Learning Differentiable Programs with Admissible Neural Heuristics

Ameesh Shah[*]

UC Berkeley

ameesh@berkeley.edu

Eric Zhan[*]

Caltech

ezhan@caltech.edu

Jennifer J. Sun

Caltech

jjsun@caltech.edu

Abhinav Verma

UT Austin

verma@utexas.edu

Yisong Yue

Caltech

yyue@caltech.edu

Swarat Chaudhuri

UT Austin

swarat@cs.utexas.edu

Abstract

We study the problem of learning differentiable functions expressed as programs in a domain-specific language. Such programmatic models can offer benefits such as composability and interpretability; however, learning them requires optimizing over a combinatorial space of program "architectures". We frame this optimization problem as a search in a weighted graph whose paths encode top-down derivations of program syntax. Our key innovation is to view various classes of neural networks as continuous relaxations over the space of programs, which can then be used to complete any partial program. This relaxed program is differentiable and can be trained end-to-end, and the resulting training loss is an approximately admissible heuristic that can guide the combinatorial search. We instantiate our approach on top of the A* algorithm and an iteratively deepened branch-and-bound search, and use these algorithms to learn programmatic classifiers in three sequence classification tasks. Our experiments show that the algorithms outperform state-of-the-art methods for program learning, and that they discover programmatic classifiers that yield natural interpretations and achieve competitive accuracy.

1 Introduction

An emerging body of work advocates *program synthesis* as an approach to machine learning. The methods here learn functions represented as programs in symbolic, domain-specific languages (DSLs) [12, 11, 49, 44, 46, 45]. Such symbolic models have a number of appeals: they can be more interpretable than neural models, they use the inductive bias embodied in the DSL to learn reliably, and they use compositional language primitives to transfer knowledge across tasks.

In this paper, we study how to learn *differentiable* programs, which use structured, symbolic primitives to compose a set of parameterized, differentiable modules. Differentiable programs have recently attracted much interest due to their ability to leverage the complementary advantages of programming language abstractions and differentiable learning. For example, recent work has used such programs to compactly describe modular neural networks that operate over rich, recursive data types [44].

To learn a differentiable program, one needs to induce the program's "architecture" while simultaneously optimizing the parameters of the program's modules. This co-design task is difficult because the space of architectures is combinatorial and explodes rapidly. Prior work has approached this challenge using methods such as greedy enumeration, Monte Carlo sampling, Monte Carlo tree

---

[*]Equal Contribution.

search, and evolutionary algorithms [46, 44, 10]. However, such approaches can often be expensive, due to not fully exploiting the structure of the underlying combinatorial search problem.

In this paper, we show that the differentiability of programs opens up a new line of attack on this search problem. A standard strategy for combinatorial optimization is to exploit (ideally fairly tight) continuous relaxations of the search space [31, 6, 48, 42, 24, 2, 45]. Optimization in the relaxed space is typically easier and can efficiently guide search algorithms towards good or optimal solutions. In the case of program learning, we propose to use various classes of neural networks as relaxations of partial programs. We frame our problem as searching a graph, in which nodes encode program architectures with missing expressions, and paths encode top-down program derivations. For each partial architecture u encountered during this search, the relaxation amounts to substituting the unknown part of u with a neural network with free parameters. Because programs are differentiable, this network can be trained on the problem's end-to-end loss. If the space of neural networks is an (approximate) proper relaxation of the space of programs (and training identifies a near-optimum neural network), then the training loss for the relaxation can be viewed as an (approximately) admissible heuristic.

We instantiate our approach, called NEAR (abbreviation for *Neural Admissible Relaxation*), on top of two informed search algorithms: A* and an iteratively deepened depth-first search that uses a heuristic to direct branching as well as branch-and-bound pruning (IDS-BB). We evaluate the algorithms in the task of learning programmatic classifiers in three behavior classification applications. We show that the algorithms substantially outperform state-of-the-art methods for program learning, and can learn classifier programs that bear natural interpretations and are close to neural models in accuracy.

To summarize, the paper makes three contributions. First, we identify a tool — heuristics obtained by training neural relaxations of programs — for accelerating combinatorial searches over differentiable programs. So far as we know, this is the first approach to exploit the differentiability of a programming language in program synthesis. Second, we instantiate this idea using two classic search algorithms. Third, we present promising experimental results in three sequence classification applications.

2   Problem Formulation

We view a program in our domain-specific language (DSL) as a pair (α, θ), where α is a discrete *(program) architecture* and θ is a vector of real-valued parameters. The architecture α is generated using a *context-free grammar* [21]. The grammar consists of a set of rules $X \to \sigma_1 \ldots \sigma_k$, where X is a *nonterminal* and $\sigma_1, \ldots, \sigma_k$ are either nonterminals or *terminals*. A nonterminal stands for a missing subexpression; a terminal is a symbol that can actually appear in a program's code. The grammar starts with an initial nonterminal, then iteratively applies the rules to produce a series of *partial architectures*: sentences made from one or more nonterminals and zero or more terminals. The process continues until there are no nonterminals left, i.e., we have a complete architecture.

The *semantics* of the architecture α is given by a function $[\![\alpha]\!](x, \theta)$, defined by rules that are fixed for the DSL. We require this function to be differentiable in θ. Also, we define a *structural cost* for architectures. Let each rule r in the DSL grammar have a non-negative real cost $s(r)$. The structural cost of α is $s(\alpha) = \sum_{r \in \mathcal{R}(\alpha)} s(r)$, where $\mathcal{R}(\alpha)$ is the multiset of rules used to create α. Intuitively, architectures with lower structural cost are simpler are more human-interpretable.

To define our learning problem, we assume an unknown distribution $D(x, y)$ over inputs x and labels y, and consider the prediction error function $\zeta(\alpha, \theta) = \mathbb{E}_{(x,y) \sim D}[\mathbf{1}([\![\alpha]\!](x, \theta) \neq y)]$, where $\mathbf{1}$ is the indicator function. Our goal is to find an architecturally simple program with low prediction error, i.e., to solve the optimization problem:

$$(\alpha^*, \theta^*) = \arg \min_{(\alpha, \theta)} (s(\alpha) + \zeta(\alpha, \theta)). \tag{1}$$

Program Learning for Sequence Classification. Program learning is applicable in many settings; we specifically study it in the sequence classification context [9]. Now we sketch our DSL for this domain. Like many others DSLs for program synthesis [15, 3, 44], our DSL is purely functional. The language has the following characteristics:

- Programs in the DSL operate over two data types: real vectors and sequences of real vectors. We assume a simple type system that makes sure that these types are used consistently.

$$\alpha \quad ::= \quad x \mid c \mid \oplus(\alpha_1, \ldots, \alpha_k) \mid \oplus_\theta(\alpha_1, \ldots, \alpha_k) \mid \textbf{if } \alpha_1 \textbf{ then } \alpha_2 \textbf{ else } \alpha_3 \mid \textbf{sel}_S \, x$$

$$\textbf{map } (\textbf{fun } x_1.\alpha_1) \, x \mid \textbf{fold } (\textbf{fun } x_1.\alpha_1) \, c \, x \mid \textbf{mapprefix } (\textbf{fun } x_1.\alpha_1) \, x$$

Figure 1: Grammar of DSL for sequence classification. Here, x, c, $\oplus$, and $\oplus_\theta$ represent inputs, constants, basic algebraic operations, and parameterized library functions, respectively. $\textbf{fun } x.e(x)$ represents an anonymous function that evaluates an expression $e(x)$ over the input x. $\textbf{sel}_S$ returns a vector consisting of a subset S of the dimensions of an input x.

$\textbf{map } (\textbf{fun } x_t.$

  $\textbf{if } DistAffine_{[.0217];-.2785}(x_t)$

  $\textbf{then } AccAffine_{[-.0007,.0055,.0051,-.0025];3.7426}(x_t) \textbf{ else } DistAffine_{[-.2143];1.822}(x_t)) \, x$

Figure 2: Synthesized program classifying a "sniff" action between two mice in the CRIM13 dataset. $DistAffine$ and $AccAffine$ are functions that first select the parts of the input that represent distance and acceleration measurements, respectively, and then apply affine transformations to the resulting vectors. In the parameters (subscripts) of these functions, the brackets contain the weight vectors for the affine transformation, and the succeeding values are the biases. The program achieves an accuracy of 0.87 (vs. 0.89 for RNN baseline) and can be interpreted as follows: if the distance between two mice is small, they are doing a "sniff" (large bias in **else** clause). Otherwise, they are doing a "sniff" if the difference between their accelerations is small.

- Programs use a set of fixed algebraic operations $\oplus$ as well as a "library" of differentiable, parameterized functions $\oplus_\theta$. Because we are motivated by interpretability, the library used in our current implementation only contains affine transformations. In principle, it could be extended to include other kinds of functions as well.

- Programs use a set of higher-order combinators to recurse over sequences. In particular, we allow the standard **map** and **fold** combinators. To compactly express sequence-to-sequence functions, we also allow a special **mapprefix** combinator. Let g be a function that maps sequences to vectors. For a sequence x, $\textbf{mapprefix}(g, x)$ equals the sequence $\langle g(x_{[1:1]}), g(x_{[1:2]}), \ldots, g(x_{[1:n]}) \rangle$, where $x_{[1:i]}$ is the i-th prefix of x.

- Programs can use a conditional branching construct. However, to avoid discontinuities, we interpret this construct in terms of a smooth approximation:

$$[\![\textbf{if } \alpha_1 > 0 \textbf{ then } \alpha_2 \textbf{ else } \alpha_3]\!](x, (\theta_1, \theta_2, \theta_3))$$
$$= \sigma(\beta \cdot [\![\alpha_1]\!](x, \theta_1)) \cdot [\![\alpha_2]\!](x, \theta_2) + (1 - \sigma(\beta \cdot [\![\alpha_1]\!](x, \theta_1))) \cdot [\![\alpha_3]\!](x, \theta_3). \tag{2}$$

Here, σ is the sigmoid function and β is a temperature hyperparameter. As $\beta \to 0$, this approximation approaches the usual if-then-else construct.

Figure 1 summarizes our DSL in the standard Backus-Naur form [47]. Figures 2 and 3 show two programs synthesized by our learning procedure using our DSL with libraries of domain-specific affine transformations (see the supplementary material). Both programs offer an interpretation in their respective domains, while offering respectable performance against an RNN baseline.

3 Program Learning using NEAR

We formulate our program learning problem as a form of graph search. The search derives program architectures top-down: it begins with the *empty* architecture, generates a series of partial architectures following the DSL grammar, and terminates when a complete architecture is derived.

In more detail, we imagine a graph $\mathcal{G}$ in which:

- The node set consists of all partial and complete architectures permissible in the DSL.

- The *source node* u_0 is the empty architecture. Each complete architecture α is a *goal node*.

- Edges are directed and capture single-step applications of rules of the DSL. Edges can be divided into: (i) *internal edges* (u, u') between partial architectures u and u', and (ii) *goal edges* (u, α)

```
map (fun x_t.
  multiply(add(OffenseAffine(x_t), BallAffine(x_t)), add(OffenseAffine(x_t), BallAffine(x_t))) x
```

Figure 3: Synthesized program classifying the ballhandler for basketball. *OffenseAffine()* and *BallAffine()* are parameterized affine transformations over the XY-coordinates of the offensive players and the ball (see the appendix for full parameters). **multiply** and **add** are computed element-wise. The program structure can be interpreted as computing the Euclidean norm/distance between the offensive players and the ball and suggests that this quantity can be important for determining the ballhandler. On a set of learned parameters (not shown), this program achieves an accuracy of 0.905 (vs. 0.945 for an RNN baseline).

between partial architecture u and complete architecture α. An internal edge (u, u') exists if one can obtain u' by substituting a nonterminal in u following a rule of the DSL. A goal edge (u, α) exists if we can complete u into α by applying a rule of the DSL.

- The cost of an internal edge (u, u') is given by the structural cost $s(r)$, where r is the rule used to construct u' from u. The cost of a goal edge (u, α) is $s(r) + \zeta(\alpha, \theta^*)$, where $\theta^* = \arg\min_\theta \zeta(\alpha, \theta)$ and r is the rule used to construct α from u.

A path in the graph $\mathcal{G}$ is defined as usual, as a sequence of nodes $u_1, \ldots, u_k$ such that there is an edge (u_i, u_{i+1}) for each $i \in \{1, \ldots, k-1\}$. The cost of a path is the sum of the costs of these edges. Our goal is to discover a least-cost path from the source u_0 to some goal node α^*. Then by construction of our edge costs, α^* is an optimal solution to our learning problem in Eq. (1).

3.1 Neural Relaxations as Admissible Heuristics

The main challenge in our search problem is that our goal edges contain rich cost information, but this information is only accessible when a path has been explored until the end. A heuristic function $h(u)$ that can predict the value of choices made at nodes u encountered early in the search can help with this difficulty. If such a heuristic is *admissible* — i.e., underestimates the cost-to-go — it enables the use of informed search strategies such as A* and branch-and-bound while guaranteeing optimal solutions. Our NEAR approach (abbreviation for *Neural Admissible Relaxation*) uses neural approximations of spaces of programs to construct a heuristic that is ϵ-close to being admissible.

Let a *completion* of a partial architecture u be a (complete) architecture $u[\alpha_1, \ldots, \alpha_k]$ obtained by replacing the nonterminals in u by suitably typed architectures α_i. Let θ_u be the parameters of u and θ be parameters of the α_i-s. The *cost-to-go* at u is given by:

Figure 4: An example of program learning formulated as graph search. Structural costs are in red, heuristic values in black, prediction errors ζ in blue, O refers to a nonterminal in a partial architecture, and the path to a goal node returned by A*-NEAR search is in teal.

$$J(u) = \min_{\alpha_1, \ldots, \alpha_k, \theta_u, \theta} ((s(u[\alpha_1, \ldots, \alpha_k]) - s(u)) + \zeta(u[\alpha_1, \ldots, \alpha_k], (\theta_u, \theta))) \quad (3)$$

where the structural cost $s(u)$ is the sum of the costs of the grammatical rules used to construct u.

To compute a heuristic cost $h(u)$ for a partial architecture u encountered during search, we substitute the nonterminals in u with neural networks parameterized by ω. These networks are *type-correct* — for example, if a nonterminal is supposed to generate subexpressions whose inputs are sequences, then the neural network used in its place is recurrent. We show an example of NEAR used in a program learning-graph search formulation in Figure 4.

We view the neurosymbolic programs resulting from this substitution as tuples $(u, (\theta_u, \omega))$. We define a semantics for such programs by extending our DSL's semantics, and lift the function ζ to
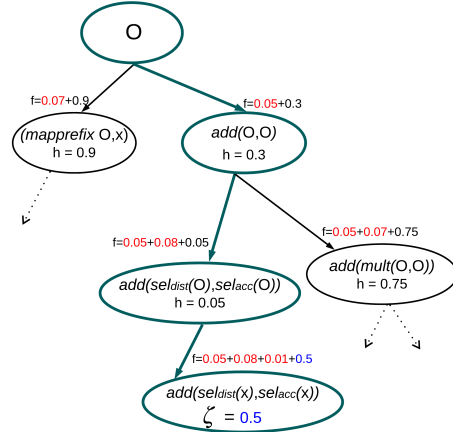
4

assign costs $\zeta(u, (\theta_u, \omega))$ to such programs. The heuristic cost for u is now given by:

$$h(u) = \min_{w,\theta} \zeta(u, (\theta_u, \omega)). \tag{4}$$

As $\zeta(u, (\theta_u, \omega))$ is differentiable in ω and θ_u, we can compute $h(u)$ using gradient descent.

ϵ-**Admissibility.** In practice, the neural networks that we use may only form an approximate relaxation of the space of completions and parameters of architectures; also, the training of these networks may not reach global optima. To account for these errors, we consider an approximate notion of admissibility. Many such notions have been considered in the past [20, 31, 43]; here, we follow a definition used by Harris [20]. For a fixed constant $\epsilon > 0$, let an ϵ-*admissible heuristic* be a function $h^*(u)$ over architectures such that $h^*(u) \leq J(u) + \epsilon$ for all u. Now consider any completion $u[\alpha_1, \ldots, \alpha_k]$ of an architecture u. As neural networks with adequate capacity are universal function approximators, there exist parameters ω^* for our neurosymbolic program such that for all $u, \alpha_1, \ldots, \alpha_k, \theta_u,$ and θ:

$$\zeta(u, (\theta_u, \omega^*)) \leq \zeta(u[\alpha_1, \ldots, \alpha_k], (\theta_u, \theta)) + \epsilon. \tag{5}$$

Because edges in our search graph have non-negative costs, $s(u) \leq s(u[\alpha_1, \ldots, \alpha_k])$, implying:

$$
\begin{aligned}
h(u) &\leq \min_{\alpha_1,\ldots,\alpha_k,\theta_u,\theta} \zeta(u[\alpha_1, \ldots, \alpha_k], (\theta_u, \theta)) + \epsilon \\
&\leq \min_{\alpha_1,\ldots,\alpha_k,\theta_u,\theta} \zeta(u[\alpha_1, \ldots, \alpha_k], (\theta_u, \theta)) + (s(u[\alpha_1, \ldots, \alpha_k]) - s(u)) + \epsilon = J(u) + \epsilon.
\end{aligned} \tag{6}
$$

In other words, $h(u)$ is ϵ-admissible.

Empirical Considerations. We have formulated our learning problem in terms of the true prediction error $\zeta(\alpha, \theta)$. In practice, we must use statistical estimates of this error. Following standard practice, we use an empirical validation error to choose architectures, and an empirical training error is used to choose module parameters. This means that in practice, the cost of a goal edge (u, α) in our graph is $\zeta^{val}(\alpha, \arg\min_\theta \zeta^{train}(\alpha, \theta))$.

One complication here is that our neural heuristics encode both the completions of an architecture and the parameters of these completions. Training a heuristic on either the training loss or the validation loss will introduce an additional error. Using standard generalization bounds, we can argue that for adequately large training and validation sets, this error is bounded (with probability arbitrarily close to 1) in either case, and that our heuristic is ϵ-admissible with high probability in spite of this error.

3.2 Integrating NEAR with Graph Search Algorithms

The NEAR approach can be used in conjunction with any heuristic search algorithm [36] over architectures. Specifically, we have integrated NEAR with two classic graph search algorithms: A^* [31] (Algorithm 1) and an iteratively deepened depth-first search with branch-and-bound pruning (IDS-BB) (Appendix A). Both algorithms maintain a *search frontier* by computing an *f-score* for each node: $f(u) = g(u) + h(u)$, where $g(u)$ is the incurred path cost from the source node u_0 to the current node u, and $h(u)$ is a heuristic estimate of the cost-to-go from node u. Additionally, IDS-BB prunes nodes from the frontier that have a higher f-score than the minimum path cost to a goal node found so far.

Algorithm 1: A* Search

Input: Graph $\mathcal{G}$ with source u_0
$S := \{u_0\}; f(u_0) := \infty;$
while $S \neq \emptyset$ **do**
    $v := \arg\min_{u \in S} f(u);$
    $S := S \setminus \{v\};$
    if v *is a goal node* **then**
        return $v, f_v;$
    else
        foreach *child u of v* **do**
            Compute $g(u), h(u), f(u);$
            $S := S \cup \{u\};$

ϵ-**Optimality.** An important property of a search algorithm is *optimality*: when multiple solutions exist, the algorithm finds an optimal solution. Both A^* and IDS-BB are optimal given admissible heuristics. An argument by Harris [20] shows that under heuristics that are ϵ-admissible in our sense, the algorithms return solutions that at most an additive constant ϵ away from the optimal solution. Let C^* denote the optimal path cost in our graph $\mathcal{G}$, and let $h(u)$ be an ϵ-admissible heuristic (Eq. (6)). Suppose IDS-BB or A* returns a goal node α_G that does not have the optimal path cost C^*. Then there must exist a node u_O on the frontier that lies along the optimal path and has yet to be expanded. This lets us establish an upper bound on the path cost of α_G:

$$g(\alpha_G) = f(\alpha_G) \leq f(u_O) = g(u_O) + h(u_O) \leq g(u_O) + J(u_O) + \epsilon \leq C^* + \epsilon. \tag{7}$$

This line of reasoning can also be extended to the Branch-and-Bound component of the NEAR-guided IDS-BB algorithm. Consider encountering a goal node during search that sets the branch-and-bound upper threshold to be a cost C. In the remainder of search, some node u_p with an f-cost greater than C is pruned, and the optimal path from u_p to a goal node will not be searched. Assuming the heuristic function h is ϵ-admissible, we can set a lower bound on the optimal path cost from u_p, $f(u_p^*)$, to be $C - \epsilon$ by the following:

$$f(u_p^*) = g(u_p) + J(u_p) \geq f(u_p) = g(u_p) + h(u_p) + \epsilon > C = g(u_p) + h(u_p) > C - \epsilon. \tag{8}$$

Thus, the IDS-BB algorithm will find goal paths are at worst an additive factor of ϵ more than any pruned goal path.

4 Experiments

4.1 Datasets for Sequence Classification

For all datasets below, we augment the base DSL in Figure 1 with domain-specific library functions that include 1) learned affine transformations over a subset of features, and 2) sliding window feature-averaging functions. Full details, such as structural cost functions used and any pre/post-processing, are provided in the appendix.

CRIM13. The *CRIM13* dataset [5] contains trajectories for a pair of mice engaging in social behaviors, annotated for different actions per frame by behavior experts; we aim to learn programs for classifying actions at each frame for fixed-size trajectories. Each frame is represented by a 19-dimensional feature vector: 4 features capture the xy-positions of the mice, and the remaining 15 features are derived from the positions, such as velocities and distance between mice. We learn programs for two actions that can be identified the tracking features: "sniff" and "other" ("other" is used when there is no behavior of interest occurring). We cut every 100 frames as a trajectory, and in total we have 12404 training, 3077 validation, and 2953 test trajectories.

Fly-vs.-Fly. We use the *Aggression* and *Boy-meets-Boy* datasets within the *Fly-vs.-Fly* environment that tracks a pair of fruit flies and their actions as they interact in different contexts [14]. We aim to learn programs that classify trajectories as one of 7 possible actions displaying aggressive, threatening, and nonthreatening behaviors. The length of trajectories can range from 1 to over 10000 frames, but we segment the data into trajectories with a maximum length of 300 for computational efficiency. The average length of a trajectory in our training set is 42.06 frames. We have 5339 training, 594 validation, and 1048 test trajectories.

Basketball. We use a subset of the basketball dataset from [50] that tracks the movements of professional basketball players. Each trajectory is of length 25 and contains the xy-positions of 5 offensive players, 5 defensive players, and the ball (22 features per frame). We aim to learn programs that can predict which offensive player has the ball (the "ballhandler") or whether the ball is being passed. In total, we have 18,000 trajectories for training, 2801 for validation, and 2693 for test.

4.2 Overview of Baseline Program Learning Strategies

We compare our NEAR-guided graph search algorithms, A*-NEAR and IDS-BB-NEAR, with four baseline program learning strategies: 1) top-down enumeration, 2) Monte-Carlo sampling, 3) Monte-Carlo tree search, and 4) a genetic algorithm. We also compare the performance of these program learning algorithms with an RNN baseline (1-layer LSTM).

Top-down enumeration. We synthesize and evaluate complete programs in order of increasing complexity measured using the structural cost $s(\alpha)$. This strategy is widely employed in program learning contexts [44, 46, 45] and is provably complete. Since our graph $\mathcal{G}$ grows infinitely, our implementation is akin to breadth-first search up to a specified depth.

Monte-Carlo (MC) sampling. Starting from the source node u_0, we sample complete programs by sampling rules (edges) with probabilities proportional to their structural costs $s(r)$. The next node chosen along a path has the best average performance of samples that descended from that node. We repeat the procedure until we reach a goal node and return the best program found among all samples.

	CRIM13-sniff			CRIM13-other			Fly-vs.-Fly			Bball-ballhandler		
	Acc.	F1	Depth	Acc.	F1	Depth	Acc.	F1	Depth	Acc.	F1	Depth
Enum.	.851	.221	3	.707	.762	2	.819	.863	2	.844	.857	6.3
MC	.843	.281	7	.630	.715	1	.833	.852	4	.841	.853	6
MCTS	.745	.338	8.7	.666	.749	1	.817	.857	4.7	.711	.729	8
Genetic	.829	.181	1.7	.727	.768	3	.850	.868	6	.843	.853	6.7
IDS-BB-NEAR	.829	.446	6	.729	.768	1.3	.876	.892	4	.889	.903	8
A*-NEAR	.821	.369	6	.706	.764	2.7	.872	.885	4	.906	.918	8
RNN	.889	.481	-	.756	.785	-	.963	.964	-	.945	.950	-

Table 1: Mean accuracy, F1-score, and program depth of learned programs (3 trials). Programs found using our NEAR algorithms consistently achieve better F1-score than baselines and match more closely to the RNN's performance. Our algorithms are also able to search and find programs of much greater depth than the baselines. Experiment hyperparameters are included in the appendix.
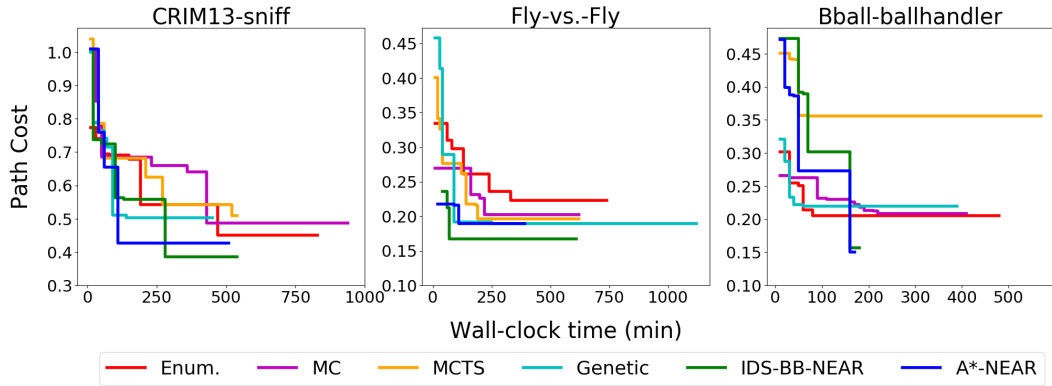

Figure 5: Median minimum path cost to a goal node found at a given time, across 3 trials (for trials that terminate first, we extend the plots so the median remains monotonic). A*-NEAR (blue) and IDS-BB-NEAR (green) will often find a goal node with a smaller path cost, or find one of similar performance but much faster.

Monte-Carlo tree search (MCTS). Starting from the source node u_0, we traverse the graph until we reach a complete program using the UCT selection criteria [23], where the value of a node is inversely proportional to the cost of its children.[2] In the backpropagation step we update the value of all nodes along the path. After some iterations, we choose the next node in the path with the highest value. We repeat the procedure until we reach a goal node and return the best program found.

Genetic algorithm. We follow the formulation in Valkov et al. [44]. In our genetic algorithm, crossover, selection, and mutation operations evolve a population of programs over a number of generations until a predetermined number of programs have been trained. The crossover and mutation operations only occur when the resulting program is guaranteed to be type-safe.

For all baseline algorithms, as well as A*-NEAR and IDS-BB-NEAR, model parameters (θ) were learned with the training set, whereas program architectures (α) were evaluated using the performance on the validation set. Additionally, all baselines (including NEAR algorithms) used F1-score [38] error as the evaluation objective ζ by which programs were chosen. To account for class imbalances, F1-scoring is commonly used as an evaluation metric in behavioral classification domains, such as those considered in our work [14, 5]. Our full implementation is available in [39].

4.3 Experimental Results

Performance of learned programs. Table 1 shows the performance results on the test sets of our program learning algorithms, averaged over 3 seeds. The same structural cost function $s(\alpha)$ is

---

[2]MCTS with this node value definition will visit shallow programs more frequently than MC sampling.
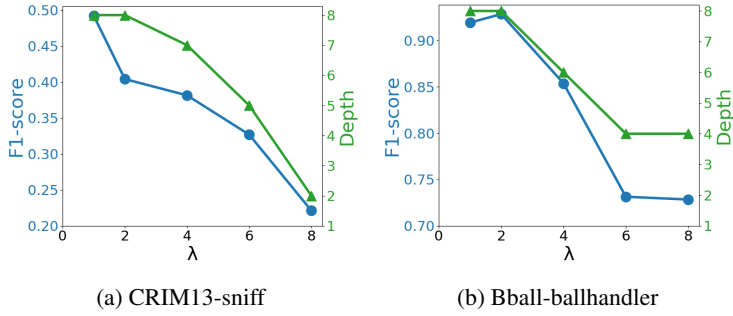
(a) CRIM13-sniff

(b) Bball-ballhandler

Figure 6: As we increase λ in Eq. (9), we observe that A*-NEAR will learn programs with decreasing program depth and also decreasing F1-score. This highlights that we can use λ to control the trade-off between structural cost and performance.

used for all algorithms, but can vary across domains (see Appendix). Our NEAR-guided search algorithms consistently outperform other baselines in F1-score while accuracy is comparable (note that our ζ does not include accuracy). Furthermore, NEAR-guided search algorithms are capable are finding deeper and more complex programs that can offer non-trivial interpretations, such as the ones shown in Figures 2 and 3. Lastly, we verify that our learned programs are comparable with highly expressive RNNs, and see that there is at most a 10% drop in F1-score when using NEAR-guided search algorithms with our DSL.

Efficiency of NEAR-guided graph search. Figure 5 tracks the progress of each program learning algorithm during search by following the median best path cost (Eq. (1)) at a given time across 3 independent trials. For times where only 2 trials are active (i.e. one trial had already terminated), we report the average. Algorithms for each domain were run on the same machine to ensure consistency, and each non-NEAR baseline was set up such to have at least as much time as our NEAR-guided algorithms for their search procedures (see Appendix). We observe that NEAR-guided search algorithms are able to find low-cost solutions more efficiently than existing baselines, while maintaining an overall shorter running time.

Cost-performance trade-off. We can also consider a modification of our objective in Eq. (1) that allows us to use a hyperparameter λ to control the trade-off between structural cost (a proxy for interpretability) and performance:

$$(\alpha^*, \theta^*) = \underset{(\alpha, \theta)}{\arg\min}(\lambda \cdot s(\alpha) + \zeta(\alpha, \theta)). \tag{9}$$

To visualize this trade-off, we run A*-NEAR with the modified objective Eq. (9) for various values of λ. Note that $\lambda = 1$ is equivalent to our experiments in Table 1. Figure 6 shows that for the *Basketball* and *CRIM13* datasets, as we increase λ, which puts more weight on the structural cost, the resulting programs found by A*-NEAR search have decreasing F1-scores but are also more shallow. This confirms our expectations that we can control the trade-off between structural cost and performance, which allows users of NEAR-guided search algorithms to adjust to their preferences. Unlike the other two experimental domains, the most performant programs learned in *Fly-vs.-Fly* were relatively shallow, so we omitted this domain as the trade-off showed little change in program depth.

We illustrate the implications of this tradeoff on interpretability using the depth-2 program in Figure 7 and the depth-8 program in Figure 8, both synthesized for the same task of detecting a "sniff" action in the CRIM13 dataset. The depth-2 program says that a "sniff" occurs if the intruder mouse is close to the right side of the cage and both mice are near the bottom of the cage, and can be seen to apply a *position bias* (regarding the location of the action) on the action. This program is simple, due to the large weight on the structural cost, and has a low F1-score. In contrast, the deeper program in Figure 8 has performance comparable to an RNN but is more difficult to interpret. Our interpretation of this program is that it evaluates the likelihood of "sniff" by applying a position bias, then using the velocity of the mice if the mice are close together and not moving fast, and using distance between the mice otherwise.

5   Related Work

Neural Program Induction.   The literature on *neural program induction* (NPI) [19, 34, 25, 37] develops methods to learn neural networks that can perform procedural (program-like) tasks, typically using architectures augmented with differentiable memory. Our approach differs from these methods

8

mapprefix (**fun** x_t.**SlidingWindowAverage**($PositionAffine(x_t)$)) x

Figure 7: Synthesized depth 2 program classifying a "sniff" action between two mice in the CRIM13 dataset. The sliding window average is over the last 10 frames. The program achieves F1 score of 0.22 (vs. 0.48 for RNN baseline). This program is synthesized using $\lambda = 8$.

map (**fun** x_t. **add**($PositionAffine(x_t)$,

$\quad$ **if** (**add**($VelocityAffine(x_t)$, $DistAffine(x_t)$) > 0)

$\quad$ **then** $VelocityAffine(x_t)$ **else** $DistAffine(x_t)$)) x

Figure 8: Synthesized depth 8 program classifying a "sniff" action between two mice in the CRIM13 dataset. The program achieves F1 score of 0.46 (vs. 0.48 for RNN baseline). This program is synthesized using $\lambda = 1$.

in that its final output is a symbolic program. However, since our heuristics are neural approximation of programs, our work can be seen as repeatedly performing NPI as the program is being produced. While we have so far used classical feedforward and recurrent architectures to implement our neural heuristics, future work could use richer models from the NPI literature to this end.

DSL-based Program Synthesis. There is a large body of research on synthesis of programs from DSLs. In many of these methods, the goal is not learning but finding a program that satisfies a hard constraint [1, 41, 32, 15]. However, there is also a growing literature on learning programs from (noisy) data [26, 13, 46, 11, 44, 45]. Of these methods, TERPRET [18] and NEURAL TERPRET [17] allows gradient descent as a mechanism for learning program parameters. However, unlike NEAR, these approaches do not allow a general search over program architectures permitted by a DSL, and require a detailed hand-written template of the program for even the simplest tasks. While the Houdini framework [44] combines gradient-based parameter learning with search over program architectures, this search is not learning-accelerated and uses a simple type-directed enumeration. As reported in our experiments, NEAR outperforms this enumeration-based approach.

Many recent methods for program synthesis use statistical models to guide the search over program architectures [3, 7, 12, 8, 11, 30, 16, 29]. In particular, Lee et al. [27] use a probabilistic model to guide an A* search over programs. Most of these models (including the one in Lee et al. [27]) are trained using corpora of synthesis problems and corresponding solutions, which are not available in our setting. There is a category of methods based on reinforcement learning (RL) [16, 4]. Unlike NEAR, these methods do not directly exploit the structure of the search space. Combining them with our approach would be an interesting topic of future work.

Structure Search using Relaxations. Our search problem bears similarities with the problems of searching over neural architectures and the structure of graphical models. Prior work has used relaxations to solve these problems [28, 40, 51, 33, 48]. Specifically, the A* lasso approach for learning sparse Bayesian networks [48] uses a dense network to construct admissible heuristics, and DARTS computes a differentiable relaxation of neural architecture search [28, 40]. The key difference between these efforts and ours is that the design space in our problem is much richer, making the methods in prior work difficult to apply. In particular, DARTS uses a composition of softmaxes over all possible candidate operations between a fixed set of nodes that constitute a neural architecture, and the heuristics in the A* lasso method come from a single, simple function class. However, in our setting, there is no fixed bound on the number of expressions in a program, different sets of operations can be available at different points of synthesis, and the input and output type of the heuristic (and therefore, its architecture) can vary based on the part of the program derived so far.

6 Conclusions

We have a presented a novel graph search approach to learning differentiable programs. Our method leverages a novel construction of an admissible heuristic using neural relaxations to efficiently search over program architectures. Our experiments show that programs learned using our approach can have

competitive performance, and that our search-based learning procedure substantially outperforms conventional program learning approaches.

There are many directions for future work. One direction is to extend the approach to richer DSLs and neural heuristic architectures, for example, those suited to reinforcement learning [45] and generative modeling [35]. Another is to combine NEAR with classical program synthesis methods based on symbolic reasoning. A third is to integrate NEAR into more complex program search problems, e.g., when there is an initial program as a starting point and the goal is to search for refinements. A fourth is to more tightly integrate with real-world applications to evaluate the interpretability of learned programs as it impacts downstream tasks.

Broader Impact

Programmatic models described using high-level DSLs are a powerful mechanism for summarizing automatically discovered knowledge in a human-interpretable way. Specifically, such models are more interpretable than state-of-the-art neural models while also tending to provide higher performance than shallower linear or decision tree models. Also, programmatic models allow for natural incorporation of inductive bias and allow the user to influence the semantic meaning of learned programs.

For these reasons, efforts on program learning, such as ours, can lead to more widespread use of machine learning in fields, such as healthcare, autonomous driving, and the natural sciences, where safety and accountability are critical and there human-held prior knowledge (such as the laws of nature) that can usefully direct the learning process. The flipside of this is that the bias introduced in program learning can just as easily be exploited by users who desire specific outcomes from the learner. Ultimately, users of program learning methods must ensure that any incorporated inductive bias will not lead to unfair or misleading programs.

Funding Acknowledgment

References

[1] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, pages 1–25. 2015.

[2] A Bagchi and A Mahanti. Admissible heuristic search in and/or graphs. *Theoretical Computer Science*, 24(2):207–219, 1983.

[3] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[4] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

[5] Xavier P Burgos-Artizzu, Piotr Dollár, Dayu Lin, David J Anderson, and Pietro Perona. Social behavior recognition in continuous video. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1322–1329. IEEE, 2012.

[6] Eugene Charniak and Saadia Husain. *A new admissible heuristic for minimal-cost proofs*. Brown University, Department of Computer Science, 1991.

[7] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[8] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. *CoRR*, abs/1703.07469, 2017.

[9] Thomas G. Dietterich. Machine learning for sequential data: A review. In *Structural, Syntactic, and Statistical Pattern Recognition, Joint IAPR International Workshops SSPR 2002 and SPR 2002, Windsor, Ontario, Canada, August 6-9, 2002, Proceedings*, pages 15–30, 2002.

[10] Kevin Ellis, Maxwell I. Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a REPL. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 9165–9174, 2019.

[11] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. In *Advances in Neural Information Processing Systems*, pages 6059–6068, 2018.

[12] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Sampling for bayesian program learning. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 1289–1297, 2016.

[13] Kevin Ellis, Armando Solar-Lezama, and Joshua B. Tenenbaum. Unsupervised learning by program synthesis. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 973–981, 2015.

[14] Eyrun Eyjolfsdottir, Steve Branson, Xavier P Burgos-Artizzu, Eric D Hoopfer, Jonathan Schor, David J Anderson, and Pietro Perona. Detecting social actions of fruit flies. In *European Conference on Computer Vision*, pages 772–787. Springer, 2014.

[15] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015.

[16] Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, S. M. Ali Eslami, and Oriol Vinyals. Synthesizing programs for images using reinforced adversarial learning. *CoRR*, abs/1804.01118, 2018.

[17] Alexander L Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable programs with neural libraries. In *International Conference on Machine Learning*, pages 1213–1222, 2017.

[18] Alexander L Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*, 2016.

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

[20] Larry R Harris. The heuristic search under conditions of error. *Artificial Intelligence*, 5(3):217–234, 1974.

[21] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

[22] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[23] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[24] Richard E Korf. Recent progress in the design and analysis of admissible heuristic functions. In *International Symposium on Abstraction, Reformulation, and Approximation*, pages 45–55. Springer, 2000.

[25] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.

[26] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[27] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 436–449, 2018.

[28] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[29] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *ICLR*, 2018.

[30] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

[31] Judea Pearl. Heuristics: Intelligent search strategies for computer problem solving. *Addision Wesley*, 1984.

[32] Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 107–126, 2015.

[33] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 4780–4789. AAAI Press, 2019.

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

[35] Daniel Ritchie, Anna Thomas, Pat Hanrahan, and Noah Goodman. Neurally-guided procedural models: Amortized inference for procedural graphics programs using neural networks. In *Advances in neural information processing systems*, pages 622–630, 2016.

[36] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002.

[37] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy P. Lillicrap. Meta-learning with memory-augmented neural networks. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 1842–1850. JMLR.org, 2016.

[38] Yutaka Sasaki. The truth of the f-measure. *Teach Tutor Master*, 01 2007.

[39] Ameesh Shah, Eric Zhan, and Jennifer Sun. Near code repository. `https://github.com/trishullab/near`, 2020.

[40] Richard Shin, Charles Packer, and Dawn Song. Differentiable neural network architecture search. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net, 2018.

[41] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[42] David Sontag, Talya Meltzer, Amir Globerson, Tommi S Jaakkola, and Yair Weiss. Tightening lp relaxations for map using message passing. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2012.

[43] Richard Anthony Valenzano, Shahab Jabbari Arfaee, Jordan Thayer, Roni Stern, and Nathan R Sturtevant. Using alternative suboptimality bounds in heuristic search. In *Twenty-Third International Conference on Automated Planning and Scheduling*, 2013.

[44] Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles A. Sutton, and Swarat Chaudhuri. HOUDINI: lifelong learning as program synthesis. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 8701–8712, 2018.

[45] Abhinav Verma, Hoang Minh Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[46] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, pages 5052–5061, 2018.

[47] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[48] Jing Xiang and Seyoung Kim. A* lasso for learning a sparse bayesian network structure for continuous variables. In Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pages 2418–2426, 2013.

[49] Halley Young, Osbert Bastani, and Mayur Naik. Learning neurosymbolic generative models via program synthesis. In *International Conference on Machine Learning (ICML)*, 2019.

[50] Yisong Yue, Patrick Lucey, Peter Carr, Alina Bialkowski, and Iain Matthews. Learning fine-grained spatial models for dynamic sports play prediction. In *2014 IEEE international conference on data mining*, pages 670–679. IEEE, 2014.

[51] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

A IDS-BB

In Algorithm 2, we provide the pseudocode for the IDS-BB algorithm introduced in the main text. This algorithm is a Heuristic-Guided Depth-First Search with three key characteristics: (1) the search depth is iteratively increased; (2) the search is ordered using a function $f(u)$ as in A*, and (3) Branch-and-Bound is used to prune unprofitable parts of the search space. We find that the use of iterative deepening in the program learning setting is useful in that it prioritizes searching shallower and less parsimonious programs early on in the search process.

Algorithm 2: Iterative Deepening Depth-First-Search

Input: Initial depth $d_{initial}$, Max depth d_{max}
Initialize *frontier* to a priority-queue with root node *root* ;
Initialize *nextfrontier* to an empty priority-queue;
$(f_{root}, f_{min}, d_{iter}) = (\infty, \infty, d_{initial})$;
$current = None$;
while *frontier is not empty* **do**
    if *current is None* **then**
        pop node with lowest f from *frontier* and assign to *current*;
    if *current is a leaf node* **then**
        $f_{min} := \min(f_{current}, f_{min})$;
        $current := None$;
    else
        if $d_{current} > d_{iter}$ **then**
            $current := None$;
        else
            Set *current* to child with lowest f;
            if $d_{current} \leq d_{max}$ **then**
                Evaluate and add all children of *current* to *frontier*;
            if *frontier is empty* **then**
                $frontier := nextfrontier$;
                $d_{iter} := d_{iter} + 1$;
return f_{min};

B Additional details on informed search algorithms

In tables 3, 4, 5, 6, and 7 we present the hyperparameters used in our implementation for all baselines. Usage of each hyperparameter can be found in our codebase. We elaborate below on hyperparameters specific to our contribution, namely A*-NEAR and IDS-BB-NEAR.

In A*-NEAR and IDS-BB-NEAR, we allow for a number of hyperparameters to be used that can additionally speed up our search. To improve efficiency, we allow for the frontier in these searches to be bounded by a constant size. In doing so, we sacrifice the completeness guarantees discussed in the main text in exchange for additional efficiency. We also allow for a scalar performance multiplier, which is a number greater than zero, that is applied to each node in the frontier when a goal node is found. The nodes on the frontier must have a lower cost than the goal node after this performance multiplier is applied; otherwise, they are pruned from the frontier in the case of branch-and-bound. When considering non-goal nodes, this multiplier is not applied. We introduce an additional parameter that decreases this performance multiplier as nodes get farther from the root; i.e become more complete programs. We also decrease the number of units given to a neural network within a *neural program approximation* as nodes get further from the root, with the intuition that neural program induction done in a more complete program will likely have less complex behavior to induce. We also allow for the branching factor of all nodes in the graph to be bounded to a user-specified width in order to bound the combinatorial explosion of program space. This constraint comes at the expected sacrifice of completeness in our program search, given that potentially optimal paths are arbitrarily not considered.

When searching over these hyperparameters, we noted that the the performance of the neural program approximations as a heuristic required a balance in their levels of accuracy with respect to the given optimization objective. If the approximations are heavily underparameterized, the NEAR heuristic

	feature dim	label dim	max seq len	# train	# valid	# test
CRIM13-sniff	19	2	100	12404	3007	2953
CRIM13-other	19	2	100	12404	3007	2953
Fly-vs.-Fly	53	7	300	5339	594	1048
Bball-ballhandler	22	6	25	18000	2801	2893

Table 2: Dataset details.

	max depth	init. # units	min # units	max # children	penalty	β
CRIM13-sniff	10	15	6	8	0.01	1.0
CRIM13-other	10	15	6	8	0.01	1.0
Fly-vs.-Fly	6	25	10	6	0.01	1.0
Bball-ballhandler	8	16	4	8	0.01	1.0

Table 3: Hyperparameters for constructing graph $\mathcal{G}$.

may not reach the same level of expressivity as the DSL, and as a result, poor performance can lead to the heuristic being inadmissible. However, if the approximations are so expressive that they are able to achieve near-perfect accuracy on the task, admissibility will still hold, but the resulting heuristic will become uninformative and inefficient. For example, a NEAR heuristic that only returns 0 will remain admissible, but the performance of A*-NEAR in this case will be no better than breadth-first search. With this in mind, we searched for a set of hyperparameters that yielded an informative level of performance.

In our experiments, we show that using the aforementioned approximative hyperparameters allows for an accelerated search while maintaining strong empirical results with our NEAR-guided search algorithms.

C   Details of Experimental Domains

C.1   Fly-v.-Fly

The *Fly-vs.-Fly* dataset [14] tracks a pair of flies and their actions as they interact in different contexts. Each timestep is represented by a 53-dimensional feature vector including 17 features outlining the fly's position and orientation along with 36 position-invariant features, such as linear and angular velocities. Our task in this domain is that of *bout-level classification*, where we are tasked to classify a given trajectory of timesteps to a corresponding single action taking place. Of the three datasets within *Fly-vs.-Fly*, we use the *Aggression* and *Boy-meets-Boy* datasets and classify trajectories over the 7 labeled actions displaying aggressive, threatening, and nonthreatening behaviors in these two datasets. We omit the use of the *Courtship* dataset for our classification task, primarily due to the heavily skewed trajectories in this dataset that vary highly in length and action type from the *Aggression* and *Boy-meets-Boy* datasets. Full details on these datasets, as well as where to download them, can be found in [14]. To ensure a desired balance in our training set, we limit the length of trajectories to 300 timesteps, and break up trajectories that exceed this length into separate trajectories with the

	# LSTM units	# epochs	learning rate	batch size
CRIM13-sniff	100	50	0.001	50
CRIM13-other	100	50	0.001	50
Fly-vs.-Fly	80	40	0.00025	30
Bball-ballhandler	64	15	0.01	50

Table 4: Training hyperparameters for RNN baseline.

	# neural epochs	# symbolic epochs	learning rate	batch size
CRIM13-sniff	6	15	0.001	50
CRIM13-other	6	15	0.001	50
Fly-vs.-Fly	6	25	0.00025	30
Bball-ballhandler	4	6	0.02	50

Table 5: Training hyperparameters for all program learning algorithms. The # neural epochs hyperparameter refers only to the number of epochs that neural program approximations were trained in NEAR strategies.

	A*-NEAR	IDS-BB-NEAR			
	frontier size	frontier size	init. depth	depth bias	perf. mult.
CRIM13-sniff	8	8	5	0.95	0.975
CRIM13-other	8	8	5	0.95	0.975
Fly-vs.Fly	10	10	4	0.9	0.95
Bball-ballhander	400	30	3	1.0	1.0

Table 6: Additional hyperparameters for A*-NEAR and IDS-BB-NEAR.

same action label for data augmentation. Our training dataset has 5339 trajectories, our validation set has 594 trajectories, and our test set has 1048 trajectories. The average length of a trajectory is 42.06 timesteps.

Training details of Fly-v.-Fly baselines. For all of our program synthesis baselines , we used the Adam [22] optimizer and cross-entropy loss. Each synthesis baseline was run on an Intel 4.9-GHz i7 CPU with 8 cores, equipped with an NVIDIA RTX 2070 GPU w/ 2304 CUDA cores.

C.2   CRIM13

The CRIM13 dataset studies the social behavior of a pair of mice annotated each frame by behavior experts [5] at 25Hz. The interaction between a resident mouse and an intruder mouse, which is introduced to the cage of the resident, is recorded. Each mice is tracked by one keypoint and a 19 dimensional feature vector based on this tracking data is provided at each frame. The feature vector consists of features such as velocity, acceleration, distance between mice, angle and angle changes. Our task in this domain is *sequence classification*: we classify each frame with a behavior label from CRIM13. Every frame is labelled with one of 12 actions, or "other". The "other" class corresponds to cases where no action of interest is occurring. Here, we focus on two binary classification tasks: other vs. rest, and sniff vs. rest. The first task, other vs. rest, corresponds to labeling whether there is an action of interest in the frame. The second task, sniff vs. rest, corresponds to whether the resident mouse is sniffing any part of the intruder mouse. These two tasks are chosen such that the RNN baseline has reasonable performance only using the tracked keypoint features of the mice. We split the train set in [5] at the video level into our train and validation set, and we present test set results on the same set as [5]. Each video is split into sequences of 100 frames. There are 12404 training trajectories, 3077 validation trajectories, and 2953 test trajectories.

We observed higher variance in F1 score for the CRIM13-sniff class in Table 8, as compared to the other experiments. For this particular class, due to the high variance of both baseline and NEAR runs, we would like to note the importance of repeating runs.

Training details of CRIM13 baselines. All CRIM13 baselines training uses the Adam [22] optimizer and cross-entropy loss. In the loss for sniff vs. rest, the sniff class is weighted by 1.5. Each synthesis baseline was run on an Intel 2.2-GHz Xeon CPU with 4 cores, equipped with an NVIDIA Tesla P100 GPU with 3584 CUDA cores.

	MC(TS)	Enum.	Genetic					
	samples /step	max # prog.	pop. size	select. size	# gens	total # evals	mutate prob.	enum. depth
CRIM13-sniff	50	300	15	8	20	100	0.1	5
CRIM13-other	50	300	15	8	20	100	0.1	5
Fly-vs.Fly	25	100	20	10	10	10	0.1	6
Bball-ballhander	150	1200	100	50	10	1000	0.01	7

Table 7: Additional hyperparameters for other program learning baselines

	CRIM13-sniff			**CRIM13-other**			**Fly-vs.-Fly**			**Bball-ballhandler**		
	Acc.	F1	Depth	Acc.	F1	Depth	Acc.	F1	Depth	Acc.	F1	Depth
Enum.	.024	.105	1	.036	.011	1	.013	.012	0	.009	.009	0.6
MC	.013	.127	1.7	.088	.031	0.6	.028	.018	2	.012	.012	0.6
MCTS	.047	.076	0	.103	.036	0	.008	.009	0.94	.003	.002	0
Genetic	.003	.015	0.6	.005	.004	1.7	.028	.030	1	.016	.019	0.6
IDDFS-NEAR	.021	.056	2	.006	.005	0.6	.023	.016	0	.006	.006	0
A*-NEAR	.026	.114	1.7	.030	.010	2.1	.003	.004	0	.034	.034	0
RNN	.008	.019	-	.005	.002	-	.006	.005	-	.001	.001	-

Table 8: Standard Deviations of accuracy, F1-score, and program depth of learned programs (3 trials).

C.3   Basketball

The basketball data tracks player positions (xy-coordinates on court) from real professional games. We used the processed version from [50], which includes trajectories over 8 seconds (3 Hz in our case of sequence length 25) centered on the left half-court. Among the offensive and defensive teams, players are ordered based on their relative positions. Labels for the ballhandler were extracted with a labeling function written by a domain expert. See Table 2 for full details of this dataset.

Training details of Basketball baselines. All Basketball experiments use Adam [22] and optimize cross-entropy loss. Each synthesis baseline was run on an Intel 3.6-GHz i7-7700 CPU with 4 cores, equipped with an NVIDIA GTX 1080 Ti GPU with 3584 CUDA cores.