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
Rice University
averma@rice.edu

Yisong Yue
Caltech
yyue@caltech.edu

Swarat Chaudhuri
The University of Texas at 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) [10, 9, 41, 36, 38, 37]. 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 [36].

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 ranging such as greedy enumeration, Monte Carlo sampling, and evolutionary

∗Equal Contribution. Correspondence to ameesh@berkeley.edu and ezhan@caltech.edu.
We view a program in our domain-specific language (DSL) as a pair \((\alpha, \theta)\), where \(\alpha\) is a discrete \textit{architecture} and \(\theta\) is a vector of real-valued parameters. The architecture \(\alpha\) is generated using a context-free grammar [17]. The grammar consists of a set of rules \(X \rightarrow \sigma_1 \ldots \sigma_k\), 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 \textit{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 \textit{semantics} of the architecture \(\alpha\) 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 \(\theta\). Also, we define a \textit{structural cost} for architectures. Let each rule \(r\) in the DSL grammar have a non-negative real cost \(s(r)\). The structural cost of \(\alpha\) is \(s(\alpha) = \sum_{r \in \mathcal{R}(\alpha)} s(r)\), where \(\mathcal{R}(\alpha)\) is the multiset of rules used to create \(\alpha\).

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 \(\zeta(\alpha, \theta) = \mathbb{E}_{(x,y) \sim D}[1([\alpha](x, \theta) \neq y)]\), where \(1\) is the indicator function. Our goal is to find an architecturally simple program with low prediction error, i.e., to solve the optimization problem:

\[
(\alpha^*, \theta^*) = \arg\min_{(\alpha, \theta)} (s(\alpha) + \zeta(\alpha, \theta)).
\]
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.

Program Learning for Sequence Classification. Program learning is applicable in many settings; we specifically study it in the sequence classification context [2]. Now we sketch our DSL for this domain. Like many others DSLs for program synthesis [13, 2, 56], 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.
- 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 $x$, maprefix$(g,x)$ 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 use a conditional branching construct. However, to avoid discontinuities, we interpret this construct in terms of a smooth approximation:

$$\begin{align*}
\text{if } \alpha_1 > 0 \text{ then } \alpha_2 \text{ else } \alpha_3 \langle x, (\theta_1, \theta_2, \theta_3) \rangle
\end{align*}$$

$$\sigma(\beta \cdot \alpha_1(x, \theta_1)) \cdot \sigma(\alpha_2(x, \theta_2) + (1 - \sigma(\beta \cdot \alpha_1(x, \theta_1))) \cdot \alpha_3(x, \theta_3).$$

Here, $\sigma$ is the sigmoid function and $\beta$ 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 [39]. 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 $\alpha$ 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 $(u, \alpha)$ between partial architecture $u$ and complete architecture $\alpha$. 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 $(u, \alpha)$ exists if we can complete $u$ into $\alpha$ using 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 $(u, \alpha)$ is $s(r) + \zeta(\alpha, \theta^*)$, where $\theta^* = \arg \min_\theta \zeta(\alpha, \theta)$ and $r$ is the rule used to construct $\alpha$ 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 $\alpha^*$. Then by construction of our edge costs, $\alpha^*$ 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 edge costs 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 $c$-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 $\alpha_i$. Let $\theta_u$ be the parameters of $u$ and $\theta$ be parameters of the $\alpha_i$-s. The cost-to-go at $u$ is given by:

$$J(u) = \min_{\alpha_1, \ldots, \alpha_k, \theta_u, \theta} \left( (s(u[\alpha_1, \ldots, \alpha_k]) - s(u)) + \zeta(u[\alpha_1, \ldots, \alpha_k], (\theta_u, \theta)) \right)$$  \hspace{1cm} (3)

where the structural cost $s(u[\alpha_1, \ldots, \alpha_k])$ 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 $\omega$. 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 $\zeta$ to assign costs $\zeta(u, (\theta_u, \omega))$ to such programs. The heuristic cost for $u$ is now given by:

$$h(u) = \min_{\omega, \theta} \zeta(u, (\theta_u, \omega))$$  \hspace{1cm} (4)

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

$c$-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 [16, 25, 35]; here, we follow a definition used by Harris [16]. For a fixed constant \( \epsilon > 0 \), let an \( \epsilon \)-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 \( \omega^* \) for our neurosymbolic program such that for all \( u, \alpha_1, \ldots, \alpha_k, \theta_u, \) and \( \theta^* \):

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

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

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

In other words, \( h(u) \) is \( \epsilon \)-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 \( (u, \alpha) \) in our graph is \( \zeta^{\text{val}}(\alpha, \arg \min_{\theta} \zeta^{\text{train}}(\alpha, \theta)) \).

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 \( \epsilon \)-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 [30] over architectures. Specifically, we have integrated NEAR with two classic graph search algorithms: \( A^* \) [25] (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.

\( \epsilon \)-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 [16] shows that under heuristics that are \( \epsilon \)-admissible in our sense, the algorithms return solutions that at most an additive constant \( \epsilon \) away from the optimal solution. Let \( C^* \) denote the optimal path cost in our graph \( G \), and let \( h(u) \) be an \( \epsilon \)-admissible heuristic (Eq. (6)). Suppose IDS-BB or \( A^* \) returns a goal node \( \alpha_G \) 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 \( \alpha_G \):

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

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 \( \epsilon \)-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:

### Algorithm 1: A* Search

**Input:** Graph \( G \) 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
\( \text{return } v, f_v; \)
else
\( \text{foreach child } u \text{ of } v \text{ do} \)
Compute \( g(u), h(u), f(u); \)
\( S := S \cup \{u\}; \)
end for
end if
end while
\[ f(u_p^*) = g(u_p) + J(u_p) \geq f(u_p) = g(u_p) + h(u_p) + \epsilon > C = g(u_p) + h(u_p) > C - \epsilon \quad (8) \]

Thus, the IDS-BB algorithm will find goal paths at worst an additive factor of \(\epsilon\) 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 \([4]\) 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 \([12]\). 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 \([42]\) 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 three baseline program learning strategies: 1) top-down enumeration, 2) Monte-Carlo sampling, and 3) 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 \([36, 38, 37]\) 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 their structural costs \(s(r)\). The next node in the path has the best average performance of samples that descend from that node. We repeat the procedure until we reach a goal node and return the best program found among all samples.

**Genetic algorithm.** We follow the formulation in Valkov et al. \([36]\). 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 (\(\theta\)) were learned with the training set, whereas program architectures (\(\alpha\)) were evaluated using the performance.
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.

Table 1

<table>
<thead>
<tr>
<th></th>
<th>CRIM13-sniff</th>
<th>CRIM13-other</th>
<th>Fly-vs.-Fly</th>
<th>Bball-ballhandler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Acc. F1 Depth</td>
<td>Acc. F1 Depth</td>
<td>Acc. F1 Depth</td>
<td>Acc. F1 Depth</td>
</tr>
<tr>
<td>Enum.</td>
<td>.851 .221 3</td>
<td>.707 .762 2</td>
<td>.819 .863 2</td>
<td>.844 .857 6.3</td>
</tr>
<tr>
<td>MC</td>
<td>.843 .281 7</td>
<td>.630 .715 1</td>
<td>.833 .852 4</td>
<td>.841 .853 6</td>
</tr>
<tr>
<td>IDS-BB-NEAR</td>
<td>.831 .452 7.5</td>
<td>.704 .760 8.7</td>
<td>.876 .892 4</td>
<td>.889 .903 8</td>
</tr>
<tr>
<td>A*-NEAR</td>
<td>.826 .448 7.7</td>
<td>.723 .770 7.7</td>
<td>.872 .885 4</td>
<td>.906 .918 8</td>
</tr>
<tr>
<td>RNN</td>
<td>.889 .481 -</td>
<td>.756 .785 -</td>
<td>.963 .964 -</td>
<td>.945 .950 -</td>
</tr>
</tbody>
</table>

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.

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 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 $\zeta$ does not include accuracy). Furthermore, NEAR-guided search algorithms are capable of finding deeper and more complex programs that can offer non-trivial interpretations, such as the ones on the validation set. Additionally, all baselines (including NEAR algorithms) used F1-score as the evaluation objective $\zeta$ 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.

Figure 6: As we increase $\lambda$ 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 $\lambda$ to control the trade-off between structural cost and performance.
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 \( \lambda \) to control the trade-off between structural cost and performance:

\[
(\alpha^*, \theta^*) = \arg \min_{(\alpha, \theta)} (\lambda \cdot s(\alpha) + \zeta(\alpha, \theta)).
\]

To visualize this trade-off, we run A*-NEAR with the modified objective Eq. (9) for various values of \( \lambda \). 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 \( \lambda \), 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.

5 Related Work

Neural Program Induction. The literature on neural program induction (NPI) 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 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 most of these methods, the goal is to find a program that satisfies a hard constraint. However, the problem of learning programs that optimize a quantitative objective has also been studied. Many recent methods in this area use statistical models to guide the synthesis process. In particular, Lee et al. use a probabilistic model to guide an A* search over programs. Most of these models (including the one in Lee et al.) 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) (LEE, etc.) 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. We are aware of only one program synthesis effort that explicitly targets the synthesis of differentiable programs. However, unlike NEAR, the combinatorial search in that work neither receives neural guidance nor does it exploit the programs’ differentiability.

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. Specifically, the A* lasso approach for learning sparse Bayesian networks uses a dense network to construct admissible heuristics, and DARTS computes a differentiable relaxation of neural architecture search. 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 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 [37] and generative modeling [29]. Another is to combine NEAR with classical program synthesis methods based on symbolic reasoning. A third is to more tightly integrate with real-world applications to evaluate the interpretability of learned programs.
References


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 $d_{\text{initial}}$, Max depth $d_{\text{max}}$

Initialize $\text{frontier}$ to a priority-queue with root node $\text{root}$;

Initialize $\text{nextfrontier}$ to an empty priority-queue;

$(f_{\text{root}}, f_{\text{min}}, d_{\text{iter}}) = (\infty, \infty, d_{\text{initial}})$;

$\text{current} = \text{None}$;

**while** $\text{frontier}$ is not empty **do**

  **if** $\text{current}$ is None **then**

    pop node with lowest $f$ from $\text{frontier}$ and assign to $\text{current}$;

  **if** $\text{current}$ is a leaf node **then**

    $f_{\text{min}} := \min(f_{\text{current}}, f_{\text{min}})$;

    $\text{current} := \text{None}$;

  **else**

    **if** $d_{\text{current}} > d_{\text{iter}}$ **then**

      $\text{current} := \text{None}$;

    **else**

      Set $\text{current}$ to child with lowest $f$;

      **if** $d_{\text{current}} \leq d_{\text{max}}$ **then**

        Evaluate and add all children of $\text{current}$ to $\text{frontier}$;

      **if** $\text{frontier}$ is empty **then**

        $\text{frontier} := \text{nextfrontier}$;

        $d_{\text{iter}} := d_{\text{iter}} + 1$;

  return $f_{\text{min}}$;

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 tree 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.

In our experiments, we show that using these approximative hyperparameters allows for an accelerated search while maintaining strong empirical results with our NEAR-guided search algorithms.
### Table 2: Dataset details.

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

### Table 3: Hyperparameters for constructing graph $G$.

<table>
<thead>
<tr>
<th></th>
<th>max depth</th>
<th>init. # units</th>
<th>min # units</th>
<th>max # children</th>
<th>penalty</th>
<th>$\beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRIM13-sniff</td>
<td>10</td>
<td>15</td>
<td>6</td>
<td>4</td>
<td>0.01</td>
<td>1.0</td>
</tr>
<tr>
<td>CRIM13-other</td>
<td>10</td>
<td>15</td>
<td>6</td>
<td>4</td>
<td>0.01</td>
<td>1.0</td>
</tr>
<tr>
<td>Fly-vs.-Fly</td>
<td>6</td>
<td>25</td>
<td>10</td>
<td>6</td>
<td>0.01</td>
<td>1.0</td>
</tr>
<tr>
<td>Bball-ballhandler</td>
<td>8</td>
<td>16</td>
<td>4</td>
<td>8</td>
<td>0.01</td>
<td>1.0</td>
</tr>
</tbody>
</table>

### Table 4: Training hyperparameters for RNN baseline.

<table>
<thead>
<tr>
<th></th>
<th># LSTM units</th>
<th># epochs</th>
<th>learning rate</th>
<th>batch size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRIM13-sniff</td>
<td>100</td>
<td>50</td>
<td>0.001</td>
<td>50</td>
</tr>
<tr>
<td>CRIM13-other</td>
<td>100</td>
<td>50</td>
<td>0.001</td>
<td>50</td>
</tr>
<tr>
<td>Fly-vs.-Fly</td>
<td>80</td>
<td>40</td>
<td>0.00025</td>
<td>30</td>
</tr>
<tr>
<td>Bball-ballhandler</td>
<td>64</td>
<td>15</td>
<td>0.01</td>
<td>50</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th></th>
<th># neural epochs</th>
<th># symbolic epochs</th>
<th>learning rate</th>
<th>batch size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRIM13-sniff</td>
<td>6</td>
<td>15</td>
<td>0.001</td>
<td>50</td>
</tr>
<tr>
<td>CRIM13-other</td>
<td>6</td>
<td>15</td>
<td>0.001</td>
<td>50</td>
</tr>
<tr>
<td>Fly-vs.-Fly</td>
<td>6</td>
<td>25</td>
<td>0.00025</td>
<td>30</td>
</tr>
<tr>
<td>Bball-ballhandler</td>
<td>4</td>
<td>6</td>
<td>0.02</td>
<td>50</td>
</tr>
</tbody>
</table>
Table 6: Additional hyperparameters for A*-NEAR and IDS-BB-NEAR. The depth bias value for CRIM13-other used a slightly different implementation (see codebase for details.)

<table>
<thead>
<tr>
<th>MC</th>
<th>Enum.</th>
<th>Genetic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>samples /step</td>
<td>max # prog.</td>
</tr>
<tr>
<td>CRIM13-sniff</td>
<td>50</td>
<td>300</td>
</tr>
<tr>
<td>CRIM13-other</td>
<td>50</td>
<td>300</td>
</tr>
<tr>
<td>Fly-vs.Fly</td>
<td>25</td>
<td>100</td>
</tr>
<tr>
<td>Bball-ballhander</td>
<td>150</td>
<td>1200</td>
</tr>
</tbody>
</table>

Table 7: Additional hyperparameters for other program learning baselines

C Details of Experimental Domains

C.1 Fly-v.-Fly

The Fly-v.-Fly dataset [12] 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-v.-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 [12]. 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 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 [18] 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 [4] at 25Hz. The interaction between a resident mouse and an intruder mouse, which is introduced to the cage of the resident, is recorded. Each mouse 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 [4] at the video level into our train and validation set, and we present test set results on the same set as [4]. Each video is split into sequences of 100 frames. There are 12404 training trajectories, 3077 validation trajectories, and 2953 test trajectories.

**Training details of CRIM13 baselines.** All CRIM13 baselines training uses the Adam [18] 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.

### C.3 Basketball

The basketball data tracks player positions (xy-coordinates on court) from real professional games. We used the processed version from [42], 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 [18] 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.

<table>
<thead>
<tr>
<th></th>
<th>CRIM13-sniff</th>
<th>CRIM13-other</th>
<th>Fly-vs.-Fly</th>
<th>Bball-ballhandler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Acc.</td>
<td>F1</td>
<td>Depth</td>
<td>Acc.</td>
</tr>
<tr>
<td>Enum.</td>
<td>.024</td>
<td>.105</td>
<td>1</td>
<td>.036</td>
</tr>
<tr>
<td>MC</td>
<td>.013</td>
<td>.127</td>
<td>1.7</td>
<td>.088</td>
</tr>
<tr>
<td>Genetic</td>
<td>.003</td>
<td>.015</td>
<td>0.6</td>
<td>.005</td>
</tr>
<tr>
<td>IDDFS-Near</td>
<td>.024</td>
<td>.022</td>
<td>0.6</td>
<td>.024</td>
</tr>
<tr>
<td>A*-Near</td>
<td>.009</td>
<td>.068</td>
<td>1</td>
<td>.012</td>
</tr>
<tr>
<td>RNN</td>
<td>.008</td>
<td>.019</td>
<td>-</td>
<td>.005</td>
</tr>
</tbody>
</table>

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