Differentiable Synthesis of Program Architectures

Guofeng Cui
Department of Computer Science
Rutgers University
gc669@cs.rutgers.edu

He Zhu
Department of Computer Science
Rutgers University
hz375@cs.rutgers.edu

Abstract

Differentiable programs have recently attracted much interest due to their interpretability, compositionality, and their efficiency to leverage differentiable training. However, synthesizing differentiable programs requires optimizing over a combinatorial, rapidly exploded space of program architectures. Despite the development of effective pruning heuristics, previous works essentially enumerate the discrete search space of program architectures, which is inefficient. We propose to encode program architecture search as learning the probability distribution over all possible program derivations induced by a context-free grammar. This allows the search algorithm to efficiently prune away unlikely program derivations to synthesize optimal program architectures. To this end, an efficient gradient-descent based method is developed to conduct program architecture search in a continuous relaxation of the discrete space of grammar rules. Experiment results on four sequence classification tasks demonstrate that our program synthesizer excels in discovering program architectures that lead to differentiable programs with higher F_1 scores, while being more efficient than state-of-the-art program synthesis methods.

1 Introduction

Program synthesis has recently emerged as an effective approach to address tasks in several fields where deep learning is applied traditionally. A synthesized program in a domain-specific language (DSL) provides a powerful abstraction for summarizing discovered knowledge from data and offers greater interpretability and transferability across tasks than a deep neural network model, while achieving competitive task performance [1–4].

A differentiable program encourages interpretability by using structured symbolic primitives to compose a set of differentiable modules with trainable parameters in its program architecture. These parameters can be efficiently learned with respect to a differentiable loss function over the program's outputs. However, synthesizing a reasonable program architecture remains challenging because the architecture search space is discrete and combinatorial. Various enumeration strategies have been developed to explore the program architecture space, including greedy enumeration [1, 2], evolutionary search [5], and Monte Carlo sampling [6]. To prioritize highly likely top-down search directions in the combinatorial architecture space, NEAR [7] uses neural networks to approximate missing expressions in a partial program whose F_1 score serves as an admissible heuristic to effective graph search algorithms such as A* [8]. However, since the discrete program architecture search space is intractably large, enumeration-based search strategies are inefficient in general.

We propose to encode program architecture search as learning the probability distribution over all possible program architecture derivations induced by a context-free DSL grammar. This problem bears similarities with searching the structure of graphical models [9] and neural architecture search. For example, to search a high-quality neural network model, DARTS [10] uses a composition of softmaxes over all possible candidate operations between a fixed set of neural network nodes to relax the discrete search space of neural architectures. However, applying this method to program synthesis

35th Conference on Neural Information Processing Systems (NeurIPS 2021).

is challenging because the program architecture search space is much richer [7]. Firstly, different sets of operations take different input and output types and may only be available at different points of a program. Secondly, there is no fixed bound on the number of expressions in a program architecture.

To address the aforementioned challenges, we learn the probability distribution of program architectures in a continuous relaxation of the search space of DSL grammar rules. We conduct program architecture search in a *program derivation graph*, in which nodes encode architectures with missing expressions, and paths encode top-down program derivations. For each partial architecture f on a graph node, we relax the categorical choice of production rules for expanding a missing expression in f to a softmax over all possible production rules with trainable weights. A program derivation graph essentially expresses all possible program derivations under a context-free grammar up to a certain depth bound (on the height of program abstract syntax trees), which can be progressively increased during search to balance accuracy and architecture complexity. We encode a program derivation graph itself as a differentiable program whose output is weighted by the outputs of all the programs involved. We seek to optimize program architecture weights with respect to an accuracy loss function defined over the encoded program's output. The learned weights allow our synthesis algorithm to efficiently prune away search directions to unlikely program derivations to discover optimal programs. Compared with enumeration-based synthesis strategies, differentiable program synthesis in the relaxed architecture space is easier and more efficient with gradient-based optimization.

One major challenge of differentiable program architecture synthesis is that a program derivation graph involves an exponential number of programs and a huge set of trainable variables including architecture weights and program parameters. To curb the large program derivation search space, we introduce node sharing in program derivation graphs and progressive graph unfolding. Node sharing allows two partial architectures to share the same child nodes if the missing expressions in the two architectures can be expanded using the same grammar rules. Progressive graph unfolding allows the synthesis algorithm to construct a program derivation graph on the fly focusing on higher-quality program derivations than all the rest. These optimization strategies significantly reduce the program architecture search space, scaling differentiable program synthesis to real-world classification tasks. We evaluate our synthesis algorithm in the context of learning classifiers for sequence classification applications. We demonstrate that our algorithm substantially outperforms state-of-the-art methods for differentiable program synthesis, and can learn programmatic classifiers that are highly interpretable and are comparable to neural network models in terms of accuracy and F_1-scores.

As a summary, this paper makes three contributions. Firstly, we encode program synthesis as learning the probability distribution of program architectures in a continuous relaxation of the discrete space defined by programming language grammar rules, enabling differentiable program architecture search. Secondly, we instantiate differentiable program architecture synthesis with effective optimization strategies including node sharing and progressive graph unfolding, scaling it to real-world classification tasks. Lastly, we present state-of-the-art results in learning programmatic classifiers for four sequence classification applications.

2 Problem Formulation

A program in a domain-specific language (DSL) is a pair (α, θ), where α is a discrete program architecture and θ is a vector of real-valued parameters of the program. Given a specification over the intended input-output behavior of an unknown program, program synthesis aims to discover the program's architecture α and optimize the program parameters θ.

In this paper, we focus on learning programmatic classifiers for sequence classification tasks [11]. We note that the proposed synthesis technique is applicable to learning any differentiable programs.

Program Architecture Synthesis. A program architecture α is typically synthesized based on a context-free grammar [12]. Such a grammar consists of a set of production rules $\alpha_k \rightarrow \{\sigma_j\}_{j=0}^{J}$ over terminal symbols Σ and nonterminal symbols Y where $\alpha_k \in Y$ and $\sigma_j \in \Sigma \cup Y$. As an example, consider the context-free grammar of a DSL for sequence classification depicted in the standard Backus-Naur form [13] in Fig. 1, adapted from [7]. A terminal in this grammar is a symbol that can appear in a program's code, e.g. x and the **map** function symbol, while a nonterminal stands for a missing expression (or subexpression), e.g. α_2 and α_3. Any program in the DSL operates over a real vector or a sequence of real vectors x. It may use constants c, arithmetic operations **Add** and **Multiply**, and an **I**f-**T**hen-**E**lse branching construct **ITE**. To avoid discontinuities for

$$\alpha \ ::= x \mid c \mid \textbf{Add}\ \alpha_1\ \alpha_2 \mid \textbf{Multiply}\ \alpha_1\ \alpha_2 \mid \textbf{ITE}\ \alpha_1 \geq 0\ \alpha_2\ \alpha_3 \mid \mathbf{F}_{\mathbf{S},\theta}(x) \mid \textbf{map}\ (\textbf{fun}\ x_1.\alpha_1)\ x \mid$$
$$\textbf{mapprefix}\ (\textbf{fun}\ x_1.\alpha_1)\ x \mid \textbf{fold}\ (\textbf{fun}\ x_1.\alpha_1)\ c\ x \mid \textbf{SlideWindowAvg}\ (\textbf{fun}\ x_1.\alpha_1)\ x$$

Figure 1: Context-free DSL Grammar for Sequence Classification (adapted from [7]).

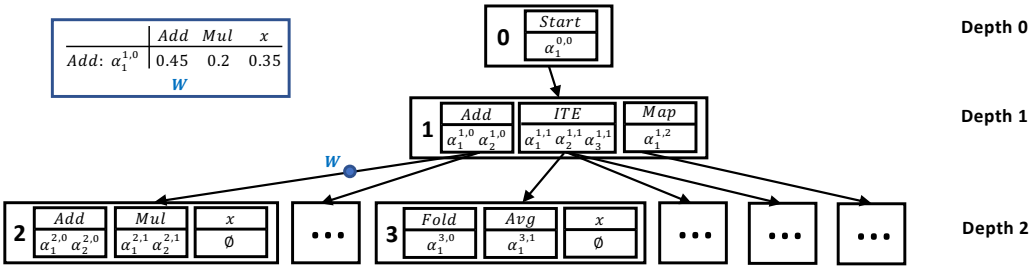

Figure 2: Program Derivation Graph of the grammar in Fig. 1.

differentiability, we interpret it in terms of a smooth approximation: $[\![\textbf{ITE}(\alpha_1 \geq 0, \alpha_2, \alpha_3)]\!](x) = \sigma([\![\alpha_1]\!](x)) \cdot [\![\alpha_2]\!](x) + (1 - \sigma([\![\alpha_1]\!](x)) \cdot [\![\alpha_3]\!](x)$ where σ is the sigmoid function. A program may invoke a customized library of differentiable, parameterized functions. In our context, these functions are in the shape $\mathbf{F}_{S,\theta}(x)$ that extract a vector consisting of a predefined subset S of the dimensions of an input x and pass the extracted vector through a linear function with trainable parameters θ. A program may also use a set of higher-order combinators to recurse over sequences including the standard **map** and **fold** combinators. The higher-order combinators take as input an anonymous function **fun** $x.e(x)$ that evaluates an expression $e(x)$ over the input x. For a sequence x, the **mapprefix** higher-order combinator returns a sequence $f(x[1:1]), f(x[1:2]), \ldots, f(x[1:n])$, where $x[1:i]$ is the i-th prefix of x. The **SlideWindowAvg** function computes the average of a sequence over a moving window of vectors.

We define the complexity of a program architecture α. Let each grammar rule r have a non-negative real-valued cost $c(r)$. The structural cost $c(\alpha)$ is the sum of the costs of the multi-set of rules used to create α. Intuitively, program architectures with lower structural cost are more interpretable. In the context of this paper, program synthesis aims to learn a simple program (in terms of structure cost) that satisfies some specifications over program input-output behaviors. In this paper, we set $c(r) = 1$ for any production rule r.

Program Synthesis Specifications. In a sequence classification task, a set of feature sequences $\{i_k\}_{k=1}^{K}$ are taken as input and we expect to classify i_k into a certain category o_k. Each i_k is a sequence of observations. Each observation captures features extracted at a frame as a 1-dimensional real-valued vector. We aim to synthesize a program $P(\cdot;\ \alpha, \theta)$ as a classifier with high accuracy and low architecture cost. Our program synthesis goal is formalized as follows:

$$\arg\min_{\theta,\alpha}\ \mathbb{E}_{i_k,o_k\sim D}[\ell\big(P(i_k;\ \alpha,\theta), o_k\big)] + c(\alpha) \tag{1}$$

where $D(i_k, o_k)$ is an unknown distribution over input sequences i_k and labels o_k. The first term of Equation (1) defines some prediction error loss ℓ of a program $P(\cdot)$ for a classification task over P's predicted labels and the ground truth labels. The second term enforces program synthesis to learn an architecturally simple classifier.

3 Differentiable Program Architecture Synthesis

We formulate program architecture derivation as a form of top-down graph traversal. Given the context-free grammar $\mathcal{G}$ of a DSL, an architecture derivation starts with the initial nonterminal (i.e. the empty architecture), then applies the production rules in $\mathcal{G}$ to produce a series of partial architectures which consist in expressions made from one or more nonterminals and zero or more terminals, and terminates when a complete architecture that does not include any nonterminals is derived.

Formally, program architecture synthesis with respect to a context-free grammar $\mathcal{G}$ is performed over a directed acyclic *program derivation graph* $G = \{V, E\}$ where V and E indicate graph nodes and edges. Fig. 2 depicts a program derivation graph for the sequence classification grammar in Fig. 1. A node $u \in V$ is a set of partial or complete program architectures permissible by $\mathcal{G}$. An edge $(u, u') \in E$ exists if one can obtain the architectures in u' by expanding a nonterminal of an architecture in u following some production rules of $\mathcal{G}$. For simplicity, Fig. 2 only shows three partial or complete architectures in any node of the program derivation graph. In the graph node at depth 1, we expand the initial nonterminal $\alpha_1^{0,0}$ to the **Add**, **ITE** and **Map** functions (each with missing expressions) using the grammar rules in Fig. 1. Notice that the edge direction in a program derivation graph indicates search order. However, program dataflow through each edge (u, u') is in the opposite direction. The output of u' is calculated first and then passed as input to u.

The main challenge of program architecture synthesis is that the search space embedded in a program derivation graph is discrete and combinatorial. Enumeration-based synthesis strategies are inefficient in general because of the intractable search space. Instead, we aim to learn the probability distribution of program architectures within a program derivation graph in a continuous relaxation of the search space. Specifically, to expand a nonterminal of a partial program architecture, we relax the categorical choice of production rules in a context-free grammar into a softmax over all possible production rules with trainable weights. For example, in Fig. 2, if we expand the initial nonterminal $\alpha_1^{0,0}$ to a partial architecture **Add** $\alpha_1^{1,0}$ $\alpha_2^{1,0}$ on node 1, we have several choices to further expand the architecture's first nonterminal $\alpha_1^{1,0}$, weighted by the probability matrix w (obtained after softmax) drawn in Fig. 2. Based on w, the synthesizer chooses to expand $\alpha_1^{1,0}$ to **Add** $\alpha_1^{2,0}$ $\alpha_2^{2,0}$ on node 2. Our main idea to learn architecture weights is to encode a program derivation graph itself as a differentiable program $\mathcal{T}_{w,\theta}$ whose output is weighted by the outputs of all programs included in $\mathcal{T}_{w,\theta}$, where w represents architecture weights and θ includes program parameters of all the mixed programs in the graph. The parameters w and θ can be jointly optimized with respect to a differentiable loss function ℓ over program outputs via bi-level optimization. Similar to DARTS [10], we train θ and w on a parameter training dataset D_θ and an architecture validation dataset D_w respectively until convergence:

$$\begin{aligned} \theta' &= \theta - \nabla_\theta \mathbb{E}_{i_k, o_k \sim D_\theta} \ell\big(\mathcal{T}_{w,\theta}(i_k), o_k\big) \\ w' &= w - \nabla_w \mathbb{E}_{i_k, o_k \sim D_w} \ell\big(\mathcal{T}_{w,\theta'}(i_k), o_k\big) \end{aligned} \tag{2}$$

However, a program derivation graph includes an exponential number of programs. Therefore, it involves a huge set of trainable variables including program architecture weights w and unknown program parameters θ. To curb the large program derivation search space, we introduce node sharing (Sec. 3.1) and progressive graph unfolding (Sec. 3.2).

3.1 Node Sharing

Intuitively, node sharing in a program derivation graph allows two partial architectures to share the same child nodes if the nonterminals in the two architectures can be expanded using the same grammar production rules. Fig. 3 depicts the compressed program derivation graph for the sequence classification grammar in Fig. 1. At depth 1, three partial architectures **Add** $\alpha_1^{1,0}$ $\alpha_2^{1,0}$, **ITE** $\alpha_1^{1,1} \geq 0$ $\alpha_2^{1,1}$ $\alpha_3^{1,1}$, and **Map** (**fun** $x_1.\alpha_1^{1,2}$) are expanded from the initial nonterminal $\alpha_1^{0,0}$. Because only one of the three partial architectures would be used to derive the final synthesized program, we allow the nonterminals $\alpha_2^{1,0}$, the second parameter of **Add**, and $\alpha_2^{1,1}$, the second parameters of **ITE**, to share the same child node 3, weighted by the probability matrix w drawn in Fig. 3. Importantly, node sharing takes function arities and types into account. The matrix w has 0 probability for the **Map** partial architecture because unlike **Add** and **ITE** it does not contain a second parameter.

Formally, in a program derivation graph, let K_u be the number of program architectures on node u. Denote $f_k^u\big(\alpha_1^{u,k}, \ldots, \alpha_{\eta(f_k^u)}^{u,k}\big)$ as the k-th (partial) architecture on u where $\eta(f_k^u)$ is the number of nonterminals contained in f_k^u and $\alpha_i^{u,k}$ is the i-th nonterminal of f_k^u. For the grammar of Fig. 1, essentially each f_k^u is a function application with missing argument expressions $\alpha_i^{u,k}$, $1 \leq i \leq \eta(f_k^u)$, and $\eta(f_k^u)$ is the arity of the function. Assume that u' is the i-th child of u from left to right in the program derivation graph. The weight w_e of the edge $e = (u, u')$ is of the shape $\mathbb{R}^{K_u \times K_{u'}}$ where the matrix rows refer to the partial architectures on u and the matrix columns refer to architectures on u'. We have $w_e[(k, k')]$ proportional to the probability of expanding the i-th nonterminal of f_k^u to $f_{k'}^{u'}$

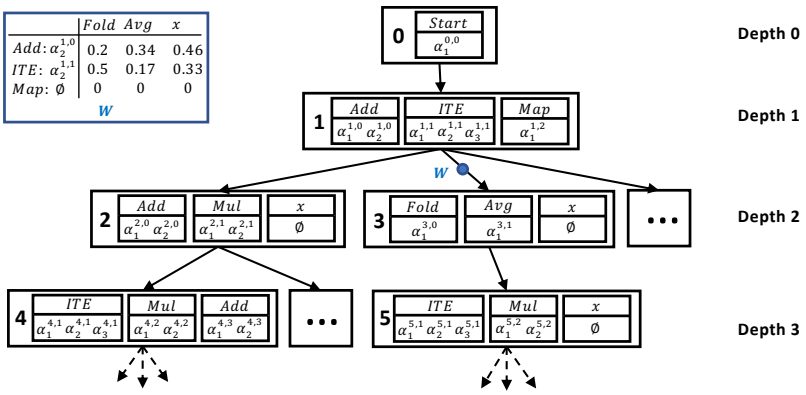

Figure 3: Node Sharing on Program Derivation Graphs.

(obtained after softmax), e.g. the w matrix drawn in Fig. 3. To make the architecture search space continuous, we relax the categorical choice of expanding a particular nonterminal $\alpha_i^{u,k}$ to a softmax over all possible grammar production rules for $\alpha_i^{u,k}$ in the program derivation graph:

$$[\![\alpha_i^{u,k}]\!](x) = \sum_{k'=0}^{K_{u'}} \frac{\exp(w_e[(k,k')])}{\sum_{j=0}^{K_{u'}} \exp(w_e[(k,j)])} \cdot [\![f_{k'}^{u'}(\alpha_1^{u',k'}, \ldots, \alpha_{\eta(f_{k'}^{u'})}^{u',k'})]\!](x) \tag{3}$$

where u' is the i-th child of u and $e = (u, u')$

Complexity. Let D be the depth of a program derivation graph for a context-free grammar, K_{max} be the number of productions rules, and η_{max} be the maximum number of nonterminals in any rules of the grammar. With node sharing, we reduce the space complexity of the program derivation graph from $O([K_{max} \cdot \eta_{max}]^{D+1})$ to $O([\eta_{max}]^{D+1})$. In a program synthesis task, K_{max} is typically much larger than η_{max}. Without compression, a program derivation graph with a large K_{max} hardly fits GPU memory.

3.2 Progressive Graph Unfolding

Node sharing significantly restricts the width of a program derivation graph. However, a derivation graph still grows exponentially with its depth, which limits the scalability of differentiable architecture search. To address this problem, we propose an on-the-fly approach that unfolds program derivation graphs progressively and prunes away unlikely candidate architectures at the end of each iteration based on their weights. Fig. 4 depicts the progressive procedure of derivation graph unfolding.

At the initial iteration, the program derivation graph is shallow as it only contains architectures up to depth d_s. We set $d_s = 2$ in Fig. 4. For any partial program architecture $f_k^u(\alpha_1^{u,k}, \ldots, \alpha_{\eta(f_k^u)}^{u,k})$ on any leaf node u of the depth-bounded graph, our algorithm substitutes neural networks for the nonterminals $\alpha_i^{u,k}$ to approximate the missing expressions. These networks are type-consistent. For example, a recurrent neural network is used to replace a missing expression whose inputs are supposed to be sequences. For a program derivation graph as such, the unknown program parameters θ come from both the parameterized functions and the neural modules. Our synthesis algorithm optimizes the architecture weights and the unknown program parameters using Equation 2.

In the next iteration, on each graph node, our synthesis algorithm retains top-N program architectures as children for each partial architecture on the node's parent, which are defined to be those assigned with higher weights on the node's incoming edge in the previous iteration. We set $N = 2$ in the example of Fig. 4. After top-N preservation on each node, our synthesis algorithm increases the depth of the program derivation graph by expanding the nonterminals (that were replaced with neural modules in the previous iteration) d_s depths deeper. Suitable neural networks are leveraged to substitute any new nonterminals at depth $2d_s + 1$. Our algorithm again jointly optimizes architecture weights and unknown program parameters and performs top-N preservation on each node based on learned architecture weights, as depicted in Fig. 4. Such a process iterates until the unfolded program

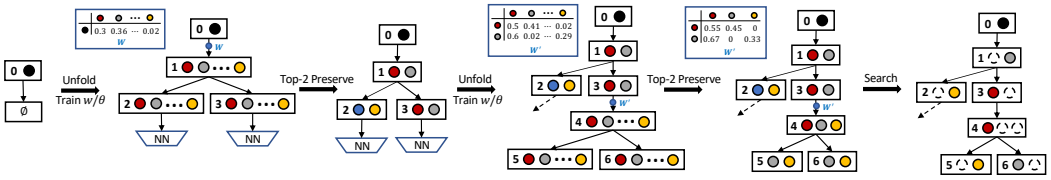

Figure 4: Differentiable program architecture synthesis with progressive graph unfolding.

derivation graph contains no nonterminals or the maximum search depth is reached. Our differentiable program architecture synthesis method is outlined in (the first while loop of) Algorithm 1.

3.3 Searching Optimal Programs

Once we have an optimized program derivation graph G, due to the top-N preservation strategy, each node retains a small number of partial architectures. From G, we could greedily obtain a discrete program architecture top-down by replacing each graph node containing mixed partial architectures with the most likely partial architecture based on learned architecture weights. However, the performance estimation ranked by architecture weights in a program derivation graph can be inaccurate due to the co-adaption among architectures via node sharing. Recent work also discovers that relaxed architecture search methods tend to overfit to certain functions that lead to more rapid gradient descent than others [14–17] and thus produce unsatisfying performance.

To overcome this potential disadvantage of differentiable architecture search, our algorithm introduces a search procedure as depicted in Fig. 4. The core idea is that while one super program derivation graph may not be able to model the entire search space accurately, multiple sub program derivation graphs can be used to effectively address the limitation by having each sub graph modeling one part of the search space.

Algorithm 1: Program Archit. Synthesis

Input : Grammar $\mathcal{G}$, Graph expansion depth d_s, Top-N parameter
Output : Synthesized Program P

```
G contains only the initial nonterminal;
while maximum depth not reached do
    Unfold G depth d_s deeper w.r.t. G;
    Optimize w and θ in G w.rt. Eq. 2;
    Top-N preservation on G's nodes;
Q := [G];
while Q ≠ ∅ do
    q := arg min_{q∈Q} f(q);
    Q := Q \ {q};
    if q is a well-typed program then
        return q;
    u is the top-left most node in q with
      more than one architecture choice;
    for each partial archit. f_k^u on u do
        q' := q[u/f_k^u];
        Compute g(q'), h(q'), s(q');
        Q := Q ∪ {q'};
```

In the search, Algorithm 1 maintains a queue Q of program derivation graphs sorted by their quality that is initialized to $[G]$. Our algorithm measures the quality of a program by both its task performance and structure cost. The algorithm dequeues one graph q from Q and extracts the top-most and left-most node u of q that still contains more than one partial architecture for search. As u co-adapts multiple architectures, we separate the entire search space into disjoint partitions by picking each available architecture $f_k^u\left(\alpha_1^{u,k}, \ldots, \alpha_{\eta(f_k^u)}^{u,k}\right)$ from the compound node u and assign a sub program derivation graph to model each partition. The algorithm computes a quality score s for each option of retaining only f_k^u on u, denoted as $q[u/f_k^u]$:

$$s(q[u/f_k^u]) = g(q[u/f_k^u]) + h(q[u/f_k^u])$$

The $g(q[u/f_k^u])$ function measures the structure cost of expanding the initial nonterminal up to u (Sec. 2) and $h(q[u/f_k^u])$ is an ϵ-Admissible heuristic estimate of the cost-to-go from node u [18]:

$$h(q[u/f_k^u]) = 1 - F_1(\mathcal{T}_{w^*,\theta^*}[u/f_k^u], D_{val}) \text{ where } w^*, \theta^* = \arg\min_{w,\theta} \mathbb{E}_{i_k,o_k\sim D}[\ell(\mathcal{T}_{w,\theta}[u/f_k^u]), o_k)]$$

where $\mathcal{T}$ encodes the program derivation graph q itself via Equation (3) as a differentiable program whose output is weighted by the output of all complete programs included in q, w and θ are the sets of architecture weights and unknown program parameters in the subgraph rooted at u in $q[u/f_k^u]$. The h function fine-tunes these trainable variables using the training dataset D to provide informed feedback

on the contribution to program quality by the choice of only retaining f_k^u on node u, measured by the program's F_1 score. In practice, to avoid overfitting, we use a separate validation dataset to obtain the F_1 score. After computing the quality score s, we add $q[u/f_k^u]$ back to the queue Q sorted based on s-scores. The search algorithm completes when the derivation graph with the least s-score from Q is a well-typed program, i.e. each graph node contains only one valid architecture choice. Our architecture selection algorithm is optimal given the admissible heuristic function h — the returned program optimally balances program accuracy and structure complexity among all the programs contained in G. The proof is given in Appendix A.

4 Experiments

We have implemented Algorithm 1 in a tool named dPads (**d**omain-specific **P**rogram **a**rchitecture **d**ifferentiable **s**ynthesis) [19], and evaluated it on four sequence classification datasets.

4.1 Datasets for Evaluation

We partition a dataset to training, validation, and test datasets. dPads uses the training dataset to optimize the architecture weights and program parameters in a program derivation graph. When searching a final program from a converged program derivation graph, we use the validation dataset to obtain the program's F_1 score to guide the search. We use the test dataset to obtain the final accuracy and F_1 score of a program. Additionally, in training we construct two separate datasets by randomly selecting 60% of a training dataset as D_θ to optimize program parameters θ and using the remaining 40% as D_w to train architecture weights w via Equation 2.

Crim13 Dataset. The dataset collects social behaviors of a pair of mice. We cut every 100 frames as a trajectory. Each trajectory frame is annotated with an action by behavior experts [20]. For each frame, a 19-dimensional feature vector is extracted including the positions and velocities of the two mice. The goal is to synthesize a program to classify each trajectory to action *sniff* or *no sniff*. In total we have 12404, 3077, and 2953 trajectories in the training set, validation set, and test set respectively.

Fly-vs-fly Dataset. We use the *Boy-meets-boy*, *Aggression* and *Courtship* datasets collected in the *fly-vs-fly* environment for monitoring two fruit flies interacting with each other [21]. Each trajectory frame is a 53-dimensional feature vector including fly position, velocity, wing movement, etc. We subsample the dataset similar to [7], which results in 5341 train trajectories, 629 validation trajectories, and 1050 test trajectories. We aim to synthesize a program to classify each trajectory as one of 7 actions displaying aggressive, threatening, and nonthreatening behaviors.

Basketball Dataset. The dataset tracks the movement of a basketball, 5 defensive players and 5 offensive players [22]. Each trajectory has 25 frames with each frame as a 22-dimensional feature vector of ball and player position information. We aim to learn a program that can predict which offensive player handles the ball or whether the ball is being passed. In total we have 18000, 2801, and 2693 trajectories in the training set, validation set, and test set respectively.

Skeletics 152 Dataset. The dataset [23] contains 152 human pose actions as well as related YouTube Videos subsampled from Kinetics-700 [24]. For each video frame, 25 3-D skeleton points are collected, resulting in a 75-dimensional feature vector per frame. We extract 100 frames from each trajectory to reduce noise. Finally, the training set contains 8721 trajectories, the validation set contains 2184 trajectories, and the test set contains 892 trajectories. We aim to learn a program to classify a pose trajectory as one of 10 actions.

As discussed in Sec. 2, the DSL for each dataset is equipped with a customized library of differentiable and parameterized functions $F_{S,\theta}(x)$. We define these functions in Appendix B. In this paper, we focus on sequence classification benchmarks. However, dPads is a general program synthesis algorithm and is not limited to sequence classification. In Appendix C.6, we evaluate dPads on cryptographic circuit synthesis to demonstrate the generalizability of dPads.

4.2 Experimental Setup

To train architecture weights w and unknown program parameters θ in a differentiable program architecture derivation graph, we use the Adam optimizer [25]. In Algorithm 1, we set $N = 2$ for top-N preservation and set graph expansion depth d_s to 2. For evaluation, we compare dPads

Table 1: Experiment results on the performance of dPads compared with NEAR [7]. All results are reported as the average of runs on five random seeds. Costs of time are set in minutes.

	Crim13-sniff			**Fly-vs-fly**			**Bball-ballhandler**			**SK152-10 actions**		
	F_1	Acc.	Time	F_1	Acc.	Time	F_1	Acc.	Time	F_1	Acc.	Time
RNN	.481	.851	-	.964	.964	-	.980	.980	-	.414	.428	-
A*-NEAR	.286	.820	164.92	.828	.764	**243.82**	.940	.934	553.01	.312	.315	210.23
IDS-BB-NEAR	.323	.834	463.36	.822	.750	465.57	.793	.768	513.33	.314	.317	848.44
dPads	**.458**	.812	**147.87**	**.887**	.853	348.25	**.945**	.939	**174.68**	**.337**	.337	**162.70**

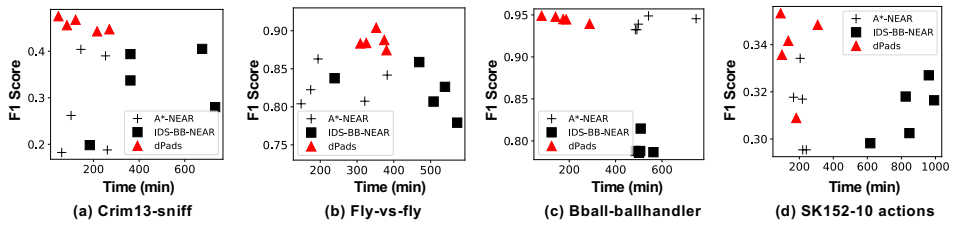

Figure 5: Experiment results on the Crim13, Fly-vs-fly, Basketball and SK152 datasets over five random seeds. x axis refers to costs of time recorded in minutes and y axis refers to F_1 scores.

with the state-of-the-art program learning algorithms A*-NEAR and IDS-BB-NEAR [7]. We only report a comparison with NEAR because NEAR significantly outperforms other program learning methods based on top-down enumeration, Monte-Carlo sampling, Monte-Carlo tree search, and genetic algorithms [7]. All experiments were performed on Intel 2.3-GHz Xeon CPU with 16 cores, equipped with an NVIDIA Quadro RTX 6000 GPU. More experiment settings including learning rates and training epochs are given in Appendix C.1 and C.2.

4.3 Experiment Results

For a fair comparison with NEAR [7], for any of the four datasets, all tools search over the same DSL. We use random seeds 0, 1000, 2000, 3000, 4000 and report average F_1 scores, accuracy rates and execution times for both methods. We also report the results achieved using a highly expressive RNN baseline which provides a task performance upper bound on F_1-scores and accuracy.

Table 1 shows the experiment results. On both the Crim13 and SK152 datasets, dPads outperforms A*-NEAR and IDS-BB-NEAR achieving higher F_1 scores and using much less time consumption. dPads also achieves competitive accuracy with NEAR on Crim13. On the Basketball dataset, although dPads achieves a bit higher F_1 score, the architectures synthesized by dPads and A*-NEAR are exactly the same. However, dPads takes 70% less time to get the result. While A*-NEAR completes the search faster than dPads on Fly-vs-fly, the program architecture synthesized by dPads leads to a program with a much better F_1 score and higher accuracy. More quantitative analyses of the experiment results are given in Appendix C.3.

We visualize the results of dPads and NEAR in terms of F_1 scores (y axis) and running times (x axis) on the 5 random seeds in Fig. 5 where red triangles refer to the results of dPads, and black plus marks and rectangles refer to the results of A*-NEAR and IDS-BB-NEAR. dPads consistently outperforms NEAR in achieving higher F_1 scores with less computation and is closer to the RNN baseline.

Although the RNN baseline provides better performance, dPads learns programs that are more interpretable. Fig. 6 depicts the best programs synthesized by dPads on Crim13 and SK152 (among all the 5 random seeds). The program for Crim13 has a simple architecture and achieves a high F_1 score 0.475 (only 0.006 less than the RNN result). It invokes two $F_{S,\theta}$ library functions: PositionAffine and DistanceAffine. This program is highly human-readable: it evaluates the likelihood of "sniff" by applying a position bias and if the distance between two mice is small they are doing a "sniff". The programmatic classifier for SK152 achieves an F_1 score 0.35 which is close to the RNN baseline. It uses the arm and leg positions of a 3-D skeleton to complete a human-action classification. We show more examples about the interpretability of programs learned by dPads in Appendix C.5.

```
Map(
  Multiply(
    PositionAffine_θ1(x_t)),
    DistanceAffine_θ2(x_t))) x
```

```
SlideWindowAvg(Add(
    Multiply(LegsAffine_θ1(x_t),
             LegsAffine_θ2(x_t)),
    Multiply(ArmsAffine_θ3(x_t),
             ArmsAffine_θ4(x_t)))) x
```

Figure 6: Synthesized Programs for Crim13-sniff (left) and SK152-10 actions (right).

Table 2: Ablation study on the importance of node sharing and progressive graph unfolding as two optimization strategies in dPads. All results are reported as the average of runs on five random seeds. Costs of time are set in minutes. OOM represents an out-of-memory error.

	Crim13-sniff			**Fly-vs-fly**			**Bball-ballhandler**			**SK152-10 actions**		
Variants of dPads	F_1	Acc.	Time	F_1	Acc.	Time	F_1	Acc.	Time	F_1	Acc.	Time
dPads w/o Node Sharing	.453	.800	334.93	-	-	> 1440	-	-	> 1440	.321	.322	252.81
dPads w/o Graph Unfolding	.449	**.818**	280.67	-	-	OOM	.848	.832	348.09	**.348**	**.346**	273.95
dPads in full	**.458**	.812	**147.87**	**.887**	**.853**	**348.25**	**.945**	**.939**	**174.68**	.337	.337	**162.70**

4.4 Ablation Studies

We introduce two more baselines to study the importance of node sharing and progressive graph unfolding. The first baseline does not use node sharing to reduce the size of a program derivation graph but still performs progressive graph unfolding. The second baseline directly expands a program derivation graph to the maximum depth but still applies node sharing. We report the comparison results over 5 random runs in Table 2. Without the two optimizations, limited by the size of GPU memory, dPads may either time-out or encounter out-of-memory error when searching programs that need deep structures to ensure high accuracy. This is because the size of a program derivation graph grows exponentially large with the height of program abstract syntax trees and the number of DSL production rules. Moreover, while being more complete, training without these two optimizations does not necessarily produce better results even when there is no memory error or timeout. On Basketball, dPads achieves .945 F_1 score. dPads without progressive graph unfolding only obtains .848 F_1 score. We suspect this is because the program derivation graph without top-N preservation and progressive unfolding is more difficult to train as it contains significantly more parameters.

We further investigate the effect of the top-N preservation strategy in program architecture synthesis (Sec. 3.2) and its impact on searching optimal programs (Sec. 3.3). We set N to 1, 2, 3 respectively and study how dPads responds to these changes. Table 3 summarizes the average results of F_1 scores, accuracy rates, time costs, and the standard deviations of these results. When $N = 1$, dPads extracts final programs greedily from optimized program derivation graphs without conducting further search. There is a significant decrease in time consumption compared with $N = 2$. However, dPads in this condition achieves less F_1 scores and the results have higher variances, which suggests that architecture weights learned using only differentiable synthesis overfit to sub-optimal programs. dPads gets *similar* F_1 scores when setting $N = 3$ compared to $N = 2$ but consumes more time as $N = 3$ incurs much larger search spaces. It even times-out on the Basketball dataset while searching an optimal program from the converged program derivation graph. This result confirms that scaling discrete program search to large architecture search spaces is challenging. dPads addresses this fundamental limitation by leveraging differentiable search of program architectures to significantly prune away unlikely search directions. Therefore, it suffices to set $N = 2$ in our experiments to balance search optimality and efficiency. Additional ablation study results are given in Appendix C.4. We discuss the limitations of dPads in Appendix D.1.

5 Related Work

Program Synthesis. Tasks in program synthesis aim to search for programs in a DSL to satisfy a specification over program inputs and outputs. There is also a growing literature on applying deep learning methods to guide the search over program architectures [6, 26–34]. There exist efforts that extend this line of research to program synthesis from noisy data [1, 2, 35–37, 3, 4]. These approaches either require a detailed hand-written program template or simply enumerate the discrete

Table 3: Ablation study on the value of N for the top-N preservation strategy used in dPads. All results are reported as the average of runs on five random seeds. Costs of time are set to minutes.

	N	dPads				
		F_1	Acc.	Time	Std. F_1	Std. Acc.
Crim13-sniff	1	.272	.627	50.85	.111	.218
	2	.458	.812	147.87	.014	.008
	3	.450	.811	441.12	.025	.008
Fly-vs-fly	1	.769	.716	95.36	.052	.062
	2	.887	.853	348.25	.010	.006
	3	.866	.818	620.48	.017	.039
Bball-ballhandler	1	.808	.785	41.14	.042	.045
	2	.945	.939	174.68	.004	.004
	3	-	-	> 1440	-	-
SK152-10 actions	1	.310	.310	40.34	.020	.024
	2	.337	.337	162.70	.017	.017
	3	.336	.338	609.14	.011	.010

space of program architectures permitted by a DSL. Additionally, most of these literature methods build models that are trained using corpora of synthesis problems and solutions, which are not available in our setting. The most closest work to our technique includes [38, 7] that enumerate the space of program architectures prioritizing search directions with feedback from machine learning models. Specifically, Lee et al. [38] uses a probabilistic model (trained from a synthesis problem corpus) to guide an A* search over discrete program syntax trees and NEAR [7] uses neural networks to approximate missing expressions in a partial program whose F_1 score serves as an admissible heuristic to guide an A* search again over discrete program syntax trees. As opposed to these efforts, our method more efficiently conducts program synthesis in a continuous relaxation of the discrete space of language grammar rules and only searches the optimal program in a much reduced search space after differentiable architecture synthesis for addressing the gradient bias problem [14].

Differentiable Architecture Search. Neural architecture search has attracted much interest as a promising approach to automate deep learning tasks [39–42]. Our program architecture synthesis algorithm is inspired by DARTS [10]. This method uses a composition of softmaxes over all possible candidate operations between a fixed set of neural network nodes to relax the neural architecture search space. Various methods further improve the efficiency and accuracy of differentiable architecture search [14–17, 43, 44]. Applying this line of algorithms to program synthesis is challenging because the space of program architectures is much richer. Different operations take different number and types of inputs/outputs and may only be available at different points of a program. There is also no fixed bound on the number of program expressions. By relaxing the discrete search space of language grammar rules with node sharing and progressive unfolding of program derivation graphs, our method addresses the aforementioned challenges. To the best of our knowledge, this is the first approach that applies differentiable architecture search to program synthesis.

6 Conclusions

This paper presents a novel differentiable approach to program synthesis. With gradient descent, our method learns the probability distribution of program architectures induced by the context-free grammar of a DSL in a continuous relaxation of the discrete space of language grammar rules. This allows the synthesis algorithm to efficiently prune away unlikely program derivations to discover optimal program architectures. We have instantiated differentiable program architecture synthesis with effective optimization strategies including node sharing and progressive graph unfolding, scaling it to real-world sequence classification tasks. Experiment results demonstrate that our algorithm substantially outperforms state-of-the-art program learning approaches.

Programmatic models in high-level DSLs are a powerful abstraction for summarizing discovered knowledge from data in a human-interpretable way. Programmatic models incorporate inductive bias through structured symbolic primitives in a DSL and open opportunities for programmers to influence the semantic meaning of learned programs. However, the programming biases in a DSL may also leave opportunities to attack on the security and fairness of a learned model. One direction for future work is to apply formal program reasoning to enhance the trustworthiness of programmatic models.

Acknowledgments and Disclosure of Funding

We thank the anonymous reviewers for their comments and suggestions. This work was supported by NSF Award #CCF-2124155 and the DARPA Symbiotic Design for Cyber Physical Systems program.

References

[1] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, pages 5045–5054. PMLR, 2018.

[2] Abhinav Verma, Hoang Minh Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. 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, December 8-14, 2019, Vancouver, BC, Canada*, pages 15726–15737, 2019.

[3] Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable programs with neural libraries. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 1213–1222. PMLR, 2017.

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

[5] Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles 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, December 3-8, 2018, Montréal, Canada*, pages 8701–8712, 2018.

[6] 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, December 8-14, 2019, Vancouver, BC, Canada*, pages 9165–9174, 2019.

[7] Ameesh Shah, Eric Zhan, Jennifer J. Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. Learning differentiable programs with admissible neural heuristics. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[8] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., USA, 1984. ISBN 0201055945.

[9] Jing Xiang and Seyoung Kim. A* lasso for learning a sparse bayesian network structure for continuous variables. In *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.

[10] 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.

[11] Thomas G. Dietterich. Machine learning for sequential data: A review. In Terry Caelli, Adnan Amin, Robert P. W. Duin, Mohamed S. Kamel, and Dick de Ridder, editors, *Structural, Syntactic, and Statistical Pattern Recognition, Joint IAPR International Workshops SSPR 2002 and SPR 2002, Windsor, Ontario, Canada, August 6-9, 2002, Proceedings*, volume 2396 of *Lecture Notes in Computer Science*, pages 15–30. Springer, 2002. doi: 10.1007/3-540-70659-3_2.

[12] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007. ISBN 978-0-321-47617-3.

[13] Glynn Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993. ISBN 978-0-262-23169-5.

[14] Xin Chen, Lingxi Xie, Jun Wu, and Qi Tian. Progressive differentiable architecture search: Bridging the depth gap between search and evaluation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[15] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part XVI*, volume 12361 of *Lecture Notes in Computer Science*, pages 544–560. Springer, 2020. doi: 10.1007/978-3-030-58517-4_32.

[16] Yiyang Zhao, Linnan Wang, Yuandong Tian, Rodrigo Fonseca, and Tian Guo. Few-shot neural architecture search. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 12707–12718. PMLR, 2021.

[17] Ruochen Wang, Minhao Cheng, Xiangning Chen, Xiaocheng Tang, and Cho-Jui Hsieh. Rethinking architecture selection in differentiable NAS. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[18] Larry R. Harris. The heuristic search under conditions of error. *Artif. Intell.*, 5(3):217–234, 1974. doi: 10.1016/0004-3702(74)90014-9.

[19] Guofeng Cui and He Zhu. dPads Source Code. `https://github.com/RU-Automated-Reasoning-Group/dPads`, 2021. [Online; accessed 26-Oct-2021].

[20] 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.

[21] Eyrun Eyjolfsdottir, Steve Branson, Xavier P.Burgos-Artizzu, Eric D. Hoopfer, Jonathan Schor, David J. Anderson, and Pietro Perona. Fly v. fly dataset, 2021. URL `https://data.caltech.edu/records/1893`.

[22] 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.

[23] Pranay Gupta, Anirudh Thatipelli, Aditya Aggarwal, Shubh Maheshwari, Neel Trivedi, Sourav Das, and Ravi Kiran Sarvadevabhatla. Quo vadis, skeleton action recognition ?, 2020.

[24] Joao Carreira, Eric Noland, Chloe Hillier, and Andrew Zisserman. A short note on the kinetics-700 human action dataset. *arXiv preprint arXiv:1907.06987*, 2019.

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

[26] Maxwell I. Nye, Luke B. Hewitt, Joshua B. Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 4861–4870. PMLR, 2019.

[27] 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*. OpenReview.net, 2017.

[28] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR, 2017.

[29] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[30] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[31] Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[32] Amit Zohar and Lior Wolf. Automatic program synthesis of long programs with a learned garbage collector. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 2098–2107, 2018.

[33] 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.

[34] Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, S. M. Ali Eslami, and Oriol Vinyals. Synthesizing programs for images using reinforced adversarial learning. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1652–1661. PMLR, 2018.

[35] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 6062–6071, 2018.

[36] Kevin Ellis, Armando Solar-Lezama, and Joshua B. Tenenbaum. Unsupervised learning by program synthesis. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *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.

[37] Brenden Lake, Ruslan Salakhutdinov, and Joshua Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350:1332–1338, 12 2015. doi: 10.1126/science.aab3050.

[38] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In Jeffrey S. Foster and Dan Grossman, editors, *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. ACM, 2018. doi: 10.1145/3192366.3192410.

[39] 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.

[40] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

[41] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European conference on computer vision (ECCV)*, pages 19–34, 2018.

[42] 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. doi: 10.1609/aaai.v33i01.33014780.

[43] Yufan Jiang, Chi Hu, Tong Xiao, Chunliang Zhang, and Jingbo Zhu. Improved differentiable architecture search for language modeling and named entity recognition. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3576–3581, 2019.

[44] Chenxi Liu, Liang-Chieh Chen, Florian Schroff, Hartwig Adam, Wei Hua, Alan L. Yuille, and Li Fei-Fei. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

A Optimality of the search procedure in Algorithm 1

In Algorithm 1, once we have an optimized program derivation graph G, due to the top-N preservation strategy, each node retains a small number of partial architectures. Algorithm 1 maintains a queue Q of program derivation graphs that is initialized to $[G]$. The algorithm dequeues one graph q from Q and extracts the top-most and left-most node u of q that contains more than one partial architecture for search. It enumerates each available partial architecture $f_k^u\big(\alpha_1^{u,k}, \ldots, \alpha_{\eta(f_k^u)}^{u,k}\big)$ on u and computes an s-score for each option of retaining only f_k^u on u, denoted as $q[u/f_k^u]$. We define

$$s(q[u/f_k^u]) = g(q[u/f_k^u]) + h(q[u/f_k^u])$$

The $g(q[u/f_k^u])$ function measures the structure cost of expanding the initial nonterminal up to u and $h(q[u/f_k^u])$ is an ϵ-Admissible heuristic estimate of the cost-to-go from node u [7, 18] for A* search:

$$\begin{aligned} h(q[u/f_k^u]) =& 1 - F_1(\mathcal{T}_{w^*,\theta^*}[u/f_k^u], D_{val}) \\ & \text{where } w^*, \theta^* = \arg\min_{w,\theta} \mathbb{E}_{i_k,o_k\sim D}[\ell\big(\mathcal{T}_{w,\theta}[u/f_k^u]), o_k\big)] \end{aligned} \tag{4}$$

where $\mathcal{T}$ encodes the program derivation graph q itself via Equation (3) as a differentiable program whose output is weighted by the output of all complete programs included in q, w and θ are the sets of architecture weights and unknown program parameters in the subgraph rooted at u in $q[u/f_k^u]$. The h function fine-tunes these trainable variables using the training dataset D to provide informed feedback on the contribution to program quality of the choice of retaining f_k^u on node u, measured by F_1 score. In practice, to avoid overfitting, we use a separate validation dataset D_{val} to obtain the F_1 score. After computing the quality score s, we add $q[u/f_k^u]$ back to the queue Q sorted based on s-scores. The search algorithm completes when the derivation graph with the least s-score from Q is a well-typed program, i.e. each graph node contains only one valid architecture choice.

We aim to prove that our search algorithm is optimal given the admissible heuristic function h. When multiple solutions exist in G (when the top-N parameter is greater than 1), the algorithm finds an optimal solution. Among all the programs contained in G, the synthesized program optimally balances program accuracy and structure complexity.

We note that our search algorithm is a variant of A* search by by interpreting $g(q[u/f_k^u])$ as cost-so-far and $h(q[u/f_k^u])$ as heuristic cost-to-go. A* search is optimal given admissible heuristics. We shows that under heuristics that are ϵ-admissible, our search algorithm returns solutions that at most an additive constant ϵ away from the optimal solution.

Firstly, we prove that that our heuristic function h is ϵ-admissible. Let a completion of a partial architecture $q[u/f_k^u]$ be a (complete) architecture $\tilde{q}[u/f_k^u]$ obtained by retaining only one partial architecture on any node of q. The cost-to-go at $q[u/f_k^u]$ is given by:

$$J(q[u/f_k^u]) = \min_{\tilde{q}[u/f_k^u]} c(\tilde{q}[u/f_k^u]) - c(q[u/f_k^u]) + 1 - F_1(\tilde{q}[u/f_k^u], D_{val})$$

where the structural cost $c(q)$ is the sum of the costs of the grammatical rules used to construct q excluding any nodes with more than 1 partial architectures (i.e. unexplored nodes in search).

The optimization in Equation (4) may only converge to a local minimum. However, since our relaxation of the search space for Equation (4) includes any possible program permitted by q, there must exist architecture weights w^* and program parameters θ^* such that

$$\forall \tilde{q}[u/f_k^u].\ 1 - F_1(\mathcal{T}[u/f_k^u], w^*, \theta^*, D_{val}) \leq 1 - F_1(\tilde{q}[u/f_k^u], D_{val}) + \epsilon$$

Thus we have:

$$\begin{aligned} h(q[u/f_k^u]) \leq\ & 1 - F_1(\mathcal{T}[u/f_k^u], w^*, \theta^*, D_{val}) \\ \leq\ & \min_{\tilde{q}[u/f_k^u]} 1 - F_1(\tilde{q}[u/f_k^u], D_{val}) + \epsilon \\ \leq\ & \min_{\tilde{q}[u/f_k^u]} c(\tilde{q}[u/f_k^u]) - c(q[u/f_k^u]) + 1 - F_1(\tilde{q}[u/f_k^u], D_{val}) + \epsilon \\ \leq\ & J(q[u/f_k^u]) + \epsilon \end{aligned} \tag{5}$$

In other words, $h(q[u/f_k^u])$ is ϵ-admissible as for a fixed constant $\epsilon > 0$, h is an ϵ-admissible heuristic function over architectures such that $h(q[u/f_k^u]) \leq J(q[u/f_k^u]) + \epsilon$ for any partial architecture f_k^u on u.

Table 4: $F_{S,\theta}(x)$ for the Crim13 dataset.

Extract Feature	Dimension
Position	0, 1, 2, 3
Distance	4
Distance Change	5
Velocity	11, 12, 13, 14
Acceleration	15, 16, 17, 18
Angle	6, 7, 10
Angle Change	8, 9

Table 5: $F_{S,\theta}(x)$ for the Fly-vs-fly dataset.

Extract Feature	Dimension
Linear	17, 25
Angular	18, 26, 27
Positional	24, 28
Ratio	22, 23
Wing	19, 20, 21

Table 6: $F_{S,\theta}(x)$ for the Basketball dataset.

Extract Feature	Dimension
Ball	0, 1
Offense	2, 3, 4, 5, 6, 7, 8, 9, 10, 11
Defence	12, 13, 14, 15, 16, 17, 18, 19, 20, 21

ϵ-Optimality. Based on the ϵ-admissible of heuristic function $h(q[u/f_k^u])$, we prove that the variant of A* search in Algorithm 1 results in a synthesized program that is at most ϵ away from the optimal solution contained in the search graph. Suppose that Algorithm 1 returns a program P^r that does not have the optimal cost C^*. Then there must exist a program derivation graph q^* in the queue Q of Algorithm 1 that contains the architecture of the optimal program P^*. Due to Equation 5 and the fact that Q is sorted, the s-score of P^r satisfies:

$$\begin{aligned} s(P^r) &\leq s(q^*[u^*/f^*]) \\ &= g([q^*[u^*/f^*]) + h(q^*[u^*/f^*]) \\ &\leq g(q^*[u^*/f^*]) + J(q^*[u^*/f^*]) + \epsilon \\ &\leq C^* + \epsilon \end{aligned}$$

where u^* is the top-most and left-most node of q^* and f^* is the optimal partial architecture to retain on node u^* to get the optimal program P^*. In other words, we have established an upper bound on the path cost of the returned synthesized program P^r.

B Context-free Grammar Details

We use the context-free grammar in Fig. 1 for synthesizing programmatic classifiers for all the datasets in our experiment. For each dataset, similar to NEAR [7], we customize parameterized functions $F_{S,\theta}(x)$ that extract a vector consisting of a predefined subset S of the dimensions of an input data item x and pass the extracted vector through a linear function with trainable parameters θ. We disclose the details of $F_{S,\theta}(x)$ for each dataset below.

Crim13 Dataset. As shown in Table 4, we define 7 feature extraction functions $F_{S,\theta}(x)$ for the Crim13 Dataset. For location information, XY positions of a pair of mice and the distance between them are recorded. Additionally, *distance change* measures the distance difference for each two consecutive frames. To track movement information, velocity and acceleration of a pair of mice are extracted in X and Y dimensions respectively. Besides, we also include the information on *angle* and *angle change*. The former contains the two relative directions between a pair of mice (one for mouse 1 relative to mouse 2, another for mouse 2 relative to mouse 1) and the difference between the two relative directions. *Angle change* represents the change of the two relative directions over time. More information on the dimensions of this dataset can be found in [20].

Fly-vs-fly Dataset. Table 5 shows the feature functions $F_{S,\theta}(x)$ we extract for the Fly-vs-fly dataset. Although the feature vectors of the dataset have 53 dimensions, we find the 5 feature functions in the table are sufficient to obtain high accuracy and F_1-scores. The *linear* feature function captures the values of velocity and distance between two flies. The *Angular* feature function extracts the value of angle velocity, relative angle between flies and facing angle of each fly. The feature function *Positional* captures the distance over a relative object and fly legs. The feature function *ratio* extracts the body ratio of two flies. The feature function *wing* extracts the angles and lengths of fly wings.

Table 7: $F_{S,\theta}(x)$ for the SK152 dataset.

Extract Feature	Point
Arms	2, 3, 4, 5, 6, 7
Legs	8, 9, 10, 11, 12, 13, 14, 19, 20, 21, 22, 23, 24
Faces	0, 1, 15, 16, 17, 18

Table 8: Dataset details

Dataset	feature dim.	category num.	max seq. len.	# train	# valid	# test
Crim13	19	2	100	12404	3077	2953
Fly-vs-fly	53	7	300	5341	629	1050
Basketball	22	6	25	18000	2801	2693
SK152	75	10	100	8721	2184	892

Basketball Dataset. As shown in table 6, we define three feature extraction functions for the Basketball dataset, extracting the positions of the basketball, 5 offensive players, and 5 defensive players. All positions are expressed in X and Y coordinates.

SK152 Dataset. This dataset uses a total of 25 points to capture human skeletons. Each point is recorded using XYZ coordinates. We define three customized feature functions *arms*, *legs* and *face*, each of which extracts a subset of the 25 features.

C Experiment Details

We provide more details about our experiment in this section.

C.1 Training Details

Datasets. Table 8 gives the full details of the four datasets used for evaluation. NEAR [7] does not release the Fly-vs-fly and Basketball datasets used to obtain the results in its paper. We sample these datasets following the guidance provided in [7]. Therefore, the datasets used in the evaluation of this paper are not completely equivalent to that of [7].

Structure Cost. To penalize complex program architectures, we implement the structure cost function g similar to NEAR [7]. Let each grammar rule r have a non-negative real-valued cost c(r). The structural cost of a (partial) architecture is the sum of the costs of the multi-set of rules used to create the architecture.

$$g(q[u/f_k^u]) = \beta \cdot \sum_{r \in q[u/f_k^u]} c(r)$$

Importantly, the above formula only counts the grammar rules used to expand the initial nonteriminal up to node u (recall that u is the top-most and left-most node of q that contains more than one partial architecture i.e. unexplored nodes are excluded). To balance the structure cost and performance of a programmatic classifier, we set the cost penalty parameter β to be 0.01 for both dPads and NEAR. In practice, we set $c(r) = 1$ for any grammar rule r. For a complete program, the g function essentially counts the number of grammar rules used to derived the program (timed with β).

C.2 Hyperparameters

RNN Baseline. The RNN baseline policies are 1-layer LSTMs. Table 9 introduces the hyperparameters used for training the RNN baselines. In general, the RNN baselines perform better than the synthesized programmatic classifiers because their richer structures allow for better data fitting at the cost of less interpretability. In the experiments, we use the cross-entropy loss to optimize classifier accuracy for both dPads and baselines.

Several other baselines that we considered include (1) Top-down enumeration that synthesizes and evaluates complete programs in order of increasing complexity measured using the structural cost, (2) Monte-Carlo sampling that constructs complete programs by sampling rules (edges) with

Table 9: Hyperparameters set for the RNN baseline.

Dataset	# LSTM units	# epochs	learning rate	batch size
Crim13	100	50	0.001	50
Fly-vs-fly	80	40	0.00025	30
Basketball	64	15	0.01	50
SK152	75	30	0.01	50

Table 10: Hyperparameters set for Differentiable Architecture Search and Selection in dPads.

Dataset	Architecture Search	Architecture Selection			Batch Size
	learning rate	graph_epoch	prog_epoch	learning rate	
Crim13	0.001	6	6	0.001	200
Fly-vs-fly	0.0005	6	6	0.0005	200
Basketball	0.001	4	6	0.02	50
SK152	0.01	4	6	0.01	200

probabilities proportional to their structural costs where the next node to expand along a path has the best average performance of samples that descended from that node, (3) Monte-Carlo tree search (MCTS) that traverses the search graph of programs using the UCT selection criteria, where the value of a node is inversely proportional to the cost of its children, and (4) Genetic algorithm that uses crossover, selection, and mutation operations to evolve a population of programs over a number of generations. Unlike dPads, these baselines perform program architecture search in the discrete space of DSL grammar rules. We do not include these baselines in the experiment section because NEAR significantly outperforms them [7]. Therefore, it suffices to compare dPads solely with NEAR.

dPads. In the evaluation of dPads, we set graph expansion depth d_s (a parameter of Algorithm 1) as 2 for all the four datasets. NEAR uses the number of grammar production rules applied to construct a program to upper-bound the search space for program learning. The largest number of production rules allowed for a synthesized program is 8 in NEAR. Instead, we set the max depth of the abstract syntax tree of a dPads's synthesized program as 4. Compare to the search space of NEAR, some programs included in the search space of dPads even need more than 10 production rules to expand from the initial nonterminal.

For top-N preservation in a program derivation graph, the goal is to retain top-N program architectures as children for each partial architecture on the node's parent, which are defined to be those assigned with higher weights on the node's incoming edge in the previous graph unfolding iteration. We set $N = 2$ in our experiments. In practice, we find that a heuristic strategy that *iteratively* prunes candidate partial architectures on a program derivation graph node and fine-tunes the derivation graph at then end of each *iteration* is more efficient than directly retaining top-N architectures. In a graph unfolding iteration, after optimizing the architecture weights and unknown program parameter over an entire program derivation graph for several epochs until the increase of the F_1 score of the whole graph is less than 1%, on each node we retain 4 program architectures as children for each partial architecture on the node's parent, then we retrain the program derivation graph for several epochs again until the increase of F_1 score is less than 1% and on each node we retain 3 program architectures for each partial program architectures on the node's parent, and finally we apply such a process again to retain only 2 program architectures to get the desired top-2 preservation in the program derivation graph.

Table 10 presents the hyperparameters used to train dPads programmatic models. The learning rates for differentiable architecture search on a program derivation graph and architecture selection from the converged program derivation graph may be different. For architecture selection, the number of graph_epoch refers to the number of epochs that is used to optimize the heuristic function h (Equation 4) when a program derivation graph has nodes containing more than one candidate architectures. Here we need to optimize both architecture weights and program parameters. The number of prog_epoch refers to the number epochs that is used to fine-tune the accuracy of a program derivation graph when every node of the graph contains exactly one architecture, i.e., the graph is the abstract syntax tree of a valid program. For Crim13 and Fly-vs-fly, both graph_epoch and prog_epoch are set to 6. For Basketball and SK152, graph_epoch is set to 4 and prog_ epoch is set to 6.

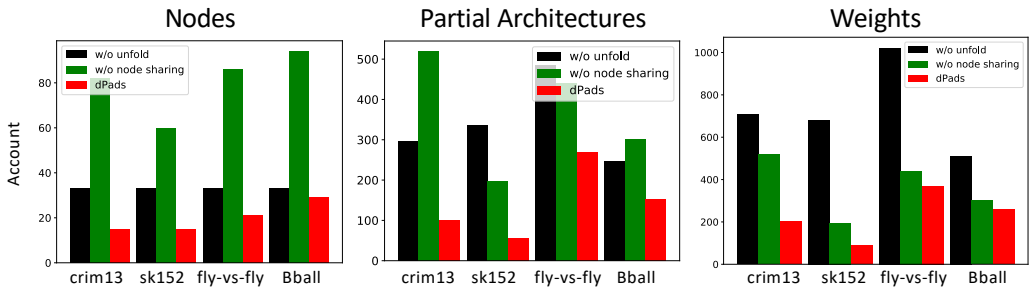

Figure 7: Results of quantifying the number of graph nodes, the total number of partial architectures (DSL functions) hosted by the nodes, and the total number of architecture weights on graph edges on program derivation graphs generated by dPads and its variants for each of the datasets.

C.3 Additional Experiment Results

Search Space Reduction. Fig. 7 quantifies the program derivation graph reduction by node sharing and progressive unfolding. We show for each benchmark the number of nodes, the total number of partial architectures (DSL functions) hosted by the nodes, and the total number of architecture weights to train on its program derivation graph. We only show the results of the deepest program derivation graph generated by dPads on each benchmark. Notice that dPads without progressive graph unfolding and dPads without node sharing generate program derivation graphs that are significantly larger.

On Fly-vs-fly, without progressive graph unfolding, the program derivation graph has 33 nodes shared by 485 partial architectures, and a total of 1021 architecture weights to train. Directly applying dPads to train under this setting encounters an out-of-memory (OOM) exception (Table 2). In contrast, with progressive graph unfolding, the deepest graph generated by dPads has only 21 nodes shared by 220 partial architectures, and a total of 370 architecture weights to train. dPads converges in less than 360 mins (Table 2).

On Basketball, without node sharing, the deepest program derivation graph generated by dPads has 94 nodes hosting 302 partial architectures, and a total of 301 architecture weights to train. If directly running dPads under this setting, the search procedure (Sec. 3.3) times-out (Table 2). In contrast, with node sharing, the program derivation graph reduces to only 29 nodes shared by 152 partial architectures, and a total of 259 architecture weights. dPads converges in less than 180 mins (Table 2).

Program Sizes. Table 11 reports the number of grammar rules used to construct synthesized programs for each benchmark. Compared with NEAR, dPads tends to synthesize more complex programs that are necessary to ensure higher F_1 scores.

On Fly-vs-fly, although NEAR runs faster, it only finds programs derived by 2.8 production rules but dPads finds much deeper programs derived by 6.8 production rules. Consequently, for this benchmark, dPads achieves much higher accuracy and F_1 scores. A similar trend can be observed for the results on Crim13 and SK152. dPads learns program classifiers for Crim13 constructed using 10.2 grammar rules averagely. These classifiers achieve an average of 0.46 F_1 scores (on the test dataset). The shallower classifiers learned by NEAR achieve an average of 0.33 F_1 scores constructed by 5.8 grammar rules averagely. While the dPads classifiers are more complex, the total s-scores of dPads classifiers $(1 - F_1(p, D_{val}) + \beta \cdot \sum_{r \in AST(p)} c(r)$ where p is a classifier), i.e. synthesized programs' classification error plus architecture cost (the search objective), are still lower than that of the programs learned by NEAR. In other words, dPads better balances structure costs and performance. On Basketball, both tools find programs with 8 production rules. In this case, the architecture spaces searched by the two tools are roughly equivalent, but dPads is 3 times faster.

The experiment results consistently demonstrate that dPads's gradient-based architecture search is much more efficient than NEAR's enumeration-based strategy. Moreover, NEAR uses neural models to estimate the performance of a partially expanded program. Our experiments find that due to overfitting or underfitting, such a neural model may be biased on a particular program. For example, on Fly-vs-fly, due to the biased estimation, NEAR stops searching the architecture space deeper than that contains programs derived by only 2.8 production rules. In contrast, (1) dPads uses a

Table 11: The average numbers of grammar rules (#R) used to construct synthesized programs together with the programs' F_1 scores. All results are reported as the average of runs on five random seeds. Costs of time are set to minutes.

	Crim13-sniff			**Fly-vs-fly**			**Bball-ballhandler**			**SK152-10 actions**		
	F_1	#R	Time	F_1	#R	Time	F_1	#R	Time	F_1	#R	Time
A*-NEAR	.286	6.8	164.92	.828	2.8	**243.82**	.940	8.0	553.01	.312	4.2	210.23
IDS-BB-NEAR	.323	5.8	463.36	.822	2.4	465.57	.793	7.0	513.33	.314	4.4	848.44
dPads	**.458**	10.2	**147.87**	**.887**	6.8	348.25	**.945**	8.0	**174.68**	**.337**	7.6	**162.70**

Table 12: Standard deviations of F_1 scores, accuracy rates, and the number of grammar rules used to construct synthesized programs. All results are reported as the average of runs on five random seeds.

	Crim13			**Fly-vs-fly**			**Basketball**			**SK152**		
	F_1	Acc.	Rules	F_1	Acc.	Rules	F_1	Acc.	Rules	F_1	Acc.	Rules
A*-NEAR	.107	.007	3.27	.025	.016	1.10	.007	.009	0.0	.017	.018	0.45
IDS-BB-NEAR	.086	.008	2.39	.030	.025	0.89	.012	.015	0.0	.012	.010	0.54
dPads	.013	.008	5.31	.011	.006	1.10	.004	.004	0.0	.017	.017	0.55

sub program derivation graph to estimate the performance of a partially expanded program that can provide more accurate assessment due to the graph's syntax resemblance to a valid program; (2) dPads only uses neural models to provide "contrastive" performance estimation for a set of programs sharing nodes in a program derivation graph when progressively unfolding the graph. As a result, on Fly-vs-fly, dPads searches the architecture space much deeper containing programs derived by 6.8 production rules. Even dPads typically searches much deeper, it often runs faster than NEAR.

Standard Deviations. We present the standard deviations of accuracy, F_1-scores, and numbers of production rules used to construct learned programs in Table 12. The results confirms the observation in [7] that NEAR has a higher variance in F_1 scores for CRIM13. dPads is more stable on this dataset.

Further Comparison with NEAR. In the comparison with NEAR, we obtain the NEAR results following the hyperparameters defined in [7]. The only exception is that for Crim13 and Fly-vs-fly, we modify the batch size for NEAR program learning to be 200 while keeping the other hyperparameters unmodified to get the results. This is for ensuring a fair comparison of running times with dPads that sets batch size to 200. In this section, we report the results of NEAR on Crim13 and Fly-vs-fly using exactly the same hyperparameters as reported in [7], setting batch size to 50 and 30 respectively, and show the results in Table 13. In this setting, A*-NEAR and IDS-BB-NEAR get higher F_1 scores that are, however, still lower than that of dPads. Moreover, NEAR costs significantly longer time in search. Thus, dPads outperforms NEAR in this setting as well.

Comparison with Enumerative Program Synthesis. Besides comparing with NEAR and RNN (as in Table 1), we also compared dPads with an enumeration strategy that synthesizes and evaluates complete programs in order of increasing complexity. This strategy is widely used in program synthesis tasks. We set the running time of the enumeration strategy twice as long as dPads's synthesis time. As shown in Table 14, dPads outperforms enumeration strategy on all benchmarks. Although enumeration gets higher accuracy on Cim13, it underfits this unbalanced dataset as the F_1 score is much lower than dPads. We have also tried a Monte-Carlo tree search strategy but found that its performance is even worse than the simple enumeration strategy on our benchmarks.

C.4 Additional Ablation Study Results

Other than top-N preservation and progressively unfolding program derivation graphs, we explored other pruning approaches, including reserving the *top-N programs* across the entire search graph. We also considered another pruning algorithm which we refer to as *First Compare First Unfold* (FCFU). Unlike dPads that separates top-N preservation and progressive graph unfolding with the search algorithm in Sec. 3.3, FCFU mixes the two optimizations with the search procedure. It manages a priority queue of program derivation graphs that is initialized to the simplest graph with the initial nonterminal only. After training converges on a program derivation graph from the priority queue, FCFU decomposes the graph into several sub graphs by separating the co-adapted architectures on

Table 13: Experiment results of NEAR following the exact hyperparameter setting specified in [7]. All results are reported as the average of runs on five random seeds. Costs of time are set to minutes.

	Crim13-sniff			**Fly-vs-fly**		
	F_1	Acc.	Time(mins)	F_1	Acc.	Time(mins)
A*-NEAR	.304	.824	519.60	.873	.827	1208.92
IDS-BB-NEAR	.328	.840	1106.28	-	-	> 1440
dPads	**.458**	.812	**147.87**	**.887**	.853	**348.25**

Table 14: Experiment results on comparing dPads with an enumeration-based synthesis strategy. The running time of the enumeration strategy is set twice as long as the synthesis time of dPads. All results are reported as the average of runs on five random seeds. Costs of time are set to minutes.

	Crim13		**Fly-vs-fly**		**Basketball**		**SK152**	
	F_1	Acc.	F_1	Acc.	F_1	Acc.	F_1	Acc.
Enumeration	.294	.856	.850	.774	.795	.767	.288	.284
dPads	.458	.812	.887	.853	.945	.939	.337	.337

Table 15: Ablation study on various pruning methods. All results are reported as the average of runs on five random seeds. Costs of time are set to minutes.

	Crim13			**Fly-vs-fly**			**Basketball**			**SK152**		
	F_1	Acc.	Time	F_1	Acc.	Time	F_1	Acc.	Time	F_1	Acc.	Time
top-5 programs	.299	.516	184.16	.652	.554	153.28	.848	.829	30.59	.283	.277	62.04
FCFU	.456	**.813**	489.53	**.889**	**.853**	606.65	-	-	> 1440	**.338**	**.339**	319.60
dPads	**.458**	.812	147.87	.887	**.853**	348.25	**.945**	**.939**	174.68	.337	.337	162.70

the top-left most compound node on the graph. On each sub graph, that node contains only one available architectures. These partitions are pushed back to the queue after fine-tunning (similar to dPads). Once a graph with each node containing at most one architecture is obtained from the queue as the least cost, we immediately unfold the graph into deeper levels and push it back to the queue unless the maximum depth is reached. FCFU prioritizes to unfold the best partial program observed so far. We compared dPads with FCFU and top-N programs over 5 random runs. The results are shown in Table 15.

It can be seen that dPads outperforms *top-N programs* where N is set to 5, achieving higher accuracy and F_1 scores. This is because *top-N programs* tends to excessively detach many valid DSL functions from graph nodes (especially when N is small), leading to suboptimal final programs. FCFU achieves comparable accuracy and F_1 scores with dPads but runs significantly slower than dPads. It even times-out on the Basketball dataset. This is because FCFU only unfolds one best partial program each time, causing the search queue to grow exponentially longer with graph decomposition. In contrast, dPads maintains a set of high-quality programs via top-N preservation on nodes and simultaneously expands all of these programs via progressive graph unfolding.

C.5 Program Examples

We show more synthesized programs by dPads for the four datasets below.

Crim13. The following is a programmatic classifier learned for Crim13.

```
Map(
    if AccelerationAffine_θ1(x_t) ≥ 0
    then if PositionAffine_θ2(x_t) ≥ 0
         then PositionAffine_θ3(x_t)
         else VelocityAffine_θ4(x_t)
    else Multiply(DistanceAffine_θ5(x_t), DistanceAffine_θ6(x_t))) x
```

In the program, AccelerationAffine, DistanceAffine, PositionAffine, and VelocityAffine are functions that first select the parts of the input that represent acceleration, distance, position, and velocity

measurements, respectively, and then apply affine transformations with parameters θ to the resulting vectors. The program contains a nested if-then-else (ITE) operation conditioned on the acceleration of two mice. Our interpretation of the program is that if the difference between the accelerations of two mice is small, in the first else branch, they are doing "sniff" if the two mice are close to each other without obvious movements (i.e. the multiplication of their distance is small); otherwise, in the first then branch, the program 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.

We show another programmatic classifier learned for Crim13 below:

```
MapPrefixes(
    Fold(
        if DistanceAffine_θ1(x_t) ≥ 0
        then PositionAffine_θ2(x_t)
        else PositionAffine_θ3(x_t))) x
```

This program has a simpler architecture compared with the one above but has a higher F_1 score (0.468 vs. 0.456) that is comparable to the RNN baseline (vs. 0.481). It exploits the distance and the position bias between two mice to evaluate if they are doing "sniff".

Fly-vs-fly. We draw two programmatic classifiers for Fly-vs-fly with similar structures below. Both

```
Fold(
  Add(
    Add(PositionalAffine_θ1(x_t),
        WingAffine_θ2(x_t)),
    RatioAffine_θ3(x_t))) x
```

```
Fold(
  Add(
    Add(AngularAffine_θ1(x_t),
        WingAffine_θ2(x_t)),
    Add(RatioAffine_θ3(x_t),
        RatioAffine_θ4(x_t)))) x
```

programs consider the wing and the body ratio features of fruit flies for classification. The left program further extracts the position information and gets a high F_1 score 0.904 while the right program extracts the fly angle velocity and relative facing angle information that results in a lower F_1 score 0.888. The better performance of the left program suggests that the position information of flies is more crucial than the angle features to classify their actions.

SK152. For SK152, we show a learned programmatic classifier that is more complex compared with the one depicted in the main paper.

```
SlideWindowAvg(
  Add(
    if ArmsXYZAffine_θ1(x_t) ≥ 0
    then ArmsXYZAffine_θ2(x_t) else ArmsXYZAffine_θ3(x_t),
    if LegsXYZAffine_θ4(x_t) ≥ 0
    then FaceXYZAffine_θ5(x_t) else LegsXYZAffine_θ6(x_t))) x
```

This program applies a sliding window to a trajectory for classification and sums the results of two ITE operations inside the window. The first ITE operation focuses on human arm behaviors and the second ITE operation leverages either face movements or leg movements based on the behavior of human legs. The program results in an F_1 score of 0.336 and an accuracy of 0.337.

Basketball. The program synthesized for Basketball to classify the ballhandler is shown below.

```
Map(
  Multiply(
    Add(BallXYAffine_θ1(x_t), OffenseXYAffine_θ2(x_t)),
    Add(OffenseXYAffine_θ3(x_t), BallXYAffine_θ4(x_t)))) x
```

In the program, OffenseXYAffineand and BallXYAffine are parameterized affine transformations over the XY-coordinates of the offensive players and the ball. The program structure can be interpreted as computing the distance between the offensive players and the ball to determine the ballhandler. As

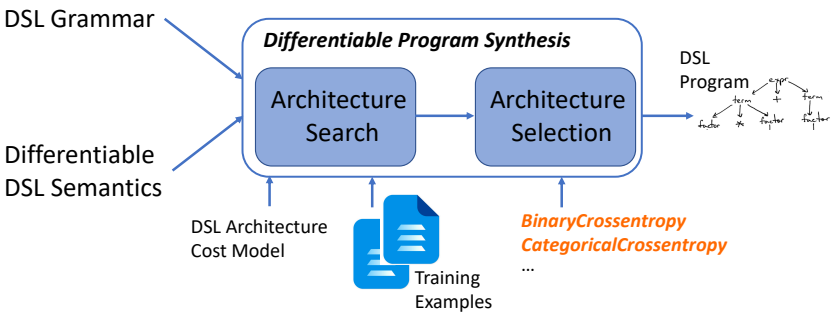

Figure 8: A high-level framework of dPads.

$$
\begin{aligned}
&\alpha_0 \to \textbf{And}(\alpha_1, \alpha_1) \mid \textbf{Not}(\alpha_1) \mid \textbf{Or}(\alpha_1, \alpha_1) \mid \textbf{Xor}(\alpha_1, \alpha_1)\\
&\alpha_1 \to \textbf{And}(\alpha_2, \alpha_2) \mid \textbf{Not}(\alpha_2) \mid \textbf{Or}(\alpha_2, \alpha_2) \mid \textbf{Xor}(\alpha_2, \alpha_2) \mid LN10\\
&\alpha_2 \to \textbf{And}(\alpha_3, \alpha_3) \mid \textbf{Not}(\alpha_3) \mid \textbf{Or}(\alpha_3, \alpha_3) \mid \textbf{Xor}(\alpha_3, \alpha_3)\\
&\alpha_3 \to \textbf{And}(\alpha_4, \alpha_4) \mid \textbf{Not}(\alpha_4) \mid \textbf{Or}(\alpha_4, \alpha_4) \mid \textbf{Xor}(\alpha_4, \alpha_4) \mid LN32\\
&\alpha_4 \to LN30 \mid LN42
\end{aligned}
$$

Figure 9: Context-free grammar for cryptographic circuit synthesis in the SyGus project.

we aim to recognize the offensive player who holds the basketball, it suffices to only consider the ball positions and the offensive players (excluding the information on defensive players). This program gets a F_1 score of 0.945 and an accuracy of 0.939.

C.6 Generalizability

In the paper, we illustrate dPads by focusing on sequence classification. However, dPads is a general program synthesis algorithm and is not limited to sequence classification.

Fig. 8 depicts a high-level framework of dPads. Other than training data, dPads takes as input a DSL, the semantics of the DSL, and a cost model of each production rule in the DSL. Importantly, dPads requires the DSL's semantics to be differentiable. This is because dPads performs gradient-based architecture search. To avoid discontinuities in programming constructs such as ITE, we require these constructs to be interpreted in terms of a smooth approximation. As such, the synthesis algorithm in dPads is exactly parameterized by a class of DSLs with differentiable semantics. The context-free grammar in Fig. 1 is such an example. It is straightforward to apply dPads to another DSL with differentiable semantics.

As an example, we demonstrate the generalizability of dPads by applying it to the cryptographic circuit synthesis task in the SyGuS (Syntax-Guided Synthesis) project[1]. The goal is to synthesize a side-channel free cryptographic circuit by following given context-free grammar while ensuring that the synthesized circuit is equivalent to the original circuit (a correctness constraint). The grammar is designed to avoid side-channel attacks, whereas the original circuit is created only for functional correctness and thus is vulnerable to such attacks. dPads takes as input a circuit grammar as depicted in Figure. 9. The grammar includes several Boolean operations **And**, **Not**, **Or** and **Xor**. It also specifies *multiple variables* (e.g. $LN10$ and $LN30$) to be used by the synthesizer to generate a desired program (circuit in this context). In this experiment, we aim to synthesize a program that must be logically equivalent to φ_{spec} (a correctness specification):

$$\varphi_{spec} : (((LN30 \ \textbf{Xor} \ LN32) \ \textbf{Xor} \ LN42) \ \textbf{Xor} \ LN10) \tag{6}$$

Notice that the above program itself cannot be expressed using the above grammar.

In order to apply dPads to such a task, the user of dPads must provide a differentiable DSL semantics. Recall that we have provided a smooth approximation of ITE in Sec.2. We define a differentiable

[1]https://sygus.org/

$$
\begin{aligned}
[\![\textbf{And}(\alpha_1, \alpha_2)]\!](v) &= \textbf{min}([\![\alpha_1]\!](v), [\![\alpha_2]\!](v)) \\
[\![\textbf{Or}(\alpha_1, \alpha_2)]\!](v) &= \textbf{max}([\![\alpha_1]\!](v), [\![\alpha_2]\!](v)) \\
[\![\textbf{Xor}(\alpha_1, \alpha_2)]\!](v) &= [\![\alpha_1]\!](v) + [\![\alpha_2]\!](v) - 2[\![\alpha_1]\!](v) \cdot [\![\alpha_2]\!](v) \\
[\![\textbf{Not}(\alpha)]\!](v) &= 1 - [\![\alpha]\!](v) \\
[\![LN10]\!](v) &= v[LN10] \\
[\![LN30]\!](v) &= v[LN30] \\
[\![LN32]\!](v) &= v[LN32] \\
[\![LN42]\!](v) &= v[LN42]
\end{aligned}
$$

Figure 10: Differentiable semantics for Boolean operations.

semantics for **And**, **Not**, **Or** and **Xor** similarly in Figure 10. Given a program φ in the DSL and a Boolean assignment v as the variables in φ, $\varphi(v)$ is deemed to be True if $[\![\varphi(v)]\!]$ is closer to 1 and $\varphi(v)$ is deemed to be False if $[\![\varphi(v)]\!]$ is closer to 0.

dPads constructs a program derivation graph to include each possible program allowed by the grammar. Given a set of input-output examples, it trains the architecture weights of the program derivation graph by minimizing the MSE loss between the graph's outputs (Equation. 3) and the ground truth outputs. In this example, an input is an assignment to the variables, e.g. $v = \{LN10 : 0,\ LN30 : 1,\ LN32 : 0,\ LN42 : 1\}$. The corresponding output is whether the input variable assignment v should be evaluated to 1 (True) or 0 (False). We collect the input-output examples using counterexample-guided inductive synthesis (CEGIS) by iteratively querying an SMT solver (such as Z3[2]) whether a synthesized program φ is logically equivalent to φ_{spec}. Any counterexample that witnesses the inequivalence of φ and φ_{spec} is added to the input-output example set. Since the DSL semantics is differentiable, dPads can efficiently learn architecture weights using gradient descent optimization and hence return the best program it synthesizes. For our example, dPads synthesizes the following solution that is verified equivalent to φ_{spec}:

$$(LN10\ \textbf{Xor}\ ((LN32\ \textbf{Xor}\ (LN42\ \textbf{Or}\ LN30))\ \textbf{Xor}\ ((LN30\ \textbf{Or}\ LN42)\ \textbf{And}\ (LN30\ \textbf{And}\ LN42))))$$

In our experience, dPads is very efficient in solving the circuit synthesis problem and can reduce the synthesis time from minutes by EUSolver[3] (an enumerative SyGuS solver) to seconds.

D Additional Discussions

D.1 Limitations

Performance gap to RNN. dPads's performance does not match the RNN baseline. This is mainly due to the limitations in expressivity imposed by the DSL. Firstly, in the DSL, we only allow for customized feature functions $F_{S,\theta}(x)$ that extract a vector consisting of a predefined subset S of the dimensions of an input sequence x and pass the extracted vector through a linear function with trainable parameters θ. For example, for Crim13, we predefine feature functions such as the XY positions, angles, velocity, acceleration, distance of a pair of mice, and distance difference for every two consecutive frames. We list the details of $F_{S,\theta}(x)$ for each dataset in Appendix B. These feature functions are extremely helpful to ensure that a synthesized program composed of these functions is interpretable. However, a program limited to these predefined feature functions may be suboptimal as only a subset of features is used. Instead, an RNN policy can understand the whole context of a sequence using all features available from the input. Secondly, for the sake of interpretation, our DSL also predefines a limited set of algebraic operations to process the outputs from the feature functions, as shown in Fig. 1. However, an RNN can use a more expressive activation function to learn about long-term dependencies in data. We have reported the performance limitation of dPads in Table 1.

Program Derivation Graph Accuracy. Another important limitation is that performance estimation of each program included in a program derivation graph ranked by architecture weights can be

[2]https://github.com/Z3Prover/z3

[3]https://bitbucket.org/abhishekudupa/eusolver/src/master/

inaccurate due to the co-adaption among partial architectures (node sharing). As one super program derivation graph may not be able to model the entire search space accurately, we have used multiple sub program derivation graphs generated on the fly via a search procedure to address this problem (Section 3.3). Each sub derivation graph models one part of the search space. However, as reported in Table. 3, the search method slows down the whole synthesis procedure and may need additional optimization.

D.2 Further Comparison with NEAR

Although dPads and NEAR [7] both formalize program synthesis as a graph search problem, the two techniques are very different. We compare dPads and NEAR in depth as follows.

NEAR. Typically, search-based program synthesizers enumerate the underlying program space in some order and for each program checks whether or not it satisfies the synthesis constraints. It is a challenging problem because the architecture search space is combinatorial. The most simple strategy that starts by searching for programs with 1 DSL production rule and iteratively increases this bound does not scale to complex programs. NEAR uses neural models to approximate unexpanded subexpressions to estimate the likelihood of eventually deriving a high-quality program by choosing a particular production rule. It leverages this kind of information to prioritize promising search directions and hence can greatly accelerate the search process. However, NEAR's search strategy is still discrete and enumeration-based.

dPads's Contributions. dPads proposes a new, scalable program synthesis technique. It views program architecture search as learning a probability distribution over all possible program architectures induced by a DSL. Unlike NEAR that enumerates and evaluates each program by training unknown parameters from scratch, dPads reduces the computation cost by training one program, a.k.a. a program derivation graph, to approximate the performance of every program in the search space. It learns the architecture weights of each program encoded in a program derivation graph in a way that ranks the performance estimation of these programs. The search procedure in dPads is more efficient because it supports gradient-based architecture optimization in a novel continuous relaxation of the program architecture search space.

dPads's Impacts. To the best of our knowledge, dPads is the first program synthesis technique that applies gradient-based architecture search. Experiment results on sequence classification demonstrate that thanks to this strategy, dPads can efficiently search in a deep program space to learn sophisticated programs that are necessary to ensure high F_1 scores.

D.3 Using Neural Networks to Approximate Missing Expressions

Both dPads and NEAR [7] use neural networks to approximate missing expressions of partial architectures. However, the key difference is that dPads does not use neural models to estimate the performance of a partially expanded program. This is because our experiments find that due to overfitting or underfitting, a neural model may be biased on a particular program. In contrast, dPads uses a sub program derivation graph to estimate the performance of a partially expanded program. We find that it can provide more accurate assessment due to the graph's syntax resemblance to a valid program. This enables dPads to be able to search deeper than NEAR in the program architecture space to synthesize more sophisticated programs that are necessary to ensure higher F_1 scores. For example, on Fly-vs-fly, NEAR's inaccurate admissible neural heuristics prevent it from searching the architecture space deeper than that contains programs derived by just 2.8 production rules. In contrast, during architecture selection, dPads uses a sub program derivation graph to estimate the performance of a partially expanded program. The sub program derivation graph provides a more accurate assessment. As a result, dPads searches the architecture space much deeper containing programs derived by 6.8 production rules and gets a much higher F_1 score (.887 vs .828).

dPads only uses neural models to provide contrastive performance estimation for a set of programs sharing nodes in a program derivation graph for progressive graph unfolding (Section 3.2). For example, on the Basketball dataset, the architecture spaces searched by the two tools are roughly equivalent. dPads is 3 times faster. This is in part because dPads uses the same set of neural networks to provide contrastive performance estimation for the set of programs sharing nodes and does require the performance estimation on a single program to be accurate. In progressive graph unfolding, this strategy effectively and efficiently prunes away a large set of unlikely search directions.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See Section 6
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] Code is shown in Algorithm 1, data and instructions are shown in Section 4.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Section 4 and Appendix.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] See section 4.3.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See section 4
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [Yes] See citations in section 4
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [Yes] See citations in section 4
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]