META-LEARNING CURIOSITY ALGORITHMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Exploration is a key component of successful reinforcement learning, but optimal approaches are computationally intractable, so researchers have focused on hand-designing mechanisms based on exploration bonuses and intrinsic reward, some inspired by curious behavior in natural systems. In this work, we propose a strategy for encoding curiosity algorithms as programs in a domain-specific language and searching, during a meta-learning phase, for algorithms that enable RL agents to perform well in new domains. Our rich language of programs, which can combine neural networks with other building blocks including nearest-neighbor modules and can choose its own loss functions, enables the expression of highly generalizable programs that perform well in domains as disparate as grid navigation with image input, acrobot, lunar lander, ant and hopper. To make this approach feasible, we develop several pruning techniques, including learning to predict a program's success based on its syntactic properties. We demonstrate the effectiveness of the approach empirically, finding curiosity strategies that are similar to those in published literature, as well as novel strategies that are competitive with them and generalize well.

1 INTRODUCTION

When an agent is learning to behave online, via reinforcement learning (RL), 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 (Ghavamzadeh et al., 2015). However, these techniques do not scale at all 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.

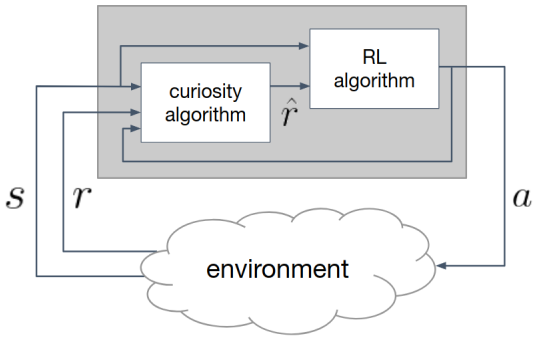

Figure 1: Our RL agent is augmented with a *curiosity module*, obtained by meta-learning over a complex space of programs, which computes a pseudo-reward $\widehat{r}$ at every time step.

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 (Pathak et al., 2017; Burda et al., 2018; Oudeyer, 2018). 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.

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 in order to expose it 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 the 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 $\sum_{k\geq 0} \gamma^k \widehat{r}_{t+k}$. The outer "evolutionary" search is attempting to find a program for the curiosity module, so 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.

Although it is, in principle, possible to discover a complete, integrated algorithm for the entire curious learning agent in the gray box, that is a much more complex search problem that is currently computationally infeasible. We are relying on the assumption that the foundational methods for reinforcement learning, including those based on temporal differencing and policy gradient, are fundamentally sound and can serve as the behavior-learning basis for our agents. It is important to note, though, that the internal RL algorithm in our architecture must be able to tolerate a non-stationary reward signal, which may necessitate minor algorithmic changes or, at least, different hyperparameter values.

In this meta-learning setting, our objective is to find a curiosity module that works well given a distribution of environments from which we can sample at meta-learning time. If the environment distribution is relatively low-variance (the tasks are all quite similar) then it might suffice to search over a relatively simple space of curiosity strategies (most trivially, the ϵ in an ϵ-greedy exploration strategy). 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 (Finn et al., 2017; Clavera et al., 2019) and, in others, for efficient exploration (Duan et al., 2016; Wang et al., 2017). 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.

This paper makes three novel contributions.

We focus on a regime of meta-reinforcement-learning in which the possible environments the agent might face are dramatically disparate and in which the agent's lifetime is very long. This is a substantially different setting than has been addressed in previous work on meta-RL and it requires substantially different techniques for representation and search.

We represent meta-learned curiosity strategies in a rich, combinatorial space of programs rather than in a fixed-dimensional numeric parameter space. The programs are represented in a *domain-specific language* (DSL) which includes sophisticated building blocks including neural networks complete with gradient-descent mechanisms, learned objective functions, ensembles, buffers, and other regressors. This language is rich enough to represent many previously reported hand-designed exploration algorithms. We believe that by performing meta-RL in such a rich space of mechanisms, we will be able to discover highly general, fundamental curiosity-based exploration methods. This generality means that a relatively computationally expensive meta-learning process can be amortized over the lifetimes of many agents in a wide variety of environments.

We make the search over programs feasible with relatively modest amounts of computation. It is a daunting search problem to find a good solution in a combinatorial space of programs, where evaluating a single potential solution requires running an RL algorithm for up to millions of time steps. We address this problem in multiple ways. By including environments of substantially different difficulty and character, we can evaluate candidate programs first on relatively simple and short-horizon domains: if they don't perform well in those domains, they are pruned early, which saves a significant amount of computation time. In addition, we predict the performance of an algorithm from its structure and operations, thus trying the most promising algorithms early in our search. Finally, we also monitor the learning curve of agents and stop unpromising programs before they reach all T environment steps.

We demonstrate the effectiveness of the approach empirically, finding curiosity strategies that are similar to those in published literature, as well as novel strategies that are competitive with them and generalize well.

2 PROBLEM FORMULATION

2.1 META-LEARNING PROBLEM

Let us assume we have an agent equipped with an RL algorithm (such as DQN or PPO, with all hyperparameters specified), $\mathcal{A}$, which receives states and rewards from and outputs actions to an environment $\mathcal{E}$, generating a stream of experienced transitions $e(\mathcal{A};\mathcal{E})_t = (s_t, a_t, r_t, s_{t+1})$. The agent continually learns a policy $\pi(t) : s_t \rightarrow a_t$, which will change in time as described by algorithm $\mathcal{A}$; so $\pi(t) = \mathcal{A}(e_{1:t-1})$ and thus $a_t \sim \mathcal{A}(e_{1:t-1})(s_t)$. Although this need not be the case, we can think of $\mathcal{A}$ 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* $\mathcal{C}$ 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 $e(\mathcal{A},\mathcal{C};\mathcal{E})_t = (s_t, a_t, \widehat{r}_t = \mathcal{C}(e_{1:t-1}), s_{t+1})$, 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 $\mathcal{C}$ 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 $\mathcal{E}$, RL algorithm $\mathcal{A}$, and curiosity module $\mathcal{C}$, 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 $\mathcal{C}$ 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 $\mathcal{E}$, and in "inner" expectation over the rewards received by the composite system in that environment, so our final objective is:

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

2.2 PROGRAMS FOR CURIOSITY

In science and computing, mathematical language has been very successful in describing varied phenomena and powerful algorithms with short descriptions. As Valiant points out: "the power [of mathematics and algorithms] comes from the implied generality, that knowledge of one equation alone will allow one to make accurate predictions about a host of situations not even conceived when the equation was first written down" (Valiant, 2013). 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+1}, a_t, r_t) triples 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.

We draw both programs from the same basic class. Fundamentally, they consist of a directed acyclic graph (DAG) of modules with polymorphically typed inputs and outputs. 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 module and from the set $\{i_t, r_t\}$ for the χ module. 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.

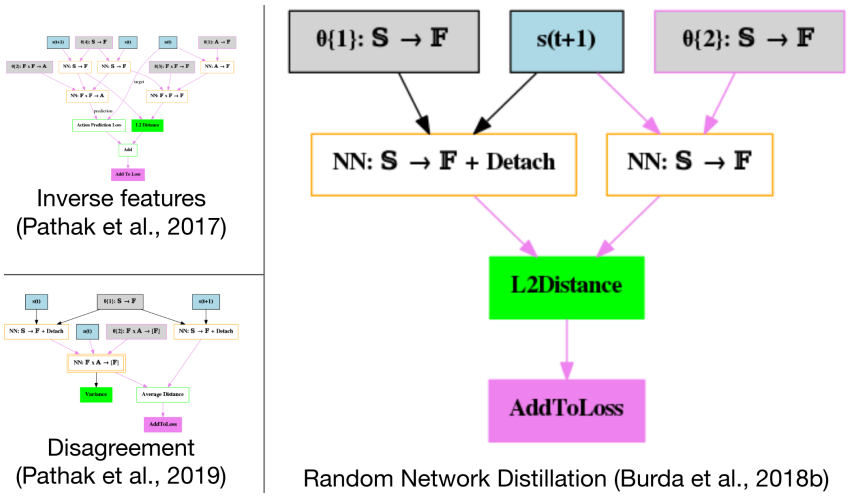

Figure 2: Example diagrams of published algorithms covered by our language (larger figures in the appendix). The green box represents the output of the intrinsic curiosity function, the pink box is the loss to be minimized. Pink arcs represent paths and networks along which gradients flow back from the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

- **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 input from their parent modules.
- **Update** modules (shown in pink), which are functional modules (such as k-Nearest-Neighbor) that either add variables to buffers or modules which add real-valued outputs to a global loss that will provide error signals for gradient descent.

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 saved, to be yielded as the output of the whole program. Before the call terminates, the FIFO buffers are updated and the adjustable parameters are updated via gradient descent using the Adam optimizer Kingma & Ba (2014). 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.

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 random network distillation (RND) (Burda et al., 2018)).

Programs representing several published designs for curiosity modules that perform internal gradient descent, including inverse features (Pathak et al., 2017), RND (Burda et al., 2018), and ensemble predictive variance (Pathak et al., 2019), are shown in figure 2 (and bigger versions can be found in appendix A.3). We can also represent algorithms similar to novelty search (Lehman & Stanley,

2008) and EX^2 (Fu et al., 2017), which include buffers and nearest neighbor regression modules. Details on the data types and module library are given in appendix A.

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 the 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. Similarly, if we are measuring an error in action space $\mathbb{A}$ we use mean-squared error for continuous action spaces and negative log-likelihood for discrete action spaces. 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.

To clarify the semantics of these programs, we walk through the operation of the RND program in figure 2. Its only input is s_{t+1}, which might be an image or an input vector, which is processed by two NNs with parameters Θ_1 and Θ_2, respectively. The structure of the NNs (and, hence, the dimensions of the Θ_i) depends on the type of s_{t+1}: if s_{t+1} is an image, then they are CNNs, otherwise a fully connected networks. Each NN outputs a 32-dimensional vector; the L_2 distance between these vectors is the output of the program on this iteration, and is also the input to a loss module. So, given an input s_{t+1}, the output intrinsic reward is large if the two NNs generate different outputs and small otherwise. After each forward pass, the weights in Θ_2 are updated to minimize the loss while Θ_1 remains constant, which causes the trainable NN to mimic the output of the randomly initialized NN. As the program's ability to predict the output of the randomized NN on an input improves, the intrinsic reward for visiting that state decreases, driving the agent to visit new states.

To limit the search space and prioritize short, meaningful programs we limit the total number of modules of the computation graph to 7. Our language is expressive enough to describe many (but far from all) curiosity mechanisms in the existing literature, as well as many other potential alternatives, but the expressiveness leads to a very large search space. Additionally, removing or adding a single operation can drastically change the behavior of a program, making the objective function non-smooth and, therefore, the space hard to search. In the next section we explore strategies for speeding up the search over tens of thousands of programs.

3 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 have designed multiple strategies for quickly discarding less promising programs and focusing more computation on a few promising programs. In doing so, we take inspiration from efforts in the AutoML community (Hutter et al., 2018).

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 programs will perform well based on syntactic features.

3.1 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. This test may, with low probability, prune a program that is not an exact duplicate, but since there is a very near neighbor under consideration, it is not very harmful to do so.

- 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.

3.2 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 (Karnin et al., 2013) in hyper-parameter optimization (Jamieson & Talwalkar, 2016).

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.

3.3 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 appendix C.

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.

4 Experiments

Our RL agent uses PPO (Schulman et al., 2017) based on the implementation by Kostrikov (2018) in PyTorch (Paszke et al., 2017). Our code, which can be found at `https://bit.ly/meta-learning-curiosity-algs`, is meant to take in any OpenAI gym environment (Brockman et al., 2016) 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.

4.1 FIRST SEARCH PHASE IN SIMPLE ENVIRONMENT

We start by searching for a good intrinsic curiosity program I in a purely exploratory environment, designed by Chevalier-Boisvert et al. (2018), which is an image-based grid world where agents navigate in an image of a 2D room either by moving forward in the pixel grid or rotating left or right. We optimize the total number of distinct pixels 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 3.1, 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.

Each machine tries to find 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 standard deviation of our estimate of the mean of the current program. In this way 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 in the appendix) 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 9 in the appendix, 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.

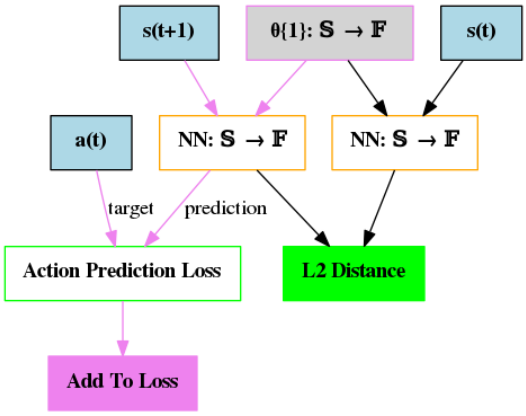

Figure 3: **Top** program in the large phase 1 search.

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 will call **Top**, is shown in figure 3; 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 appends zeros to the action to match dimensions and applies MSE loss, depending on the type of the action space. Note that this is not the same as rewarding taking a different action in the previous time-step. To the best of our knowledge, the algorithm represented by this program has not been proposed before, although its simplicity makes us think it may have. 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.

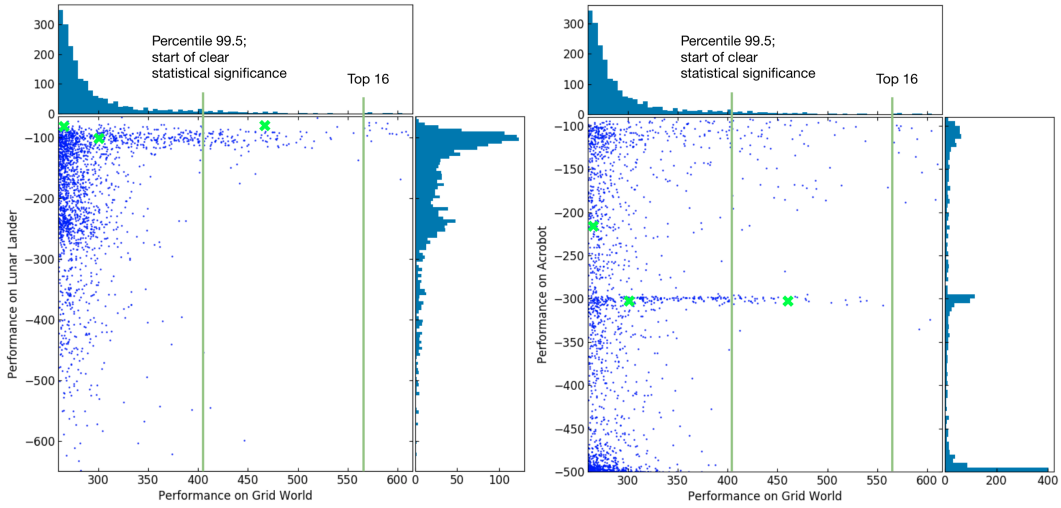

Figure 4: We show correlation between program performance in gridworld and performance in harder environments (lunar lander on the left, acrobot on the right), using the top 2,000 programs in gridworld. Performance is evaluated using mean reward across *all* learning episodes, averaged over trials (two trials for acrobot / lunar lander and five for gridworld).We can see that almost all intrinsic curiosity programs that had statistically significant performance for grid world also do well on the other two environments. In green, we show the performance of three published works; in increasing gridworld performance: disagreement (Pathak et al., 2019), inverse features (Pathak et al., 2017) and random distillation (Burda et al., 2018).

Many of the highest-scoring programs are small variations on **Top**, including versions that predict the action from s_t instead of s_{t+1}. Of the top 16 programs, 13 are variants of **Top** and 3 are variants of an interesting program that is more difficult to understand because it does a combination of random network distillation and state-transition prediction, with some weight sharing, shown in figure 11 in the appendix.

4.2 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 2478 candidates (with 5 or less operations) the best reward combiner was $\widehat{r}_t = \frac{(1+i_t-t/T)\cdot i_t+t/T\cdot r_t}{1+i_t}$. Notice that for $0 < i_t \ll 1$ (usually the case) this is approximately $\widehat{r}_t = i_t^2 + (1-t/T)i_t + (t/T)r_t$, which is a down-scaled version of intrinsic reward plus a linear interpolation that ranges from all intrinsic reward at $t = 0$ to all extrinsic reward at $t = T$. 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 4 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 370 (the start of the statistically significant performances), performance on the other two environments is also good in more than 90% of cases.

Finally, we evaluate the 16 best programs on grid world (most of which also did well on lunar lander and acrobot) on two MuJoCo environments (Todorov et al., 2012): *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.

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 1: 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.

We then compare the best 16 programs on grid world 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. Adding more more meta-training tasks would be as simple as standardising the performance within each task (to make results comparable) and then select 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.

5 Related work

In some regards our work is similar to neural architecture search (NAS) (Stanley & Miikkulainen, 2002; Elsken et al., 2018; Pham et al., 2018; Zoph & Le, 2016) or hyperparameter optimization for deep networks (Mendoza et al., 2016), 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 Zoph et al. (2018)) 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 (Hutter et al., 2018) that searches in a space of programs, for example in the case of SAT solving (KhudaBukhsh et al., 2009) or auto-sklearn (Feurer et al., 2015). Although we take inspiration from ideas in that community (Jamieson & Talwalkar, 2016; Li et al., 2016), our algorithms also specify their own optimization objectives (vs being specified by the user) which need to work well in syncrony with an expensive deep RL algorithm.

There has been work on meta-learning with genetic programming (Schmidhuber, 1987), searching over mathematical operations within neural networks (Ramachandran et al., 2017; Gaier & Ha, 2019), searching over programs to solve games (Wilson et al., 2018; Kelly & Heywood, 2017; Silver et al., 2019) and to optimize neural network weights (Bengio et al., 1995; Bello et al., 2017), and neural networks that learn programs (Reed & De Freitas, 2015; Pierrot et al., 2019). In contrast, our work uses neural networks as basic operations within larger algorithms. Finally, modular meta-learning (Alet et al., 2018) trains the weights of small neural modules and transfers to new tasks by searching for a good composition of modules using a relatively simple composition scheme; as such, it can be seen as a (restricted) dual of our approach.

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 (Lehman & Stanley, 2008) and EX^2 (Fu et al., 2017); 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 (Srivastava et al., 2013; Kulkarni et al., 2016; Florensa et al., 2018), learning progress (Oudeyer et al., 2007; Schmidhuber, 2008; Azar et al.,

2019), generating diverse skills (Eysenbach et al., 2018), stochastic neural networks (Florensa et al., 2017; Fortunato et al., 2017), count-based exploration (Tang et al., 2017) or object-based curiosity measures (Forestier & Oudeyer, 2016). Finally, part of our motivation stems from Taïga et al. (2019) showing that some bonus-based curiosity algorithms have trouble generalising to new environments.

Related work on parametric-based meta-RL and efforts to increase its generalization can be found in appendix B. More relevant to our work, there have been research efforts on meta-learning exploration policies. Duan et al. (2016); Wang et al. (2017) 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. Stadie et al. (2018) improves their exploration and that of Finn et al. (2017) by considering the importance of sampling in RL policies. Gupta et al. (2018) combine gradient-based meta-learning with a learned latent exploration space in which they add structured noise for meaningful exploration. Closer to our formulation, Zheng et al. (2018) 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, Chiang et al. (2019); Faust et al. (2019) 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.

Probably closest to our work, in evolved policy gradients Houthooft et al. (2018) meta-learn a neural network that computes a loss function based on interactions of the agent with an environment. The weights of this network are optimized via evolution strategies to efficiently optimize new policies from scratch to satisfy new goals. They show that they can generalize more broadly than MAML and RL^2 by meta-training a robot to go to different positions to the east of the start location and then meta-test making the robot quickly learn to go to a location 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. For all methods transferring parametric representations, it is not clear how one would adapt the learned neural networks to an unseen environment with different action dimensionality or a different observation space. In contrast, algorithms leverage polymorphic data types that adapt the neural networks to the environment they are running in.

6 Conclusions

In this work we show that programs are a powerful, succinct, representation for algorithms for generating curious exploration, and these programs can be meta-learned efficiently via active search. Results from this work are two-fold. First, by construction, algorithms resulting from this search will have broad generalization and will thus be a useful default for RL settings, where reliability is key. Second, the algorithm search code will be open-sourced to facilitate further research on exploration algorithms based on new ideas or building blocks, which can be added to the search. In addition, we note that the approach of meta-learning programs instead of network weights may have further applications beyond finding curiosity algorithms, such as meta-learning optimization algorithms or even meta-learning meta-learning algorithms.

REFERENCES

Ferran Alet, Tomas Lozano-Perez, and Leslie P. Kaelbling. Modular meta-learning. In *Proceedings of The 2nd Conference on Robot Learning*, pp. 856–868, 2018.

Mohammad Gheshlaghi Azar, Bilal Piot, Bernardo Avila Pires, Jean-Bastian Grill, Florent Altché, and Rémi Munos. World discovery models. *arXiv preprint arXiv:1902.07685*, 2019.

Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V Le. Neural optimizer search with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 459–468. JMLR. org, 2017.

Samy Bengio, Yoshua Bengio, and Jocelyn Cloutier. On the search for new learning rules for anns. *Neural Processing Letters*, 2(4):26–30, 1995.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. *arXiv preprint arXiv:1810.12894*, 2018.

Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. Minimalistic gridworld environment for openai gym. `https://github.com/maximecb/gym-minigrid`, 2018.

Hao-Tien Lewis Chiang, Aleksandra Faust, Marek Fiser, and Anthony Francis. Learning navigation behaviors end-to-end with autorl. *IEEE Robotics and Automation Letters*, 4(2):2007–2014, 2019.

Ignasi Clavera, Anusha Nagabandi, Ronald S Fearing, Pieter Abbeel, Sergey Levine, and Chelsea Finn. Learning to adapt: Meta-learning for model-based control. In *International Conference on Learning Representations*, 2019.

Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018.

Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is all you need: Learning skills without a reward function. *arXiv preprint arXiv:1802.06070*, 2018.

Aleksandra Faust, Anthony Francis, and Dar Mehta. Evolving rewards to automate reinforcement learning. *arXiv preprint arXiv:1905.07628*, 2019.

Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A Rusu, Alexander Pritzel, and Daan Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 28*, pp. 2962–2970. Curran Associates, Inc., 2015. URL `http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf`.

Chelsea Finn. *Learning to Learn with Gradients*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2018. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-105.html`.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *arXiv preprint arXiv:1703.03400*, 2017.

Carlos Florensa, Yan Duan, and Pieter Abbeel. Stochastic neural networks for hierarchical reinforcement learning. *arXiv preprint arXiv:1704.03012*, 2017.

Carlos Florensa, David Held, Xinyang Geng, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1515–1528, Stockholmsmssan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL http://proceedings.mlr.press/v80/florensa18a.html.

Sébastien Forestier and Pierre-Yves Oudeyer. Modular active curiosity-driven discovery of tool use. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3965–3972. IEEE, 2016.

Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

Justin Fu, John Co-Reyes, and Sergey Levine. Ex2: Exploration with exemplar models for deep reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 2577–2587, 2017.

Adam Gaier and David Ha. Weight agnostic neural networks. *arXiv preprint arXiv:1906.04358*, 2019.

Mohammad Ghavamzadeh, Shie Mannor, Joelle Pineau, and Aviv Tamar. Bayesian reinforcement learning: A survey. *Foundations and Trends in Machine Learning*, 8(5–6), 2015.

Abhishek Gupta, Russell Mendonca, YuXuan Liu, Pieter Abbeel, and Sergey Levine. Meta-reinforcement learning of structured exploration strategies. In *Advances in Neural Information Processing Systems*, pp. 5302–5311, 2018.

Rein Houthooft, Yuhua Chen, Phillip Isola, Bradly Stadie, Filip Wolski, OpenAI Jonathan Ho, and Pieter Abbeel. Evolved policy gradients. In *Advances in Neural Information Processing Systems*, pp. 5400–5409, 2018.

Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (eds.). *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2018. In press, available at http://automl.org/book.

Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pp. 240–248, 2016.

Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*, pp. 1238–1246, 2013.

Stephen Kelly and Malcolm I Heywood. Multi-task learning in atari video games with emergent tangled program graphs. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 195–202. ACM, 2017.

Ashiqur R KhudaBukhsh, Lin Xu, Holger H Hoos, and Kevin Leyton-Brown. Satenstein: Automatically building local search sat solvers from components. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

Ilya Kostrikov. Pytorch implementations of reinforcement learning algorithms. https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail, 2018.

Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in neural information processing systems*, pp. 3675–3683, 2016.

Joel Lehman and Kenneth O Stanley. Exploiting open-endedness to solve problems through the search for novelty. In *ALIFE*, pp. 329–336, 2008.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016.

Hector Mendoza, Aaron Klein, Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Towards automatically-tuned neural networks. In *Workshop on Automatic Machine Learning*, pp. 58–65, 2016.

Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman. Gotta learn fast: A new benchmark for generalization in rl. *arXiv preprint arXiv:1804.03720*, 2018.

Pierre-Yves Oudeyer. Computational theories of curiosity-driven learning. *arXiv preprint arXiv:1802.10546*, 2018.

Pierre-Yves Oudeyer, Frdric Kaplan, and Verena V Hafner. Intrinsic motivation systems for autonomous mental development. *IEEE transactions on evolutionary computation*, 11(2):265–286, 2007.

Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*, 2015.

Adam Paszke, Sam Gross, and Adam Lerer. Automatic differentiation in PyTorch. In *International Conference on Learning Representations*, 2017.

Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 16–17, 2017.

Deepak Pathak, Dhiraj Gandhi, and Abhinav Gupta. Self-supervised exploration via disagreement. *arXiv preprint arXiv:1906.04161*, 2019.

Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

Thomas Pierrot, Guillaume Ligner, Scott 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. *arXiv preprint arXiv:1905.12941*, 2019.

Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

Jürgen Schmidhuber. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook*. PhD thesis, Technische Universität München, 1987.

Jürgen 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*, pp. 48–76. Springer, 2008.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Tom Silver, Kelsey R Allen, Alex K Lew, Leslie Pack Kaelbling, and Josh Tenenbaum. Few-shot bayesian imitation learning with logic over programs. *arXiv preprint arXiv:1904.06317*, 2019.

Rupesh Kumar Srivastava, Bas R Steunebrink, and Jürgen Schmidhuber. First experiments with powerplay. *Neural Networks*, 41:130–136, 2013.

Bradly C Stadie, Ge Yang, Rein Houthooft, Xi Chen, Yan Duan, Yuhuai Wu, Pieter Abbeel, and Ilya Sutskever. Some considerations on learning to explore via meta-reinforcement learning. *arXiv preprint arXiv:1803.01118*, 2018.

Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

Adrien Ali Taïga, William Fedus, Marlos C Machado, Aaron Courville, and Marc G Bellemare. Benchmarking bonus-based exploration methods on the arcade learning environment. *arXiv preprint arXiv:1908.02388*, 2019.

Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, OpenAI Xi Chen, Yan Duan, John Schulman, Filip DeTurck, and Pieter Abbeel. # exploration: A study of count-based exploration for deep reinforcement learning. In *Advances in neural information processing systems*, pp. 2753–2762, 2017.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033. IEEE, 2012.

Leslie Valiant. *Probably Approximately Correct: NatureÕs Algorithms for Learning and Prospering in a Complex World*. Basic Books (AZ), 2013.

Vivek Veeriah, Matteo Hessel, Zhongwen Xu, Richard Lewis, Janarthanan Rajendran, Junhyuk Oh, Hado van Hasselt, David Silver, and Satinder Singh. Discovery of useful questions as auxiliary tasks. *arXiv preprint arXiv:1909.04607*, 2019.

JX Wang, Z Kurth-Nelson, D Tirumala, H Soyer, JZ Leibo, R Munos, C Blundell, D Kumaran, and M Botivnick. Learning to reinforcement learn. arxiv 1611.05763, 2017.

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O Stanley. Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv preprint arXiv:1901.01753*, 2019.

Dennis G Wilson, Sylvain Cussat-Blanc, Hervé Luga, and Julian F Miller. Evolving simple programs for playing atari games. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 229–236. ACM, 2018.

Zhongwen Xu, Hado P van Hasselt, and David Silver. Meta-gradient reinforcement learning. In *Advances in neural information processing systems*, pp. 2396–2407, 2018.

Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Sergey Levine, and Chelsea Finn. Meta-world: A benchmark and evaluation for multi-task and meta-reinforcement learning, 2019. URL `https://github.com/rlworkgroup/metaworld`.

Zeyu Zheng, Junhyuk Oh, and Satinder Singh. On learning intrinsic rewards for policy gradient methods. In *Advances in Neural Information Processing Systems*, pp. 4644–4654, 2018.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.

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.

A.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.

A.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.

A.3 TWO OTHER PUBLISHED ALGORITHMS COVERED BY OUR DSL

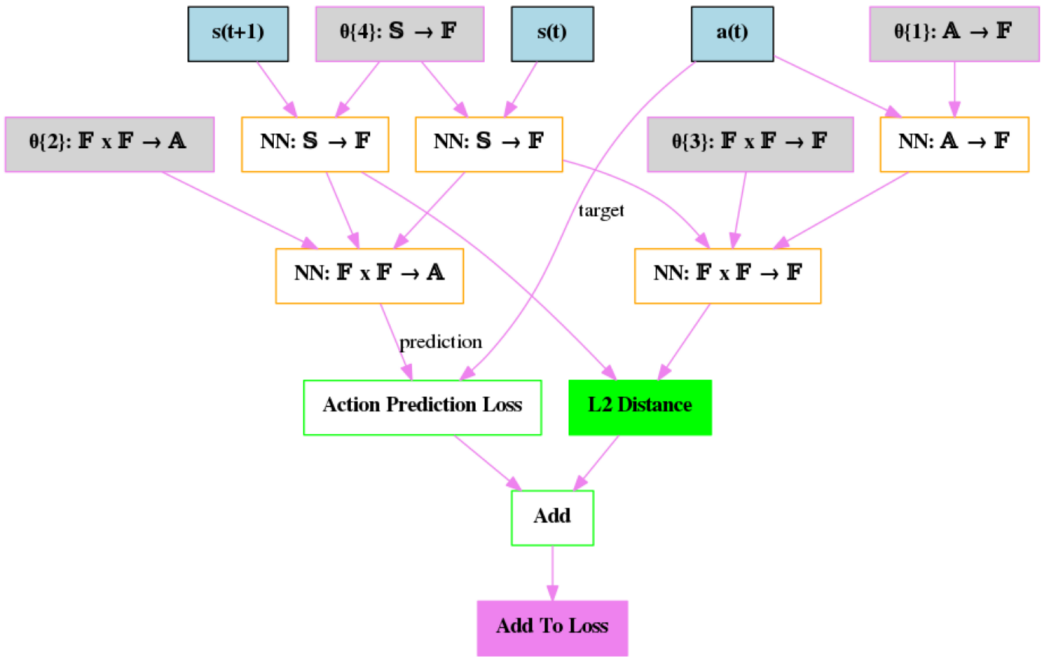

Figure 5: Curiosity by predictive error on inverse features by Pathak et al. (2017). In pink, paths and networks where gradients flow back from the minimizer.

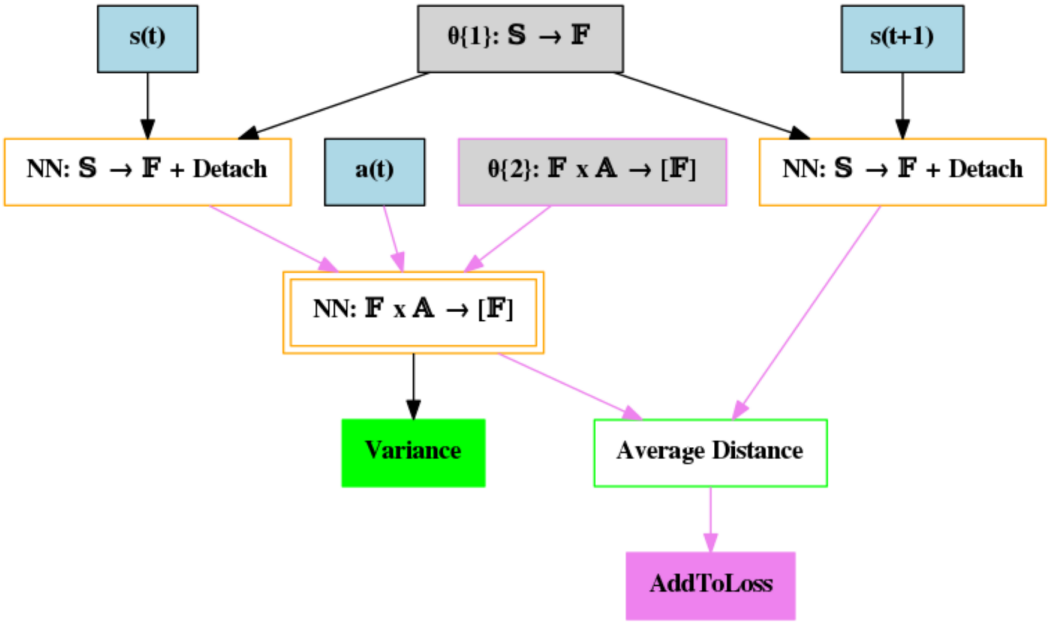

Figure 6: Curiosity by ensemble predictive variance Pathak et al. (2019). In pink, paths and networks where gradients flow back from the minimizer.

B RELATED WORK ON META-RL AND GENERALIZATION

Most work on meta-RL has focused on learning transferable feature representations or parameter values for quickly adapting to new tasks (Finn et al., 2017; Finn, 2018; Clavera et al., 2019) or improving performance on a single task (Xu et al., 2018; Veeriah et al., 2019). 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 (Fernando et al., 2017; Rusu et al., 2016) or proper pretraining (Parisotto et al., 2015). However, as noted by Nichol et al. (2018), 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, Yu et al. (2019) recently proposed a benchmark of many tasks. Wang et al. (2019) 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 3.2. 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.

C PREDICTING ALGORITHM PERFORMANCE

Figure 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.

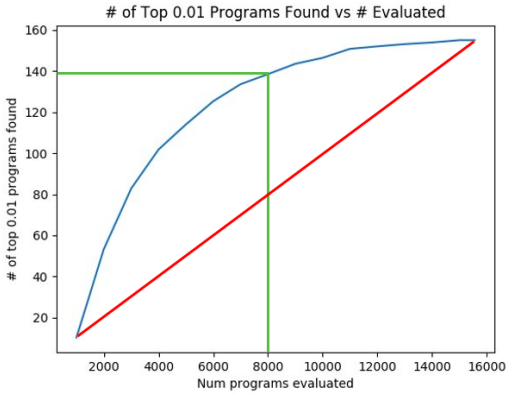

Figure 8: Predicting algorithm performance allows us to find the best programs faster. We investigate the number of the top 1% of programs found vs. the number of programs evaluated, and observe that the optimized search (in blue) finds 88% of the best programs after only evaluating 50% of the programs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

D PERFORMANCE ON GRID WORLD

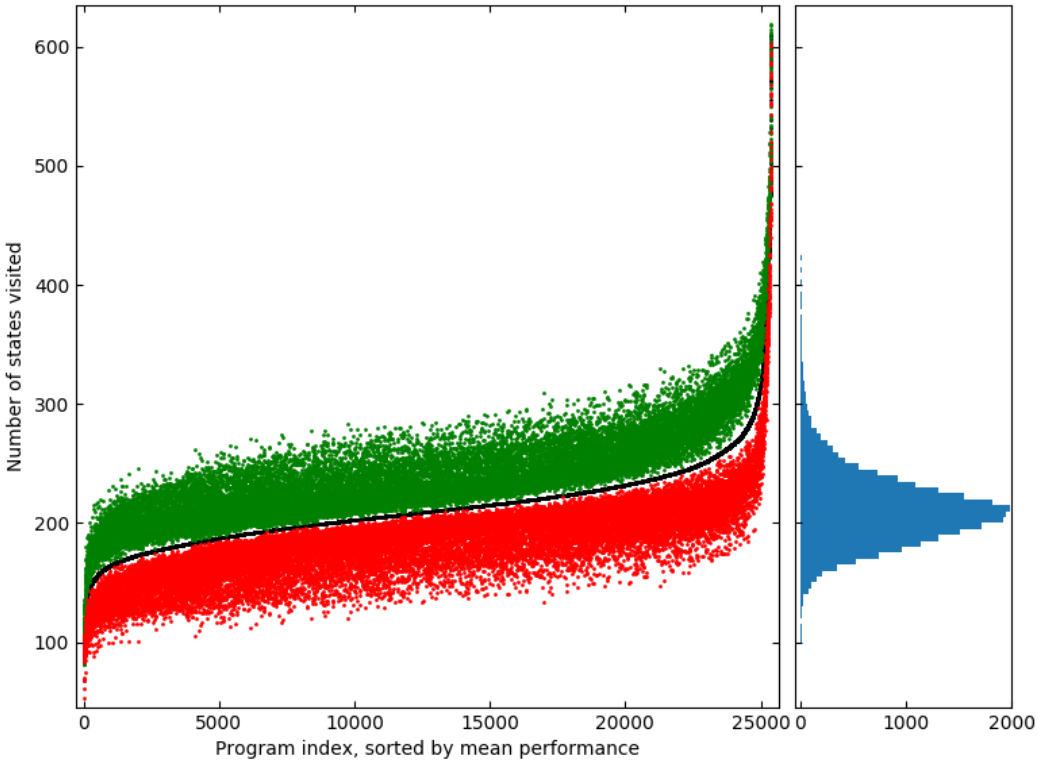

Figure 9: In black, mean performance across 5 trials for all 26,000 programs evaluated (out of their finished trials). In green mean plus one standard deviation for the mean estimate and in red one minus one standard deviation for the mean estimate. On the right, you can see program means form roughly 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.

E INTERESTING PROGRAMS FOUND BY OUR SEARCH

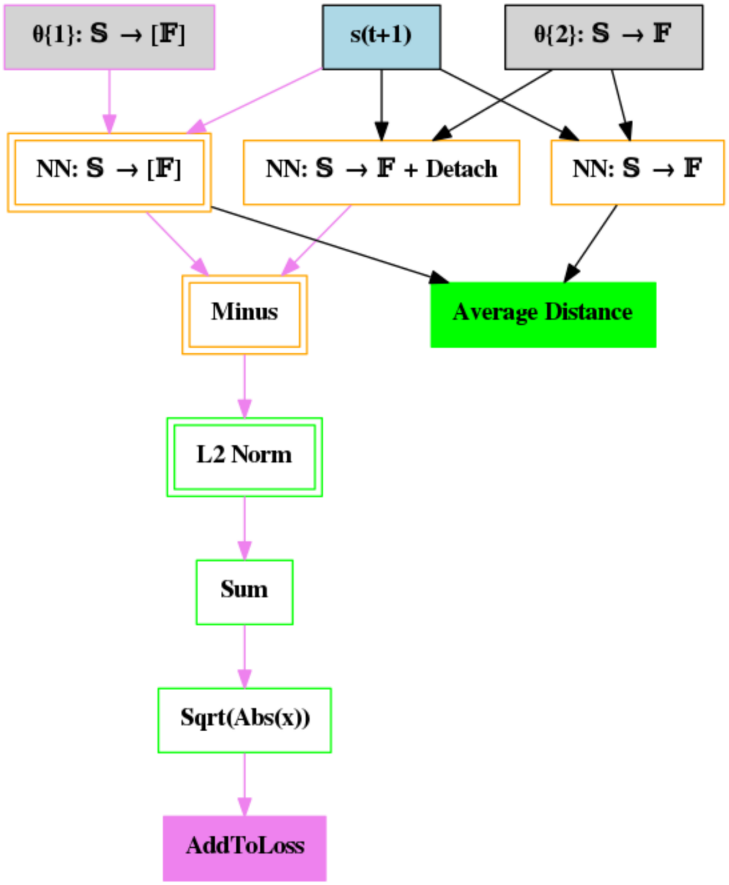

Figure 10: Top variant in preliminary search on grid world; variant on random network distillation using an ensemble of trained networks instead of a single one.

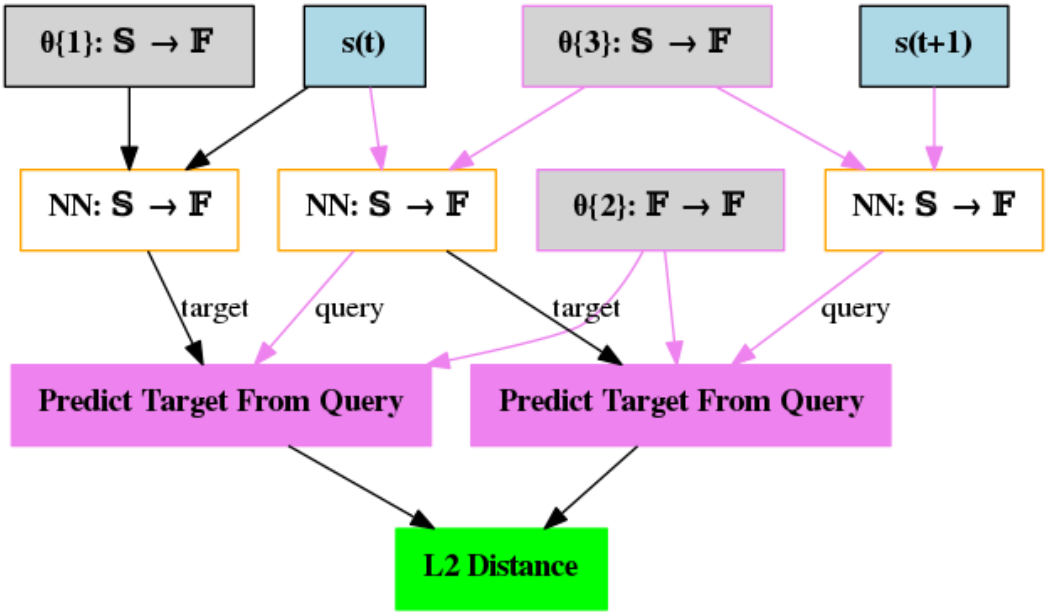

Figure 11: Good algorithm found by our search (3 of the top 16 programs on grid world are variants of this program). On its left part it does random network distillation but does not use that error as a reward. Instead it does an extra prediction based on the state transition on the right and compares both predictions. Notice that, to make both predictions, the same $\mathbb{F} \to \mathbb{F}$ network was used to map from the query to the target, thus sharing the weights between both predictions.