Learning to Synthesize Programs as Interpretable and Generalizable Policies

Dweep Trivedi[*] [†] **Jesse Zhang**[*] [1] **Shao-Hua Sun**[1] **Joseph J. Lim**[‡] [1]
[1]University of Southern California
`{dtrivedi, jessez, shaohuas, limjj}@usc.edu`

Abstract

Recently, deep reinforcement learning (DRL) methods have achieved impressive performance on tasks in a variety of domains. However, neural network policies produced with DRL methods are not human-interpretable and often have difficulty generalizing to novel scenarios. To address these issues, prior works explore learning programmatic policies that are more interpretable and structured for generalization. Yet, these works either employ limited policy representations (e.g. decision trees, state machines, or predefined program templates) or require stronger supervision (e.g. input/output state pairs or expert demonstrations). We present a framework that instead learns to synthesize a program, which details the procedure to solve a task in a flexible and expressive manner, solely from reward signals. To alleviate the difficulty of learning to compose programs to induce the desired agent behavior from scratch, we propose to first learn a program embedding space that continuously parameterizes diverse behaviors in an unsupervised manner and then search over the learned program embedding space to yield a program that maximizes the return for a given task. Experimental results demonstrate that the proposed framework not only learns to reliably synthesize task-solving programs but also outperforms DRL and program synthesis baselines while producing interpretable and more generalizable policies. We also justify the necessity of the proposed two-stage learning scheme as well as analyze various methods for learning the program embedding. Website at `https://clvrai.com/leaps`.

1 Introduction

Recently, deep reinforcement learning (DRL) methods have demonstrated encouraging performance on a variety of domains such as outperforming humans in complex games [1–4] or controlling robots [5–11]. Despite the recent progress in the field, acquiring complex skills through trial and error still remains challenging and these neural network policies often have difficulty generalizing to novel scenarios. Moreover, such policies are not interpretable to humans and therefore are difficult to debug when these challenges arise.

To address these issues, a growing body of work aims to learn programmatic policies that are structured in more interpretable and generalizable representations such as decision trees [12], state-machines [13], and programs described by domain-specific programming languages [14, 15]. Yet, the programmatic representations employed in these works are often limited in expressiveness due to constraints on the policy spaces. For example, decision tree policies are incapable of naïvely generating repetitive behaviors, state machine policies used in [13] are computationally complex to

[*]Contributed equally.
[†]Work partially done as a visiting scholar at USC.
[‡]AI Advisor at NAVER AI Lab.

scale to policies representing diverse behaviors, and the programs of [14, 15] are constrained to a set of predefined program templates. On the other hand, program synthesis works that aim to represent desired behaviors using flexible domain-specific programs often require extra supervision such as input/output pairs [16–20] or expert demonstrations [21, 22], which can be difficult to obtain.

In this paper, we present a framework to instead synthesize human-readable programs in an expressive representation, solely from rewards, to solve tasks described by Markov Decision Processes (MDPs). Specifically, we represent a policy using a program composed of control flows (*e.g.* if/else and loops) and an agent's perceptions and actions. Our programs can flexibly compose behaviors through perception-conditioned loops and nested conditional statements. However, composing individual program tokens (*e.g.* `if`, `while`, `move()`) in a trial-and-error fashion to synthesize programs that can solve given MDPs can be extremely difficult and inefficient.

To address this problem, we propose to first learn a latent program embedding space where nearby latent programs correspond to similar behaviors and allows for smooth interpolation, together with a program decoder that can decode a latent program to a program consisting of a sequence of program tokens. Then, when a task is given, this embedding space allows us to iteratively search over candidate latent programs to find a program that induces desired behavior to maximize the reward. Specifically, this embedding space is learned through reconstruction of randomly generated programs and the behaviors they induce in the environment in an unsupervised manner. Once learned, the embedding space can be reused to solve different tasks without retraining.

To evaluate the proposed framework, we consider the Karel domain [23], featuring an agent navigating through a gridworld and interacting with objects to solve tasks such as stacking and navigation. The experimental results demonstrate that the proposed framework not only learns to reliably synthesize task-solving programs but also outperforms program synthesis and deep RL baselines. In addition, we justify the necessity of the proposed two-stage learning scheme as well as conduct an extensive analysis comparing various approaches for learning the latent program embedding spaces. Finally, we perform experiments which highlight that the programs produced by our proposed framework can both generalize to larger state spaces and unseen state configurations as well as be interpreted and edited by humans to improve their task performance.

2 Related Work

Neural program induction and synthesis. Program induction methods [20, 24–36] aim to implicitly induce the underlying programs to mimic the behaviors demonstrated in given task specifications such as input/output pairs or expert demonstrations. On the other hand, program synthesis methods [16–19, 21, 37–58] explicitly synthesize the underlying programs and execute the programs to perform the tasks from task specifications such input/output pairs, demonstrations, language instructions. In contrast, we aim to learn to synthesize programs solely from reward described by an MDP without other task specifications. Similarly to us, a two-stage synthesis method is proposed in [46]. Yet, the task is to match truth tables for given test programs rather than solve MDPs. Their first stage requires the entire ground-truth table for each program synthesized during training, which is infeasible to apply to our problem setup (*i.e.* synthesizing imperative programs for solving MDPs).

Learning programmatic policies. Prior works have also addressed the problem of learning programmatic policies [59–61]. Bastani et al. [12] learns a decision tree as a programmatic policy for pong and cartpole environments by imitating an oracle neural policy. However, decision trees are incapable of representing repeating behaviors on their own. Silver et al. [49] addresses this by including a loop-style token for their decision tree policy, though it is still not as expressive as synthesized loops. Inala et al. [13] learns programmatic policies as finite state machines by imitating a teacher policy, although finite state machine complexity can scale quadratically with the number of states, making them difficult to scale to more complex behaviors.

Another line of work instead synthesizes programs structured in Domain Specific Languages (DSLs), allowing humans to design tokens (*e.g.* conditions and operations) and control flows (*e.g.* while loops, if statements, reusable functions) to induce desired behaviors and can produce human interpretable programs. Verma et al. [14, 15] distill neural network policies into programmatic policies. Yet, the initial programs are constrained to a set of predefined program templates. This significantly limits the scope of synthesizable programs and requires designing such templates for each task. In contrast, our method can synthesize diverse programs, without templates, which can flexibly represent the complex behaviors required to solve various tasks.

3 Problem Formulation

We are interested in learning to synthesize a program structured in a given DSL that can be executed to solve a given task described by an MDP, purely from reward. In this section, we formally define our definition of a program and DSL, tasks described by MDPs, and the problem formulation.

Program ρ	:=	DEF run m(s m)
Repetition n	:=	Number of repetitions
Perception h	:=	Domain-dependent perceptions
Condition b	:=	perception h | not perception h
Action a	:=	Domain-dependent actions
Statement s	:=	while c(b c) w(s w) | s_1; s_2 | a |
		repeat R=n r(s r) | if c(b c) i(s i) |
		ifelse c(b c) i(s_1 i) else e(s_2 e)

Figure 1: The domain-specific language (DSL) for constructing programs.

Program and Domain Specific Language. The programs, or programmatic policies, considered in this work are defined based on a DSL as shown in Figure 1. The DSL consists of control flows and an agent's perceptions and actions. A perception indicates circumstances in the environment (*e.g.* `frontIsClear()`) that can be perceived by an agent, while an action defines a certain behavior that can be performed by an agent (*e.g.* `move()`, `turnLeft()`). Control flow includes `if/else` statements, loops, and boolean/logical operators to compose more sophisticated conditions. A policy considered in this work is described by a program ρ which is executed to produce a sequence of actions given perceptions from the environment.

MDP. We consider finite-horizon discounted MDPs with initial state distribution $\mu(s_o)$ and discount factor γ. For a fixed sequence $\{(s_0, a_0), ..., (s_t, a_t)\}$ of states and actions obtained from a rollout of a given policy, the performance of the policy is evaluated based on a discounted return $\sum_{t=0}^{T} \gamma^t r_t$, where T is the horizon of the episode and $r_t = \mathcal{R}(s_t, a_t)$ the reward function.

Objective. Our objective is $\max_\rho \mathbb{E}_{a \sim \text{EXEC}(\rho), s_0 \sim \mu}[\sum_{t=0}^{T} \gamma^t r_t]$, where EXEC returns the actions induced by executing a program policy ρ in the environment. Note that one can view this objective as a special case of the standard RL objective, where the policy is represented as a program which follows the grammar of the DSL and the policy rollout is obtained by executing the program.

4 Approach

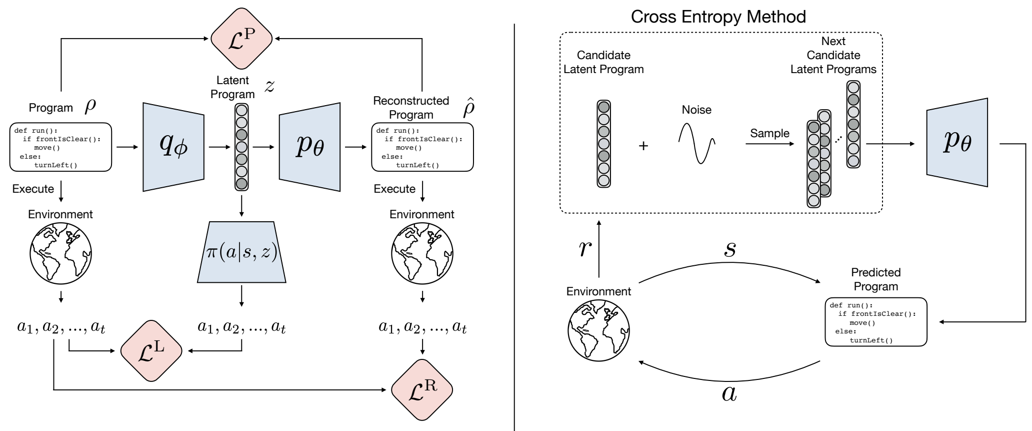
Our goal is to develop a framework that can synthesize a program (*i.e.* a programmatic policy) structured in a given DSL that can be executed to solve a task of interest. This requires the ability to synthesize a program that is not only valid for execution (*e.g.* grammatically correct) but also describes desired behaviors for solving the task from only the reward. Yet, learning to synthesize such a program from scratch for every new task can be difficult and inefficient.

To this end, we propose our **L**earning **E**mbeddings for l**A**tent **P**rogram **S**ynthesis framework, dubbed LEAPS, as illustrated in Figure 2. LEAPS first learns a latent program embedding space that continuously parameterizes diverse behaviors and a program decoder that decodes a latent program to a program consisting of a sequence of program tokens. Then, when a task is given, we iteratively search over this embedding space and decode each candidate latent program using the decoder to find a program that maximizes the reward. This two-stage learning scheme not only enables learning to synthesize programs to acquire desired behaviors described by MDPs solely from reward, but also allows reusing the learned embedding space to solve different tasks without retraining.

In the rest of this section, we describe how we construct the model and our learning objectives for the latent program embedding space in Section 4.1. Then, we present how a program that describes desired behaviors for a given task can be found through a search algorithm in Section 4.2.

4.1 Learning a Program Embedding Space

To learn a latent program embedding space, we propose to train a variational autoencoder (VAE) [62] that consists of a program encoder q_ϕ which encodes a program ρ to a latent program z and a program decoder p_θ which reconstructs the program from the latent. Specifically, the VAE is trained through



```
def run():
  if frontIsClear():
    move()
  else:
    turnLeft()
```

```
def run():
  if frontIsClear():
    move()
  else:
    turnLeft()
```

```
def run():
  if frontIsClear():
    move()
  else:
    turnLeft()
```

(a) Learning Program Embedding Stage

(b) Latent Program Search Stage

Figure 2: (a) **Learning program embedding stage**: we propose to learn a program embedding space by training a program encoder q_ϕ that encodes a program as a latent program z, a program decoder p_θ that decodes the latent program z back to a reconstructed program $\hat{\rho}$, and a policy π that conditions on the latent program z and acts as a neural program executor to produce the execution trace of the latent program z. The model optimizes a combination of a program reconstruction loss $\mathcal{L}^{\text{P}}$, a program behavior reconstruction loss $\mathcal{L}^{\text{R}}$, and a latent behavior reconstruction loss $\mathcal{L}^{\text{L}}$. $a_1, a_2, .., a_t$ denotes actions produced by either the policy π or program execution. (b) **Latent program search stage**: we use the Cross Entropy Method to iteratively search for the best candidate latent programs that can be decoded and executed to maximize the reward to solve given tasks.

reconstruction of randomly generated programs and the behaviors they induce in the environment in an unsupervised manner. Architectural details are listed in Section L.6.

Since we aim to iteratively search over the learned embedding space to achieve certain behaviors when a task is given, we want this embedding space to allow for smooth behavior interpolation (*i.e.* programs that exhibit similar behaviors are encoded closer in the embedding space). To this end, we propose to train the model by optimizing the following three objectives.

4.1.1 Program Reconstruction

To learn a program embedding space, we train a program encoder q_ϕ and a program decoder p_θ to reconstruct programs composed of sequences of program tokens. Given an input program ρ consisting of a sequence of program tokens, the encoder processes the input program one token at a time and produces a latent program embedding z. Then, the decoder outputs program tokens one by one from the latent program embedding z to synthesize a reconstructed program $\hat{\rho}$. Both the encoder and the decoder are recurrent neural networks and are trained to optimize the β-VAE [63] loss:

$$\mathcal{L}^{\text{P}}_{\theta,\phi}(\rho) = -\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\rho)}[\log p_\theta(\boldsymbol{\rho}|\mathbf{z})] + \beta D_{\text{KL}}(q_\phi(\mathbf{z}|\boldsymbol{\rho})\|p_\theta(\mathbf{z})). \tag{1}$$

4.1.2 Program Behavior Reconstruction

While the loss in Eq. 1 enforces that the model encodes syntactically similar programs close to each other in the embedding space, we also want to encourage programs with the same semantics to have similar program embeddings. An example that demonstrates the importance of this is the *program aliasing* issue, where different programs have identical program semantics (*e.g.* `repeat(2): move()` and `move() move()`). Thus, we introduce an objective that compares the execution traces of the input program and the reconstructed program. Since the program execution process is not differentiable, we optimize the model via REINFORCE [64]:

$$\mathcal{L}^{\text{R}}_{\theta,\phi}(\rho) = -\mathbb{E}_{z\sim q_\phi(z|\rho)}[R_{\text{mat}}(p_\theta(\rho|z), \rho)], \tag{2}$$

where $R_{\text{mat}}(\hat{\rho}, \rho)$, the reward for matching the input program's behavior, is defined as

$$R_{\text{mat}}(\hat{\rho}, \rho) = \mathbb{E}_\mu [\frac{1}{N} \sum_{t=1}^{N} \underbrace{\mathbb{1}\{\text{EXEC}_i(\hat{\rho}) == \text{EXEC}_i(\rho) \; \forall i = 1, 2, ...t\}}_{\text{stays 0 after the first } t \text{ where } \text{EXEC}_t(\hat{\rho}) \text{ != } \text{EXEC}_t(\rho)}], \tag{3}$$

where N is the maximum of the lengths of the execution traces of both programs, and $\text{EXEC}_i(\rho)$ represents the action taken by program ρ at time i. Thus this objective encourages the model to embed behaviorally similar yet possibly syntactically different programs to similar latent programs.

4.1.3 Latent Behavior Reconstruction

To further encourage learning a program embedding space that allows for smooth behavior interpolation, we devise another source of supervision by learning a program embedding-conditioned policy. Denoted $\pi(a|z, s_t)$, this recurrent policy takes the program embedding z produced by the program encoder and learns to predict corresponding agent actions. One can view this policy as a neural program executor that allows gradient propagation through the policy and the program encoder by optimizing the cross entropy between the actions obtained by executing the input program ρ and the actions predicted by the policy:

$$\mathcal{L}^{\text{L}}_\pi(\rho, \pi) = -\mathbb{E}_\mu [\sum_{t=1}^{M} \sum_{i=1}^{|\mathcal{A}|} \mathbb{1}\{\text{EXEC}_i(\hat{\rho}) == \text{EXEC}_i(\rho)\} \log \pi(a_i|z, s_t)], \tag{4}$$

where M denotes the length of the execution of ρ. Optimizing this objective directly encourages the program embeddings, through supervised learning instead of RL as in $\mathcal{L}^{\text{R}}$, to be useful for action reconstruction, thus further ensuring that similar behaviors are encoded together and allowing for smooth interpolation. Note that this policy is only used for improving learning the program embedding space not for solving the tasks of interest in the later stage.

In summary, we propose to optimize three sources of supervision to learn the program embedding space that allows for smooth interpolation and can be used to search for desired agent behaviors: (1) $\mathcal{L}^{\text{P}}$ (Eq. 1), the β-VAE objective for program reconstruction, (2) $\mathcal{L}^{\text{R}}$ (Eq. 2), an RL environment-state matching loss for the reconstructed program, and (3) $\mathcal{L}^{\text{L}}$ (Eq. 4), a supervised learning loss to encourage predicting the ground-truth agent action sequences. Thus our combined objective is:

$$\min_{\theta, \phi, \pi} \lambda_1 \mathcal{L}^{\text{P}}_{\theta,\phi}(\rho) + \lambda_2 \mathcal{L}^{\text{R}}_{\theta,\phi}(\rho) + \lambda_3 \mathcal{L}^{\text{L}}_\pi(\rho, \pi), \tag{5}$$

where λ_1, λ_2, and λ_3 are hyperparameters controlling the importance of each loss. Optimizing the combination of these losses encourages the program embedding to be both semantically and syntactically informative. More training details can be found in Section L.6.

4.2 Latent Program Search: Synthesizing a Task-Solving Program

Once the program embedding space is learned, our goal becomes searching for a latent program that maximizes the reward described by a given task MDP. To this end, we adapt the Cross Entropy Method (CEM) [65], a gradient-free continuous search algorithm, to iteratively search over the program embedding space. Specifically, we (1) sample a distribution of latent programs, (2) decode the sampled latent programs into programs using the learned program decoder p_θ, (3) execute the programs in the task environment and obtain the corresponding rewards, and (4) update the CEM sampling distribution based on the rewards. This process is repeated until either convergence or the maximum number of sampling steps has been reached.

5 Experiments

We first introduce the environment and the tasks in Section 5.1 and describe the experimental setup in Section 5.2. Then, we justify the design of LEAPS by conducting extensive ablation studies in Section 5.3. We describe the baselines used for comparison in Section 5.4, followed by the experimental results presented in Section 5.5. In Section 5.6, we conduct experiments to evaluate the ability of our method to generalize to a larger state space without further learning. Finally, we investigate how LEAPS' interpretability can be leveraged by conducting experiments that allow humans to debug and improve the programs synthesized by LEAPS in Section 5.7

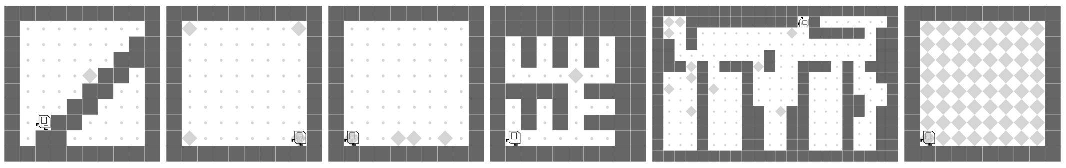


Figure 3: **The Karel problem set**: the domain features an agent navigating through a gridworld with walls and interacting with markers, allowing for designing tasks that demand certain behaviors. The tasks are further described in Section K with visualizations in Figure 15.

5.1 Karel domain

To evaluate the proposed framework, we consider the Karel domain [23], as featured in [17, 19, 21], which features an agent navigating through a gridworld with walls and interacting with markers. The agent has 5 actions for moving and interacting with marker and 5 perceptions for detecting obstacles and markers. The tasks of interest are shown in Figure 3. Note that most tasks have randomly sampled agent, wall, marker, and/or goal configurations. When either training or evaluating, we randomly sample initial configurations upon every episode reset. More details can be found in Section K.

5.2 Programs

To produce programs for learning the program embedding space, we randomly generated a dataset of 50,000 unique programs. Note that the programs are generated independently of any Karel tasks; each program is created only by sampling tokens from the DSL, similar to the procedures used in [16–19, 21, 22]. This dataset is split into a training set with 35,000 programs a validation set with 7,500 programs, and a testing set with 7,500 programs. The validation set is used to select the learned program embedding space to use for the program synthesis stage.

For each program, we sample random Karel states and execute the program on them from different starting states to obtain 10 environment rollouts to compute the program behavior reconstruction loss $\mathcal{L}^{\text{R}}$ and the latent behavior reconstruction loss $\mathcal{L}^{\text{L}}$ when learning the program embedding space. We perform checks to ensure rollouts cover all execution branches in the program so that they are representative of all aspects of the program's behavior. The maximum length of the programs is 44 tokens and the average length is 17.9. We plot a histogram of their lengths in Figure 14 (in Appendix). More dataset generation details can be found in Section J.

5.3 Ablation Study

We first ablate various components of our proposed framework in order to (1) justify the necessity of the proposed two-stage learning scheme and (2) identify the effects of the proposed objectives. We consider the following baselines and ablations of our method (illustrated Section I).

- Naïve: a program synthesis baseline that learns to directly synthesize a program from scratch by recurrently predicting a sequence of program tokens. This baseline investigates if an end-to-end learning method can solve the problem. More details can be found in Section L.4.
- LEAPS-P: the simplest ablation of LEAPS, in which the program embedding space is learned by only optimizing the program reconstruction loss $\mathcal{L}^{\text{P}}$ (Eq. 1).
- LEAPS-P+R: an ablation of LEAPS which optimizes both the program reconstruction loss $\mathcal{L}^{\text{P}}$ (Eq. 1) and the program behavior reconstruction loss $\mathcal{L}^{\text{R}}$ (Eq. 2).
- LEAPS-P+L: an ablation of LEAPS which optimizes both the program reconstruction loss $\mathcal{L}^{\text{P}}$ (Eq. 1) and the latent behavior reconstruction loss $\mathcal{L}^{\text{L}}$ (Eq. 4).
- LEAPS (LEAPS-P+R+L): LEAPS with all the losses, optimizing our full objective in Eq. 5.
- LEAPS-rand-{8/64}: similar to LEAPS, this ablation also optimizes the full objective (Eq. 5) for learning the program embedding space. Yet, when searching latent programs, instead of CEM, it simply randomly samples 8/64 candidate latent programs and chooses the best performing one. These baselines justify the effectiveness of using CEM for searching latent programs.

Table 1: Program behavior reconstruction rewards (standard deviations) across all methods.

	WHILE	IFELSE+WHILE	2IF+IFELSE	WHILE+2IF+IFELSE	Avg Reward
Naïve	0.65 (0.33)	0.83 (0.07)	0.61 (0.33)	0.16 (0.06)	0.56
LEAPS-P	0.95 (0.13)	0.82 (0.08)	0.58 (0.35)	0.33 (0.17)	0.67
LEAPS-P+R	0.98 (0.09)	0.77 (0.05)	0.63 (0.25)	0.52 (0.27)	0.72
LEAPS-P+L	**1.06** (0.00)	0.84 (0.10)	0.77 (0.23)	0.33 (0.13)	0.75
LEAPS-rand-8	0.62 (0.24)	0.49 (0.09)	0.36 (0.18)	0.28 (0.14)	0.44
LEAPS-rand-64	0.78 (0.22)	0.63 (0.09)	0.55 (0.20)	0.37 (0.09)	0.58
LEAPS	**1.06** (0.08)	**0.87** (0.13)	**0.85** (0.30)	**0.57** (0.23)	**0.84**

Program Behavior Reconstruction. To determine the effectiveness of the proposed two-stage learning scheme and the learning objectives, we measure how effective each ablation is at reconstructing the behaviors of input programs. We use programs from the test set (shown in Figure 9 in Appendix), and utilize the environment state matching reward $R_{\text{mat}}(\hat{\rho}, \rho)$ (Eq. 3), with a 0.1 bonus for synthesizing a syntactically correct program. Thus the return ranges between $[0, 1.1]$. We report the mean cumulative return, over 5 random seeds, of the final programs after convergence.

The results are reported in Table 1. Each test is named after its control flows (*e.g.* IFELSE+WHILE has an if-else statement and a while loop). The naïve program synthesis baseline fails on the complex WHILE+2IF+IFELSE program, as it rarely synthesizes conditional and loop statements, instead generating long sequences of action tokens that attempt to replicate the desired behavior of those statements (see synthesized programs in Figure 10). We believe that this is because it is incentivized to initially predict action tokens to gain more immediate reward, making it less likely to synthesize other tokens. LEAPS and its variations perform better and synthesize more complex programs, demonstrating the importance of the proposed two-stage learning scheme in biasing program search. We also note that LEAPS-P achieves the worst performance out of the CEM search LEAPS ablations, indicating that optimizing the program reconstruction loss $\mathcal{L}^{\text{P}}$ (the VAE loss) alone does not yield satisfactory results. Jointly optimizing $\mathcal{L}^{\text{P}}$ with either the program behavior reconstruction loss $\mathcal{L}^{\text{R}}$ or the latent behavior reconstruction loss $\mathcal{L}^{\text{L}}$ improves the performance, and optimizing our full objective with all three achieves the best performance across all tasks, indicating the effectiveness of the proposed losses. Finally, LEAPS outperforms LEAPS-rand-8/64, suggesting the necessity of adopting better search algorithms such as CEM.

Program Embedding Space Smoothness. We investigate if the program and latent behavior reconstruction losses encourage learning a behaviorially smooth embedding space. To quantify behavioral smoothness, we measure how much a change in the embedding space corresponds to a change in behavior by comparing execution traces. For all programs we compute the pairwise Euclidean distance between their embeddings in each model. We then calculate the environment state matching distance R_{mat} between the decoded programs by executing them from the same initial state.

Table 2: Program embedding space smoothness. For each program, we execute the ten nearest programs in the learned embedding space of each model to calculate the mean state-matching reward R_{mat} against the original program. We report R_{mat} averaged over all programs in each dataset.

	LEAPS-P	LEAPS-P+R	LEAPS-P+L	LEAPS
Training	0.22	0.22	**0.31**	**0.31**
Validation	0.22	0.21	**0.27**	**0.27**
Testing	0.22	0.22	**0.28**	0.27

The results are reported in Table 2. LEAPS and LEAPS-P+L perform the best, suggesting that optimizing the latent behavior reconstruction objective $\mathcal{L}^{\text{L}}$, in Eq. 4, is essential for improving the smoothness of the latent space in terms of execution behavior. We further analyze and visualize the learned program embedding space in Section A and Figure 4 (in Appendix).

5.4 Baselines

We evaluate LEAPS against the following baselines (illustrated in Figure 13 in Appendix Section I).

- DRL: a neural network policy trained on each task and taking raw states (Karel grids) as input.
- DRL-abs: a recurrent neural network policy directly trained on each Karel task but taking *abstract* states as input (*i.e.* it sees the same perceptions as LEAPS, *e.g.* `frontIsClear()==true`).
- DRL-abs-t: a DRL transfer learning baseline in which for each task, we train DRL-abs policies on all other tasks, then fine-tune them on the current task. Thus it acquires a prior by learning to

Table 3: Mean return (standard deviation) of all methods across Karel tasks, evaluated over 5 random seeds. DRL methods, program synthesis baselines, and LEAPS ablations are separately grouped.

	StairClimber	FourCorner	TopOff	Maze	CleanHouse	Harvester
DRL	**1.00** (0.00)	0.29 (0.05)	0.32 (0.07)	**1.00** (0.00)	0.00 (0.00)	**0.90** (0.10)
DRL-abs	0.13 (0.29)	0.36 (0.44)	0.63 (0.23)	**1.00** (0.00)	0.01 (0.02)	0.32 (0.18)
DRL-abs-t	0.00 (0.00)	0.05 (0.10)	0.17 (0.11)	**1.00** (0.00)	0.01 (0.02)	0.16 (0.18)
HRL	-0.51 (0.17)	0.01 (0.00)	0.17 (0.11)	0.62 (0.05)	0.01 (0.00)	0.00 (0.00)
HRL-abs	-0.05 (0.07)	0.00 (0.00)	0.19 (0.12)	0.56 (0.03)	0.00 (0.00)	-0.03 (0.02)
Naïve	0.40 (0.49)	0.13 (0.15)	0.26 (0.27)	0.76 (0.43)	0.07 (0.09)	0.21 (0.25)
VIPER	0.02 (0.02)	0.40 (0.42)	0.30 (0.06)	0.69 (0.05)	0.00 (0.00)	0.51 (0.07)
LEAPS-rand-8	0.10 (0.17)	0.10 (0.14)	0.28 (0.05)	0.40 (0.50)	0.00 (0.00)	0.07 (0.06)
LEAPS-rand-64	0.18 (0.40)	0.20 (0.11)	0.33 (0.07)	0.58 (0.41)	0.03 (0.06)	0.12 (0.05)
LEAPS	**1.00** (0.00)	**0.45** (0.40)	**0.81** (0.07)	**1.00** (0.00)	**0.18** (0.14)	0.45 (0.28)

first solve other Karel tasks. Rewards are reported for the policies from the task that transferred with highest return. We only transfer DRL-abs policies as some tasks have different state spaces.

- HRL: a hierarchical RL baseline in which a VAE is first trained on action sequences from program execution traces used by LEAPS. Once trained, the decoder is utilized as a low-level policy for learning a high-level policy to sample actions from. Similar to LEAPS, this baseline utilizes the dataset to produce a prior of the domain. It takes raw states (Karel grids) as input.
- HRL-abs: the same method as HRL but taking abstract states (*i.e.* local perceptions) as input.
- VIPER [12]: A decision-tree programmatic policy which imitates the behavior of a deep RL teacher policy via a modified DAgger algorithm [66]. This decision tree policy cannot synthesize loops, allowing us to highlight the performance advantages of more expressive program representation that LEAPS is able to take advantage of.

All the baselines are trained with PPO [67] or SAC [68], including the VIPER teacher policy. More training details can be found in Section L.

5.5 Results

We present the results of the baselines and our method evaluated on the Karel task set based on the environment rewards in Table 3. The reward functions are sparse for all tasks, and are normalized such that the final cumulative return is within $[-1, 1]$ for tasks with penalties and $[0, 1]$ for tasks without; reward functions for each task are detailed in Section K.

Overall Task Performance. Across all but one task, LEAPS yields the best performance. The LEAPS-rand baselines perform significantly worse than LEAPS on all Karel tasks, demonstrating the need for using a search algorithm like CEM during synthesis. The performance of VIPER is bounded by its RL teacher policy, and therefore is outperformed by the DRL baselines on most of the tasks. Meanwhile, DRL-abs-t is generally unable to improve upon DRL-abs across the board, suggesting that transferring Karel behaviors with RL from one task to another is ineffective. Furthermore, both the HRL baselines achieve poor performance, likely because agent actions alone provide insufficient supervision for a VAE to encode useful action trajectories on unseen tasks—unlike programs. Finally, the poor performance of the naïve program synthesis baseline highlights the difficulty and inefficiency of learning to synthesize programs from scratch using only rewards. In the appendix, we present programs synthesized by LEAPS in Figure 11, example optimal programs for each task in Section F (Figure 9), rollout visualizations in Figure 16, and additional results analysis in Section H.

Repetitive Behaviors. Solving StairClimber and FourCorner requires acquiring repetitive (or looping) behaviors. StairClimber, which can be solved by repeating a short, 4-step stair-climbing behavior until the goal marker is reached, is not solved by DRL-abs. LEAPS fully solves the task given the same perceptions, as this behavior can be simply represented with a while loop that repeats the stair-climbing skill. However VIPER performs poorly as its decision tree cannot represent such loops. Similarly, the baselines are unable to perform as well on FourCorner, a task in which the agent must pickup a marker located in each corner of the grid. This behavior takes at least 14 timesteps to complete, but can be represented by two nested loops. Similar to StairClimber, the bias introduced by the DSL and our generated dataset (which includes nested loops), results in LEAPS being able to perform much better.

Exploration. TOPOFF rewards the agent for adding markers to locations with existing markers. However, there are no restrictions for the agent to wander elsewhere around the environment, thus making exploration a problem for the RL baselines, and thereby also constraining VIPER. LEAPS performs best on this task, as the ground-truth program can be represented by a simple loop that just moves forward and places markers when a marker is detected. MAZE also involves exploration, however its small size (8×8) results in many methods, including LEAPS, solving the task.

Complexity. Solving HARVESTER and CLEANHOUSE requires acquiring complex behaviors, resulting in poor performance from all methods. CLEANHOUSE requires an agent to navigate through a house and pick up all markers along the walls on the way. This requires repeated execution of a skill, of varied length, which navigates around the house, turns into rooms, and picks up markers. As such, all baselines perform very poorly. However, LEAPS is able to perform substantially better because these behaviors can be represented by a program of medium complexity with a while loop and some nested conditional statements. On the other hand, HARVESTER involves simply navigating to and picking up a marker on every spot on the grid. However, this is a difficult program to synthesize given our random dataset generation process; the program we manually derive to solve HARVESTER is long and more syntactically complex than most training programs. As a result, DRL and VIPER outperform LEAPS on this task.

Learned Program Embedding Space. More analysis on our learned program embedding space can be found in the appendix. We present CEM search trajectory visualizations in Section B, demonstrating how the search population's rewards change over time. To qualitatively investigate the smoothness of the learned program embedding space, we linearly interpolate between pairs of latent programs and display their corresponding decoded programs in Section C. In Section D, we illustrate how predicted programs evolve over the course of CEM search.

5.6 Generalization

We are also interested in learning whether the baselines and the programs synthesized by LEAPS can generalize to novel scenarios without further learning. Specifically, we investigate how well they can generalize to larger state spaces. We expand both STAIRCLIMBER and MAZE to 100×100 grid sizes (from 12×12 and 8×8, respectively). We directly evaluate the policies or programs obtained from the original tasks with smaller state spaces for all methods except DRL (its observation space changes), which we retrain from scratch. The results are shown in Table 4. All baselines perform significantly worse than before on both tasks. On the contrary, the programs synthesized by LEAPS for the smaller task instances achieve zero-shot generalization to larger task instances without losing any performance. Larger grid size experiments for the other Karel tasks and additional unseen configuration experiments can be found in Section G.

Table 4: Rewards on 100×100 grids.

	STAIRCLIMBER	MAZE
DRL	0.00 (0.00)	0.00 (0.00)
DRL-abs	0.00 (0.00)	0.04 (0.05)
VIPER	0.00 (0.00)	0.10 (0.12)
LEAPS	**1.00** (0.00)	**1.00** (0.00)

5.7 Interpretability

Interpretability in machine learning [69, 70] is particularly crucial when it comes to learning a policy that interacts with the environment [71–78]. The proposed framework produces programmatic policies that are interpretable from the following aspects as outlined in [70].

- Trust: interpretable machine learning methods and models may more easily be trusted since humans tend to be reluctant to trust systems that they do not understand. Programs synthesized by LEAPS can naturally be better trusted since one can simply read and interpret them.
- Contestability: the program execution traces produce a chain of reasoning for each action, providing insights on the induced behaviors and thus allowing for contesting improper decisions.
- Safety: synthesizing readable programs allows for diagnosing issues earlier (*i.e.* before execution) and provides opportunities to intervene, which is especially critical for safety-critical tasks.

In the rest of this section, we investigate how the proposed framework enjoys interpretability from the three aforementioned aspects. Specifically, synthesized programs are not only readable to human users but also interactive, allowing non-expert users with a basic understanding of programming to diagnose and make edits to improve their performance. To demonstrate this, we asked non-expert humans to read, interpret, and edit suboptimal LEAPS policies to improve their performance. Participants edited LEAPS programs on 3 Karel tasks with suboptimal reward: TOPOFF, FOURCORNER, and HARVESTER. With just 3 edits, participants obtained a mean reward improvement of 97.1%, and with 5 edits, participants improved it by 125%. This justifies how our synthesized policies can be manually diagnosed and improved, a property which DRL methods lack. More details and discussion can be found in Section E.

6 Discussion

We propose a framework for solving tasks described by MDPs by producing programmatic policies that are more interpretable and generalizable than neural network policies learned by deep reinforcement learning methods. Our proposed framework adopts a flexible program representation and requires only minimal supervision compared to prior programmatic reinforcement learning and program synthesis works. Our proposed two-stage learning scheme not only alleviates the difficulty of learning to synthesize programs from scratch but also enables reusing its learned program embedding space for various tasks. The experiments demonstrate that our proposed framework outperforms DRL and programmatic baselines on a set of Karel tasks by producing expressive and generalizable programs that can consistently solve the tasks. Ablation studies justify the necessity of the proposed two-stage learning scheme as well as the effectiveness of the proposed learning objectives.

While the proposed framework achieves promising results, we would like to acknowledge two assumptions that are implicitly made in this work. First, we assume the existence of a program executor that can produce execution traces of programs. This program executor needs to be able to return perceptions from the environment state as well as apply actions to the environment. While this assumption is widely made in program synthesis works, a program executor can still be difficult to obtain when it comes to real-world robotic tasks. Fortunately, in research fields such as computer vision or robotics, a great amount of effort has been put into satisfying this assumption such as designing modules that can return high-level abstraction of raw sensory input (*e.g.* with object detection networks, proximity/tactile sensors, etc.).

Secondly, we assume that it is possible to generate a distribution of programs whose behaviors are at least remotely related to the desired behaviors for solving the tasks of interest. It can be difficult to synthesize programs which represent behaviors that are more complex than ones in the training program distribution, although one possible solution is to employ a better program generation process to generate programs that induce more complex behaviors. Also, the choice of DSL plays an important role in how complex the programs can be. Ideally, employing a more complex DSL would allow our proposed framework to synthesize more advanced agent behaviors.

In the future, we hope to extend the proposed framework to more challenging domains such real-world robotics. We believe this framework would allow for deploying robust, interpretable policies for safety-critical tasks such as robotic surgeries. One way to make LEAPS applicable to robotics domains would be to simultaneously learn perception modules and action controllers. Other possible solutions include incorporating program execution methods [79–84] that are designed to allow program execution or designing DSLs that allow pre-training of perception modules and action controllers. Also, the proposed framework shares some characteristics with works in multi-task RL [79, 80, 85–89] and meta-learning [90–99]. Specifically, it learns a program embedding space from a distribution of tasks/programs. Once the program embedding space is learned, it can be be reused to solve different tasks without retraining.

Yet, extending LEAPS to such domains can potentially lead to some negative societal impacts. For example, our framework can still capture unintended bias during learning or suffer from adversarial attacks. Furthermore, policies deployed in the real world can create great economic impact by causing job losses in some sectors. Therefore, we would encourage further work to investigate the biases, safety issues, and potential economic impacts to ensure that the deployment in the field does not cause far-reaching, negative societal impacts.

Acknowledgments

The authors appreciate the fruitful discussions with Karl Pertsch, Youngwoon Lee, Ayush Jain, and Grace Zhang. The authors would like to thank Ting-Yu S. Su for contributing to the learned program embedding spaces visualizations. The authors are grateful to Sidhant Kaushik, Laura Smith, and Siddharth Verma for offering their time to participate in the human debugging interpretability experiment.

Funding Transparency Statement

This project was partially supported by USC startup funding and by NAVER AI Lab.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human level control through deep reinforcement learning. *Nature*, 2015.

[2] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

[3] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 2018.

[4] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 2019.

[5] Alexandre Campeau-Lecours, Hugo Lamontagne, Simon Latour, Philippe Fauteux, Véronique Maheu, François Boucher, Charles Deguire, and Louis-Joseph Caron L'Ecuyer. Kinova modular robot arms for service robotics applications. In *Rapid Automation: Concepts, Methodologies, Tools, and Applications*. 2019.

[6] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *International Conference on Robotics and Automation*, 2017.

[7] OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 2020.

[8] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. In *International Conference on Learning Representations*, 2020.

[9] Jun Yamada, Youngwoon Lee, Gautam Salhotra, Karl Pertsch, Max Pflueger, Gaurav S Sukhatme, Joseph J Lim, and Peter Englert. Motion planner augmented reinforcement learning for robot manipulation in obstructed environments. In *Conference on Robot Learning*, 2020.

[10] Grace Zhang, Linghan Zhong, Youngwoon Lee, and Joseph J Lim. Policy transfer across visual and dynamics domain gaps via iterative grounding. In *Robotics: Science and Systems*, 2021.

[11] Youngwoon Lee, Edward S Hu, Zhengyu Yang, and Joseph J Lim. To follow or not to follow: Selective imitation learning from observations. In *Conference on Robot Learning*, 2019.

[12] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Neural Information Processing Systems*, 2018.

[13] Jeevana Priya Inala, Osbert Bastani, Zenna Tavares, and Armando Solar-Lezama. Synthesizing programmatic policies that inductively generalize. In *International Conference on Learning Representations*, 2020.

[14] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, 2018.

[15] Abhinav Verma, Hoang Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. In *Neural Information Processing Systems*, 2019.

[16] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International Conference on Machine Learning*, 2017.

[17] Rudy R Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*, 2018.

[18] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019.

[19] Eui Chul Shin, Illia Polosukhin, and Dawn Song. Improving neural program synthesis with inferred execution traces. In *Neural Information Processing Systems*, 2018.

[20] Miguel Lázaro-Gredilla, Dianhuan Lin, J Swaroop Guntupalli, and Dileep George. Beyond imitation: Zero-shot task transfer on robots by learning concepts as cognitive programs. *Science Robotics*, 2019.

[21] Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph Lim. Neural program synthesis from diverse demonstration videos. In *International Conference on Machine Learning*, 2018.

[22] Raphaël Dang-Nhu. Plans: Neuro-symbolic program learning from videos. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Neural Information Processing Systems*, 2020.

[23] Richard E Pattis. *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, Inc., 1981.

[24] Danfei Xu, Suraj Nair, Yuke Zhu, Julian Gao, Animesh Garg, Li Fei-Fei, and Silvio Savarese. Neural task programming: Learning to generalize across hierarchical tasks. In *International Conference on Robotics and Automation*, 2018.

[25] Jacob Devlin, Rudy R Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural program meta-induction. In *Advances in Neural Information Processing Systems*, 2017.

[26] Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. In *International Conference on Learning Representations*, 2015.

[27] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[28] Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. In *International Conference on Learning Representations*, 2016.

[29] Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable programs with neural libraries. In *International Conference on Machine Learning*, 2017.

[30] Scott Reed and Nando De Freitas. Neural programmer-interpreters. In *International Conference on Learning Representations*, 2016.

[31] Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. In *International Conference on Learning Representations*, 2017.

[32] Da Xiao, Jo-Yu Liao, and Xingyuan Yuan. Improving the universality and learnability of neural programmer-interpreters with combinator abstraction. In *International Conference on Learning Representations*, 2018.

[33] Michael Burke, Svetlin Penkov, and Subramanian Ramamoorthy. From explanation to synthesis: Compositional program induction for learning from demonstration. *arXiv preprint arXiv:1902.10657*, 2019.

[34] Yujun Yan, Kevin Swersky, Danai Koutra, Parthasarathy Ranganathan, and Milad Hashemi. Neural execution engines: Learning to execute subroutines. In *Neural Information Processing Systems*, 2020.

[35] Yujia Li, Felix Gimeno, Pushmeet Kohli, and Oriol Vinyals. Strong generalization and efficiency in neural programs. *arXiv preprint arXiv:2007.03629*, 2020.

[36] De-An Huang, Suraj Nair, Danfei Xu, Yuke Zhu, Animesh Garg, Li Fei-Fei, Silvio Savarese, and Juan Carlos Niebles. Neural task graphs: Generalizing to unseen tasks from a single video demonstration. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[37] Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a differentiable forth interpreter. In *International Conference on Machine Learning*, 2017.

[38] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *International Conference on Learning Representations*, 2017.

[39] Richard Shin, Illia Polosukhin, and Dawn Song. Improving neural program synthesis with inferred execution traces. In *Neural Information Processing Systems*, 2018.

[40] Yunchao Liu, Jiajun Wu, Zheng Wu, Daniel Ritchie, William T. Freeman, and Joshua B. Tenenbaum. Learning to describe scenes with programs. In *International Conference on Learning Representations*, 2019.

[41] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. In *International Conference on Language Resources and Evaluation*, 2018.

[42] Yuan-Hong Liao, Xavier Puig, Marko Boben, Antonio Torralba, and Sanja Fidler. Synthesizing environment-aware activities via activity sketches. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

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

[44] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*, 2020.

[45] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations*, 2017.

[46] Paweł Liskowski, Krzysztof Krawiec, Nihat Engin Toklu, and Jerry Swan. Program synthesis as latent continuous optimization: Evolutionary search in neural embeddings. In *Genetic and Evolutionary Computation Conference*, 2020.

[47] Daniel A Abolafia, Mohammad Norouzi, Jonathan Shen, Rui Zhao, and Quoc V Le. Neural program synthesis with priority queue training. *arXiv preprint arXiv:1801.03526*, 2018.

[48] Joey Hong, David Dohan, Rishabh Singh, Charles Sutton, and Manzil Zaheer. Latent programmer: Discrete latent codes for program synthesis. In *International Conference on Machine Learning*, 2021.

[49] Tom Silver, Kelsey R Allen, Alex K Lew, Leslie Pack Kaelbling, and Josh Tenenbaum. Few-shot bayesian imitation learning with logical program policies. In *Association for the Advancement of Artificial Intelligence*, 2020.

[50] Jiajun Wu, Joshua B Tenenbaum, and Pushmeet Kohli. Neural scene de-rendering. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[51] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[52] Ferran Alet, Javier Lopez-Contreras, James Koppel, Maxwell Nye, Armando Solar-Lezama, Tomas Lozano-Perez, Leslie Kaelbling, and Joshua Tenenbaum. A large-scale benchmark for few-shot program induction and synthesis. In *International Conference on Machine Learning*, 2021.

[53] Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. *arXiv preprint arXiv:2107.00101*, 2021.

[54] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[55] Joey Hong, David Dohan, Rishabh Singh, Charles Sutton, and Manzil Zaheer. Latent programmer: Discrete latent codes for program synthesis. In *International Conference on Machine Learning*, 2021.

[56] Catherine Wong, Kevin Ellis, Joshua B Tenenbaum, and Jacob Andreas. Leveraging language to learn program abstractions and search heuristics. In *International Conference on Machine Learning*, 2021.

[57] Xinyun Chen, Petros Maniatis, Rishabh Singh, Charles Sutton, Hanjun Dai, Max Lin, and Denny Zhou. Spreadsheetcoder: Formula prediction from semi-structured context. In *International Conference on Machine Learning*, 2021.

[58] Maxwell Nye, Yewen Pu, Matthew Bowers, Jacob Andreas, Joshua B. Tenenbaum, and Armando Solar-Lezama. Representing partial programs with blended abstract semantics. In *International Conference on Learning Representations*, 2021.

[59] Dongkyu Choi and Pat Langley. Learning teleoreactive logic programs from problem solving. In *International Conference on Inductive Logic Programming*, 2005.

[60] Elly Winner and Manuela Veloso. Distill: Learning domain-specific planners by example. In *International Conference on Machine Learning*, 2003.

[61] Mikel Landajuela, Brenden K Petersen, Sookyung Kim, Claudio P Santiago, Ruben Glatt, Nathan Mundhenk, Jacob F Pettit, and Daniel Faissol. Discovering symbolic policies with deep reinforcement learning. In *International Conference on Machine Learning*, 2021.

[62] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. In *International Conference on Learning Representations*, 2014.

[63] Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. In *International Conference on Learning Representations*, 2016.

[64] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 1992.

[65] Reuven Y Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 1997.

[66] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *International Conference on Artificial Intelligence and Statistics*, 2011.

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

[68] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, 2018.

[69] Zachary C Lipton. The mythos of model interpretability. In *ICML Workshop on Human Interpretability in Machine Learning*, 2016.

[70] Owen Shen. Interpretability in ml: A broad overview, 2020. URL `https://mlu.red/muse/52906366310`.

[71] Jesse Zhang, Brian Cheung, Chelsea Finn, Sergey Levine, and Dinesh Jayaraman. Cautious adaptation for reinforcement learning in safety-critical settings. In *International Conference on Machine Learning*, 2020.

[72] Lukas Hewing, Juraj Kabzan, and Melanie N. Zeilinger. Cautious model predictive control using gaussian process regression. *IEEE Transactions on Control Systems Technology*, 2019.

[73] Jaime F Fisac, Anayo K Akametalu, Melanie N Zeilinger, Shahab Kaynama, Jeremy Gillula, and Claire J Tomlin. A general safety framework for learning-based control in uncertain robotic systems. *IEEE Transactions on Automatic Control*, 2018.

[74] A. Hakobyan, G. C. Kim, and I. Yang. Risk-aware motion planning and control using cvar-constrained optimization. *IEEE Robotics and Automation Letters*, 2019.

[75] Dorsa Sadigh and Ashish Kapoor. Safe control under uncertainty with probabilistic signal temporal logic. In *Proceedings of Robotics: Science and Systems*, 2016.

[76] Felix Berkenkamp, Matteo Turchetta, Angela P. Schoellig, and Andreas Krause. Safe model-based reinforcement learning with stability guarantees. In *Advances in Neural Information Processing Systems*, 2017.

[77] Brijen Thananjeyan, Ashwin Balakrishna, Ugo Rosolia, Felix Li, Rowan McAllister, Joseph E. Gonzalez, Sergey Levine, Francesco Borrelli, and Ken Goldberg. Safety augmented value estimation from demonstrations (saved): Safe deep model-based rl for sparse cost robotic tasks. *IEEE Robotics and Automation Letters*, 2020.

[78] Anil Aswani, Humberto Gonzalez, S Shankar Sastry, and Claire Tomlin. Provably safe and robust learning-based model predictive control. *Automatica*, 2013.

[79] Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In *International Conference on Machine Learning*, 2017.

[80] Junhyuk Oh, Satinder Singh, Honglak Lee, and Pushmeet Kohli. Zero-shot task generalization with multi-task deep reinforcement learning. In *International Conference on Machine Learning*, 2017.

[81] Shao-Hua Sun, Te-Lin Wu, and Joseph J. Lim. Program guided agent. In *International Conference on Learning Representations*, 2020.

[82] Yichen Yang, Jeevana Priya Inala, Osbert Bastani, Yewen Pu, Armando Solar-Lezama, and Martin Rinard. Program synthesis guided reinforcement learning. *arXiv preprint arXiv:2102.11137*, 2021.

[83] Youngwoon Lee, Shao-Hua Sun, Sriram Somasundaram, Edward Hu, and Joseph J. Lim. Composing complex skills by learning transition policies. In *International Conference on Learning Representations*, 2019.

[84] Zelin Zhao, Karan Samel, Binghong Chen, and Le Song. Proto: Program-guided transformer for program-guided tasks. *arXiv preprint arXiv:2110.00804*, 2021.

[85] Yee Whye Teh, Victor Bapst, Wojciech Marian Czarnecki, John Quan, James Kirkpatrick, Raia Hadsell, Nicolas Heess, and Razvan Pascanu. Distral: Robust multitask reinforcement learning. *arXiv preprint arXiv:1707.04175*, 2017.

[86] Richard Socher Tianmin Shu, Caiming Xiong. Hierarchical and interpretable skill acquisition in multi-task reinforcement learning. *International Conference on Learning Representations*, 2018.

[87] Sungryull Sohn, Junhyuk Oh, and Honglak Lee. Hierarchical reinforcement learning for zero-shot generalization with subtask dependencies. In *Advances in Neural Information Processing Systems*, 2018.

[88] Ayush Jain, Andrew Szot, and Joseph Lim. Generalization to new actions in reinforcement learning. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 4661–4672. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/jain20b.html.

[89] Karl Pertsch, Youngwoon Lee, and Joseph J. Lim. Accelerating reinforcement learning with learned skill priors. In *Conference on Robot Learning*, 2020.

[90] Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. In *Advances in Neural Information Processing Systems*. 2017.

[91] Risto Vuorio, Shao-Hua Sun, Hexiang Hu, and Joseph J Lim. Multimodal model-agnostic meta-learning via task-aware modulation. In *Neural Information Processing Systems*, 2019.

[92] Oriol Vinyals, Charles Blundell, Tim Lillicrap, Daan Wierstra, et al. Matching networks for one shot learning. In *Advances in Neural Information Processing Systems*, 2016.

[93] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. In *International Conference on Machine Learning*, 2017.

[94] Risto Vuorio, Shao-Hua Sun, Hexiang Hu, and Joseph J Lim. Toward multimodal model-agnostic meta-learning. *arXiv preprint arXiv:1812.07172*, 2018.

[95] Wei-Yu Chen, Yen-Cheng Liu, Zsolt Kira, Yu-Chiang Frank Wang, and Jia-Bin Huang. A closer look at few-shot classification. In *International Conference on Learning Representations*, 2019.

[96] Yoonho Lee and Seungjin Choi. Gradient-Based Meta-Learning with Learned Layerwise Metric and Subspace. In *International Conference on Machine Learning*, 2018.

[97] Alex Nichol and John Schulman. Reptile: a scalable metalearning algorithm. *arXiv preprint arXiv:1803.02999*, 2018.

[98] Garima Pruthi, Frederick Liu, Satyen Kale, and Mukund Sundararajan. Estimating training data influence by tracing gradient descent. In *Neural Information Processing Systems*, 2020.

[99] Yun-Chun Chen, Chao-Te Chou, and Yu-Chiang Frank Wang. Learning to learn in a semi-supervised fashion. In *European Conference on Computer Vision*, 2020.

[100] I.T. Jolliffe. *Principal Component Analysis*. Springer Verlag, 1986.

[101] Mihalj Bakator and Dragica Radosav. Deep learning and medical diagnosis: A review of literature. *Multimodal Technologies and Interaction*, 2018.

[102] Dinggang Shen, Guorong Wu, and Heung-Il Suk. Deep learning in medical image analysis. *Annual review of biomedical engineering*, 2017.

[103] Biraja Ghoshal and Allan Tucker. Estimating uncertainty and interpretability in deep learning for coronavirus (covid-19) detection. *arXiv preprint arXiv:2003.10769*, 2020.

[104] Chia-Jung Chang, Wei Guo, Jie Zhang, Jon Newman, Shao-Hua Sun, and Matt Wilson. Behavioral clusters revealed by end-to-end decoding from microendoscopic imaging. *bioRxiv*, 2021.

[105] Amitojdeep Singh, Sourya Sengupta, and Vasudevan Lakshminarayanan. Explainable deep learning models in medical image analysis. *Journal of Imaging*, 2020.

[106] Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 1966.

[107] Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*, 2020.

[108] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Conference on Software Engineering*, 2013.

[109] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018.

[110] Qi Xin and Steven P Reiss. Leveraging syntax-related code for automated program repair. In *International Conference on Automated Software Engineering*, 2017.

[111] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 2019.

[112] Eric Schulte, Stephanie Forrest, and Westley Weimer. Automated program repair through the evolution of assembly code. In *International Conference on Automated Software Engineering*, 2010.

[113] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, 2020.

[114] Thomas Durieux and Martin Monperrus. Dynamoth: dynamic code synthesis for automatic program repair. In *International Workshop on Automation of Software Test*, 2016.

[115] Yi Li, Shaohua Wang, and Tien N Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *International Conference on Software Engineering*, 2020.

[116] Liushan Chen, Yu Pei, and Carlo A Furia. Contract-based program repair without the contracts. In *International Conference on Automated Software Engineering*, 2017.

[117] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2019.

[118] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Association for the Advancement of Artificial Intelligence*, 2017.

[119] Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embedding for program repair. *arXiv preprint arXiv:1711.07163*, 2017.

[120] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. Deepdelta: learning to repair compilation errors. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[121] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. An inductive synthesis framework for verifiable reinforcement learning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019.

[122] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[123] Karol Hausman, Jost Tobias Springenberg, Ziyu Wang, Nicolas Heess, and Martin Riedmiller. Learning an embedding space for transferable robot skills. In *International Conference on Learning Representations*, 2018.

[124] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.