Program Synthesis Approaches to Improving Generalization in Reinforcement Learning

by

Martin Franz Schneider

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

Author . . . .
Department of Electrical Engineering and Computer Science
May 18, 2020

Certified by . . . .
Leslie Pack Kaelbling
Professor
Thesis Supervisor

Certified by . . . .
Tomás Lozano-Pérez
Professor
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Program Synthesis Approaches to Improving Generalization in Reinforcement Learning

by

Martin Franz Schneider

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

To perform real-world tasks, robots need to rapidly explore and model their environment. However, existing methods either explore slowly, are data-inefficient, or need to leverage significant prior knowledge that limits their generalization. In this thesis, we explore applications of techniques from the program synthesis literature to improve the generalization and data-efficiency of reinforcement learning agents. Two complementary approaches are explored. First, we explore leveraging program synthesis techniques to meta-learn exploration strategies, and automatically synthesize new explorations strategies competitive with state of the art benchmarks. Second, we explore applying program synthesis to the problem of learning factored world models and achieve promising preliminary results. We see these results as promising examples of the potential of integrating program synthesis techniques with the rest of our modern modern reinforcement learning and robotics toolkits, increasing generalization in the process.

Thesis Supervisor: Leslie Pack Kaelbling
Title: Professor

Thesis Supervisor: Tomás Lozano-Pérez
Title: Professor

Acknowledgments

First, I would like to thank my mentor and collaborator Ferran Alet for meta-training me so that I may better learn and research on my own. His guidance at both the high and low level has sharpened me in more ways than I can count.

I would like to thank Professors Leslie Kaelbling and Tomás Lozano-Pérez for their sage yet fun-spirited guidance, their clarity about finding and working on important problems, and the supportive atmosphere they've instilled throughout the lab.

For very useful course corrections and discussions, I would like to thank Professor Josh Tenenbaum, Professor Pulkit Agrawal, and the LIS lab members as a whole; the path forward will be one of many perspectives.

To the friends who have made studying computer science not just interesting, but joyful - thank you.

Last, but definitely not least, I would like to thank my parents for nurturing my interests wherever they may go, and for my family for their immense support and care.

Contents

Chapter 1

Introduction

To perform complex tasks, agents need to learn about their environments. This learning is different from the classical setting of supervised learning because the agent's actions determine the training data it will receive, and the agent needs structures to support long term decision making. Current methods as a whole explore slowly, are data-inefficient, or need to leverage significant prior knowledge that limits their generalization. In this thesis I leverage program synthesis techniques to address these problems from two angles.

In chapter 2, I consider *curiosity* strategies that guide an agent to explore its environment effectively. These strategies encourage the agent to seek out novel experiences with the hope that in doing so it will discover new dynamics, find new sources of rewards, or improve its transition model. Specifically, I explore ways to automatically synthesize novel curiosity strategies automatically given some distribution of tasks.

In chapter 3, I develop methods to learn transition models using a program-like representation. This representation has a good inductive bias for learning transition models in environments that contain objects and can be leveraged for efficient planning. I also explore methods that build on this representation to automatically learn to plan more efficiently.

Chapter 2

Meta-Learning Curiosity Algorithms

2.1 Overview

We hypothesize that curiosity is a mechanism found by evolution that encourages meaningful exploration early in an agent's life in order to expose it to experiences that enable it to obtain high rewards over the course of its lifetime. We formulate the problem of generating curious behavior as one of meta-learning: an outer loop will search over a space of curiosity mechanisms that dynamically adapt the agent's reward signal, and an inner loop will perform standard reinforcement learning using the adapted reward signal. However, current meta-RL methods based on transferring neural network weights have only generalized between very similar tasks. To broaden the generalization, we instead propose to meta-learn algorithms: pieces of code similar to those designed by humans in ML papers. Our rich language of programs combines neural networks with other building blocks such as buffers, nearest-neighbor modules and custom loss functions. We demonstrate the effectiveness of the approach empirically, finding two novel curiosity algorithms that perform on par or better than human-designed published curiosity algorithms in domains as disparate as grid navigation with image inputs, acrobot, lunar lander, ant and hopper.

2.2 Introduction

When a reinforcement-learning agent is learning to behave, it is critical that it both explores its domain and exploits its rewards effectively. 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 [51, 8, 46]. 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.

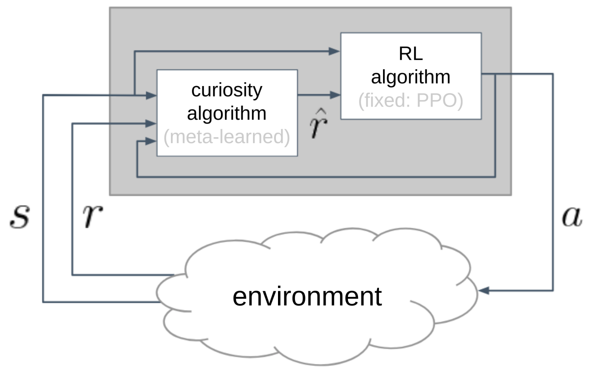

Figure 2-1: Our RL agent is augmented with a *curiosity module*, obtained by meta-learning over a complex space of programs, which computes a pseudo-reward $\widehat{r}$ at every time step.

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 an agent's life. This exploration exposes 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 an inner loop will perform standard reinforcement learning using the adapted reward signal. This process is illustrated in figure 2-1; 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 as to optimize the agent's lifetime return $\sum_{t=0}^{T} r_t$, or another global objective like the mean performance on the last few trials.

In this meta-learning setting, our objective is to find a curiosity module that works well given a distribution of environments from which we can sample at meta-learning time. Meta-RL has been widely explored recently, in some cases with a focus on reducing the amount of experience needed by initializing the RL algorithm well [23, 11] and, in others, for efficient exploration [12, 68]. 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 propose to do meta-learning in a rich, combinatorial space of programs rather than transferring neural network weights. 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 perform on par or better than those in published literature. Interestingly, the top 2 algorithms, to the best of our knowledge, had not been proposed before, despite making sense in hindsight. We conjecture the first one (shown in figure 2-3) is deceptively simple and that the complexity of the other one makes it relatively implausible for humans to discover.

2.3 Problem formulation

2.3.1 Meta-learning problem

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

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

2.3.2 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" [67]. Therefore, in order to obtain curiosity modules that can generalize over a very broad range of tasks and that are sophisticated enough to provide exploration guidance over very long horizons, we describe them in terms of general programs in a domain-specific language. Algorithms in this language will map a history of (s_t, s_{t+1}, a_t, r_t) tuples into a proxy reward $\widehat{r}_t$.

Inspired by human-designed systems that compute and use intrinsic rewards, and to simplify the search, we decompose the curiosity module into two components: the first, I, outputs an intrinsic reward value i_t based on the current experienced transition (s_t, a_t, s_{t+1}) (and past transitions $(s_{1:t-1}, a_{1:t-1})$ indirectly through its memory); the second, χ, takes the current time-step t, the actual reward r_t, and the intrinsic reward i_t (and, if it chooses to store them, their histories) and combines them to yield the proxy reward $\widehat{r}_t$. To ease generalization across different timescales, in practice, before feeding t into χ we normalize it by the total length of the agent's lifetime, T.

Both programs consist of a directed acyclic graph (DAG) of modules with polymorphically typed inputs and outputs. As shown in figure 2-2, there are four classes of modules:

- **Input** modules (shown in blue), drawn from the set $\{s_t, a_t, s_{t+1}\}$ for the I component and from the set $\{i_t, r_t\}$ for the χ component. They have no inputs, and their outputs have the type corresponding to the types of states and actions in whatever domain they are applied to, or the reals numbers for rewards.

- **Buffer and parameter** modules (shown in gray) of two kinds: FIFO queues that provide as output a finite list of the k most recent inputs, and neural network weights initialized at random at the start of the program and which may (pink border) or may not (black border) get updated via back-propagation depending on the computation graph.

- **Functional** modules (shown in white), which compute output values given the inputs 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.

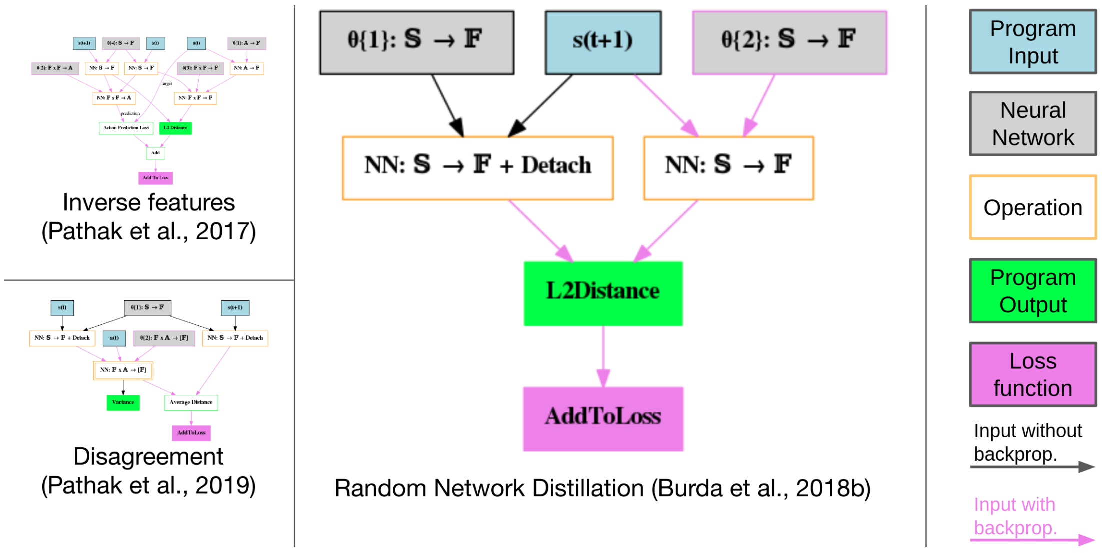

Figure 2-2: Example diagrams of published algorithms covered by our language. The green box represents the output of the intrinsic curiosity function, the pink box is the loss to be minimized. Pink arcs represent paths and networks along which gradients flow back from the minimizer to update parameters.

On each call to a program (corresponding to one time-step of the system) the current input values and parameter values are propagated through the functional modules, and the output node's output is given to the RL algorithm. Before the call terminates, the FIFO buffers are updated and the adjustable parameters are updated via gradient descent using the Adam optimizer [39]. Most operations are differentiable and thus able to propagate gradients backwards. Some operations are not differentiable, including buffers (to avoid backpropagating through time) and "Detach" whose purpose is stopping the gradient from flowing back. In practice, we have multiple copies of the same agent running at the same time, with both a shared policy and shared curiosity module. Thus, we execute multiple reward predictions on a batch and then update on a batch.

Programs representing several published designs for curiosity modules that perform internal gradient descent, including inverse features [51], random network distillation (RND) [8], and ensemble predictive variance [52], are shown in figure 2-2. We can also represent algorithms similar to novelty search [42] and EX^2 [28], which include buffers and nearest neighbor regression modules.

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

Key to our program search are *polymorphic data types*: the inputs and outputs to each module are typed, but the instantiation of some types, and thus of some operations, depends on the environment. We have four types: reals , state space of the given environment , action space of the given environment and feature space , used for intermediate computations and always set to ^{32} in our current implementation. For example, a neural network module going from to will be instantiated as a convolutional neural network if is an image and as a fully connected neural network of the appropriate dimension if is a vector. Similarly, if we are measuring an error in action space 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 2-2. 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.

2.4 Improving the efficiency of our search

We wish to find curiosity programs that work effectively in a wide range of environments, from simple to complex. However, evaluating tens of thousands of programs in the most expensive environments would consume decades of GPU computation. Therefore, we designed multiple strategies for quickly discarding less promising programs and focusing computation on a few promising programs. In doing so, we take inspiration from efforts in the AutoML community [33].

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

2.4.1 Pruning invalid algorithms without running them

Many programs are obviously bad curiosity programs. We have developed two heuristics to immediately prune these programs without an expensive evaluation.

- Checking that programs are not duplicates. Since our language is highly expressive, there are many non-obvious ways of getting equivalent programs. To find duplicates, we designed a randomized test where we identically seed two programs, feed them both identical fake environment data for tens of steps and check whether their outputs are identical.

- Checking that the loss functions cannot be minimized independently of the input data. Many programs optimize some loss depending on neural network regressors. If we treat inputs as uncontrollable variables and networks as having the ability to become any possible function, then for every variable, we can determine whether neural networks can be optimized to minimize it, independently of the input data. For example, if our loss function is $|NN_\theta(s)|^2$ the neural network can learn to make it 0 by disregarding s and optimizing the weights θ to 0. We discard any program that has this property.

2.4.2 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 [36] in hyper-parameter optimization [34].

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.

2.4.3 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.

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.

2.5 Experiments

Our RL agent uses PPO [61] based on the implementation by pytorchrl in PyTorch [50]. Our code (`https://github.com/mfranzs/meta-learning-curiosity-algorithms`) can take in any OpenAI gym environment [7] 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.

2.5.1 First search phase in simple environment

We start by searching for a good intrinsic curiosity program I in a purely exploratory environment, designed by [9], which is an image-based grid world where agents navigate in an image of a 2D room either by moving forward in the grid or rotating left or right. We optimize the total number of distinct cells visited across the agent's lifetime. This allows us to evaluate intrinsic reward programs in a fast and simple environment, without worrying about combining it with external reward.

To bias towards simple, interpretable algorithms and keep the search space manageable, we search for programs with at most 7 operations. We first discard duplicate and invalid programs, as described in section 2.4.1, resulting in about 52,000 programs. We then randomly split the programs across 4 machines, each with 8 Nvidia Tesla K80 GPUs for 10 hours; thus a total of 13 GPU days.

Each machine finds the highest-scoring 625 programs in its section of the search space and prunes programs whose partial learning curve is statistically significantly lower than the current top 625 programs. To do so, after every episode of every trial, we check whether $mean_{program}(step) \leq mean_{top625}(step) - 2std_{top625} - std_{program}$.Thus, we account for both inter-program variability among the top 625 programs and intra-program variability among multiple trials of the same program.

We use a 10-nearest-neighbor regressor to predict program performance and choose the next program to evaluate with an ϵ-greedy strategy, choosing the best predicted program 90% of the time and a random program 10% of the time. By doing this, we try the most promising programs early in our search. This is important for two reasons: first, we only try 26,000 programs, half of the whole search space, which we estimated from earlier results 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. We find that most programs perform relatively poorly, with a long tail of programs that are statistically significantly better, comprising roughly 0.5% of the whole program space.

The highest scoring program (a few other programs have lower average performance but are statistically equivalent) is surprisingly simple and meaningful, comprised of only 5 operations, even though the limit was 7. This program, which we call FAST (Fast Action-Space Transition), is shown in figure 2-3; it trains a single neural network (a CNN or MLP depending on the type of state) to predict the action from s_{t+1} and then compares its predictions based on s_{t+1} with its predictions based on s_t, generating 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. 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.

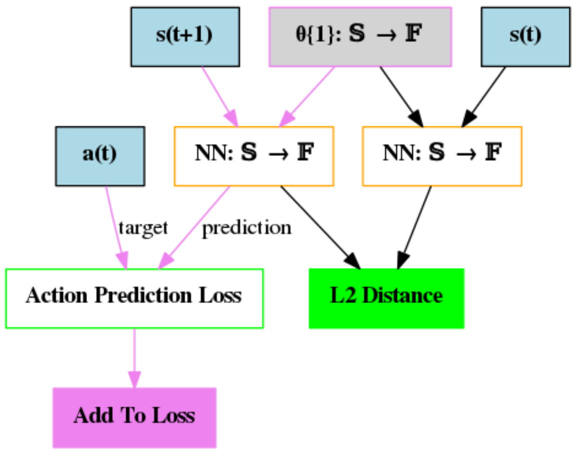

Figure 2-3: Fast Action-Space Transition(FAST): top-performing intrinsic curiosity algorithm discovered in our phase 1 search.

Of the top 16 programs, 13 are variants of FAST, including versions that predict the action from s_t instead of s_{t+1}. The other 3 are variants of a more complex program that is hard to understand at first glance, but we finally determined to be using ideas similar to cycle-consistency in the GAN literature [71] (we thus name it Cycle-consistency intrinsic motivation). Interestingly, to the best of our knowledge neither algorithm had been proposed before: we conjecture the former was too simple for humans to believe it would be effective and the latter too hard for humans to design, as it was already very hard to understand in hindsight.

2.5.2 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. Among a set of 2,500 candidates (with 5 or fewer operations) the best reward combiner discovered by our search was $\widehat{r_t} = \frac{(1+i_t-t/T)\cdot i_t + t/T\cdot r_t}{1+i_t}$. Notice that for $0 < i_t \ll 1$ (usually the case) this is approximately $\widehat{r_t} \approx i_t^2 + (1-t/T)i_t + (t/T)r_t$, 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, rather than image-based, inputs. The results in figure 2-4 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 400 (the lowest score that is statistically significantly good), performance on the other two environments is also good in more than 90% of cases.

Finally, we evaluate on two MuJoCo environments [66]: *hopper* and *ant*. These environments have more than an order of magnitude longer exploration horizon than acrobot and lunar lander, exploring for 500K time-steps, as well as continuous action-spaces instead of discrete. We then compare the best 16 programs on grid world (most of which also did well on lunar lander and acrobot) to four weak baselines (constant 0,-1,1 intrinsic reward and Gaussian noise reward) and three published algorithms expressible in our language (shown in figure 2-2). 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

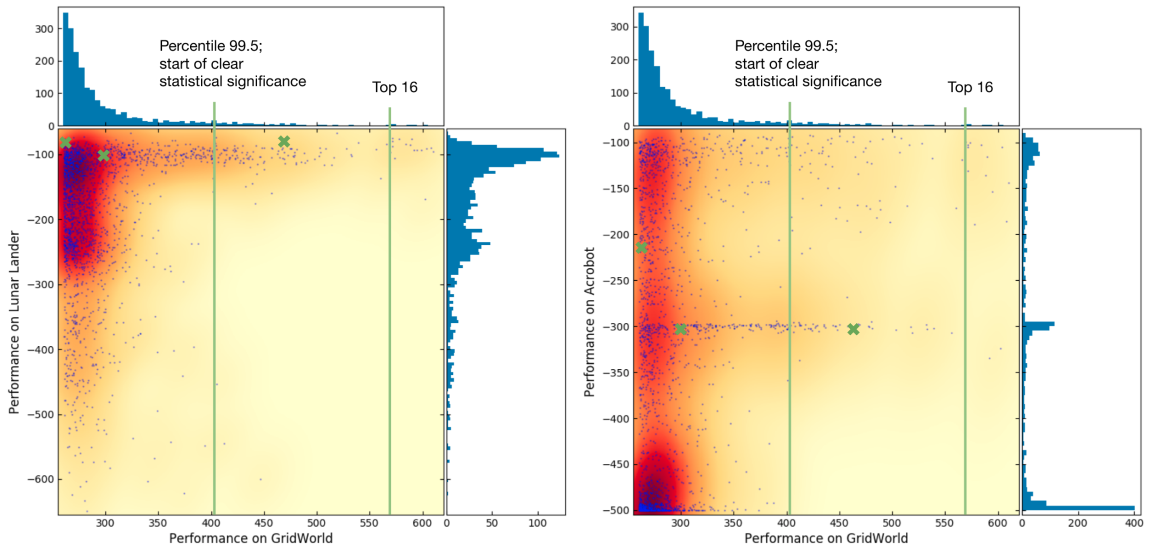

Figure 2-4: Correlation between program performance in gridworld and in harder environments (lunar lander on the left, acrobot on the right), using the top 2,000 programs in gridworld. Performance is evaluated using mean reward across *all* learning episodes, averaged over trials (two trials for acrobot / lunar lander and five for gridworld). The high number of algorithms performing around -300 in the middle of the right plot is an artifact of averaging the performance of two seeds and the mean performance in Acrobot having two peaks. Almost all intrinsic curiosity programs that had statistically significant performance for grid world also do well on the other two environments. In green, the performance of three published works; in increasing gridworld performance: disagreement [52], inverse features [51] and random distillation [8].

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

the reward combiner found on lunar lander. For both environments we find that the performance of our top programs is statistically equivalent to published work and significantly better than the weak baselines, confirming that we meta-learned good curiosity programs.

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

2.6 Related work

In some regards our work is similar to neural architecture search (NAS) [64, 72, 14, 53] or hyperparameter optimization for deep networks [45], which aim at finding the best neural network architecture and hyper-parameters for a particular task. However, in contrast to most (but not all, see zoph2018learning) 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 [33] that searches in a space of programs, for example in the case of SAT solving [38] or auto-sklearn [22] and concurrent work on learning loss functions to replace cross-entropy for training a fixed architecture on MNIST and CIFAR [30, 31]. Although we took inspiration from ideas in that community [34, 43], our algorithms specify both how to compute their outputs and their own optimization objectives in order to work well in synchrony with an expensive deep RL algorithm.

There has been work on meta-learning with genetic programming [59], searching over mathematical operations within neural networks [55, 29], searching over programs to solve games [69, 37, 62] and to optimize neural networks [6, 5], and neural networks that learn programs [56, 54]. Our work uses neural networks as basic operations within larger algorithms. Finally, modular meta-learning [1, 2] trains the weights of small neural modules and transfers to new tasks by searching for a good composition of modules; as such, it can be seen as a (restricted) dual of our approach.

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 2-2 we can represent several prominent curiosity algorithms. We can also generate meaningful algorithms similar to novelty search [42] and EX^2 [28]; 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 [63, 41, 25], learning progress [47, 60, 4], generating diverse skills [20], stochastic neural networks [24, 27], count-based exploration [65] or object-based curiosity measures [26]. Finally, part of our motivation stems from taiga2019benchmarking showing that some bonus-based curiosity algorithms have trouble generalising to new environments.

There have been research efforts on meta-learning exploration policies: [12, 68] 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. stadie2018some improves their exploration and that of [23] by considering the importance of sampling in RL policies. gupta2018meta combine gradient-based meta-learning with a learned latent exploration space in which they add structured noise for meaningful exploration. Closer to our formulation, zheng2018learning 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, [10, 21] 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.

Closest to our work, evolved policy gradients (EPG, [32]) use evolutionary strategies to meta-learn a neural network that acts as a loss function and is used to train a policy network. EPG generalizes by meta-training with target locations east of the start location and meta-testing with target locations to the west. In contrast, we showed that by meta-learning programs, we can generalize between radically different environments, not just goal variations of a single environment. Concurrent to our work, [40] also show generalization capabilities between environments similar to ours (lunar lander, hopper and half-cheetah). Their approach transfers a parametric representation, for which it is unclear how to adapt the learned neural losses to an unseen environment with a different observation space. Their approach thus does not encode states into the loss function, which is critical for efficient exploration. In contrast, our algorithms can leverage polymorphic data types that adapt the neural networks to the environment they are running in, adapting both the size and the type of network (CNN vs MLP) running in each environment.

Chapter 3

A Program Synthesis Approach to Learning Entity-Level Transition Models

3.1 Motivation

Transition models that predict $(s_t, a) \to s_{t+1}$ can be leveraged for planning and reasoning. Entity-level transition models factor their prediction into separate functions for each entity. In domains where most dynamics are controlled by a single entity or by pairwise interactions, entity-level transition models have been shown to increase generalization between states and reduce sample complexity. Note that this requires an assumption that the agent already has a perception mechanism that can obtain a representation of the state in terms of the state of each independent entity.

We propose an approach for learning a entity-level symbolic transition model in an active manner from state transitions within a simulator. Our approach attempts to expand the hypothesis space of transition models that the algorithm can efficiently consider; it partially succeeds in this regard.

Our approach differs from related approaches in a few ways - it learns a purely symbolic world model, factored into the dynamics of different entities, **without** an apriori encoding any of the domain's transition rules. Note that a strong caveat here is that our program search space still encodes a reasonably strong bias, encoding knowledge about entities interacting through local interactions, and knowledge about the complexity of possible interactions.

3.2 Assumptions

Our approach makes a number of assumptions about its environment.

First, the state is described in terms of entities. We assume that we're provided a state space in terms of entities whose state can be described by $(x, y, type, resource\_dictionary)$. This requires our model to sit on top of a vision module that parses this environment. This assumption is motivated by the observation that factoring a state into entities seems to help humans learn and generalize in a data efficient way. However, our structure includes an assumption that the entities can be learned by a lower-level module without higher level feedback; this seems potentially problematic and needs to be addressed in future work.

Second, each entity has a single type. We assume that each entity is identified with a single entity type and that all entities of the same type act in the same way. (Note, however, that entities can also hold varying amounts of resources on which their dynamics are conditioned). This is perhaps a reasonable assumption in a specific realm of thinking, but is unlikely to hold more broadly. A solution that identifies entities from raw visual data could hopefully fare better here, especially given that such a solution would need to already leverage an understanding of the different roles each entity could be playing.

Third, there is no action at a distance. We assume that every transition is only a function of the entities within some surrounding $n \times n$ region around the entity (we use 5×5 in our experiments). This encodes a bias that nothing happens through action at a distance. This is a reasonable assumption in some regards, and is a valid assumption in all VGDL domains. However, it prohibits modelling effects like "switches connected to doors at arbitrary distances" or "light shining across large distances". Removing this limitation would require the model to consider a significantly larger hypothesis space. It's likely that other heuristics will need to be built in or learned to constrain this search. For example, a reasonable bias might be that "every action at a distance is triggered by the interaction of two neighboring entities, such as an agent touching a switch". An alternative bias might be that "every action at a distance is triggered by a change in some other entity's state, such as a switch flipping from on to off". This bias would allow transitions to be chained and would likely result in a smaller hypothesis space. However, VGDL does not support action at a distance and we have thus not yet explored this idea further.

Fourth, the environment is discrete, fully-observed, nonstochastic, and markovian. Every entity's position is locked onto a grid, and in principle the next state can always be predicted perfectly from the current state. This set of assumptions is perhaps the biggest blocker to extending this approach to a broader set of interesting domains (such as continuous games or partially-observed manipulation environments).

For clarity, we do **not** assume that we need oracle simulator access to query $(s_{t-1}, a) \rightarrow s_t$ at an arbitrary state. Instead, we only have online access to the simulator through an interface that takes an action in the current state. Our planner instead leverages its learned model, which it can query from arbitrary locations.

3.3 Background

3.3.1 The Video Game Description Language (VGDL)

We test our approach in game domains specified in the Video Game Description Language. The Video Game Description Language (VGDL) [58] is a flexible language for expressing simple gridworld games. A game consists of a *game specification file* and a sequence of *level layout files*.

The level layout file is an $n \times m$ grid of letters, each representing an entity. The mapping of letters to entity types is specified in the game specification file. Multiple level layout files can be constructed for a single game.

The game specification file specifies four components:

1. A set of *entity types*, each with a base type from the VGDL language. (These are called *sprites* in VGDL, but are called *entity types* here for consistency.) Base types

are linked to a program that determines the base entity dynamics. Example base types include MovingAvatar, Passive, Immoveable, RandomNPC, OrientedFlicker, Door, ShootAvatar, etc.

2. A *level mapping* from single characters to entity types; ex "g" is mapped to Goal. This is used in the level layout file to describe an initial game state.

3. A set of *interactions* that specify how entities interact on collision. For example, the interaction "if an agent touches poison and has 0 medicine, kill the agent" would be written as `avatar poison > killIfHasLess resource=medicine limit=0`. The interaction "increase an agent's 'medicine' resource by 1 upon touching a medicine entity" would be specified as `avatar medicine > changeResource resource=medicine value=1`. Interactions can also lead to changes in rewards.

4. A set of termination conditions, such as touching the goal.

This language defines a large space of possible games. Learning a model of the game from scratch is thus a non-trivial task.

For example, the game Sokoban could be specified with entities for Agent, Box, Hole, and Wall to generate levels like the ones shown in 3-1

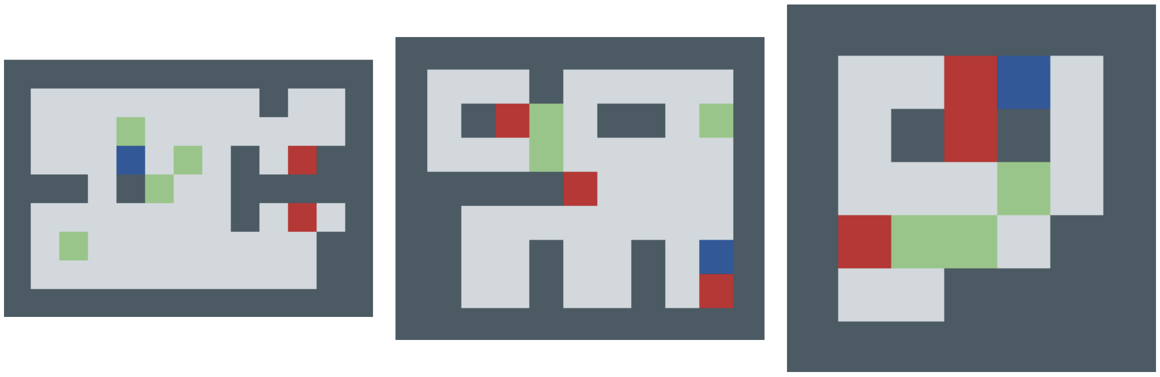
Figure 3-1: Example levels of VGDL *Sokoban*. They grey squares are Walls, the blue square is an Agent, the green squares are push-able Blocks, and the red squares are Holes. The Agent's goal is to push all of the Blocks into the Holes, deleting all of the Blocks in the process.

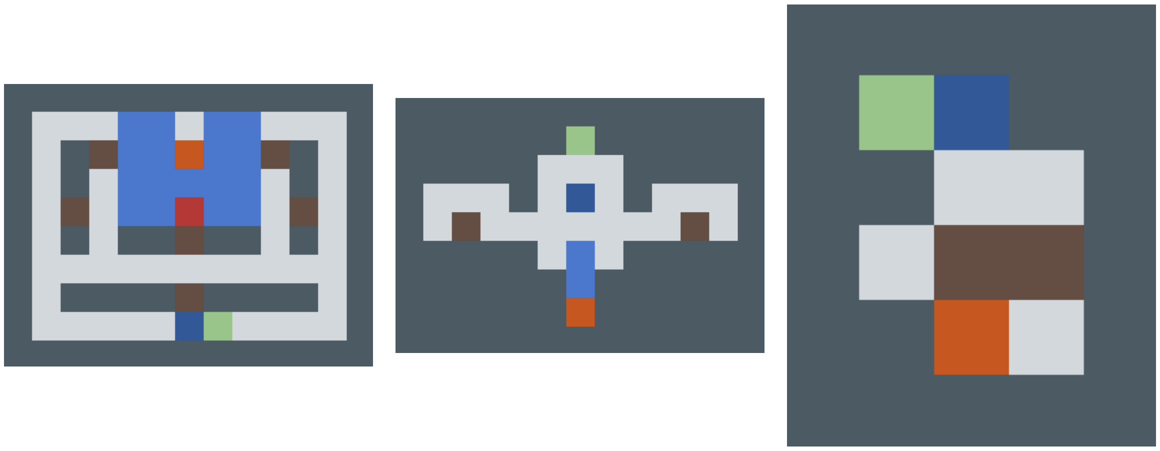

Figure 3-2: Example levels of VGDL *Bait*. The grey squares are walls, the dark blue square is an Agent, the light blue squares are Water, the red square is a Hole, the orange square is a Key, the green square is a Goal, and the brown squares are push-able Blocks. Water is destroyed when a Block is pushed onto it. The dark-blue Agent's objective is get the orange Key then get to the green Goal.

3.4 Related Work

We survey related ideas in learning world models, learning entity-level world models, and in planning.

3.4.1 Learning Neural World Models

One approach to learning a world model is to learn an end-to-end neural network to predict the entire state s_{t+1} from s_t .

In "Model Based Reinforcement Learning for Atari"[15], Kaiser et al. train a Deep Video Prediction network to learn a simulator of the environment, and then train the reinforcement learning agent within the learned simulator. This approach is designed to increase the agent's data efficiency and is tested within the "low-data" regime in Atari (100k interactions between the agent and environment). The simulator can then be queried an arbitrary number of times, allowing powerful model-free reinforcement learning algorithms to be trained with a large amount of (simulated) data. This approach led to significantly better performance than simply training the model-free RL algorithm for 100k steps directly. Note, however, that this world model is not directly accessible by the agent / policy network in any way.

"Imagination-Augmented Agents for Deep Reinforcement Learning"[19] by Weber et al. leverage a similar setup, except that the learned model is provided directly to the policy network. An "imagination module" is again parameterized by a Deep Video Prediction network. On every frame the agent uses the imagination module to simulate a rollout of the next T frames. This rollout is then encoded into a fixed-length vector by a LSTM "rollout encoder" network. This rollout encoder network is trained as part of the policy network and can be seen as learning to "interpret" the predicted rollout. The actions in the rollout trajectory are taken from a small "rollout policy" network that is trained in a supervised fashion to mimic the actions of the larger policy network. This extra policy-mimicking network is needed because the main policy needs input from the rollout encoder and also helps to speed up inference. This approach shares an element with ours in that it uses the learned world model to locally choose actions; however, it does not explicitly perform planning.

We now move to a set of approaches that all leverage planning rather than policy gradients or Q-learning.

"Entity Abstraction in Visual Model-Based Reinforcement Learning"[18] by Veerapaneni et al. presents a compelling approach to learning a world model and entity factorization jointly from raw pixels. They treat entities as latent state variables H_T and frame the process of identifying these state variables as an inference process of computing $p(H_T|H_{<T}, X_{\leq T})$. Through training, a latent representation for H_T will be learned that can then be used to reconstruct X_T or to predict the next state. Very little domain-specific bias is encoded a-priori. The most significant biases are an architectural factorization into different entities, symmetric pairwise entity processing functions, and knowledge of depth encoded in the 3D renderer (the "observation model").

Their model learns this distribution through an iterative inference process introduced in [44]. The model is factored into three components - an Object Recognition Model $Q\left(H_{1:K}^{(t)}|H_{1:K}^{(t-1)}, X^{(t)}, A^{(t-1)}\right)$, a Observation Model $G(X|H_{1:K})$, and a Dynamics Model $\mathcal{D}\ \ell\left(H_k'|H_k, H_{[\neq k]}, A\right)$. These distributions are parameterized by neural networks trained through variational inference .

First, the Object Recognition Model $Q\left(H_{1:K}^{(t)}|H_{1:K}^{(t-1)}, X^{(t)}, A^{(t-1)}\right)$ predicts the next latent state from the previous latent state and new observation and actions. Second, the dynamics model the predicts the state of the entities after a single timestep. It's factored into three types of dynamics: 1) modelling global effects on individual objects (such as gravity) 2) modelling action effects on individual objects, and 3) modeling pairwise effects between objects. It does this in a symmetric manner, applying the same functions to the state of each entity. This encourages the model to 1) learn a general physics model that can be applied to an arbitrary number of objects, and 2) learn to represent latent information about entity types within the state representation. Third, the observation model predicts the visual input that would be observed for a given latent state history $H_{1:K}$. The model outputs a rendering and a depthwise segmentation mask for each entity; these are then composed on top of each other, breaking ties with the depth data.

OP3 is tested on a rendered 3D block-stacking dataset, and also has some preliminary results on real world images. The approach is still data intensive and worked best on the isolated blockworld domains, but presents an exciting direction towards learning factored world models directly from raw visual data. It would be especially interesting to explore building in modular / programmatic structures into the architecture, to facilitate generalization across entities of different types.

3.4.2 Learning Symbolic World Models (Rule Learning)

We now survey related approaches that leverage symbolic factored models (rather than a neural network) to predict s_{t+1} from (s_t, a)

"Schema Networks"[35] by Kansky et al. also learn a factored transition model of their environment, and express it in terms of small "networks". (These are not neural networks, but rather simpler rule-like structures.) They test primarily in the game Atari Breakout. In their setup, each pixel is treated as a separate entity, leveraging the assumption that a vision module can track each pixel across frames and that each color represents a different type of object. The state is remapped into a binary variables (breaking apart continuous / categorical variables into multiple binary ones). Networks are learned to predict the state of a binary variable on s_{t+1} from state of the n closest neighboring entities the previous

T states. These rules are learned at a lifted level without reference to specific entities; to use them for prediction, they are matched ("grounded") to specific entities in the state for which the rule's preconditions hold. Rules are formulas in the form of logical expressions over the binary state variables of nearby entities; for example a rule might condition on the existence of an entity with "color=red" and "relative_offset=1". These rules are similar to the programs explored in our work in that they're factored, look at neighboring entities, and are symbolic. However, the details of the structure and the optimization process are quite different; for example, new Schema Networks are greedily added by selecting new rules to maximize prediction error, rather than performing a larger program search. Additionally, our approaches leverage dramatically different exploration, planning, and state representation techniques.

An alternative approach to rule learning is presented in "Learning Symbolic Models of Stochastic Domains"[49] by Pasula et al. They learn "action rules" to model the local effects of taking an action in a given situation. For example, consider a motion primitive called "pickup". Their model could learn that a rule that "if a block B has a relative offset on the y axis of 10 units below the gripper G , there's a 70% chance that "the predicate `inhand(B, G)` will be true after the predicate finishes executing, and a 30% chance of no change". This allows the model to represent stochasticity - in this example, to model the probability that the motion primitive fails. More formally, action rules take the form of lifted logical predicate expressions that output a distribution of possible outcomes, each of which are represented by the set of new boolean atoms that will be turned on. Their rule learning process can be interpreted from a program synthesis perspective. Given a dataset D of observed transitions, a unique rule is first constructed to model the result of every individual transition; this rule simply states the entire previous state as the predicate and the entire resulting state as the outcome. $|D|$ rules are created in total. Then, rules are generalized using an anti-unification -style process, by checking ways to replace specific entities with lifted entity variables. Using the above example of picking up a block, if there exist two observations, one with "block-1" and another with "block-2", the search process could discover that a rule that replaces "block-1" with a lifted variable B could explain both of these transitions. Then, given a proposed predicate, the distribution of possible outcomes can be computed simply by checking matching states in the dataset and looking at the distribution of next states. This approach requires an existing set of motion primitive s, as well as an existing set of symbolic relational predicates between entities (such as $on(X, Y)$, $inhand(X)$, etc.). In that sense, the idea lives at a different level of abstraction with a different setting than the one we're exploring. However, the goal of learning transition rules is similar, and their rule-synthesis process has desirable properties from an efficiency perspective. It would be interesting to explore ways to combine these ideas with the ideas we present.

"Learning sparse relational transition models" by Xia et al. leverage a similar sparse transition model to the work by Pasula et al., but instead use neural networks to parameterize the transition models. This parameterization provides a smoother optimization landscape than Pasula et al.'s program search process and allows them to learn more complex, continuous dynamics.

Perhaps the closest related work is "Human-Level Reinforcement Learning through Theory-Based Modeling, Exploration, and Planning" [17] by Tsividis et al. This paper presents EMPA, which learns a symbolic factored transition model of its domain by performing bayesian inference to reconstruct the original representation of the domain. It uses a strong prior about the space of possible environment dynamics to do so efficiently. EMPA has three components - "Exploring", "Modelling", and "Planning".

The paper also uses the VGDL domain (and solve many more domains in VGDL than our current approach). EMPA is pre-built with a full internal specification of the VGDL language. Internally, it directly calls into the VGDL library - this allows the agent to consider $p(observations|vgdl\_specification)$ by constructing a corresponding VGDL implementation and looking at the successor states. EMPA factors its inference into different entities. This allows it to learn in a highly data-efficient manner, often identifying the environment dynamics necessary to solve the goal within 10 to 50 steps. This data efficiency is an impressive demonstration of how to learn efficiently once you've identified the relevant space of dynamics that you need to consider. VGDL is highly expressive, and there are a massive number of domains that can be constructed (by constructing different game specification files and level layout files), so EMPA is tasked with a significant inference challenge. It solves this by leveraging a strong assumption that the full space of possible entity type dynamics are known beforehand (although note that EMPA still needs to perform inference over how interactions can be constructed, which impacts entity dynamics as well).

EMPA explores by setting subgoals for itself to get pairs of entities to touch in previously unobserved ways. Given that interactions in VGDL are triggered by touching, this was shown to be a highly effective exploration strategy. We also use this strategy (which we call Explore-Novel-Pair below), and extend it with an additional exploration strategy for our programs, which we call Explore-Unknown-Outcome .

EMPA uses a modified version of the Iterated Width (IW) Planning Algorithm. It leverages three types of goals, which are combined to create a reward function - 1) High level goals of getting the size of a set to equal something (such as "pick up all diamonds" or "touch the goal flag to make it disappear") 2) Sub goals, which are any state that changes the count of the high level set (such as "pick up a diamond") 3) Goal gradients, which are any state that moves the agent closer to a sub goal. Additionally, their planner includes three stages of fallback modes: 1) Planning to reach a goal terminal condition , 2) Planning to reach a subgoal 3) Staying alive by planning to any nearby state that doesn't result in a loss.

Our approach is inspired by these ideas, and explores the tradeoffs and methodology around baking in less prior knowledge into the model. Although we both attempt to learn a symbolic world model that can be leveraged for planning, the formulations of this model are quite different. Our approach attempts to remove the need to provide VGDL to the agent and instead have it re-learn (an approximation to) VGDL for itself. Rather than modelling in terms of inferring a VGDL specification, we learn lower-level programs to predict environment dynamics. Leveraging these models, we both plan with the Iterated Width (IW) plannign algorithm. EMPA additionally build in more planning structure in the form of heuristics about subgoals and goal gradients to speed up planning, while we leverage our model to learn IW-2 higher level actions. However, our technique for learning IW-2 higher level actions can be used in any domain where we have a world model that's factored into objects, and could thus also be integrated into EMPA if desired.

Finally, we note that using entity-centric transition models that look at the neighborhood around the entity has (perhaps unsurprisingly) also been found to be useful in model-free reinforcement learning. For example, "Rotation, Translation, and Cropping for Zero-Shot Generalization"[70] by Ye et al. demonstrates that centering the policy network's input around the agent helps with generalization to new levels of a VGDL game and reduces sample complexity. We leverage a similar principle.

3.4.3 Iterated Width Planning

Iterated Width (IW) [16] planning is a planning algorithm based on breadth-first-search that prunes states that are too similar to already observed states. A IW Location is a tuple of length w consisting of a set of boolean atoms. A state s with a set of active boolean atoms B has a set of IW Locations constructed from all possible size w combinations of boolean atoms. The IW Planning algorithm maintains a set L of already-seen IW Locations. New states are only expanded if they contain at least one IW Location that is not present in the set L . After expanding a state, all IW Locations in the state are added to L . The value w that defines the length of the IW Location tuples is called the "width" of the planner. This parameter is quite important, so the width is often specified in the name; for example "IW-1" means that $w = 1$. Given a branching factor b and n boolean atoms per state, IW- w will explore at most $O(bn^w)$ states; this is exponential in the w term, so it's important to keep the width as small as possible. Interestingly, many domains have low width if expressed in the right representation. A width of 2 is sufficient to solve many popular planning domains (and we only explore width of 1 or 2 in our work). However, it's important to note that the width is very sensitive to the representation design. For example, representing the state with separate variables for x and y positions already makes simple grid navigation a width-2 task, but combining them into a joint (x, y) position lowers this back down to width-1. It is thus useful to find ways to spend as much time as possible planning in lower widths; we explore an approach to do so in our work.

3.5 The World-Modeller

As the agent explores its domain, it learns a world model $(s_t, a) \rightarrow s_{t+1}$. This model is then leveraged by the planner to find action sequences that achieve the agent's goals. The model is built from observed state transitions, some directly caused by the agent's actions. Sometimes these state transitions are observed in a passive sense, without the agent trying to cause them. Other times, the agent explicitly tries to experiment with its environment to see what state transition will occur.

3.5.1 A Factored Transition Model

The modelling module learns a transition function to predict s_{t+1} from (s_t, a) . It does so in a factored way, learning to model the transitions of each entity independently. Specifically, the model is composed of a set of *transition classifiers*, each of which are tied to a single entity type and and predict if an entity of that type will undergo a relative state transition $(\Delta x, \Delta y, new\_type)$. (Alternatively $(resource\_name, \Delta r)$ if the classifier is modelling a resource change). For example, a Box entity type might have a transition classifier of ($\Delta x = 1, \Delta y = 0, new\_type = box)$ to represent being pushed to the right; this classifier might learn that this transition only happens when an agent is to the left and pushing right.

The next state is predicted in a factored manner, independently for each entity. For each entity, the set of transition classifiers with a matching entity type are found. These classifiers are then split into groups; one group for classifiers that modify the agent's position, and an extra group for each research that the agent can have. These groups ensure that such as an entity's position can simultaneously change by $\Delta x = 1$ and $\Delta\texttt{medicine} = 1$, but not simultaneously by $\Delta x = 1$ and $\Delta x = -1$. For each group, if any single classifier matches, the relative state delta is applied (if multiple match, the change is selected arbitrarily).

Each transition classifier outputs one of *Yes - it will happen*, *No - it will not happen*, or *Unknown*. When any transition classifier outputs *Unknown*, the overall state is marked as *uncertain*. This tracking allows the agent to reason with a coarse level of certainty about its predictions. As explained later, it also allows the agent to plan paths to state transitions for which it is unsure of the outcome.

Each classifier internally maintains a dataset of *positive datapoints* of states on which the transition was observed and *negative datapoints* of states on which the transition was not observed. These datasets are built in a conservative manner, where new datapoints are only stored if a previous model made an incorrect prediction. A new *positive* datapoint is collected if the model failed to predict a that transition would occur, and a new *negative* datapoint is collected only if the model incorrectly predicted a transition would occur. The datasets generally stayed small in our experiments, with only tens of datapoints stored for each. Our classifier training process has time complexity linear in the number of datapoints collected, so keeping this size small was important for speeding up this process.

3.5.2 Classifiers are Represented by Programs

The classifier could be constructed in a number of ways. In this work, we explore representing the classifier as a program/logical expression. Our program takes as input the $k \times k$ region surrounding the entity ((5×5) in our experiments); the programs will query this region by checking if there exists an entity of a specific type at a specific relative offset from the entity in the center. Using our running example of a box being pushed to the right, a program could encode `if(entity_type_at_offset(-1, 0, Agent) and not entity_type_at_of`
`0, Wall)) return Yes`

Our programs are constructed through a simple domain specific language of just two operations:

- ANDRule: This operation represents a single rule, such as
  `if(entity_type_at_offset(-1, 0, Agent) and not`
  `entity_type_at_offset(1, 0, Wall)) return Yes`. Specifically, this operation looks at a set of squares (determined by relative offsets from the entity being transformed), and for each square checks if the entity(s) in that square are of the type specified by the program. The operation stores an outcome to return if it matches.

  Additionally, the rule stores the set of all observed tuples of entity types in the squares that it's monitoring. Using the above example, if an Agent is ever observed to the left and a Box to the right, (Agent, Box) would be stored. If the rule sees a tuple that it does not have in its observed set, it will return *Unknown*. This is designed to help the agent learn generalizations of its rules.If the rule finds a match, it will return its stored outcome (either *Yes* or *No*). Otherwise, it will return *Pass*.

The above rule would thus be stored as

```
{
yes:{(-1, 0): {"agent"}},
no: {(1, 0): {"wall"}},
observed: {("agent", "wall"), ("agent", "empty"),
("agent", "box")},
outcome: "Yes"
}
```

- TakeFirstActiveRule: This operation maintains a list of of ANDRules. It loops through each rule and returns the first outcome that is not *Pass*. If all rules return *Pass*, this operation returns *Unknown*. This operation is similar to a decision list. [57]

3.5.3 A *Synthesis Through Unification*-style Program Synthesizer Finds High-Scoring Programs

Due to our small DSL, every program is of the form TakeFirstActiveRule(List[ANDRule]). Our program synthesis module utilizes a *synthesis-through-unification*[3]-style procedure to build programs iteratively, adding one ANDRule at a time. On each attempt to find the next ANDRule, the synthesizer will perform an enumerative search over a few parameters:

1. The number of squares (relative positions from the main entity) to condition on. Programs with smaller numbers of squares are tested first, biasing the search towards simpler programs. This value is capped with a hyperparameter.

2. The specific tuples of squares to consider.

3. Which squares are in the *positive set* and which are in the *negative set*.

4. The matching entity sets for each square.

5. The position in the TakeFirstActiveRule list at which to put the new ANDRule.

Each candidate program is built by adding the new ANDRule to the program being built and evaluating the program's score on the full collected dataset. The best ANDRule will be greedily selected and added to the program. The search process will halt if a perfect (score =1) program has been found; otherwise a search will begin for the next operation to add (up to the maximum number of operations).

3.5.4 Program Synthesis Optimizations to Speed up Our Search

We found it critical to build in a number of optimizations into our search process. These optimizations should probably be considered limitations of the work, as they increase the complexity while decreasing the generality. In future work we would like to explore methods for learning these biases.

First, only consider squares with multiple observed values in our dataset: If a transition classifier's datasets have always had the same entity in a given square offset, that square has no discriminatory power. That square is thus not considered as a candidate in our search process. For example, the center square in the detector input always includes the entity being transformed; this square is thus ignored.

Second, eject when a perfect program is found: Once a program with score=1 is found, the search process halts and that program is returned.

Third, resume searches from the last synthesized perfect program: Consider the case where a transition classifier has found a perfect program P with o operations. At some point a new datapoint might be observed that the program mis-classifies . Rather than scrapping the entire existing program, the synthesizer will instead restart the search with $P[: o-1]$, the program P with all but the last-found operation left. Our synthesizer can be seen as having two modes: a *search for perfect program* mode and a *search for the* *highest scoring program mode*. In our deterministic VGDL domains, all transitions can in principle be predicted with perfect accuracy, so the synthesizer mostly acts in the first mode. It sometimes enters the second mode when it's building on a partially incorrect program; in this case the program will eventually hit its max number of operations and be re-synthesized from scratch.

Fourth, only consider the central cross of squares: In VGDL, almost all transitions can be modeled by looking at entities only on either the same horizontal or vertical axis. A more general heuristic should in principle be learnable by biasing the search towards squares used in previously selected programs; we did not explore implementing this optimization.

Fifth, only consider neighboring pairs of squares: We further constraint the hypothesis space by only considering chains of neighboring/touching squares. This is again a reasonable bias, especially in VGDL, but will be a barrier in generalizing the approach.

Sixth, extend ANDRule *observed sets* when possible: Individual ANDRules return *unknown* for any tuple of entity IDs that they have not explicitly observed before. Rather than re-synthesizing the entire program once a novel tuple is observed, the synthesizer first checks if the program would have otherwise predicted the right answer for this novel tuple. If so, the tuple is simply added to the program's *observed set*, allowing the program to return that right answer. This allows observed sets to grow with minimal computation, allowing programs to generalize efficiently in a very controlled manner.

3.5.5 Resource Handling

In VGDL, every entity can hold a set of resources, each with a static string name and a variable integer counter. Resources can change upon entity collisions, as specified by the game's Interaction set. We handle resources in the same way as "x, y" coordinate transitions, by learning a classifier for when a resource delta (ex "gaining 1 medicine") will occur.

This is handled in a few places throughout the agent. In the Prediction-Fidelity Checking step, resource transitions are monitored to create new transitions and add collect extra datapoints for mispredicted existing transitions.The entity state is extended with a dictionary of the resources of each entity, and the state - transition predictor factors changes in resources from changes in (x, y) state.

The program input space is extended to allow programs to condition on an entity having a given resource. Specifically, programs can query the exact resource value of the entity at a relative $(\Delta x, \Delta y)$ offset from the Main Entity . For example, a program for checking if a door opens could check if "the entity at $(-1, 0)$ is an agent, the entity at $(-1, 0)$ has a resource Key with value 1, and the action is *press right*". Note that this space does not currently support inequalities; this would be simple to add but comes at the tradeoff of a larger search space.

In the IW-Planner a boolean atom is added for every tuple of the type (entity_id, resource_name, resource_value). Valid resource values are bounded between 0 and the max resource type specified in the domain's VGDL specification file. This prevents endless resource acquisition in the planning step; such as when the agent has an incorrect world model of "acquire a key every time the agent touches the key, but do not delete the key".

3.5.6 Handling Appearing / Disappearing Entities

In addition to representing position changes and resource level changes, transition classifiers can also represent agents disappearing / appearing. Disappearing entities can be treated in roughly the same way as position changes are, as long one is careful about the state representation. Appearing entities must be treated slightly differently, as there is not an existing entity to center the transition classifier's input around. Appearing entities are currently handled by the modeller by checking every square separately to see if an entity will appear. A faster approach might leverage the principle of binding to a given entity in the rule; for example if a rule requires an Agent in a location, the checker only needs to check squares aat which that agent would be at the right relative offset. We did not explore this in our work.

3.6 The Planner

Our planner takes a description of the state, the learned world model, and a goal predicate. It leverages a modified form of Iterated Width (IW) Planning Algorithm to find the shortest path to the goal.

3.6.1 Stages of the Planner

The planner has four different planning modes:

1. Reach-Goal: Find a path to the goal, on which our model never predicts *Unknown*

2. Attempt-Goal: Find a path to the goal, allowing paths that have uncertainty. Recall that a state s_{t+1} can be marked with a *uncertain* flag if any transition for any entity on state s_t output *Unknown* (or if s_t was already flagged as uncertain). However, the uncertainty might come from an entity that is irrelevant to agent reaching its goal. This planner thus simply tries to execute the path to the goal anyway, in the spirit of *optimism under uncertainty*.

3. Explore-Novel-Pair : Find a path that terminates in the observation of novel pair of entities (a, b) touching on a side s (left, right, above, or below). (See *Exploration Mechanisms*.)

4. Explore-Unknown-Outcome: Find a path that terminates a state where a model output *Unknown*

The planner runs each of these modes sequentially, returning the first successfully returned path. If no path is found with IW-1, the planner reruns the four modes with IW-2, resulting in eight total planning attempts. If no plan is found, a random action is taken. IW-2 is dramatically slower than IW-1, so placing all of the IW-1 planning attempts before the IW-2 planning attempts was found to significantly reduce planning time. Once the agent has solved a specific domain configuration a single time in IW-2, it learns a HLA that allows it to reliably find that path again.

3.6.2 Exploration Mechanisms

Two guided exploration mechanisms are used - *Explore-Novel-Pair* and *Explore-Unknown-Outcome*.

Explore-Novel-Pair

Taking inspiration from EMPA, we use an exploration procedure that seeks to get novel pairs of objects to touch. For example, if we've never seen a box touch the hole from the left side, the planner will attempt to find a path to a state on which we can observe this relationship. Observing this interaction can lead to four possible outcomes, three of which which provide useful information to the world-modelling module. 1) A new entity transition is observed. The modeller instantiates a new transition, which it will then work on refining over time. 2) An existing transition is observed that the agent had predicted *Unknown* or *No (will not happen)* for. The world-modeller module will collect a new datapoint to reformulate its hypothesis. 3) A transition that the world-model predicted would happen did not happen. The world-modeller module will collect a new datapoint to reformulate its hypothesis. 4) No transform happens. Nothing is learned, but the touch is recorded and will not be pursued again.

Explore-Unknown-Outcome

Transition classifiers can output *Unknown* when a state does not exist within their set of observations. In this case, we want to collect an observation that would resolve this uncertainty. To do so, the Explore-Unknown-Outcome heuristic searches for a path to the closest state with any unknowns. Empirically, the agent spends a majority of its time in this state, identifying deficiencies in its model and seeking paths to correct them.

3.7 IW-2 Higher Level Actions

3.7.1 Motivation

The number of states explored by an IW planner is exponential in terms of the width. IW-2 planning is tractable in many domains but is already significantly slower than IW-1, and widths of IW-3 or greater are rarely used. We thus would like to be able to solve as many domains in IW-1 as possible.

As a running example, consider the game Sokoban, where an agent needs to be able to efficiently plan ways to push blocks around. Finding the plan in figure 3.7.1 requires IW-2. (IW-1 could fail if it explores the path that places the agent beneath the block before it explores the path involving pushing the box down).

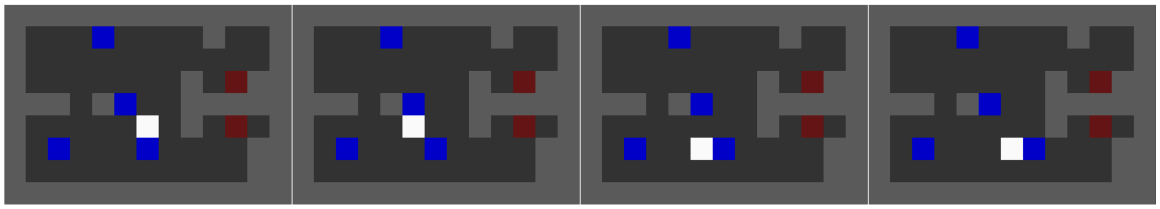

Figure 3-3: This plan was found by IW-2, but is not guaranteed to be found by IW-1. In IW-1, the planner might have already seen the white agent everywhere else before considering this sequence. Then, the second frame would be immediately pruned because the agent was already in that location and the blue box below the agent has not moved.

However, finding a way to push this block from the top right to the bottom left could mostly be done through IW-1 planning - except for the small intermediate section where we have this more complicated manipulation. One way to reduce the width of a domain is to expand the set of available actions to include higher level composite actions. For example, if we had an action "push a block right, then down", our IW-1 planner could find a complete path.

We model this ability as a mechanism to store cached action paths for these higher level tricks. Rather than building in these higher level actions as motion primitives, we propose a method to learn them automatically.

3.7.2 Representation and Planning

A higher level action (HLA) is represented by a set of involved entities , a cached action sequence, and a predicted end state for the involved entities. Higher level actions found in IW-2 can be used in IW-1, allowing IW-1 to find plans that would have otherwise required IW-2. This comes at a cost of an increased branching factor in IW-1, as the outcome of every HLA must be checked. However, we still found IW-1 with HLAs to be dramatically faster in our experiments than IW-2 as the number of HLAs was significantly smaller than the number of atoms in the environment.

On every step, the planner searches through all higher level actions to find any that match the current state. To match, the state needs to have a subset of entities that are in the same relative configuration as the set of involved entities. Potential matches are found efficiently by specifying a main entity, and looking for matching entities at the correct relative offsets from that main entity. (All entities must be at exact relative offsets from each other. Relaxing this requirement is an interesting future work direction.)

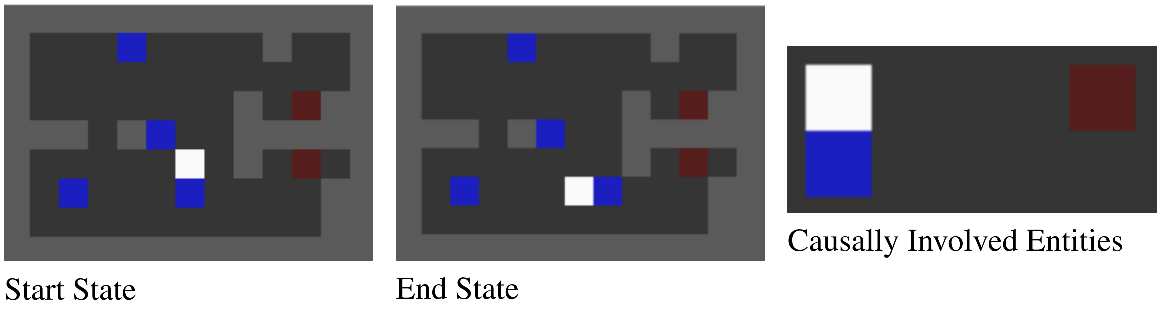

Figure 3-4: After an IW-2 plan is found, the entities that are causally involved in reaching the end state are identified. These entities are extracted, and there relative offsets are stored as the HLA matching template. Here, these three entities are involved as the white agent is pushing the blue box to the red goal.

Once a matching HLA is found, the planner simulates the result of executing that HLA on the current state. This simulation process allows the planner to ensure that every entity ends up in its expected final state, and allows it to reason about other entities that would be affected by the application of this HLA. Note that it's fine if a HLA matches, but fails to be useful because of other entities that got in the way - at worse the planner explores a few extra states (still bounded by the global IW heuristic). This simulation process is important.

For example, simply teleporting the involved entities to the end states found in the original HLA observation does not work as it would ignore the effects of other un-involved entities that would get effected by the HLA.

As future work, it would also be interesting to explore an alternative where a goal predicate representing the HLA's end state is stored instead of a cached action sequence. The agent could search within a simplified domain (such as one with only the important entities) for the goal predicate, still resulting in a speedup compared to searching in the real domain. Additionally, it would be interesting to explore the implications of a formulation that leads to the goal not appearing in the set of causally involved entities in the example in 3-4.

3.7.3 Learning Higher Level Actions

Our higher level actions are sub-plans that can be found with IW-2 but not IW-1. At a high level, there are 4 steps:

1. Find a plan with IW-2 that doesn't work with IW-1.
2. Find the shortest sub-plan that cannot be solved with IW-1. (Test every pair of starting and ending points in the overall plan and find the shortest unsolveable one.) This provides a start state and an end state .
3. At the end state, identify the entities that are causally involved in the reaching the goal of the overall plan. We denote this set as the *goal entities*. This uses the proposed causal filtering process described below.
4. At the start state, identify the entities that are causally involved in moving the goal entities from their starting states to their ending states. We denote this set as the *involved entities*, as this higher level action will match to another state if these entities appear in the same relative state. This uses the proposed causal filtering process described below.

The higher level action is then represented as a tuple of the (set of matching entities, and the sequence of actions that leads from a matching start state to a matching end state).

3.7.4 Causal Filtering Process

Given a state s and a goal predicate g , we want to identify the smallest subset of entities that are *causally involved* in reaching the goal predicate. We define a causally involved entity as "an entity whose modification would lead to the discovery of a different overall plan". More precisely, our algorithm to find causal entities is as follows:

1. Make an initial plan p from the state to the goal g
2. For each entity e :
    (a) Construct a modified state s'_e from s with entity e removed. (Or with e added back in if it disappeared in the goal end state).
    (b) Construct a new plan p'_e from s'_e to the goal
    (c) Denote this entity as causally involved if $p'_e \neq p$

3.7.5 Limitations

This process assumes that every entity can be treated independently. However, this is not necessarily the case. For example, consider a domain consisting of a single hallway with the agent to the left of two sequential doors with a goal on the right. Both doors open with the press of a single key. In the proposed process both doors would be marked as *not important*, as deleting a single one of them would still lead to the agent deciding to press the key. This is problematic, as the HLA will match to many more scenes than are relevant.

We consider two (unimplemented) solutions. Both leverage the observation that it's possible for the agent to check if it has found a superset of entities to remove (by attempting to plan), but it's only hard for the agent to confirm that this set is minimal. Let us call this an *under-specified HLA*.

First, we could search over larger groups of entities to delete simultaneously. If the HLA is under-specified with groups of size k, the agent could restart and consider deleting groups of size $k + 1$ instead.

Second, we could add an extra phase of adding back incorrectly-removed entities. If there are a group of entities that they all must be removed for a change to be observed, then adding back any entity will cause the change to disappear. Thus, we could simply loop through each removed entity and check if adding it back changes the plan; if so, we would mark it as important.

3.7.6 Consistency-Checking Higher Level Actions

The agent learns higher level actions whenever it executes a IW-2 plan. Note that higher level actions are discovered within the planner (rather than by interaction with the real environment), which leverages a potentially incorrect or incomplete world model. It's thus possible that a *faulty HLA* is discovered, leading to the HLA failing to reach the end state that it was originally planned for.

Whenever a higher level action finishes executing, it is checked for consistency against the predicted end state. If the HLA's important entities in this end state do not match their predicted location, the HLA is marked as inconsistent and removed.

3.8 Results

We test our approach on four VGDL domains: Sokoban, Bait, Preconditions, and Relational.

In *Sokoban*, the agent is tasked with removing all boxes from the domain by pushing them into holes. In *Bait*, the agent is tasked with navigating obstacles to get a key before going to a goal. In *Preconditions* the agent is tasked with collecting resources like *Medicine* to get through barriers like *Poison* to find a a path to the goal. In *Relational* the agent needs to learn how to manipulate and remove all probes with *Converters*.

Each agent was run on a single Intel Xeon Gold 6248 CPU core for at most 24 hours. We measure both the total number of environment interactions and the amount of computation time needed to solve the environment. These numbers are not perfectly correlated because the agent can spend significant amounts of time thinking between environment interactions to find new plans or synthesize new higher level actions.

The results of running our agent in each level of these domains are shown in table 3.1 below. All successful experiments finish in only a few thousand environment interactions. In some domains, the agent fails to find a plan; upon analysis this was due to the planner greedily selecting subgoals that traps it and prevents it from reaching the overall goal.

We then investigated the generalization of the learned model across different levels within a single environment. Results are shown in table 3.2. We see that transferring the world model from one level to the next reduces the amount of total learning time.

In table 3.3, we measure the generalization of the learned model across different environments. Here, we run each domain in sequence, transferring the world model and HLAs learned in earlier levels rather than re-training the expeirment from scratch in each level. We see that this transfer reduces the amount of time that the agent needs to solve in the subsequent domain.

Experiment	**Environment Interactions**	**Time**
Sokoban Level 0	3073	12 hours
Sokoban Level 1	2984	7 hours
Sokoban Level 2	1855	2 hours
Sokoban Level 3	x	x
Sokoban Level 4	493	12 min
Bait Level 0	758	7 min
Bait Level 1	x	x
Bait Level 2	x	x
Bait Level 3	x	x
Bait Level 4	x	x
Precondition Level 0	227	1 min
Precondition Level 1	1068	2 hours
Precondition Level 2	227	2 min
Relational Level 0	2857	8 hours
Relational Level 1	x	x
Relational Level 2	x	x

Table 3.1: Results in multiple levels of four VGDL domains. Number of environment interactions and total time to solve the level are listed. An x indicates that the level was not solved.

In table 3.4 we show ablations of various parts of our model, compared on level two of Sokoban and Precondition. We see that the most impactful optimizations are only considering squares with multiple values, simulating the next state, and only finding perfect programs; ablating these optimizations caused the programs to not finish. The exploration mechanisms were the next most important. The other optimizations had relatively minimal impact.

Experiment	**Interactions w. Transfer**	**Without Transfer**	**Reduction**
Sokoban Levels 0, 2, 4	3958	5021	79%
Precondition Levels 0, 1, 2	921	1522	60%

Table 3.2: We compare transferring the learned world models across levels within a domain against re-learning the world model from scratch in each levels of our VGDL domains. The "Interactions w. Transfer" column shows the total number of environment interactions that the agent needed to solve each level in the domain in sequence. The "Without Transfer" column shows the sum number of environment interactions, initializing a new agent in each level separately.

Experiment	**Interactions w. Transfer**	**Without Transfer**	**Reduction**
Sokoban Level 0 to Bait Level 0	3660	3831	95%
Sokoban Level 2 to Bait Level 0	2399	2613	91%
Sokoban Level 4 to Bait Level 0	734	1251	58%

Table 3.3: We compare transferring the learned world models across different domains against re-learning the world model from scratch in each domain. The "Interactions w. Transfer" column shows the total number of environment interactions that the agent needed to solve each domain in sequence. The "Without Transfer" column shows the sum number of environment interactions, initializing a new agent in each domain separately.

Ablation	**Sokoban Level 2**	**Precondition Level 2**
Extend Current Program	123%	111%
Only Cross Squares	105%	99%
Only Neighboring Squares	85%	100%
Only Squares with Multiple Values	x	x
Learn Higher Level Actions	136%	100%
Simulate Next HLA State	x	100%
Only Find Perfect Programs	x	x
Explore-Novel-Pair	161%	118%
Explore-Unknown-Outcome	144%	103%

Table 3.4: We perform ablations of the main experiment hyperparameters and denote the percent time longer that the ablated experiment took over the non-ablated experiment. For each domain we denote the level number that was used in the test. An x indicates that the ablated experiment did not finish.

3.9 Observations

As a result of this work, we have made a few high-level observations that we believe can be carried over to learning rules with other setups and other domains.

- **Planning With Incorrect Models Is Fine If You Can Notice You're Incorrect** In its early phases, our planner leverages a highly buggy world model. It thus occasionally finds a path that it believes will lead it to a goal, but that fails on the way. When the planner sees an unexpected outcome, it simply halts the current path and collects a new datapoint, in a Model Predictive Control -style.This is a relatively inexpensive way to explore, as it either leads to a success or the collection of data that immediately improves the model.

  Taking inspiration from an Optimism Under Uncertainty perspective, our planner tries to plan a path to the goal before falling back on trying to plan a path to improve its model. This has a another desirable property that computation time for learning rules has higher weight on the rules that will let the agent achieve its goals, rather than achieving arbitrary changes in the environment. However, it seems likely that there is still additional room for improvement here, due to still relatively formal nature in which the model plans. For example, leveraging better relaxed planning (such as choosing to ignore some entities / effects in the planner stage) could let the agent spend even more time in the phase of exploring possible paths to the goal.

- **Immediately Re-synthesizing Erroneous Programs Avoids Head-Bashing Behavior** Whenever a program makes an invalid prediction, a new datapoint is collected and the program is immediately re-synthesized. This has a nice effect in that the agent will never repeatedly make the same mistake - such as repeatedly bashing itself into a wall. (In contrast, this failure mode can appear in model-free RL agents, which is often fixed by adding a LSTM on top of the policy network. [48]). Of course, this comes at a tradeoff with the computation time needed to re-synthesize the program.

- **Programs Undergo Waves of Increasing Complexity then Reformulation** In our experiments, we observed that programs will often undergo waves of complexity. First, they grow in number of ANDRule s, layering on new hacky rules that fix individual datapoints. The program eventually hits the maximum number of allowed operations and begins resynthesizing from scratch. More general rules that capture multiple datapoints can now emerge by greedily selecting the new best rule, resulting in a smaller program overall.

These phases are reminiscent of a an *accumulating* and then *uniframing* process. New datapoints are accumulated as individual rules to remember, and then are eventually uniframed into a central framework once the datapoints make a clear statistical suggestion.

- **Over-Generalized HLAs Can Block the IW-1 Planner** We observe that over-generalized HLAs can block the IW-1 Planner. IW is very sensitive to the order in which states are explored, as this order determines the positions of the non-novel entities. This can cause it to get *blocked* if states happen to be explored in the wrong order. For example, as shown in 3-5 once an agent is observed on a square s , the planner will be unable to perform other manipulations that require the agent to again be on square s (as long as the manipulation has 1 non-novel state). This caused a problem in preliminary experiments, and also is part of the reason that we need to take include *Blockers* in our higher level actions (See 3.10.1).

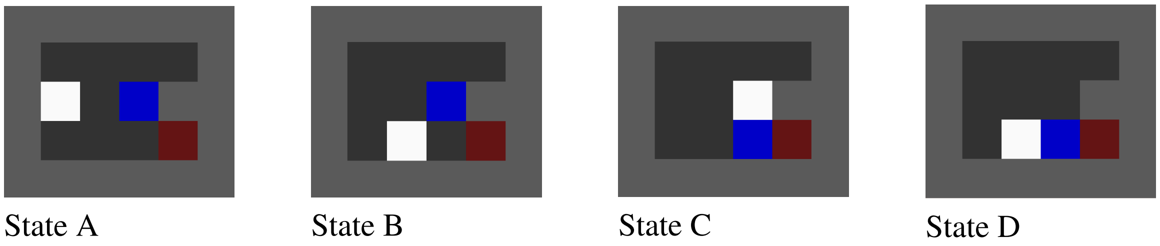

Figure 3-5: State ordering matters in IW planning. Consider the initial state A. The white square is an agent, the blue square is a pushable box, and the red square is a hole to place the box in. Imagine an event where the planner has already found state B and C. Then, state D will be pruned in IW-1 because no entities are in novel locations. State B thus interferes with state D being explored in this ordering, and a HLA that leads to state B being explored first would prevent a path to state D from being foud.

3.10 Future Work

3.10.1 *Blockers* and *Enablers*

Our procedure for finding higher level actions attempts to identify the important entities in the domain. However, it might be fruitful to use a finer-grained analysis of different entities.

One consideration is that some entities are directly involved in reaching the goal and thus act as *enablers* while other entities require the agent to find a longer path to the goal and act as *blockers*. For example, in Sokoban, the box beng pushed would be enabler, but walls in the way act as blockers. Blockers could be formally identified as entities that lead to a longer path. Blockers need to be captured as important entities to allow the HLA to match to situations where the longer plan is required. However, removing the blocker would also allow the HLA to apply more generally. It's unclear if generating extra variants of HLAs with blockers removed would lower overall computation cost or not (as a match-checking cost for each HLA is incurred in the planning process.

Enablers could alternatively be defined as the set of entities that are involved in causing the plan to succeed (rather than the set of entities that are involved in causing the plan to be found). This would be significantly faster to compute (as no replanning would be needed, instead only requiring tests with the internal model). However, blockers would not be found by this process, so an extra mechanism would need to be added to learn situations in which the HLA does not apply.

It could be useful to have a mechanism for learning situations in which HLAs do not apply due to other entities that get in the way. Currently, HLAs are simply removed if they have unexpected effects. Alternative procedures could instead collect datasets for states where the HLA fails to apply and learn a classifier for these states. However, the structure of such a classifier is unclear.

3.10.2 Finding Important Entities More Efficiently

We note that more efficient processes can be leveraged to find these important entities than raw enumeration, by considering deleting groups of multiple entities at a time. Specifically, if we have a k important entities (unknown a-priori) and n total entities, we can find the important entities in $O(\log(n)k)$ time by performing k+1 binary searches over the entities of unknown importance. k is generally significantly smaller smaller then n in all of our experiments, so this would be faster. Note that here we're measuring runtime in terms of "calls to the planner" but this scheme would potentially see further efficiency boosts by planning in simpler environments.

3.10.3 Faster Program Synthesis

Our transition classifiers are represented as programs and are learned through program synthesis. However, due to our limited DSL, they do not achieve generalization properties that programs in principle could provide. The tradeoff with expanding the DSL is of course that a larger hypothesis space leads to slower inference. It remains an exciting avenue for future work to explore ways to search through this space efficiently.

"Library Learning for Neurally-Guided Bayesian Program Induction" [13] by Elis et al. is an example of recent promising work that learns to synthesize programs more effectively and that automatically learns higher level program structures. This approach was tested on list processing and image generation problems. These problems have smooth curriculums, which this approach implicitly heavily relies on. However, we've observed that our domains also seem to yield a good curriculum when combined with our exploration strategies. For example, one could imagine low-level programs that memorize individual state transitions emerging first, before finding programs that find higher level ways to model state transitions.

One could also explore a anti-unification -style synthesis method. Pasula et al. [49] explore something similar, but do not frame their inference as program synthesis. Reframing the approach as program synthesis could plausibly allow this technique to be used while also leveraging higher level program-like structures.

3.10.4 Meta-Reasoning About Subgoals

The planner currently greedily selects the shortest path it sees to any subgoal. This can cause it to become *trapped* if a path to a subgoal is not globally optimal. As a running example, consider the case in Sokoban where a block is pushed against a wall; if this happens, the block can never be pushed back away from the wall again, trapping the agent. The agent might inadvertently cause this to happen, and then get stuck trying to rescue the block later on.

One solution would be to simply remove the notion of greedily planning to subgoals. This is not wholly unreasonable as the planner learns to plan more efficiently over time (by learning new higher level actions), so it eventually might be able to efficiently find a full path. However, IW-planning needs some notion of subgoals baked in to keep the width low. For example, Sokoban has width 3 if it has multiple boxes and no subgoals, but shrinks to width 2 if eliminating boxes are treated as a special subgoal. IW_G is a variant of IW that effectively restarts IW after a subgoal is found (by maintaining separate tables of seen IW locations for different numbers of achieved subgoals). However, this would not prevent the agent from trapping itself, without extra heuristics built in.

It seems desirable for the agent to learn meta-knowledge about its plans. One could imagine this in a few forms. First, it could learn explicit rules, such as "pushing a block against a wall means that no plan can be found to push the block to other goals". Second, this could be formulated as a value function, where states with blocks against a wall have lower value. However, the exact mechanisms are unclear and an avenue for future work.

Chapter 4

Conclusion

In this work, we show that programs can encode high level knowledge that facilitates the efficient encoding of exploration strategies and world models. We first develop an approach to automatically synthesize exploration strategies through meta-learning and program synthesis, and demonstrate that this approach yields algorithms with performance competitive to state of the art human-designed benchmarks. We then explore an approach to learn factored world models through program synthesis and achieve preliminary results, solving a few VGDL domains with minimal prior knowledge.

We see these two demonstrations as promising examples of the potential of using program synthesis techniques to increase the generalization of machine learning models in reinforcement learning. A major downside of this approach is its lack of a smooth optimization space. We present some techniques to speed up optimization (in terms of hand-designed and learned heuristics), but significant future work remains. Solving this challenge and finding efficient optimization algorithms or more malleable representations is thus an important avenue for future work to make these approaches tractable in ever more complex settings.

Bibliography

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

[2] Ferran Alet, Erica Weng, Tomas Lozano-Perez, and Leslie Kaelbling. Neural relational inference with fast modular meta-learning. In *Advances in Neural Information Processing Systems (NeurIPS) 32*. 2019.

[3] Rajeev Alur, Pavol Černỳ, and Arjun Radhakrishna. Synthesis through unification. In *International Conference on Computer Aided Verification*, pages 163–179. Springer, 2015.

[4] 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.

[5] 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*, pages 459–468. JMLR. org, 2017.

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

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

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

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

[10] 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.

[11] 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.

[12] 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.

[13] Kevin Ellis, Lucas Morales, Mathias Sablé Meyer, Armando Solar-Lezama, and Joshua B Tenenbaum. Search, compress, compile: Library learning in neurally-guided bayesian program learning. In *Advances in neural information processing systems*, 2018.

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

[15] Kaiser et al. Model based reinforcement learning for atari. *ICLR 2020*.

[16] Lipovetzky et al. Width and serialization of classical planning problems. *ECAI 2012*.

[17] Tsividis et al. Human-level reinforcement learning through theory-based modeling, exploration, and planning.

[18] Veerapaneni et al. Entity abstraction in visual model-based reinforcement learning.

[19] Weber et al. Imagination-augmented agents for deep reinforcement learning. *NIPS 2017*.

[20] 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.

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

[22] 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, editors, *Advances in Neural Information Processing Systems 28*, pages 2962–2970. Curran Associates, Inc., 2015.

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

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

[25] Carlos Florensa, David Held, Xinyang Geng, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1515–1528, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[26] 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)*, pages 3965–3972. IEEE, 2016.

[27] 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.

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

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

[30] Santiago Gonzalez and Risto Miikkulainen. Improved training speed, accuracy, and data utilization through loss function optimization. *arXiv preprint arXiv:1905.11528*, 2019.

[31] Santiago Gonzalez and Risto Miikkulainen. Evolving loss functions with multivariate taylor polynomial parameterizations, 2020.

[32] 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*, pages 5400–5409, 2018.

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

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

[35] Ken Kansky, Tom Silver, David A Mély, Mohamed Eldawy, Miguel Lázaro-Gredilla, Xinghua Lou, Nimrod Dorfman, Szymon Sidor, Scott Phoenix, and Dileep George. Schema networks: Zero-shot transfer with a generative causal model of intuitive physics. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1809–1818. JMLR. org, 2017.

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

[37] 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*, pages 195–202. ACM, 2017.

[38] 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.

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

[40] Louis Kirsch, Sjoerd van Steenkiste, and Jürgen Schmidhuber. Improving generalization in meta reinforcement learning using learned objectives. *arXiv preprint arXiv:1910.04098*, 2019.

[41] 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*, pages 3675–3683, 2016.

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

[43] 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.

[44] Joseph Marino, Milan Cvitkovic, and Yisong Yue. A general method for amortizing variational filtering. In *Advances in Neural Information Processing Systems*, pages 7857–7868, 2018.

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

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

[47] 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.

[48] Emilio Parisotto and Ruslan Salakhutdinov. Neural map: Structured memory for deep reinforcement learning. *arXiv preprint arXiv:1702.08360*, 2017.

[49] Hanna M Pasula, Luke S Zettlemoyer, and Leslie Pack Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352, 2007.

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

[51] 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*, pages 16–17, 2017.

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

[53] 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.

[54] 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.

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

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

[57] Ronald L Rivest. Learning decision lists. *Machine learning*, 2(3):229–246, 1987.

[58] Tom Schaul. A video game description language for model-based or interactive learning. *Proceedings of the IEEE Conference on Computational Intelligence in Games*.

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

[60] 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*, pages 48–76. Springer, 2008.

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

[62] 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.

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

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

[65] 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*, pages 2753–2762, 2017.

[66] 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*, pages 5026–5033. IEEE, 2012.

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

[68] 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.

[69] 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*, pages 229–236. ACM, 2018.

[70] Chang Ye, Ahmed Khalifa, Philip Bontrager, and Julian Togelius. Rotation, translation, and cropping for zero-shot generalization. *arXiv preprint arXiv:2001.09908*, 2020.

[71] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232, 2017.

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