MuJoCo tasks: *ant* and *hopper*. We found that our meta-learned algorithms performed statistically significantly better than baseline algorithms and statistically equivalent to published algorithms. Moreover, they also performed either statistically equivalently or better in the meta-training task (grid navigation) and the two previous meta-test task (acrobot and lunar lander). When inspecting those top programs, we found that they were all minor variations of two different algorithms which, to the best of our knowledge, have not been previously proposed. We discuss them in the following sections.

III. Discovered Curiosity Algorithms

A. Fast Action Space Transition (FAST) intrinsic motivation

13 of the top 16 algorithms in the grid navigation task were versions of the same algorithm; we show one of them (the best in our search) in figure 3. The algorithm trains a single neural network (a CNN or MLP depending on the type of state) to predict the action from s_{t+1} and then compares its predictions based on s_{t+1} with its predictions based on s_t, generating a high intrinsic reward when the difference is large. The *action prediction loss* module either computes a softmax followed by NLL loss or appends zeros to the action to match dimensions and applies MSE loss, depending on the type of the action space. In other words, the network predicting the action is learning to imitate the policy (or the *inverse* policy if predicting from s_{t+1}), since it does not have direct access to the neural network policy of the RL agent. We hypothesize that the reward dynamics vary over the state landscape based on the novelty of a given state, in the form of two phases. First, getting to a novel region of the

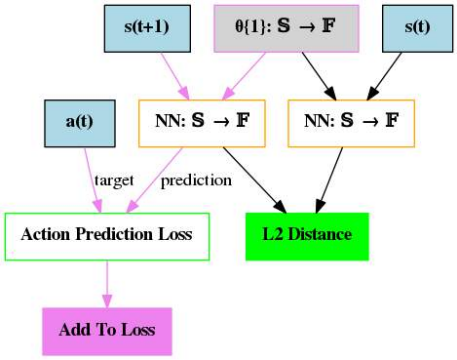

Fig. 3. Fast Action-Space Transition (FAST): top-performing intrinsic curiosity algorithm discovered in our phase 1 search. A more formal description is given in equation 1.

space will be rewarded by noise from a poorly-trained policy-mimicking model. Second,taking actions on consecutive steps in regions where we have a correct policy-mimicking model will also yield reward. We call this algorithm FAST (Fast Action Space Transition) intrinsic motivation, because it rewards taking big jumps in a space that predicts the action. In equations:

FAST ACTION SPACE TRANSITION (FAST)

$$i_t = ||NN_\theta(s_{t+1}) - NN_\theta(s_t)||_2$$
$$\min_\theta \mathscr{L}(NN_\theta(s_{t+1}), a(t)) \quad (1)$$

As mentioned, we can also set the loss to predict the action from the current state; i.e., $\min_\theta \mathscr{L}(NN_\theta(s_t), a(t))$.

To the best of our knowledge, despite its simplicity (or maybe because of it) the algorithm represented by this program has not been proposed in the literature before.

B. Cycle-consistency intrinsic motivation

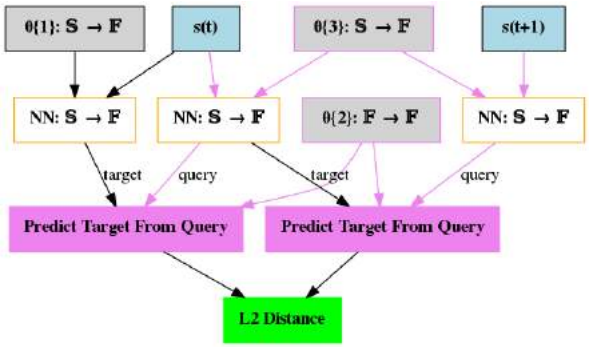

Fig. 4. *Cycle-consistency* intrinsic motivation: 3 of the top 16 programs on grid world are variants of this program. A more formal description is given in equation 2.

The other 3 algorithms of the top 16 in the grid navigation task were variations of a much more complex algorithm that we depict in figure 4. The algorithm combines several concepts seen in the literature, such as an untrained network like RND [4] and predicting another state in feature space like [3], [11], but also includes weight sharing between both predictions, which makes the algorithm hard to interpret at first sight. However, one can give meaning to the role of all 3 neural networks by considering what their role should be in order to minimize the loss. To do so, let us name the networks: $\theta\{1\}$ (as labeled in the figure) as r_{θ_1} (for random embedding), $\theta\{2\}$ as b_{θ_2} (for backwards) and $\theta\{3\}$ as fr_{θ_3} (for forward and random embedding) and look at the algorithm in equation form:

BACK AND FORTH INTRINSIC MOTIVATION

$$i_t = \|b_{\theta_2}(fr_{\theta_3}(s_t)) - b_{\theta_2}(fr_{\theta_3}(s_{t+1}))\|$$
θ_1 is kept at its random initialization
$$\min_{\theta_2} \|b_{\theta_2}(fr_{\theta_3}(s_t)) - r_{\theta_1}(s_t)\| + \|b_{\theta_2}(fr_{\theta_3}(s_{t+1})) - fr_{\theta_3}(s_t)\|$$
$$\min_{\theta_3} \|b_{\theta_2}(fr_{\theta_3}(s_t)) - r_{\theta_1}(s_t)\| \quad (2)$$

We can see that r_{θ_1} will indeed be a random embedding because the network is randomly initialized and is not trained. Then, we observe that the second term in the loss for θ_2, which does not involve θ_3 and thus θ_2 has to minimize alone, is $\|b_{\theta_2}(fr_{\theta_3}(s_{t+1})) - fr_{\theta_3}(s_t)\|$. In this term, b_{θ_2} receives a transformation of s_{t+1} and has to make it very similar to the same transformation applied to s_t; therefore, this term is similar to cycle-consistency found in some other parts of machine learning [15] and b_{θ_2} must act like a backward model. Finally, looking at the minimization of θ_3 receives the original s_t and has to output a vector such that the backward model will bring it close to the random embedding of s_t. Therefore θ_3 must learn a forward model composed with the random embedding of θ_1. Finally, we see that the algorithm outputs $\|b_{\theta_2}(fr_{\theta_3}(s_t)) - b_{\theta_2}(fr_{\theta_3}(s_{t+1}))\|$, going forward and backward for both s_{t+1} and s_t and comparing the difference. In summary, this distance combines errors in the cycle-consistency of predictions (which will be higher in unvisited parts of the state) with distance in the random embedding space between $s(t)$ and $s(t+1)$, i.e. moving to a very different state.

C. Combining intrinsic and extrinsic reward

Many intrinsic curiosity works only endow the agent with intrinsic motivation, disregarding external reward, and show that this leads to meaningful behavior. This behavior maximizes the overall objective for tasks where the goal is purely exploratory (like Mario or our grid navigation), but not for more focused tasks such as running to a particular location. Similarly, in lunar lander and acrobot, training an agent just on intrinsic motivation leads to very poor performance; this is because the environment finishes when the agent reaches the goal state (bringing the arm up), which stops the intrinsic reward, encouraging the agent to never reach the goal.

We therefore meta-learned a reward combiner that took the stream of all intrinsic rewards (coming from a meta-learned algorithm, such as the ones discussed in the previous two sections) and the stream of all extrinsic rewards r_t, along

with a number indicating how far along its learning life the agent is, i.e. t/T with T being the total number of time-steps in the entire life of the agent, possibly over many episodes. The combiner then outputs a single number, the reward $\hat{r}_t$ for the policy to optimize. Our reward combiner was developed in *lunar lander* (the simplest environment with meaningful extrinsic reward) based on the best program among a preliminary set of 16,000 programs (which resembled Random Network Distillation, its computation graph is shown in appendix E). Among a set of 2,500 candidates (with 5 or less operations) the best reward combiner discovered by our search was $\widehat{r}_t = \frac{(1+i_t-t/T)\cdot i_t + t/T \cdot r_t}{1+i_t}$. When $t = 0$, this evaluates to i_t. When $t = T$, this evaluates to $\frac{i^2+r}{1+i}$, which is r when $0 \leq i << 1$. Thus, the combiner roughly interpolates from purely intrinsic reward to purely extrinsic reward, if the intrinsic reward has started to drop down to 0 by the end. In future work, we hope to co-adapt the search for intrinsic reward programs and combiners as well as find multiple reward combiners. More details on the search can be found in appendix A.2.

IV. Conclusions

In this work, we proposed to meta-learn algorithms and show that by transferring programs we can generalize between tasks much more varied than previously possible in meta-RL, even between those with different input or output spaces. We showed that the algorithms resulting from this search can be interpreted as doing something meaningful, and introduced two algorithms: Fast Action Space Transition (FAST) and *Cycle-consistency* instrinsic motivation. Our relatively modest compute (2 GPU-weeks) and a simple search method restricted us to a medium-sized search space, but we expect that future work could search over significantly bigger spaces. It thus may be possible to automatically search for new algorithms from even more fundamental building blocks.

References

[1] F. Alet, M. F. Schneider, T. Lozano-Perez, and L. P. Kaelbling, "Meta-learning curiosity algorithms," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=BygdyxHFDS

[2] M. Ghavamzadeh, S. Mannor, J. Pineau, and A. Tamar, "Bayesian reinforcement learning: A survey," *Foundations and Trends in Machine Learning*, vol. 8, no. 5–6, 2015.

[3] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2017, pp. 16–17.

[4] Y. Burda, H. Edwards, A. Storkey, and O. Klimov, "Exploration by random network distillation," *arXiv preprint arXiv:1810.12894*, 2018.

[5] P.-Y. Oudeyer, "Computational theories of curiosity-driven learning," *arXiv preprint arXiv:1802.10546*, 2018.

[6] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," *arXiv preprint arXiv:1703.03400*, 2017.

[7] I. Clavera, A. Nagabandi, R. S. Fearing, P. Abbeel, S. Levine, and C. Finn, "Learning to adapt: Meta-learning for model-based control," in *International Conference on Learning Representations*, 2019.

[8] Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel, "Rl2: Fast reinforcement learning via slow reinforcement learning," *arXiv preprint arXiv:1611.02779*, 2016.

[9] J. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick, "Learning to reinforcement learn. arxiv 1611.05763," 2017.

[10] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[11] D. Pathak, D. Gandhi, and A. Gupta, "Self-supervised exploration via disagreement," *arXiv preprint arXiv:1906.04161*, 2019.

[12] J. Lehman and K. O. Stanley, "Exploiting open-endedness to solve problems through the search for novelty." in *ALIFE*, 2008, pp. 329–336.

[13] J. Fu, J. Co-Reyes, and S. Levine, "Ex2: Exploration with exemplar models for deep reinforcement learning," in *Advances in Neural Information Processing Systems*, 2017, pp. 2577–2587.

[14] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[15] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2223–2232.

[16] F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2018, in press, available at http://automl.org/book.

[17] Z. Karnin, T. Koren, and O. Somekh, "Almost optimal exploration in multi-armed bandits," in *International Conference on Machine Learning*, 2013, pp. 1238–1246.

[18] K. Jamieson and A. Talwalkar, "Non-stochastic best arm identification and hyperparameter optimization," in *Artificial Intelligence and Statistics*, 2016, pp. 240–248.

[19] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[20] I. Kostrikov, "Pytorch implementations of reinforcement learning algorithms," https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail, 2018.

[21] A. Paszke, S. Gross, and A. Lerer, "Automatic differentiation in PyTorch," in *International Conference on Learning Representations*, 2017.

[22] M. Chevalier-Boisvert, L. Willems, and S. Pal, "Minimalistic gridworld environment for openai gym," https://github.com/maximecb/gym-minigrid, 2018.

[23] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 5026–5033.

[24] R. K. Srivastava, B. R. Steunebrink, and J. Schmidhuber, "First experiments with powerplay," *Neural Networks*, vol. 41, pp. 130–136, 2013.

[25] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction

and intrinsic motivation," in *Advances in neural information processing systems*, 2016, pp. 3675–3683.

[26] C. Florensa, D. Held, X. Geng, and P. Abbeel, "Automatic goal generation for reinforcement learning agents," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. Stockholmsmssan, Stockholm Sweden: PMLR, 10–15 Jul 2018, pp. 1515–1528. [Online]. Available: http://proceedings.mlr.press/v80/florensa18a.html

[27] P.-Y. Oudeyer, F. Kaplan, and V. V. Hafner, "Intrinsic motivation systems for autonomous mental development," *IEEE transactions on evolutionary computation*, vol. 11, no. 2, pp. 265–286, 2007.

[28] J. Schmidhuber, "Driven by compression progress: A simple principle explains essential aspects of subjective beauty, novelty, surprise, interestingness, attention, curiosity, creativity, art, science, music, jokes," in *Workshop on anticipatory behavior in adaptive learning systems*. Springer, 2008, pp. 48–76.

[29] M. G. Azar, B. Piot, B. A. Pires, J.-B. Grill, F. Altché, and R. Munos, "World discovery models," *arXiv preprint arXiv:1902.07685*, 2019.

[30] B. Eysenbach, A. Gupta, J. Ibarz, and S. Levine, "Diversity is all you need: Learning skills without a reward function," *arXiv preprint arXiv:1802.06070*, 2018.

[31] C. Florensa, Y. Duan, and P. Abbeel, "Stochastic neural networks for hierarchical reinforcement learning," *arXiv preprint arXiv:1704.03012*, 2017.

[32] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, *et al.*, "Noisy networks for exploration," *arXiv preprint arXiv:1706.10295*, 2017.

[33] H. Tang, R. Houthooft, D. Foote, A. Stooke, O. X. Chen, Y. Duan, J. Schulman, F. DeTurck, and P. Abbeel, "# exploration: A study of count-based exploration for deep reinforcement learning," in *Advances in neural information processing systems*, 2017, pp. 2753–2762.

[34] S. Forestier and P.-Y. Oudeyer, "Modular active curiosity-driven discovery of tool use," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2016, pp. 3965–3972.

[35] A. A. Taïga, W. Fedus, M. C. Machado, A. Courville, and M. G. Bellemare, "Benchmarking bonus-based exploration methods on the arcade learning environment," *arXiv preprint arXiv:1908.02388*, 2019.

[36] B. C. Stadie, G. Yang, R. Houthooft, X. Chen, Y. Duan, Y. Wu, P. Abbeel, and I. Sutskever, "Some considerations on learning to explore via meta-reinforcement learning," *arXiv preprint arXiv:1803.01118*, 2018.

[37] A. Gupta, R. Mendonca, Y. Liu, P. Abbeel, and S. Levine, "Meta-reinforcement learning of structured exploration strategies," in *Advances in Neural Information Processing Systems*, 2018, pp. 5302–5311.

[38] Z. Zheng, J. Oh, and S. Singh, "On learning intrinsic rewards for policy gradient methods," in *Advances in Neural Information Processing Systems*, 2018, pp. 4644–4654.

[39] H.-T. L. Chiang, A. Faust, M. Fiser, and A. Francis, "Learning navigation behaviors end-to-end with autorl," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 2007–2014, 2019.

[40] A. Faust, A. Francis, and D. Mehta, "Evolving rewards to automate reinforcement learning," *arXiv preprint arXiv:1905.07628*, 2019.

[41] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[42] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[43] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *arXiv preprint arXiv:1808.05377*, 2018.

[44] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," *arXiv preprint arXiv:1802.03268*, 2018.

[45] H. Mendoza, A. Klein, M. Feurer, J. T. Springenberg, and F. Hutter, "Towards automatically-tuned neural networks," in *Workshop on Automatic Machine Learning*, 2016, pp. 58–65.

[46] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8697–8710.

[47] A. R. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Satenstein: Automatically building local search sat solvers from components," in *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[48] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 2962–2970. [Online]. Available: http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf

[49] S. Gonzalez and R. Miikkulainen, "Improved training speed, accuracy, and data utilization through loss function optimization," *arXiv preprint arXiv:1905.11528*, 2019.

[50] ——, "Evolving loss functions with multivariate taylor polynomial parameterizations," 2020.

[51] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *arXiv preprint arXiv:1603.06560*, 2016.

[52] J. Schmidhuber, "Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook," Ph.D. dissertation, Technische Universität München, 1987.

[53] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *arXiv preprint arXiv:1710.05941*, 2017.

[54] A. Gaier and D. Ha, "Weight agnostic neural networks," *arXiv preprint arXiv:1906.04358*, 2019.

[55] D. G. Wilson, S. Cussat-Blanc, H. Luga, and J. F. Miller, "Evolving simple programs for playing atari games," in *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2018, pp. 229–236.

[56] S. Kelly and M. I. Heywood, "Multi-task learning in atari video games with emergent tangled program graphs," in *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2017, pp. 195–202.

[57] T. Silver, K. R. Allen, A. K. Lew, L. P. Kaelbling, and J. Tenenbaum, "Few-shot bayesian imitation learning with logic over programs," *arXiv preprint arXiv:1904.06317*, 2019.

[58] S. Bengio, Y. Bengio, and J. Cloutier, "On the search for new learning rules for anns," *Neural Processing Letters*, vol. 2, no. 4, pp. 26–30, 1995.

[59] I. Bello, B. Zoph, V. Vasudevan, and Q. V. Le, "Neural optimizer search with reinforcement learning," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 459–468.

[60] S. Reed and N. De Freitas, "Neural programmer-interpreters," *arXiv preprint arXiv:1511.06279*, 2015.

[61] T. Pierrot, G. Ligner, S. Reed, O. Sigaud, N. Perrin, A. Laterre, D. Kas, K. Beguir, and N. de Freitas, "Learning compositional neural programs with recursive tree search and planning," *arXiv preprint arXiv:1905.12941*, 2019.

[62] F. Alet, T. Lozano-Perez, and L. P. Kaelbling, "Modular meta-learning," in *Proceedings of The 2nd Conference on Robot Learning*, 2018, pp. 856–868.

[63] F. Alet, E. Weng, T. Lozano-Perez, and L. Kaelbling, "Neural relational inference with fast modular meta-learning," in *Advances in Neural Information Processing Systems (NeurIPS) 32*, 2019.

[64] S. Thrun and L. Pratt, *Learning to learn*. Springer Science & Business Media, 1998.

[65] J. Clune, "Ai-gas: Ai-generating algorithms, an alternate paradigm for producing general artificial intelligence," *arXiv preprint arXiv:1905.10985*, 2019.

[66] C. Finn, "Learning to learn with gradients," Ph.D. dissertation, EECS Department, University of California, Berkeley, Aug 2018. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-105.html

[67] Z. Xu, H. P. van Hasselt, and D. Silver, "Meta-gradient reinforcement learning," in *Advances in neural information processing systems*, 2018, pp. 2396–2407.

[68] V. Veeriah, M. Hessel, Z. Xu, R. Lewis, J. Rajendran, J. Oh, H. van Hasselt, D. Silver, and S. Singh, "Discovery of useful questions as auxiliary tasks," *arXiv preprint arXiv:1909.04607*, 2019.

[69] C. Fernando, D. Banarse, C. Blundell, Y. Zwols, D. Ha, A. A. Rusu, A. Pritzel, and D. Wierstra, "Pathnet: Evolution channels gradient descent in super neural networks," *arXiv preprint arXiv:1701.08734*, 2017.

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

[71] E. Parisotto, J. L. Ba, and R. Salakhutdinov, "Actor-mimic: Deep multitask and transfer reinforcement learning," *arXiv preprint arXiv:1511.06342*, 2015.
[72] A. Nichol, V. Pfau, C. Hesse, O. Klimov, and J. Schulman, "Gotta learn fast: A new benchmark for generalization in rl," *arXiv preprint arXiv:1804.03720*, 2018.
[73] T. Yu, D. Quillen, Z. He, R. Julian, K. Hausman, S. Levine, and C. Finn, "Meta-world: A benchmark and evaluation for multi-task and meta-reinforcement learning," 2019. [Online]. Available: https://github.com/rlworkgroup/metaworld
[74] R. Wang, J. Lehman, J. Clune, and K. O. Stanley, "Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions," *arXiv preprint arXiv:1901.01753*, 2019.
[75] R. Houthooft, Y. Chen, P. Isola, B. Stadie, F. Wolski, O. J. Ho, and P. Abbeel, "Evolved policy gradients," in *Advances in Neural Information Processing Systems*, 2018, pp. 5400–5409.
[76] L. Kirsch, S. van Steenkiste, and J. Schmidhuber, "Improving generalization in meta reinforcement learning using learned objectives," *arXiv preprint arXiv:1910.04098*, 2019.

APPENDIX

Acknowledgments

We thank Kelsey Allen, Peter Karkus, Kevin Smith, Josh Tenenbaum and the rest of the Honda-CMM MIT team for their insightful feedback. We thank Chris Lu for his idea on what the algorithm in figure 4 is computing. We also want to thank Bernadette Bucher, Chelsea Finn, Abhishek Gupta, Deepak Pathak, Lerrel Pinto, Oleh Rybkin, Karl Schmeckpeper and Joaquin Vanschoren for valuable conversations. Finally, we also want to thank Maria Bauza and Tej Chajed for their feedback on early drafts and Clement Gehring for his help setting up the experiments.

We gratefully acknowledge support from NSF grants 1523767 and 1723381, AFOSR grant FA9550-17-1-0165, ONR grant N00014-18-1-2847, the Honda Research Institute, SUTD Temasek Laboratories and the MIT Quest for Intelligence. Any opinions, findings, and conclusions or recommendations expressed in this material do not necessarily reflect the views of our sponsors.

A. Details of our domain-specific language for curiosity algorithms

We have the following types. Note that $\mathbb{S}$ and $\mathbb{A}$ get defined differently for every environment.

- $\mathbb{R}$: real numbers such as r_t or the dot-product between two vectors.
- $\mathbb{R}^+$: numbers guaranteed to be positive, such as the distance between two vectors. The only difference to our program search between $\mathbb{R}$ and $\mathbb{R}^+$ is in pruning programs that can optimize objectives without looking at the data. For $\mathbb{R}^+$ we check whether they can optimize down to 0, for $\mathbb{R}$ we check whether they can optimize to arbitrarily negative values.
- state space $\mathbb{S}$: the environment state, such as a matrix of pixels or a vector with robot joint values. The particular form of this type is adapted to each environment.
- action space $\mathbb{A}$: either a 1-hot description of the action or the action itself. The particular form of this type is adapted to each environment.
- feature-space $\mathbb{F} = \mathbb{R}^{32}$: a space mostly useful to work with neural network embeddings. For simplicity, we only have a single feature space.
- List[$\mathbb{X}$]: for each type we may also have a list of elements of that type. All operations that take a particular type as input can also be applied to lists of elements of that type by mapping the function to every element in the list. Lists also support extra operations such as average or variance.

1) Curiosity operations:

Operation	Input type(s)	State	Output type
Add	$\mathbb{R}$, $\mathbb{R}$		$\mathbb{R}$
RunningNorm	$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$
VariableAsBuffer	$\mathbb{X}$	$List[\mathbb{X}]$	$List[\mathbb{X}]$
NearestNeighborRegressor	$\mathbb{F}$, $\mathbb{F}$	$List[\mathbb{F}]$	$\mathbb{F}$
SubtractOneTenth	$\mathbb{R}$		$\mathbb{R}$
NormalDistribution			$\mathbb{R}$
Subtract	$\mathbb{R}$, $\mathbb{R}$		$\mathbb{R}$
Sqrt(Abs(x))	$\mathbb{R}$		$\mathbb{R}^+$
NN $\mathbb{F},\mathbb{F}\to\mathbb{F}$	$\mathbb{F}$, $\mathbb{F}$	$\Theta_{\mathbb{F},\mathbb{F}\to\mathbb{F}}$	$\mathbb{F}$
NN $\mathbb{F},\mathbb{F}\to\mathbb{A}$	$\mathbb{F}$, $\mathbb{F}$	$\Theta_{\mathbb{F},\mathbb{F}\to\mathbb{A}}$	$\mathbb{A}$
NN $\mathbb{F}\to\mathbb{A}$	$\mathbb{F}$	$\Theta_{\mathbb{F}\to\mathbb{A}}$	$\mathbb{A}$
NN $\mathbb{A}\to\mathbb{F}$	$\mathbb{A}$	$\Theta_{\mathbb{A}\to\mathbb{F}}$	$\mathbb{F}$
(C)NN	$\mathbb{S}$	$\Theta_{\mathbb{S}\to\mathbb{F}}$	$\mathbb{F}$
(C)NN, Detach	$\mathbb{S}$	$\Theta_{\mathbb{S}\to\mathbb{F}}$	$\mathbb{F}$
(C)NNEnsemble	$\mathbb{S}$	$5x\Theta_{\mathbb{S}\to\mathbb{F}}$	$List[\mathbb{F}]$
NN Ensemble $\mathbb{F}\to\mathbb{F}$	$\mathbb{F}$	$5x\Theta_{\mathbb{F}\to\mathbb{F}}$	$List[\mathbb{F}]$
NN Ensemble $\mathbb{F},\mathbb{F}\to\mathbb{F}$	$\mathbb{F}$, $\mathbb{F}$	$5x\Theta_{\mathbb{F},\mathbb{F}\to\mathbb{F}}$	$List[\mathbb{F}]$
NN Ensemble $\mathbb{F},\mathbb{A}\to\mathbb{F}$	$\mathbb{F}$, $\mathbb{A}$	$5x\Theta_{\mathbb{A},\mathbb{F}\to\mathbb{F}}$	$List[\mathbb{F}]$
MinimizeValue	$\mathbb{R}$	Adam	
L2Norm	$\mathbb{X}$		$\mathbb{R}^+$
L2Distance	$\mathbb{X}$, $\mathbb{X}$		$\mathbb{R}$
ActionSpaceLoss	$\mathbb{X}$, $\mathbb{A}$		$\mathbb{R}^+$
DotProduct	$\mathbb{X}$, $\mathbb{X}$		$\mathbb{R}$
Add	$\mathbb{X}$, $\mathbb{X}$		$\mathbb{X}$
Detach	$\mathbb{X}$		$\mathbb{X}$
Mean	$List[\mathbb{R}]$		$\mathbb{R}$
Variance	$List[\mathbb{X}]$		$\mathbb{R}^+$
Mean	$List[\mathbb{X}]$		$\mathbb{X}$
Mapped L2 Norm	$List[\mathbb{X}]$		$List[\mathbb{R}]$
Average Distance	$List[\mathbb{X}]$, $\mathbb{X}$		$\mathbb{R}$
Minus	$List[\mathbb{X}]$, $\mathbb{X}$		$List[\mathbb{X}]$

Note that $\mathbb{X}$ stands for the option of being $\mathbb{F}$ or $\mathbb{A}$. NearestNeighborRegressor takes a query and a target, automatically creates a buffer of the target (thus keeps a list as a state) and answers based on the buffer. RunningNorm keeps track of the variance of the input and normalizes by that variance.

2) Reward combiner operations:

Operation	Input type(s)	State	Output type
Constant $\{0.01,0.1,0.5,1\}$			$\mathbb{R}$
NormalDistribution			$\mathbb{R}$
Add	$\mathbb{R}$, $\mathbb{R}$		$\mathbb{R}$
Max	$\mathbb{R}$, $\mathbb{R}$		$\mathbb{R}$
Min	$\mathbb{R}$, $\mathbb{R}$		$\mathbb{R}$
WeightedNormalizedSum	$\mathbb{R}$, $\mathbb{R}$, $\mathbb{R}$, $\mathbb{R}$		$\mathbb{R}$
RunningNorm	$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$
VariableAsBuffer	$\mathbb{R}$	$List[\mathbb{R}]$	$List[\mathbb{R}]$
Subtract	$\mathbb{R}$, $\mathbb{R}$		$\mathbb{R}$
Multiply	$\mathbb{R}$, $\mathbb{R}$		$\mathbb{R}$
Sqrt(Abs(x))	$\mathbb{R}$		$\mathbb{R}^+$
Mean	$List[\mathbb{R}]$		$\mathbb{R}$

Note that $WeightedNormalizedSum(a,b,c,d) = \frac{ab+cd}{|a|+|c|}$. RunningNorm keeps track of the variance of the input and normalizes by that variance.

B. Two other published algorithms covered by our DSL

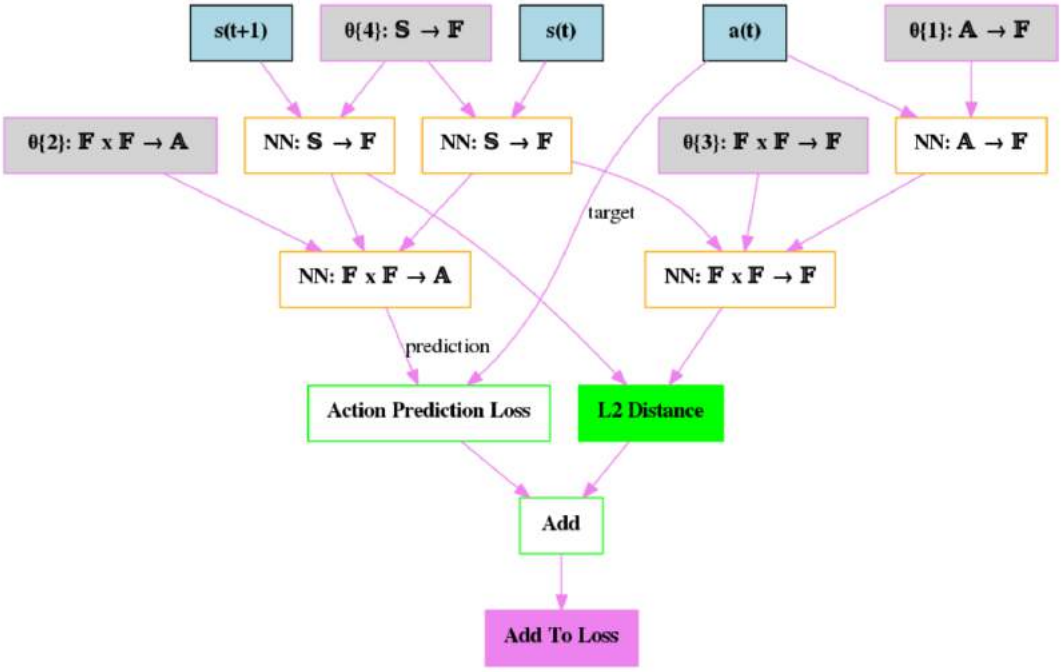

Fig. 5. Curiosity by predictive error on inverse features by [3]. In pink, paths and networks where gradients flow back from the minimizer.

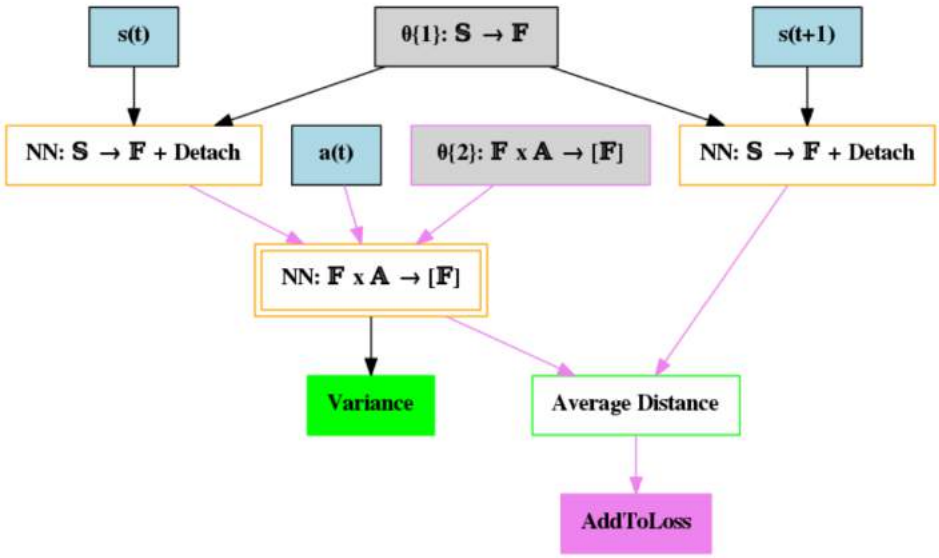

Fig. 6. Self-supervised exploration via disagreement [11]. In pink, paths and networks where gradients flow back from the minimizer.

C. Improving the efficiency of our search

We wish to find curiosity programs that work effectively in a wide range of environments, from simple to complex. However, evaluating tens of thousands of programs in the most expensive environments would consume decades of GPU computation. Therefore, we designed multiple strategies for quickly discarding less promising programs and focusing computation on a few promising programs. In doing so, we take inspiration from efforts in the AutoML community [16].

We divide these pruning efforts into three categories: simple tests that are independent of running the program in any environment, "filtering" by ruling out some programs based on poor performance in simple environments, and "meta-meta-RL": learning to predict which curiosity programs will produce good RL agents based on syntactic features.

1) Pruning invalid algorithms without running them: Many programs are obviously bad curiosity programs. We have developed two heuristics to immediately prune these programs without an expensive evaluation.

- Checking that programs are not duplicates. Since our language is highly expressive, there are many non-obvious ways of getting equivalent programs. To find duplicates, we designed a randomized test where we identically seed two programs, feed them both identical fake environment data for tens of steps and check whether their outputs are identical.
- Checking that the loss functions cannot be minimized independently of the input data. Many programs optimize some loss depending on neural network regressors. If we treat inputs as uncontrollable variables and networks as having the ability to become any possible function, then for every variable, we can determine whether neural networks can be optimized to minimize it, independently of the input data. For example, if our loss function is $|NN_\theta(s)|^2$ the neural network can learn to make it 0 by disregarding s and optimizing the weights θ to 0. We discard any program that has this property.

2) Pruning algorithms in cheap environments: Our ultimate goal is to find algorithms that perform well on many different environments, both simple and complex. We make two key observations. First, there may be only tens of reasonable programs that perform well on all environments but hundreds of thousands of programs that perform poorly. Second, there are some environments that are solvable in a few hundred steps while others require tens of millions. Therefore, a key idea in our search is to try many programs in cheap environments and only a few promising candidates in the most expensive environments. This was inspired by the effective use of sequential halving [17] in hyper-parameter optimization [18].

By pruning programs aggressively, we may be losing multiple programs that perform well on complex environments. However, by definition, these programs will tend to be less general and robust than those that succeed in all environments. Moreover, we seek generalization not only for its own sake, but also to ease the search since, even if we only cared about the most expensive environment, performing the complete search only in this environment would be impractical.

3) Predicting algorithm performance: Perhaps surprisingly, we find that we can predict program performance directly from program structure. Our search process bootstraps an initial training set of (program structure, program performance) pairs, then uses this training set to select the most promising next programs to evaluate. We encode each program's structure with features representing how many times each operation is used, thus having as many features as number of operations in our vocabulary. We use a k-nearest-neighbor regressor, with $k = 10$. We then try the most promising programs and update the regressor with their results. Finally, we add an ε-greedy exploration policy to make sure we explore all the search space. Even though the correlation between predictions and actual values is only moderately high (0.54 on a holdout test), this is enough to discover most of the top programs searching only half of the program space, which is our ultimate goal. Results are shown in figures 7, 8.

Fig. 7. Predicting algorithm performance from the structure of the program alone. Comparison between predicted and actual performance on a test set; showing a correlation of 0.54. In black, the identity line.

We can also prune algorithms during the training process of the RL agent. In particular, at any point during the meta-search, we use the top K current best programs as benchmarks for all T time-steps. Then, during the training of a new candidate program we compare its current performance at time t with the performance at time t of the top K programs and stop the run if its performance is significantly lower. If the program is not pruned and reaches the final time-step T with one of the top K performances, it becomes part of the benchmark for the future programs.

D. Experiments

Our RL agent uses PPO [19] based on the implementation by [20] in PyTorch [21].

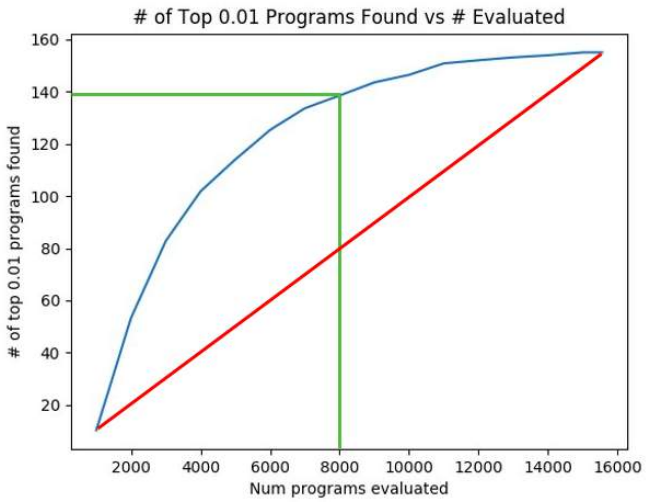

Fig. 8. Predicting algorithm performance allows us to find the best programs faster. We investigate the number of the top 1% of programs found vs. the number of programs evaluated, and observe that the optimized search (in blue) finds 88% of the best programs after only evaluating 50% of the programs (highlighted in green). The naive search order would have only found 50% of the best programs at that point.

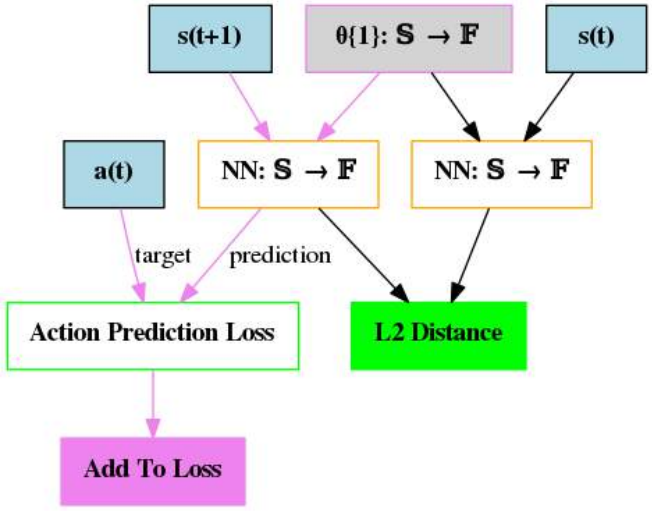

Fig. 9. Fast Action-Space Transition(FAST): top-performing intrinsic curiosity algorithm discovered in our phase 1 search.

Our code (https://github.com/mfranzs/meta-learning-curiosity-algorithms) can take in any OpenAI gym environment [14] with a specification of the desired exploration horizon T.

We evaluate each curiosity algorithm for multiple trials, using a seed dependent on the trial but independent of the algorithm, which leads to the PPO weights and curiosity data-structures being initialized identically on the same trials for all algorithms. As is common in PPO, we run multiple rollouts (5, except for MuJoCo which only has 1), with independent experiences but shared policy and curiosity modules. Curiosity predictions and updates are batched across these rollouts, but not across time. PPO policy updates are batched both across rollouts and multiple timesteps.

1) First search phase in simple environment: We start by searching for a good intrinsic curiosity program I in a purely exploratory environment, designed by [22], which is an image-based grid world where agents navigate in an image of a 2D room either by moving forward in the grid or rotating left or right. We optimize the total number of distinct cells visited across the agent's lifetime. This allows us to evaluate intrinsic reward programs in a fast and simple environment, without worrying about combining it with external reward.

To bias towards simple, interpretable algorithms and keep the search space manageable, we search for programs with at most 7 operations. We first discard duplicate and invalid programs, as described in section C.1, resulting in about 52,000 programs. We then randomly split the programs across 4 machines, each with 8 Nvidia Tesla K80 GPUs for 10 hours; thus a total of 13 GPU days.

Each machine finds the highest-scoring 625 programs in its section of the search space and prunes programs whose partial learning curve is statistically significantly lower than the current top 625 programs. To do so, after every episode of every trial, we check whether the mean performance of the current program is below the mean performance (at that point during the trial) of the top 625 programs minus two standard deviations of their performance minus one deviation of our estimate of the mean of the current program. Thus, we account for both inter-program variability among the top 625 programs and intra-program variability among multiple trials of the same program.

We use a 10-nearest-neighbor regressor to predict program performance and choose the next program to evaluate with an ε-greedy strategy, choosing the best predicted program 90% of the time and a random program 10% of the time. By doing this, we try the most promising programs early in our search. This is important for two reasons: first, we only try 26,000 programs, half of the whole search space, which we estimated from earlier results (shown in figure 8) would be enough to get 88% of the top 1% of programs. Second, the earlier we run our best programs, the higher the bar for later programs, thus allowing us to prune them earlier, further saving computation time. Searching through this space took a total of 13 GPU days. As shown in figure 10, we find that most programs perform relatively poorly, with a long tail of programs that are statistically significantly better, comprising roughly 0.5% of the whole program space.

The highest scoring program (a few other programs have lower average performance but are statistically equivalent) is surprisingly simple and meaningful, comprised of only 5 operations, even though the limit was 7. This program, which we call FAST (Fast Action-Space Transition), is shown in figure 9; it uses a single neural network (a CNN or MLP depending on the type of state) to predict the action from s_{t+1} and then compares its predictions based on s_t with its predictions based on s_{t+1}, generating high intrinsic reward when the difference is large. The *action prediction loss* module either computes a softmax followed by NLL loss or

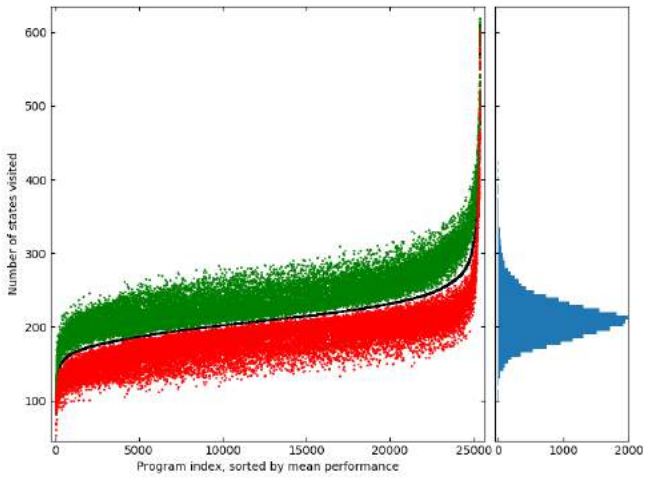

Fig. 10. In black, mean performance across 5 trials for all 26,000 programs evaluated (out of their finished trials). In green mean plus one standard deviation for the mean estimate and in red one minus one standard deviation for the mean estimate. On the right, you can see program means form roughly a gaussian distribution of very big noise (thus probably not significant) with a very small (between 0.5% and 1% of programs) long tail of programs with statistically significant performance (their red dots are much higher than almost all green dots), composed of algorithms leading to good exploration.

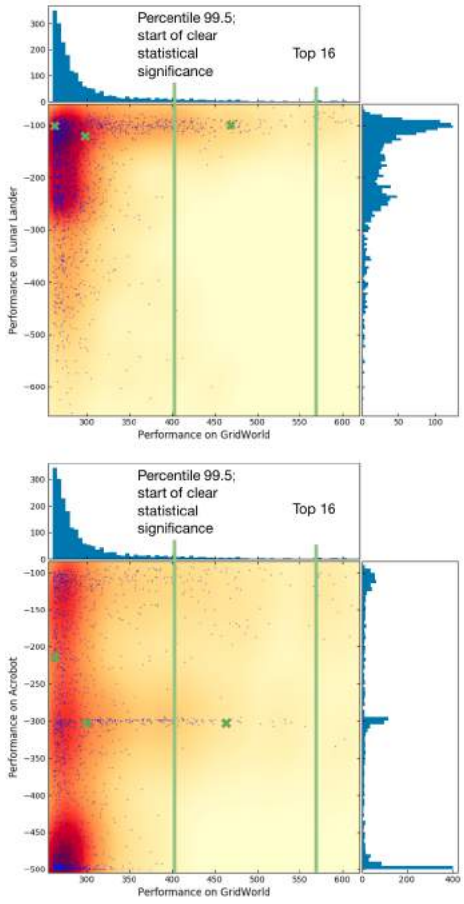

Fig. 11. Correlation between program performance in gridworld and in harder environments (lunar lander on top, acrobot on the bottom), using the top 2,000 programs in gridworld. Performance is evaluated using mean reward across *all* learning episodes, averaged over trials (two trials for acrobot / lunar lander and five for gridworld). The high number of algorithms performing around -300 in the middle of the bottom plot is an artifact of averaging the performance of two seeds and the mean performance in Acrobot having two peaks. Almost all intrinsic curiosity programs that had statistically significant performance for grid world also do well on the other two environments. In green, the performance of three published works; in increasing gridworld performance: disagreement [11], inverse features [3] and random distillation [4].

appends zeros to the action to match dimensions and applies MSE loss, depending on the type of the action space. Some versions of FAST predict the action from s_t instead of s_{t+1}, so this seems to not be a critical decision. Note that this is not the same as rewarding taking a different action in the previous time-step. The network predicting the action is learning to imitate the policy learned by the internal RL agent, because the curiosity module does not have direct access to the RL agent's internal state.

Of the top 16 programs, 13 are variants of FAST, including versions that predict the action from s_t instead of s_{t+1}. The other 3 are variants of a pretty complex program that is hard to understand at first glance, combining 3 neural networks, 2 different prediction losses and leveraging weight sharing in a clever way; the diagram and explanation is in figure 4. Interestingly, to the best of our knowledge neither algorithm had been proposed before: we conjecture the former was too simple for humans to believe it would be effective and the latter too hard for humans to design, as it was already very hard to understand in hindsight.

2) Transferring to new environments: Our reward combiner was developed in *lunar lander* (the simplest environment with meaningful extrinsic reward) based on the best program among a preliminary set of 16,000 programs (which resembled Random Network Distillation, its computation graph is shown in appendix E). Among a set of 2,500 candidates (with 5 or less operations) the best reward combiner discovered by our search was $\widehat{r_t} = \frac{(1+i_t-t/T)\cdot i_t+t/T\cdot r_t}{1+i_t}$. When $t=0$, this evaluates to i_t. When $t=T$, this evaluates to $\frac{i^2+r}{1+i}$, which is r when $0 \leq i << 1$. Thus, the combiner roughly interpolates from purely intrinsic reward to purely extrinsic reward, if the intrinsic reward has started to drop down to 0 by the end. In future work, we hope to co-adapt the search for intrinsic reward programs and combiners as well as find multiple reward combiners.

Given the fixed reward combiner and the list of 2,000 selected programs found in the image-based grid world, we evaluate the programs on both *lunar lander* and *acrobot*, in their discrete action space versions. Notice that both environments have much longer horizons than the image-based grid world (37,500 and 50,000 vs 2,500) and they have vector-based inputs, not image-based. The results in figure 11 show good correlation between performance on grid world and on each of the new environments. Especially interesting is that, for both environments, when intrinsic reward in grid world is above 400 (the start of the statistically significant performances), performance on the other two environments is also good in more than 90% of cases.

Class	Ant	Hopper
Baseline algorithms	[-95.3, -39.9]	[318.5, 525.0]
Meta-learned algorithms	[+67.5, +80.0]	[589.2, 650.6]
Published algorithms	[+67.4, +98.8]	[627.7, 692.6]

TABLE I

META-LEARNED ALGORITHMS PERFORM SIGNIFICANTLY BETTER THAN CONSTANT REWARDS AND STATISTICALLY EQUIVALENTLY TO PUBLISHED ALGORITHMS FOUND BY HUMAN RESEARCHERS (SEE 2). THE TABLE SHOWS THE CONFIDENCE INTERVAL (ONE STANDARD DEVIATION) FOR THE MEAN PERFORMANCE (ACROSS TRIALS, ACROSS ALGORITHMS) FOR EACH ALGORITHM CATEGORY. PERFORMANCE IS DEFINED AS MEAN EPISODE REWARD FOR ALL EPISODES.

Finally, we evaluate on two MuJoCo environments [23]: *hopper* and *ant*. These environments have more than an order of magnitude longer exploration horizon than acrobot and lunar lander, exploring for 500K time-steps, as well as continuous action-spaces instead of discrete. We then compare the best 16 programs on grid world (most of which also did well on lunar lander and acrobot) to four weak baselines (constant 0,-1,1 intrinsic reward and Gaussian noise reward) and the three published algorithms expressible in our language (shown in figure 2). We run two trials for each algorithm and pool all results in each category to get a confidence interval for the mean of that category. All trials used the reward combiner found on lunar lander. For both environments we find that the performance of our top programs is statistically equivalent to published work and significantly better than the weak baselines, confirming that we meta-learned good curiosity programs.

Note that we meta-trained our intrinsic curiosity programs only on one environment (GridWorld) and showed they generalized well to other very different environments: they perform better than published works in this meta-train task and one meta-test task (Acrobot) and on par in the other 3 tasks meta-test tasks. Adding more meta-training tasks would be as simple as standardising the performance within each task (to make results comparable) and then selecting the programs with best mean performance. We chose to only meta-train on a single, simple, task because it (surprisingly!) already gave great results; highlighting the broad generalization of meta-learning program representations.

E. Algorithm used to optimize reward combiner

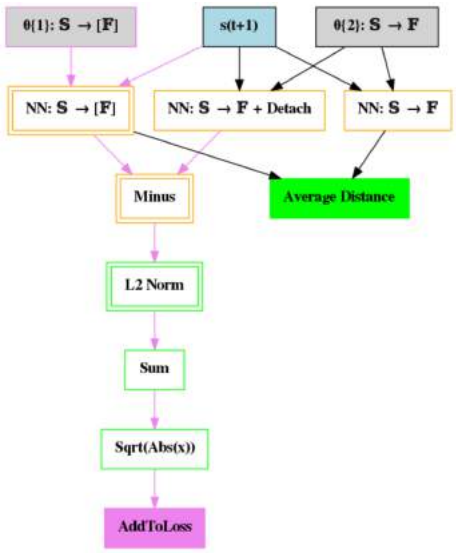

Fig. 12. Top variant in preliminary search on grid world; variant on random network distillation using an ensemble of trained networks instead of a single one.

F. Related work

There has been much interesting work in designing intrinsic curiosity algorithms. We take inspiration from many of them to design our domain-specific language. In particular, we rely on the idea of using neural network training as an implicit memory, which scales well to millions of time-steps, as well as buffers and nearest-neighbour regressors. As we showed in figure 2 we can represent several prominent curiosity algorithms. We can also generate meaningful algorithms similar to novelty search [12] and EX^2 [13]; which include buffers and nearest neighbours. However, there are many exploration algorithm classes that we do not cover, such as those focusing on generating goals [24], [25], [26], learning progress [27], [28], [29], generating diverse skills [30], stochastic neural networks [31], [32], count-based exploration [33] or object-based curiosity measures [34]. Finally, part of our motivation stems from [35] showing that some bonus-based curiosity algorithms have trouble generalising to new environments.

There have been research efforts on meta-learning exploration policies: [8], [9] learn an LSTM that explores an environment for one episode, retains its hidden state and is spawned in a second episode in the same environment; by training the network to maximize the reward in the second episode alone it learns to explore efficiently in the first episode. [36] improves their exploration and that of [6] by considering the importance of sampling in RL policies. [37] combine gradient-based meta-learning with a learned latent exploration space in which they add structured noise for meaningful exploration. Closer to our formulation, [38] parametrize an intrinsic reward function which influences

policy-gradient updates in a differentiable manner, allowing them to backpropagate through a single step of the policy-gradient update to optimize the intrinsic reward function for a single task. In contrast to all three of these methods, we search over algorithms, which will allows us to generalize more broadly and to consider the effect of exploration on up to $10^5 - 10^6$ time-steps instead of the $10^2 - 10^3$ of previous work. Finally, [39], [40] have a setting similar to ours where they modify reward functions over the entire agent's lifetime, but instead of searching over intrinsic curiosity algorithms they tune the parameters of a hand-designed reward function.

In some regards our work is similar to neural architecture search (NAS) [41], [42], [43], [44] or hyperparameter optimization for deep networks [45], which aim at finding the best neural network architecture and hyper-parameters for a particular task. However, in contrast to most (but not all, see [46]) NAS work, we want to generalize to many environments instead of just one. Moreover, we search over programs, which include non-neural operations and data structures, rather than just neural-network architectures, and decide what loss functions to use for training. Our work also resembles work in the AutoML community [16] that searches in a space of programs, for example in the case of SAT solving [47] or auto-sklearn [48] and concurrent work on learning loss functions to replace cross-entropy for training a fixed architecture on MNIST and CIFAR [49], [50]. Although we take inspiration from ideas in that community [18], [51], our algorithms specify both how to compute their outputs and their own optimization objectives in order to work well in synchrony with an expensive deep RL algorithm.

There has been work on meta-learning with genetic programming [52], searching over mathematical operations within neural networks [53], [54], searching over programs to solve games [55], [56], [57] and to optimize neural networks [58], [59], and neural networks that learn programs [60], [61]. Our work uses neural networks as basic operations within larger algorithms. Finally, modular meta-learning [62], [63] trains the weights of small neural modules and transfers to new tasks by searching for a good composition of modules; as such, it can be seen as a (restricted) dual of our approach.

Meta-learning [52], [64], [65] aims at learning transferable representations from many tasks in order to learn a new task more efficiently. Most work on meta-RL has focused on learning transferable feature representations or parameter values for quickly adapting to new tasks [6], [66], [7] or improving performance on a single task [67], [68]. However, the range of variability between tasks is typically limited to variations of the same goal (such as moving at different speeds or to different locations) or generalizing to different environment variations (such as different mazes or different terrain slopes). There have been some attempts to broaden the spectrum of generalization, showing transfer between Atari games thanks to modularity [69], [70] or proper pre-training [71]. However, as noted by [72], Atari games are too different to get big gains with current feature-transfer methods; they instead suggest using different levels of the game *Sonic* to benchmark generalization. Moreover, [73] recently proposed a benchmark of many tasks. [74] automatically generate different terrains for a bipedal walker and transfer policies between terrains, showing that this is more effective than learning a policy on hard terrains from scratch; similar to our suggestion in section C.2. In contrast to these methods, we aim at generalization between completely different environments, even between environments that do not share the same state and action spaces.

Closest to our work, evolved policy gradients (EPG, [75]) use evolutionary strategies to meta-learn a neural network that acts as a loss function and is used to train a policy network. EPG generalizes by meta-training with target locations east of the start location and meta-testing with target locations to the west. In contrast, we showed that by meta-learning programs, we can generalize between radically different environments, not just goal variations of a single environment. Concurrent to our work, [76] also show generalization capabilities between environments similar to ours (lunar lander, hopper and half-cheetah). Their approach transfers a parametric representation, for which it is unclear how to adapt the learned neural losses to an unseen environment with a different observation space. Their approach thus does not encode states into the loss function, which is critical for efficient exploration. In contrast, our algorithms can leverage polymorphic data types that adapt the neural networks to the environment they are running in, adapting both the size and the type of network (CNN vs MLP) running in each environment.