BRANCHING BOOTSTRAPPED ABSTRACTION DISCOVERY

Aditya Ganeshan
adityaganeshan@gmail.com

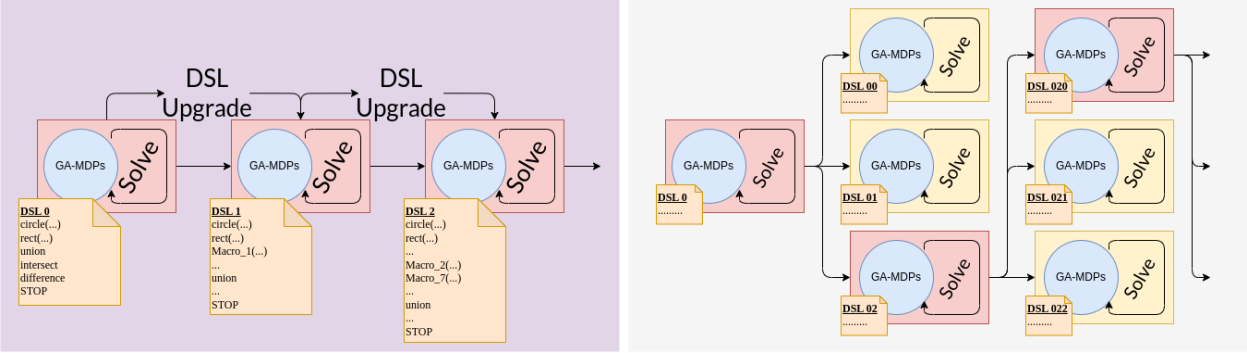

Figure 1: Library learning can be modeled as a sequential transition between related MDPs. Previous Library learning techniques iterate between a learning phase to (approximately) solve the MDP, and a compression phase to transition to the next MDP which consists of a better library. **(left)** Previous library learning methods greedily update the MDP which can result in convergence to a sub-optimal library of functions. **(right)** In contrast, we propose performing branching parallel MDP transitions, which will allow us to avoid convergence to a sub-optimal library.

1 Introduction

Programs offer a way to represent the underlying procedure to generate visual data, and are used widely across industrial tasks such as fabrication and procedural content generation. Access to a visual datum's programmatic representation helps with various downstream tasks such as editing and structural analysis. However, manual designing programs can be expensive and in most cases infeasible. Learning to automatically infer visual programs offers a solution, and many recent techniques effectively employ neural networks with symbolic techniques towards this goal [1, 2].

While these results are impressive, such techniques often work on a highly specialized DSL which have been hand crafted to simply the task and restrict the search space of valid programs. In the absence of such a *library* of specialized functions, which is often the case, these approaches have limited success. To solve this predicament, previous works have proposed Library Learning approaches [3, 4] to discover a specialized DSL for the given task. They generally consist of two phases: a SOLVE phase, where the VPI task is modelled as a Markov Decision Process (MDP) and a policy network π is trained to generate reward-maximizing trajectories in the MDP; and a DSL-UPGRADE phase, where the DSL is upgraded, resulting in a new but related MDP to solve. Such approaches have however experienced a limited use due to two reasons: (a) the convergence of such procedures to a sub-optimal library and (b) the high cost of such procedures.

In this work, we propose *Branching Bootstrapped Abstraction Discovery* (BRANCHING BAD), a library-learning system designed to alleviate both of these issues. First, we note that previous library learning systems can converge to sub-optimal library due to greedily integrating discovered abstractions. Once a discovered abstraction is integrated into the MDP's action space, it influences all the subsequent SOLVE and UPGRADE steps. Poor abstractions (even when eventually discarded), can harm subsequently discovered abstractions as well as lower the policy network's ability to solve the MDP, resulting in a lower quality library. To circumvent this issue, we propose to perform a branching search of abstractions to integrate. At each DSL-UPGRADE step, we stochastically select abstraction candidates, and create k new MDPs with k different DSLs for the solving (over all the previous tasks). After solving each of the k branches

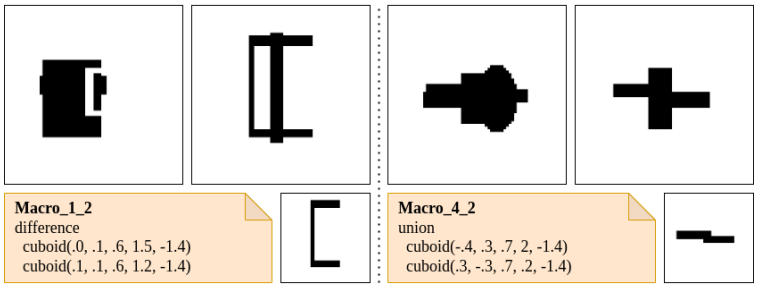

Figure 2: We show two examples of macro-actions discovered during the abstraction discovery process. Each expression is accompanied by two example input shapes which are constructed using the discovered macros, and the execution of the macro itself to the right of the macro expression.

in parallel, the best branch is selected and subsequently branched into k new MDPs in the next DSL-UPGRADE step. This allows us to 'try' different abstraction candidates, in turn helping us reject poor abstractions without harming the library's evolution. Additionally, the ability to 'try' different abstraction candidates, allows us to use cheaper yet noisier abstraction discovery techniques since poor abstractions can be easily rejected. In Figure 1, we compares prior abstraction discovery methods (left) to our proposed method (right) emphasizing the branching search used in our method.

Secondly, we identify and rectify the computational bottleneck of MDP-to-MDP transitions. Typically, updating the DSL leads to a change in the MDP's action space, requiring architectural changes to the policy network π. The architectural changes necessitates re-initialization of some parameters of the network, leading to longer training times and higher cost of solving each MDP. To alleviate this issue, we leverage the separation between token-space and network parameters in the transformers architecture [5] and facilitate effective transfer learning between MDPs. With our model, each DSL UPGRADE step only modifies the token-space allowing re-use of the learnt policy without any parameter re-initialization, thus reducing the cost of solving the new MDPs.

We use BRANCHINGBAD on the 2D CSG domain [6], to craft a library of 2D CSG abstractions, as well as infer 2D CSG programs for novel test inputs. Our results show that bootstrapped library learning is indeed superior to VPI methods which do not update their DSLs. More importantly, we show that BRANCHINGBAD is superior to naive greedy abstraction discovery. With BRANCHINGBAD, we are able to infer programs which more closely reconstruct the input shape, while being shorter. Furthermore, frequently used abstractions discovered with BRANCHINGBAD seem to capture objects semantics such as "chair-back" or "table-base". Figure 2 presents some of the discovered abstractions, and their usage in test inputs.

In summary, we present BRANCHINGBAD, a novel bootstrapped library learning system which uses branched search and efficient policy adaptation to induce a high quality library for visual program inference task. Fundamentally, our method leverages stochastic search with rejection to surpass prior greedy approaches. We hope this work leads to further investigation into integrating backtracking stochastic search techniques with library learning techniques.

2 Related Work

Visual Program Inference: (VPI) is a sub-problem within program synthesis. Program synthesis is a storied field, with roots back to the inception of Artificial Intelligence, where the objective is to produce programs that meet some specification [7]. Under our framing, the specification is an input visual datum that the synthesized program should reconstruct when executed.

Key challenge in visual program inference is the presence of discrete elements, which complicates end-to-end learning approaches. None the less, prior works have proposed end-to-end learning methods with program relaxation [2, 8, 9] and by employing noisy neural executor [10, 11]. Other approaches have attempted visual program inference with Reinforcement learning [6, 12] as well as bootstrapped learning methods [1, 13]. However, all the approaches have previously been employed only with a fixed DSL, in contrast in this work, we aim learning a rich DSL, while also learning VPI in an unsupervised fashion.

Bayesian library learning: Typically, library learning approaches find common sub-computations shared across many programs and factor them out into subroutines (i.e. abstractions), for future usage. When a dataset of programs is given as input, this step can be decoupled from visual program inference [4, 14, 15, 16]. Alternatively, some methods have investigated how abstraction discovery (AD) phases can improve visual program inference performance [3, 17]. In an

iterative procedure, an abstraction phase greedily rewrites a dataset of programs with abstractions, then a recognition model learns on rewritten programs to discover higher-order abstractions and solve more complex inference problems. In this work we extend EC2 [17] addressing its drawbacks such as high computational requirement and local convergence of library due to greedy library updates.

Abstraction in RL: Hierarchical decision processes have been studied in RL in broadly two classes. *Discovering Options*: Options encapsulate a multi-step action sequence by defining the tuple of Initiation Set, option policy, and termination condition. Prior works have discovered options using various approaches such as Option-Critic [18] and Skill-chaining [19]. In this work, we focus on simply introducing macro-actions as new primitive actions, rather than as options. Though this makes the framework weaker, we hope extend it to the richer formulation of options in the future. *Feudal methods*: These approaches learn a hierarchy of *managers* who assign sub-goals to *workers*. Our work does not approach abstraction in this framework.

Non-stationary MDPs: Another line of work [20, 21] investigates reinforcement learning with non-stationary Markov decision process, where action spaces changes over time. Notable, our efficient policy adaptation formulation is similar in spirit to LAICA [21], where the policy parameterization is split into two parts, β which requires no re-initialization, and $\hat{\phi}$ which is modified when the MDP changes. A noticeable drawback of LAICA is the requirement of supervised learning updates (via variational auto-encoding) specifically for updating $\hat{\phi}$, apart from RL updates for learning β. In contrast, our method learns both the token space and the policy parameterization together in the bootstrapped learning process.

3 Preliminaries

Goal Conditioned Reinforcement Learning: We can model sequential decision making problems as Markov Decision Processes (MDPs). An MDP can be described as a five-tuple $\langle S, A, P, R, \gamma \rangle$, where S is a finite set of states; A is a finite set of actions; P is the transition function; R is a bounded reward function, and $\gamma \in [0, 1]$ is a discount factor. Formally, we learn a policy $\pi : S \to A$ to maximize the expected cumulative returns:

$$J(\pi) = \mathop{\mathbb{E}_{a_t \sim \pi(.|s_t)}}_{s_{t+1} \sim P(.|s_t, a_t)} \Big[\sum_t \gamma^t R(s_t, a_t) \Big] \tag{1}$$

While in standard RL settings, only a single reward function is sufficient, in many cases the learnt policy π must be *conditional* to certain inputs (or goals). Towards that end, MDPs have be extended to goal augmented MDPs (GA-MDPs) by including an extra tuple $\langle \mathcal{G}, p_g, \phi \rangle$, where $\mathcal{G}$ denotes a finite set of Goals, p_g describes a distribution over the goals, and $\phi : S \to \mathcal{G}$ describes a mapping function from the state to a specific goal. Additionally, we modify the policy π to be a mapping $\pi : S \times G \to A$ from current state and goal to actions in A.

Visual Program Inference as an MDP: We define the visual program inference (VPI) task as follows: given a target distribution S of visual inputs (e.g. shapes), we want to learn a policy $\pi_\theta(z|x)$, where $x \sim S$, which infers a program z whose execution $E(z)$ reconstructs the input shape x. Following Occam's razor, we seek programs that are parsimonious. More formally, we seek a π_{θ^*} that maximizes our objective function $\mathcal{R}$:

$$\theta^* = argmax_\theta \mathbb{E}_{x \sim S} \Big[\sum_z \mathcal{R}(x, z) \pi_\theta(z|x) \Big], \tag{2}$$

$$\mathcal{R}(x, z) = recon(x, E(z)) - \alpha |z|, \tag{3}$$

where $recon$ measures the reconstruction accuracy between x and $E(z)$, $|z|$ measures the length of the program, and α modulates the strength of these two terms.

We can model VPI as an MDP with the following mapping: The state S is described as the $\mathbb{R}^n$ space in which the shapes x are described in, the actions A are tokens in the domain specific language, P is encoded in the executor E, and the reward R is mapped to our objective $\mathcal{R}$. We can further extend this formulation to GA-MDPs by mapping G as $\mathbb{R}^n$, and state-to-goal mapping ϕ as the identity function. For example, in the case of inferring 2D CSG programs, the state (and goal) spaces are in $\mathbb{R}^{H \times W}$, and the actions are in $A = \{union, intersection, ..., rectangle(x, y), circle(x, y), ...\}$.

Learning VPI policies: Recently, by modelling RL as a sequence modelling problem, prior works [22] have shown that large auto-regressive models [23] can be employed to learn reward maximizing policies. Specifically, such approaches learn to replicate expert trajectories $\tau = (RTG_0, s_0, a_0, ...RTG_T, s_T, a_T)$ consisting of the return to go ($RTG_t = \sum_t^T R_t$), states and actions values. The policy π uses the previous tokens to predict actions a_t and is optimized by a mean squared error loss or maximum likelihood loss between the expert actions and predicted actions.

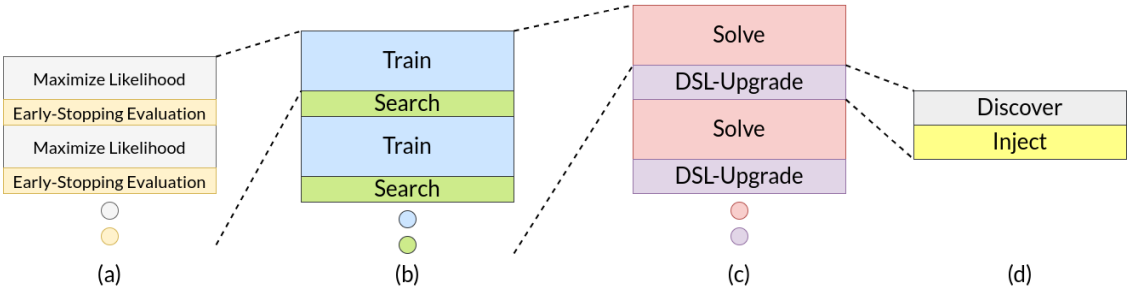

Figure 3: We describe each component of our sequential bootstrapped abstraction discovery process. **(c)** depicts the overall process of applying SOLVE and DSL-UPGRADE sequentially (cf. Section 4.1). **(b)** shows how the SOLVE step is composed of the SEARCH and TRAIN steps which are applied sequentially. **(a)** shows the internals of TRAIN which involves maximizing likelihood on searched programs which checking the early stopping criterion (cf. Section 4.2). Finally **(d)** shows the two steps of DSL-UPGRADE namely abstraction discovery and abstraction injection (cf. Section 4.3).

When a labelled dataset of expert trajectories is given, this approach can learn to replicate the expert trajectories and show acceptable generalization to novel test tasks. However, in many cases, such expert trajectories are not available. Hence, prior works [1, 13], have introduced bootstrapped-learning paradigm, where the policy π is trained on its own trajectories. Such approaches alternate between SEARCH phases, that discover 'good' programs with the policy π, and TRAIN phases, that use discovered programs to train the policy itself. In this project, we will use this paradigm for learning reward maximizing policy π on each given MDP.

Library Learning: To make the VPI task easier, it is desirable to have a well engineered DSL which exposes only useful task-specific DSL commands as actions. However, designing such highly domain specific DSLs is challenging and time-consuming. Instead, library-learning approaches [4, 3, 17] aim to make VPI easier by discovering useful *action-abstractions* and introducing them to the DSL.

Viewed as a probabilistic inference problem, we can describe the objective of library learning as follows:

$$J(\mathcal{D}, \theta) = \mathbb{P}[\mathcal{D}, \theta]\mathbb{E}_{x\sim S}\Big[\sum_{z\in\mathcal{D}} \mathcal{R}(x, z)\pi_{\theta,\mathcal{D}}(z|x)\Big], \tag{4}$$

$$\mathcal{D}^* = argmax_{\mathcal{D}} \int J(D, \theta)d\theta, \qquad \theta^* = argmax_\theta J(D^*, \theta), \tag{5}$$

where $\mathcal{D}$ describes the DSL. To learn DSLs with action abstractions, we first perform a learning phase (the search-train loop or explore-compile loop [17]), followed by a compression phase, where macro actions are discovered and introduced into the DSL. During the compression phase, these macro actions are often detected by domain specific heuristics and their inclusion into the DSL is decided on the basis of their length ($\mathbb{P}(\mathcal{D}) \propto \exp(-\lambda \sum_{p\in\mathcal{D}} size(p))$ and usefulness ($\propto \mathcal{R}(x, z)\pi_{\theta,\mathcal{D}}(z|x)$).

4 Approach

In this section, we describe our approach as follows: first, we formally describe library learning as a sequential transition between related MDPs which we solve by the SOLVE and DSL-UPGRADE steps. We then describe each of these two steps in detail. Finally, we present BRANCHING BAD focusing on the two improvements - efficient policy adaptation and branching search of abstraction integration.

4.1 Library learning via sequential MDP transitions

As described in Section 3, given a fixed DSL $\mathcal{D}$, we can view the task of learning an unsupervised VPI network as a goal-conditioned reinforcement learning task. At the start of the library learning process, our DSL $\mathcal{D}$, which contains primitive actions is denoted by $\mathcal{D}_0$. We denote the corresponding MDP as $\text{MDP}_{\mathcal{D}_0}$.

First, given $\text{MDP}_{\mathcal{D}_i}$, in the SOLVE step, we train a policy network π to discover and learn objective maximizing trajectories, conditioned on inputs $x \sim S$, i.e. objective maximizing programs under the DSL $\mathcal{D}_i$. In this step the policy parameters θ are updated with stochastic gradient descent to maximize the objective function $\mathcal{R}$ (refer equation 5). This step is performed using SEARCH-TRAIN bootstrapped learning method, and described in detail in Section 4.2.

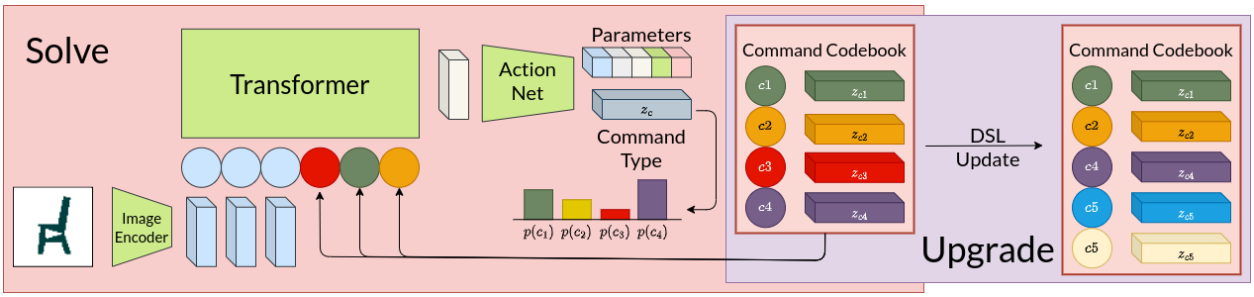

Figure 4: We achieve efficient policy adaptation by decoupling command selection from policy parameterization. The command probabilities are computed using the cosine distance between the embeddings for each command in the code-book and the command vector z_c produced by the network (in green). During the SOLVE step both the command code-book and the policy (in green) are updated. During the DSL-UPGRADE step only the command code-book is updated, with pruning of poorly performing macros and inclusion of new macros.

Once the SOLVE step converges, we start the DSL-UPGRADE step. In this step, given $\text{MDP}_{\mathcal{D}_i}$ and learnt policy π, we first retrieve a set of objective maximizing programs for $x \sim S$. Then, an abstraction discovery process is used to detect frequently occurring sub-programs. Such subprograms are then incorporated into $\text{MDP}_{\mathcal{D}_{i+1}}$ as novel parameterized macro-actions. We additionally prune out poorly performing macros, and perform abstraction injection, both of which are described in detail in Section 4.3.

In this way, each DSL-UPGRADE step leads to a new MDP with a new DSL $\mathcal{D}$, which is then used in the SOLVE step to discover better programs. The interleaved application of these two steps is depicted in Figure 3 (c). All prior library learning approaches [17, 4, 3] follow this direction and we show this in Figure 1 (left).

4.2 SOLVE Step

For the SOLVE step we follow the framework outlined in PLAD [1]. In this approach, we train the network by searching for "good" programs, and maximizing their likelihood. First, the policy network π along with a simple search procedure such a beam-search is employed to find objective maximizing programs z_i for a set of inputs $x_i \sim S$. This is termed the SEARCH step. Second, given a set of such objective maximizing programs, the policy network is then trained to increase their likelihood $\theta^* = argmax_\theta E_{(z_i,x_i)} p(z|x)$. This is termed the TRAIN step. These two steps are then interleaved until the process converges. The interleaved application of these two steps is depicted in Figure 3 (b).

Note that naively integrating these two steps can lead to over-fitting to sub-optimal programs. To alleviate this issue, we 1) use a validation set of input to perform early stopping of the TRAIN step, and 2) increase the likelihood of multiple approximate distributions given be (x_i, z_i) and $(E(z_i), z_i)$, i.e., the policy network is trained to predict z_i given the execution $E(z_i)$ of z_i as input. This process is depicted in Figure 3 (a).

4.3 DSL-UPGRADE Step

The DSL-UPGRADE step aims to modify the DSL D_i, changing the MDP's action space by introducing or pruning macro-actions. Our goal is to capture commonly occurring action sequences in a set of trajectories predicted by the policy network, and create macro-actions which can execute into such sequences. These macro-actions create a hierarchy of actions which can then be used by the agent to more effectively maximize the objective $\mathcal{R}$. Towards this end, the DSL-UPGRADE step is performed with the following two actions

Abstraction-Discovery: First, given the trained policy π, we recover a set of objective-maximizing programs $z_i \in Z$. Given this set of programs Z, we search for commonly occurring sub-expressions $\hat{z}$ in the programs. Naively searching for equivalence between expressions can be prohibitively expensive, therefore, we instead rely on approximate execution equivalence. All the sub-expressions are evaluated on a set of input values (in our case a $(64, 64)$ grid of coordinates spanning the input space), and commonly occurring sub-expressions are detected by tracking cache collisions in a cache of sub-expressions. Furthermore, we evaluate each sub-expression in a canonical frame allowing us to detect matches under translation and scale invariance. Then, the sub-expressions are sorted according to the expected benefit of converting each into a macro-action (based by their length and frequency over Z). Finally, the top-k sub-expressions are selected to be included as new macro actions for the next MDP.

Abstraction-Injection: While abstraction discovery introduces new macro actions to perform, the policy network naively may choose to not use them, especially since it has learnt to maximize the likelihood of the previous set of

	$\|DSL\|$ (↓)	IoU (↑)	Avg. $\|z\|$ (↓)	Objective $\mathcal{J}$ (↑)
PRETRAINED	**6**	60.2	5.89	50.76
SEARCH-TRAIN	6	77.8	5.757	68.85
NAIVE BAD	18	74.3	**4.53**	65.70
BRANCHING BAD	15	**78.1**	4.65	**69.62**

Table 1: We compare BRANCHING BAD against other approaches for VPI. PRETRAINED denotes the model only trained on synthetic program samples (generated from the primary DSL grammar). SEARCH-TRAIN denotes the model which learns without creating new macro actions. NAIVE BAD denotes the model which performs Bootstrapped Abstraction Discovery (BAD) in a sequential greedy approach.

actions. To aid the adaptation of the policy to the new set of macro actions we search for programs with the new macro actions which are better than previously discovered programs. This search is conducted via "code-grafting", a program rewriting method where sub-expressions are *spliced* into pre-existing programs. We use a brute-force splicing of the new macros over all the programs in Z, rejecting the spliced programs if they decrease the objective $\mathcal{R}$. These spliced programs are then used during the TRAIN step if they are better than the programs discovered during the SEARCH step.

Note that this brute-force search can be cheaply performed when program executions can be batched, as in our case. As noted in other works, over use of rewriting techniques can result in poor training of the policy network. Therefore, we use a injection factor $\Gamma \in [0, 1]$ (typically set to 0.5) to control the amount of rewrite injection.

4.4 Branching Search for Abstraction

Performing sequential greedy updates to the DSL can lead to convergence to a local maxima. Instead, we propose to perform multiple parallel DSL updates to branch into different DSLs (and their corresponding MDPs) in parallel. Towards this end, given a DSL $\mathcal{D}_i$, we create β new DSLs (where β is the branching factor). To create the β new DSLs, we modify the DSL-UPGRADE step to stochastically sample abstraction candidates from the top $k \times \beta$ sub-expressions. These β DSLs and their corresponding MDPs are then solved in parallel (in practice, due to engineering bottleneck we do this sequentially). Once all the β branched have converged with their respective DSLs, we select the best branch, based on the objective $\mathcal{J}$ (cf. equation 2) as $\mathcal{D}_{i+1}$. From $\mathcal{D}_{i+1}$, β new branches are again created for the SOLVE step, and this process is followed until convergence. This is depicted in Figure 1 (right).

We note that this process effectively decreases the computational efficiency of the method by a factor of $1/\beta$. On the other hand, with the ability to sample and reject macro-actions with minimal repercussions on the evolution of the DSL, we are able to use weaker (and cheaper) abstraction discovery technique (as described in section 4.3), thereby allowing to to still discover a useful DSL, despite using weaker abstraction discovery technique.

4.5 Efficient Policy Adaptation

For any given MDP, we have to learn a policy network π to infer the reward maximizing program. However, learning the network from scratch is expensive. Instead, we leverage two heuristics to accelerate learning on the given DSL. First, we initiate the weights of the new policy from the previous policy. Secondly, we modify how the policy π infers the program to enable rapid adaptation to modified action space. Specifically, the policy decodes program tokens by predicting a command embeddings z_c for each time-step (along with parameters which are optionally used based on the type of command). Additionally, we store a similarly sized embedding for each DSL token in a command code-book $C = \{z_i | i \in \mathbb{N}\}$. Actions at a given time step are then selected by using the command embedding z_c as a query, and the DSL token embeddings as keys as follows:

$$a = argmax_{z_i \in C} \textit{softmax}(z_c z_i^T). \tag{6}$$

Hence, when the DSL is changed, we are only required to change the command code-book, while keeping the network architecture constant. In contrast, prior works require changing at least the network's last layer to adapt it to new DSLs. Figure 4 shows our policy and action architecture which enables effective adaptation.

β	$\|DSL\|$	IoU (↑)	Avg. $\|z\|$ (↓)	$\mathcal{J}$ (↑)
1	18	74.3	**4.53**	65.70
3	15	**78.1**	4.65	**69.62**

k	$\|DSL\|$	IoU (↑)	Avg. $\|z\|$ (↓)	$\mathcal{J}$
1	**15**	**75.0**	5.55	65.17
3	18	74.3	4.53	**65.70**
5	20	72.9	**4.23**	64.55

Table 2: We perform ablation on the key features of our approach. Left: We vary β, the branching factor of BRANCHING BAD. Right: We vary k, the number of new macro-actions introduced in each DSL-UPGRADE step.

Γ	$\|DSL\|$	IoU	Avg. $\|z\|$	$\mathcal{J}$
0	**13**	74.0	5.50	64.45
0.5	18	**74.3**	**4.53**	**65.70**
1	15	73.6	4.73	65.0

θ_{update}	$\|DSL\|$	IoU	Avg. $\|z\|$	$\mathcal{J}$
NIL	18	**74.3**	4.53	**65.70**
Action net	20	72.91	**4.32**	64.43
+ Transformer	**9**	56.1	5.38	56.01

Table 3: Left: We perform ablation on Γ, the abstraction injection rate (refer Section 4.3. Right: We perform ablation on different levels of policy re-initialization after each DSL-UPGRADE step.

5 Experiments

5.1 Details

Domain: We evaluate our method on the 2D CSG domain [6]. The primary DSL of 2D CSG is equipped with only 6 commands (*circle*, *rectangle*, *union*, *intersection*, *difference*, *stop*). Parameterized commands (such as *circle* take 5 additional parameters namely 2d translation, 2D scale and 1 rotation parameter, whereas the Boolean commands (such as *union*) require two expressions as input. With this DSL, programs are written in polish notation. Macro actions are also added in the form of parameterized commands, and we show examples in Figure 2. For learning the VPI network and the DSL, we use the CAD dataset [6] which consists of 10000 train, 1000 val and 1000 test images of 2D projection of manufactured objects such as chairs and tables.

Metrics: We evaluate the reconstruction accuracy of the inferred programs in terms of Intersection over Union (IoU) and 2D Chamfer Distance between the program's execution and the target. We evaluate the conciseness of program in terms of their length. Finally the overall score is calculated as $\mathcal{J} = \eta \times |DSL| + \mathcal{R}$, where η is set to 0.001. Note that $\mathcal{R}$ accounts for both the reconstruction accuracy as well as program conciseness.

Baselines: We compare our work against three baselines: a) PRE-TRAINED model: We train the policy network on random programs sampled from the primary DSL $\mathcal{D}_0$, b) SEARCH-TRAIN Model: This model keeps the DSL $\mathcal{D}_0$ fixed and learns to (approximately) solve the GA-MDP without introducing novel macro actions, and finally c) NAIVE BAD Model: This model is trained with a non-branching bootstrapped abstraction discovery method as described in Section 4.1.

5.2 Results

First, we compare BRANCHING BAD to other baselines on the 2D CSG domain. We report the results in Table 1. First, we note that both the abstraction discovery methods are superior to SEARCH-TRAIN in terms of the length of the program. Further, while SEARCH-TRAIN is able to surpass the reconstruction accuracy of NAIVE BAD, it does so with longer programs. Finally, we see that BRANCHING BAD is superior to both SEARCH-TRAIM and NAIVE BAD. This demonstrates that the branching search allows us to discover more suitable abstractions over a greedy sequential approach.

5.3 Ablations

We first perform ablation on the two central design factors in our method, the branching factor β, and the number of macros added in each DSL-UPGRADE step k. We report both our experiments in Table 2. First, in Table 2 (left) we see that the performance (score $\mathcal{J}$) increase as we use more branches. We expect this trend to continue as the ablation with $k = 5$ is completed. Currently the computational cost increases linearly w.r.t. the branching factor β, which may limit the number of branches we can feasibly try. Table 2 (right) reports the ablation on k. Due to computational limitations, we perform our ablative studies on NAIVE BAD. We see that $k = 3$ and $k = 1$ are superior to $k = 5$. Introducing many

macro actions at once increase the complexity of the MDP without helping the policy network discover more optimal solution resulting in poor overall quality.

Abstraction Injection Rate: Another key component of our approach is the abstraction injection which is used in the DSL-UPGRADE step to create new programs which utilize the newly discovered macros. We use an injection rate Γ to control the amount of injection performed. When $\Gamma = 0$ we perform on abstraction injection, i.e. while the policy can use the new macro actions, it does not have novel examples programs which use the new macros. On the other hand, when $\Gamma = 1.0$ we introduce as many objective maximizing programs as possible (using the code-grafting technique). We report the ablation in Table 3 (left), noting that $\Gamma = 0.5$, performs better than both the alternatives.

Policy Adaptation Ablation: To verify the benefits of our approach for efficient policy adaptation (i.e. no re-initialization of the network parameters), we compare out method to two ablations: 1) where the Action-Net is reinitialized after each DSL-UPGRADE step, and 2) where the Action-Net and the transformer are reinitialized after each DSL-UPGRADE step. The report the ablation in Table 3, noting that larger re-initialization results in lower performance (with a training failure when the transformer is reinitialized every each DSL-UPGRADE step). This shows that benefit of using our policy adaptation method which requires in parameter re-initialization.

6 Conclusions

We introduced BRANCHING BAD, a new abstraction discovery method which combines low-cost abstraction discovery techniques with a branched search process to efficiently learn a library of useful macros/functions. We also introduced a network architecture which decouples policy parameterization from action selection, aiding effective policy adaptation between MDP transitions. Our method is able to surpass naive bootstrapped abstraction discovery both in terms of the quality of the library as well as the objective score maximized by the learnt policy π. A major drawback of the current formulation is the rejection of all the computation performed in the $k - 1$ branches which get rejected. Instead, macros from all the branches may be viable candidates and in future we hope to investigate how to merge different branches to speed up abstraction discovery.

References

[1] R. Kenny Jones, Homer Walke, and Daniel Ritchie. Plad: Learning to infer shape programs with pseudo-labels and approximate distributions. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[2] Kacper Kania, Maciej Zieba, and Tomasz Kajdanowicz. Ucsg-net- unsupervised discovering of constructive solid geometry tree. In *Advances in Neural Information Processing Systems*, 2020.

[3] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021.

[4] R. Kenny Jones, David Charatan, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. Shapemod: Macro operation discovery for 3d shape programs. *ACM Transactions on Graphics (TOG), Siggraph 2021*.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[6] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. Csgnet: Neural shape parser for constructive solid geometry. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[7] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 2017.

[8] Fenggen Yu, Qimin Chen, Maham Tanveer, Ali Mahdavi Amiri, and Hao Zhang. Dualcsg: Learning dual csg trees for general and compact cad modeling, 2023.

[9] Daxuan Ren, Jianmin Zheng, Jianfei Cai, Jiatong Li, Haiyong Jiang, Zhongang Cai, Junzhe Zhang, Liang Pan, Mingyuan Zhang, Haiyu Zhao, and Shuai Yi. Csg-stump: A learning friendly csg-like representation for interpretable shape parsing. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

[10] Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. Learning to Infer and Execute 3D Shape Programs. In *ICLR*, 2019.

[11] Yiwei Hu, Paul Guerrero, Milos Hasan, Holly Rushmeier, and Valentin Deschaintre. Node graph optimization using differentiable proxies. In *ACM SIGGRAPH Conference Proceedings*, 2022.

[12] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. In *NeurIPS*, 2019.

[13] Chen Lian, Jonathan Berant, Quoc V. Le, Ken Forbus, and Ni Lao. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, 2017.

[14] Yuezhi Yang and Hao Pan. Discovering design concepts for CAD sketches. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[15] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. Babble: Learning better abstractions with e-graphs and anti-unification. *Proc. ACM Program. Lang.*, (POPL), 2023.

[16] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proc. ACM Program. Lang.*, (POPL), 2023.

[17] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Joshua B. Tenenbaum. Library Learning for Neurally-Guided Bayesian Program Induction. In *NeurIPS*, 2018.

[18] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, 2017.

[19] Akhil Bagaria and George Konidaris. Option discovery using deep skill chaining. In *International Conference on Learning Representations*, 2020.

[20] Yash Chandak, Shiv Shankar, Nathaniel D. Bastian, Bruno Castro da Silva, Emma Brunskil, and Philip S. Thomas. Off-policy evaluation for action-dependent non-stationary environments, 2023.

[21] Yash Chandak, Georgios Theocharous, Chris Nota, and Philip S. Thomas. Lifelong learning with a changing action set, 2020.

[22] Hiroki Furuta, Yutaka Matsuo, and Shixiang Shane Gu. Generalized decision transformer for offline hindsight information matching, 2022.

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017.