# Meta-learning curiosity algorithms

**Ferran Alet***, **Martin F. Schneider***, **Tomás Lozano-Pérez & Leslie Pack Kaelbling**
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
{alet,martinfs,tlp,lpk}@mit.edu

## 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 Problem formulation

### 1.1 Meta-learning problem

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. 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. 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. In this meta-learning setting, our objective is to find a curiosity module that works well given a broad distribution of environments from which we can sample at meta-learning time.

Let us assume we have an agent $\mathcal{A}$ equipped with an RL algorithm (such as DQN or PPO, with all hyperparameters specified), 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 $\pi(t) : s_t \to a_t$ to maximize the discounted reward $\sum_i \gamma^i r_{t+i}, \gamma < 1$.

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 $(s_t, a_t, \widehat{r}_t = \mathcal{C}(e_{1:t-1}), s_{t+1})$, without having access to the original $r_t$. The inner RL

*Equal contribution.

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

algorithm still purely exploits its reward $\widehat{r}$, but a correctly designed $\mathcal{C}$ may induce exploration. Our final objective is: $\max_{\mathcal{C}} \left[\mathbb{E}_{\mathcal{E}} \left[\mathbb{E}_{r_t \sim e(\mathcal{A},\mathcal{C};\mathcal{E})} \left[\sum_{t=0}^{T} r_t\right]\right]\right]$ .

### 1.2 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 we describe them in terms of general programs in a domain-specific language. 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, $\chi$, takes the current time-step $t$ (normalized by dividing by the agent's lifetime $T$), the actual reward $r_t$, and the intrinsic reward $i_t$ and combines them to yield the proxy reward $\widehat{r}_t$.

Both programs consist of a directed acyclic graph (DAG) of modules with polymorphically typed inputs and outputs: input modules (shown in blue), buffer and parameter modules (in gray), functional modules (in white), and update modules (in pink). The instantiation of some types and operations depends on the environment. A single node is specified as the output. Each call to a program corresponds to one time-step of the system. 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). In practice, we execute multiple reward predictions on a batch and then update on a batch. See appendix A for more details.

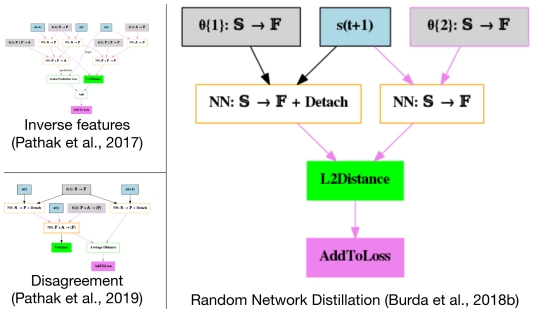


Figure 1: Example diagrams of published algorithms covered by our language (larger figures in the appendix). The green box represents the output of the intrinsic curiosity function, the pink box is the loss to be minimized. Pink arcs represent paths and networks along which gradients flow back from the minimizer to update parameters.

To limit the search space and prioritize short, meaningful programs we limit the total number of operations 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 that we need to search efficiently.

## 2 Improving the efficiency of our search

We wish to find curiosity programs that work effectively in a wide range of environments. However, evaluating tens of thousands of programs in the most expensive environments would consume decades of GPU computation, so we have designed strategies for quickly finding good programs.

**Pruning invalid algorithms without running them:** First, we check that programs are not duplicates using a randomized test where we identically seed two programs, feed them identical fake data for tens of steps and compare their outputs. Second, we check that the loss functions cannot be minimized independently of the input data, ex. by setting a network's weights to zero.

**Pruning algorithms in cheap environments:** Our ultimate goal is to find algorithms that perform well on many different environments, both simple and complex. We observe that most valid programs will be extremely poor curiosity modules and that some environments 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 expensive ones.

**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 and use a $k$-nearest-neighbors regressor. 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 D.

**Quickly terminating bad programs**: At any point during the meta-search, we use the $K$ current best programs as benchmarks. Specifically, 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 $K$ programs minus two standard deviations of their performance minus one standard deviation of our estimate of the mean of the current program, and if so we terminate the program.

# 3 Experiments

Our RL agent uses PPO (Schulman et al., 2017) based on Kostrikov (2018)'s implementation in PyTorch. Our code at `https://bit.ly/meta-learning-curiosity-algs` takes OpenAI gym environments (Brockman et al., 2016).

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.

## 3.1 First search phase in simple environment

We start by searching for a good intrinsic curiosity program $I$ in a fast image-based grid world exploratory environment, designed by Chevalier-Boisvert et al. (2018). We optimize the total number of distinct squares visited across the agent's lifetime.

Our search optimizations allow us to only evaluate half of all synthesized programs (52,000 total, of length at most 7). 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.

The highest scoring program (a few other programs have lower average performance but are statistically equivalent) is surprisingly simple and meaningful. This program, which we will call **Top**, is shown in appendix A.3. Of the top 16 programs, 13 are variants of **Top** and 3 are variants of a program shown in figure 11 in the appendix.

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

Figure 2: Meta-learned algorithms perform significantly better than constant rewards and statistically equivalently to published algorithms found by human researchers (see 1). 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.

## 3.2 Transferring to new environments

Given the fixed reward combiner described in appendix C and the list of the best 2,000 selected programs found in 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 3 show good correlation between performance on grid world the new environments.

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. We then compare the best 16 programs on grid world to four weak baselines (constant 0,-1,1 and Gaussian noise reward) and the three published algorithms expressible in our language (shown in figure 1). 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.

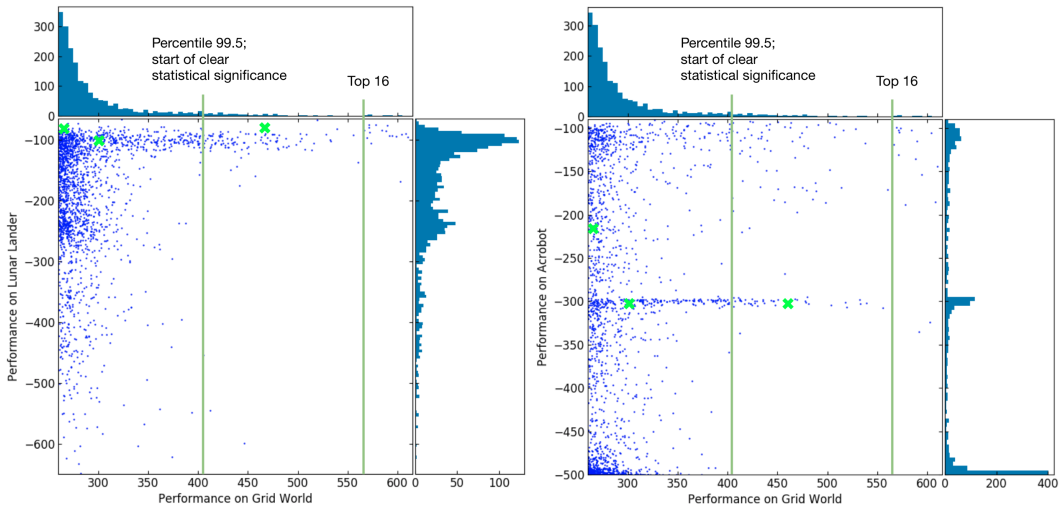


Figure 3: We show correlation between program performance in gridworld and performance in harder environments (lunar lander on the left, acrobot on the right), using the top 2,000 programs in gridworld. Performance is evaluated using mean reward across *all* learning episodes, averaged over trials (two 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).

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 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.

## 4 Related work

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 more review please see appendix B.

## 5 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 is 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


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.

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.

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.

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, 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.

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.

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. 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.

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.

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 Botvinick. 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.

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.

There are four classes of modules: **1. 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 $\chi$ module. **2. Buffer and parameter** modules (shown in gray) of two kinds: FIFO queues that provide a finite list of the $k$ most recent inputs, and neural network weights which may (pink border) or may not get updated via back-propagation depending on the computation graph. **3. Functional** modules (shown in white) such as a "Neural Network" or "L2 Distance". **4. Update** modules (shown in pink), that either add variables to buffers (such as "k-Nearest-Neighbor") or that add real-valued outputs to a global loss 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.

Our programs have four data 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.

The instantiation of some types and operations depends on the environment. Neural networks are either convolutional or fully connected and action prediction losses either use mean squared error or negative log likelihood. 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. 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 means that the same curiosity program can be applied to states and actions regardless of dimensionality, shapes (ex. images vs. vectors) or type (ex. continuous vs. discrete).

### 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 The top program discovered by our program search

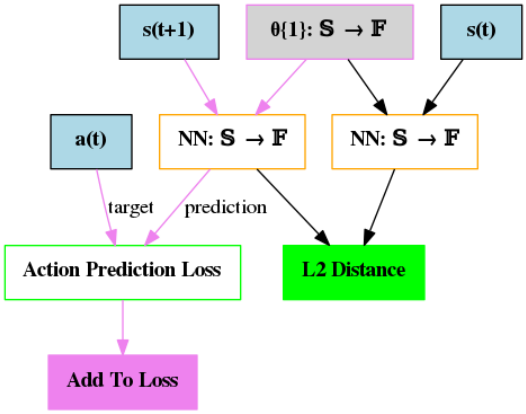


Figure 4: **Top** program in the large phase 1 search. It uses a single neural network to predict the action from $s_{t+1}$ and then compares its predictions based on $s_t$ with its predictions based on $s_{t+1}$, generating high intrinsic reward when the difference is large. The *action prediction loss* 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.

### A.4 Two other published algorithms covered by our DSL

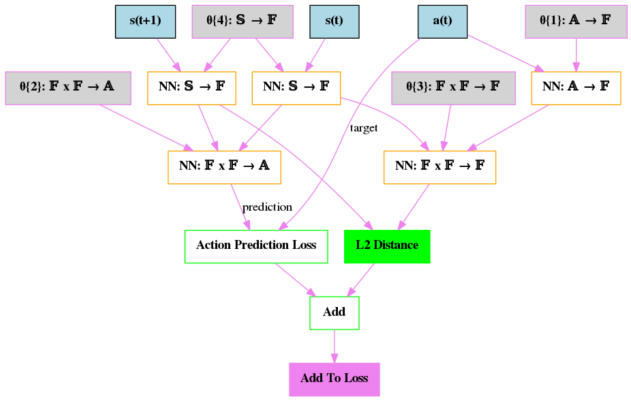


Figure 5: Curiosity by predictive error on inverse features by Pathak et al. (2017). In pink, paths and networks where gradients flow back from the minimizer.

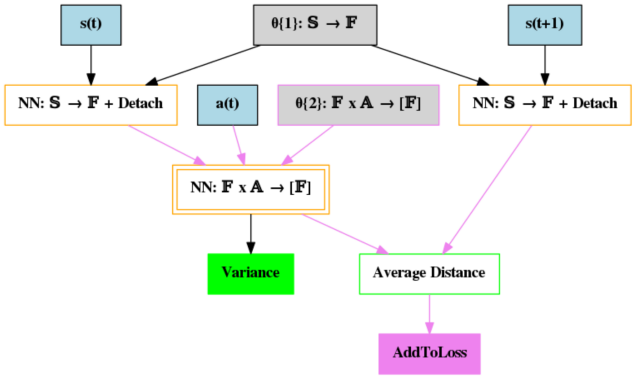


Figure 6: Curiosity by ensemble predictive variance Pathak et al. (2019). In pink, paths and networks where gradients flow back from the minimizer.

## B Extra 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 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 1 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.

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 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.

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. 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.

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. 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.

# C Searching for a reward combiner

Our reward combiner was developed in *lunar lander* (the simplest environment with meaningful extrinsic reward) based on the best program (shown in appendix F) among a preliminary set of 16,000 programs. 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.

## D 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.

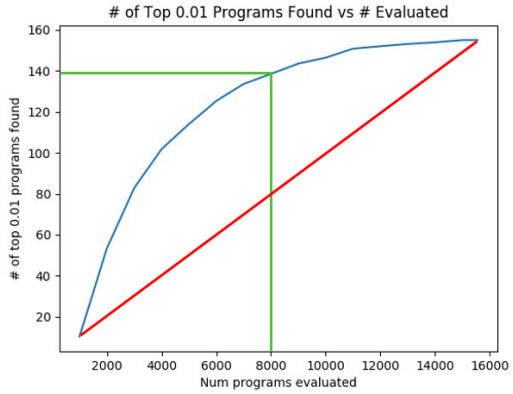


Figure 8: Predicting algorithm performance allows us to find the best programs faster. We investigate the number of the top 1% of programs found vs. the number of programs evaluated, and observe that the optimized search (in blue) finds 88% of the best programs after only evaluating 50% of the programs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

## E Performance on grid world

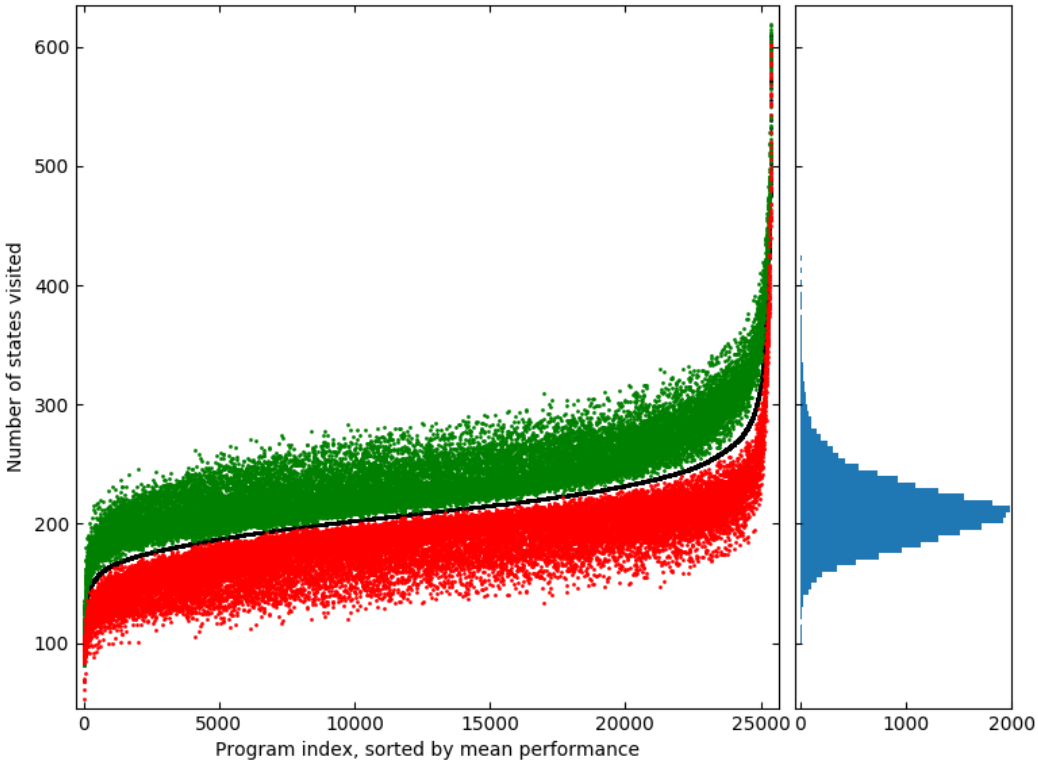


Figure 9: In black, mean performance across 5 trials for all 26,000 programs evaluated (out of their finished trials). In green mean plus one standard deviation for the mean estimate and in red one minus one standard deviation for the mean estimate. On the right, you can see program means form roughly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

## F Interesting programs found by our search

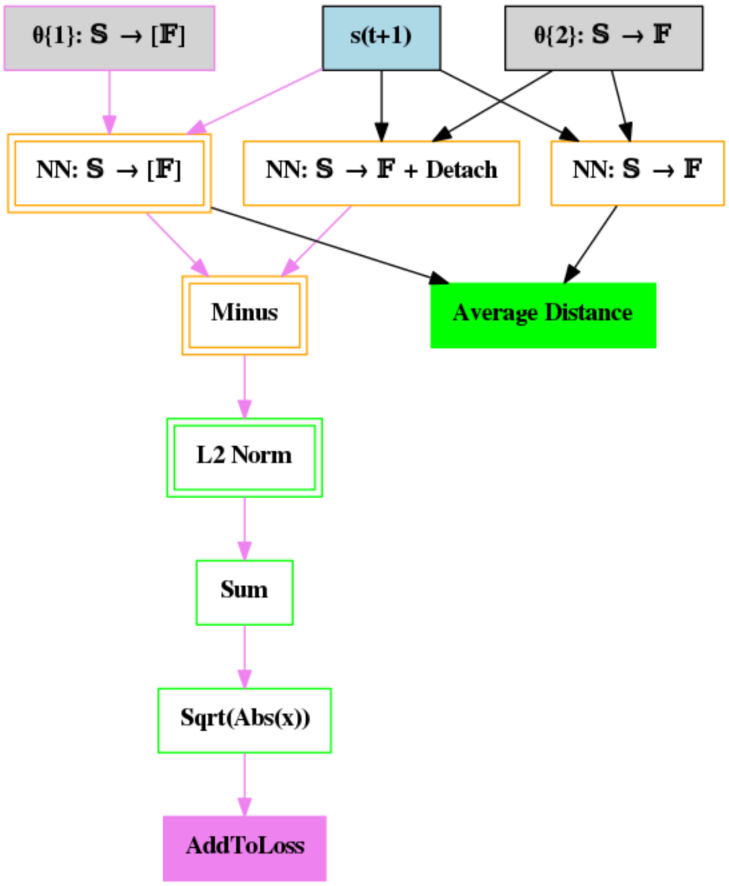


Figure 10: Top variant in preliminary search on grid world; variant on random network distillation using an ensemble of trained networks instead of a single one.

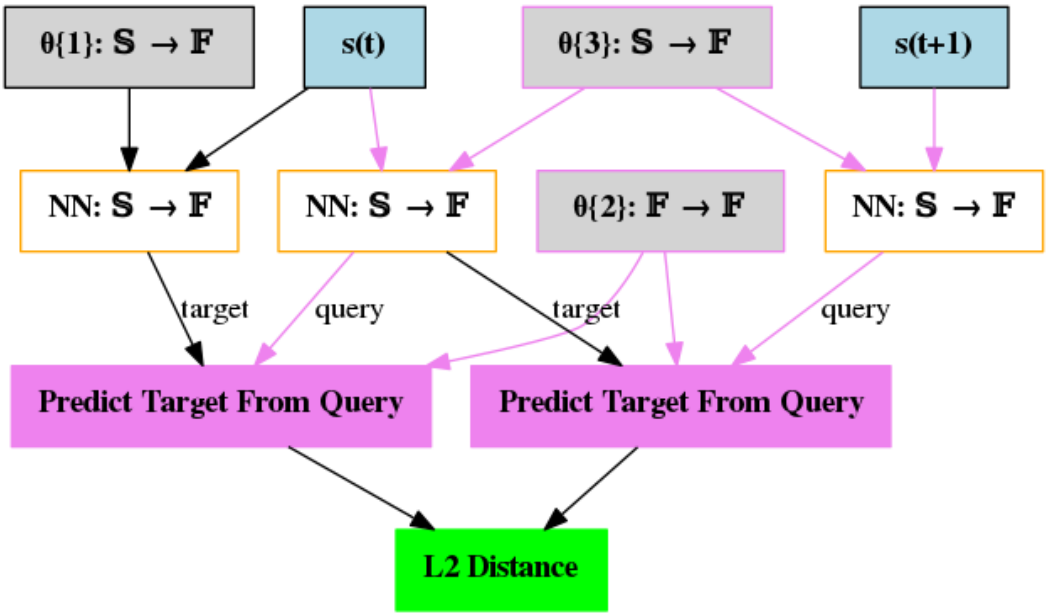


Figure 11: Good algorithm found by our search (3 of the top 16 programs on grid world are variants of this program). On its left part it does random network distillation but does not use that error as a reward. Instead it does an extra prediction based on the state transition on the right and compares 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.