Differentiable Synthesis of Program Architectures

Anonymous Author(s)
Affiliation
Address
email

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 F1 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 symbolic domain-specific languages (DSLs) 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 achieves competitive performance.

A differentiable program encourages interpretability by using structured symbolic primitives to compose a set of differentiable modules with trainable parameters in its 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 search space of architectures is discrete and combinatorial. Various enumeration strategies have been developed to explore the program architecture space, including greedy enumeration [1, 2], evolutionary search [3], and Monte Carlo approaches [4]. To prioritize highly likely search directions in a combinatorial architecture space, NEAR [5] uses neural networks to approximate missing expressions in a partial program whose F1 score serves as an admissible heuristic to effective graph search algorithms such as A* [6]. However, since the discrete program architecture search space is intractably large, such enumeration-based strategies are inefficient in natural.

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 [7] and neural architecture search. Specifically, to support differentiable learning, DARTS [8–10] uses a composition of softmaxes over all possible candidate operations between a fixed set of neural network nodes to relax the search space of neural architectures. However, applying this method to program synthesis is challenging

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

because the program architecture search space is much richer. 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 in 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 iteratively increased 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 involved programs. We seek to optimize program architecture weights with respect to a loss function 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 this relaxed space is easier and more efficient with gradient-based architecture optimization.

One major challenge of differentiable program architecture synthesis is that a program derivation graph includes an exponential number of programs, involving a huge set of trainable variables over program architectures and parameters. To curb the large program derivation search space, we introduce shared nodes in program derivation graphs and iterative graph unfolding. Shared nodes allow two partial architectures to share the same child nodes encoding the fact that the missing expressions in the two architectures may be expanded using the same grammar rules. Iterative 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 together significantly reduce the search space of program architectures, scaling differentiable program synthesis to real-world classification datasets. We evaluate our synthesis algorithm in the context of synthesizing programmatic classifiers in four sequence classification applications. We demonstrate that our algorithm substantially outperforms state-of-the-art methods for differentiable program synthesis, and can learn classifiers that are highly interpretable and are comparable to neural network models in terms of accuracy and F1-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 search space of grammar rules, enabling differentiable program architecture search. Secondly, we instantiate differentiable program architecture synthesis with effective optimization strategies including iterative graph unfolding and node sharing, 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]. 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 [5]. 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 real vectors or sequences 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

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

Figure 1: Context-free DSL Grammar for Sequence Classification (adapted from [5]).

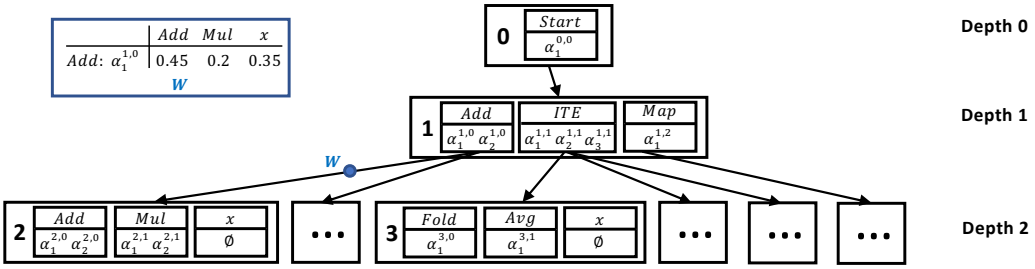

Figure 2: Program Architecture Search Graph of the grammar in Fig. 1.

differentiability, we interpret it in terms of a smooth approximation: $[\![\textbf{ITE}(\alpha_1, \alpha_2, \alpha_3)]\!](x) = \sigma([\![\alpha_1]\!](x)) \cdot [\![\alpha_2]\!](x) + (1 - \sigma([\![\alpha_1]\!](x)) \cdot [\![\alpha_3]\!](x)$ 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 takes 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 frames.

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 multiset 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.

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. For each sequence i_k with a list of frames, different kinds of features may be contained, such as images or simple 1-dimensional real-valued vectors. 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:

$$\arg\min_{\theta,\alpha}\ \mathcal{L}\big(P(\cdot;\ \alpha, \theta)\big) + c(\alpha) \tag{1}$$

where $\mathcal{L}\big(P(\cdot;\ \alpha, \theta)\big) = \mathbb{E}_{i_k, o_k \sim D}[\ell\big(P(i_k;\ \alpha, \theta), o_k\big)]$ and $D(i_k, o_k)$ is an unknown distribution over inputs i_k and labels o_k. The first term $\mathcal{L}$ 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 programmatic 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 $\mathcal{G}$ in some order 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 natural 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 of DSL grammar rules. Specifically, to expand a nonterminal of a partial program architecture, our method relaxes 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** $\alpha_1^{1,0}$ $\alpha_2^{1,0}$ in 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 will choose to expand $\alpha_1^{1,0}$ to **Add** $\alpha_1^{2,0}$ $\alpha_2^{2,0}$ in node 2. Our main idea to train program 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 program architecture weights and θ includes unknown program parameters of all the mixed programs in the graph. The parameters w and θ can be co-optimized with respect to a differentiable loss function over program outputs. Similar to DARTS [8], we train w and θ iteratively as follows:

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

where a validation dataset val and a training dataset $train$ are leveraged to train w and θ respectively.

However, a program derivation graph includes an exponential number of programs. Therefore, it involves a huge set of trainable variables over program architecture weights and unknown program parameters. To curb the large program derivation search space, we introduce node sharing (Sec. 3.1) and iterative 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 encoding the fact that the nonterminals in the two architectures may 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** $\alpha_1^{1,1}$ $\alpha_2^{1,1}$ $\alpha_3^{1,1}$, 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 in 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 candidate

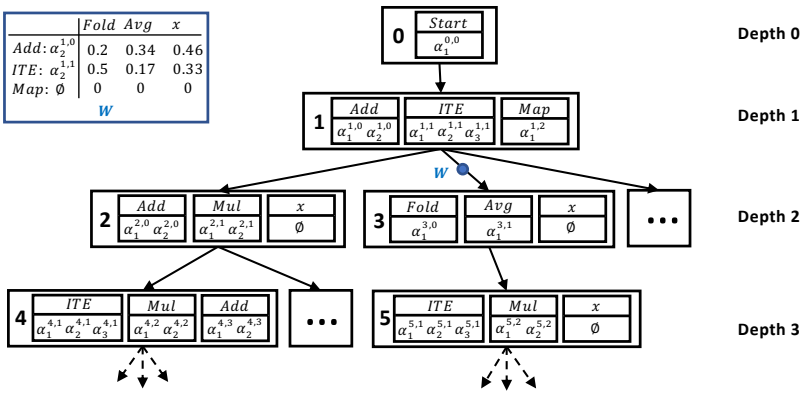

Figure 3: Example search graph for our architecture search method.

architectures on u'. We have $w_e^{(k,k')}$ proportional to the probability of expanding the i-th nonterminal of f_k^u to $f_{k'}^{u'}$ (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:

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

Complexity. Let D be the depth of a program derivation graph, K_{max} be the number of productions rules, and η_{max} be the maximum number of nonterminals in any rules of a context-free 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, the program derivation graph of a grammar with a large K_{max} even hardly fits GPU memory.

3.2 Iterative 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 iteratively and prune away unlikely candidate architectures at the end of each iteration based on their weights. Fig. 4 depicts the iterative graph unfolding procedure.

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 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 function f_k^u and the neural modules. The goal is to optimize the architecture weights and the unknown program parameters using Equation 2. The qualities of candidate partial architectures on each graph node are ranked based on the learned architecture weights.

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 noterminals (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 co-optimizes architecture weights and unknown program parameters and performs top-N preservation on each node based on

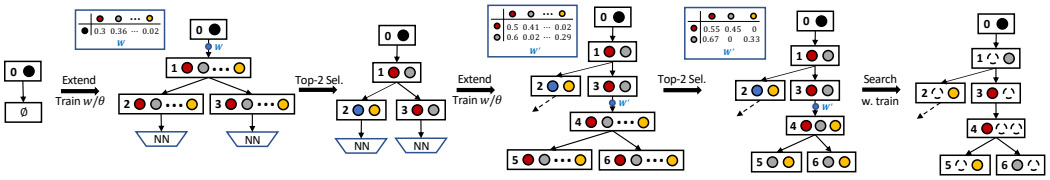

Figure 4: Differentiable program architecture synthesis with iterative graph unfolding.

learned architecture weights. Such a process iterates until the unfolded program derivation graph contains no nonterminals or the maximum search depth is reached. Our differentiable program architecture synthesis method is outlined in 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, recent work discovers that relaxed architecture search methods tend to overfit to certain functions that lead to more rapid gradient descent than others [9] and thus produce unsatisfying performance. As depcited in Fig. 4, our algorithm combines differentiable program architecture synthesis with a variant of the A* graph search algorithm on G to address the problem of biased gradients.

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 G *contains nonterminals* **do**
  Extend G depth d_s deeper w.r.t. $\mathcal{G}$;
  Optimize w and θ in G w.rt. Eq. 2;
  Top-N preservation in G's nodes;
$Q := [G]$;
while $Q \neq \emptyset$ **do**
  $q := \arg\min_{q \in Q} f(q)$;
  $Q := Q \setminus \{q\}$;
  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 $g(q')$, $h(q')$, $f(q')$;
    $Q := Q \cup \{q'\}$;

In the search, Algorithm. 1 maintains a queue Q of program derivation graphs that is initialized to $[G]$. The algorithm pops 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. For A* search, it enumerates each available partial architecture $f_k^u(\alpha_1^{u,k}, \ldots, \alpha_{\eta(f_k^u)}^{u,k})$ on u and computes an f-score for each option of retaining only f_k^u on u, denoted as $q[u/f_k^u]$. We define $f(q[u/f_k^u]) = g(q[u/f_k^u]) + h(q[u/f_k^u])$. 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 [5, 14] for A* search:

$$h(q[u/f_k^u]) = \min_{w,\theta} \mathcal{L}(\mathcal{T}[u/f_k^u], w, \theta)$$

where $\mathcal{T}$ encodes the program derivation graph q itself 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 $q[u/f_k^u]$. For an effective A* search, we fine-tune these trainable variables according to Equation 2 to provide informed feedback on the contribution to program accuracy $\mathcal{L}$ of the choice of retaining f_k^u on node u. After computing the f-score, we add $q[u/f_k^u]$ back to the queue Q sorted based on f-scores. The search algorithm completes when the derivation graph with the least f-score from Q is a well-typed program, i.e. each graph node contains only one valid architecture choice. Our search algorithm based on A* is optimal given the admissible heuristic function h. Among all the programs contained in G, the synthesized program optimally balances structure complexity and accuracy.

4 Experiments

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

4.1 Datasets for Evaluation

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 [15]. 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 [16]. Each trajectory frame is a 53-dimensional feature vector including fly position, velocity, wing movement, etc. We subsample the dataset similar to [5], 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 [17]. 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 [18] contains 152 human pose actions as well as related YouTube Videos subsampled from Kinetics-700 [19]. 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.

4.2 Experiment Setup

To train architecture weights w and unknown program parameters θ in a differentiable program architecture derivation graph, we use the Adam optimizer [20]. We construct two different datasets for optimizing equation 2 by randomly selecting 40% of a training set as D_{val} to train w and using the remained 60% as D_{train} to optimize θ. In Algorithm. 1, we set $N = 2$ for top-N preservation and set graph expansion depth d_s to 2. We also consider F1-scores in the A*-like graph search in Algorithm 1. For evaluation, we compare dPads with the state-of-the-art program learning algorithms A*-NEAR and IDS-BB-NEAR [5]. We only report a comparison with NEAR because it significantly outperforms other program learning methods based on top-down enumeration, Monte-Carlo sampling, Monte- Carlo tree search, and genetic algorithms [5]. All experiments were performed on Intel 2.3-GHz Xeon CPU with 16 cores, equipped with an NVIDIA Quadro RTX 6000 GPU. More experiment setting including learning rates and train epochs are given in the supplementary materials.

4.3 Experiment Results

For a fair comparison with NEAR [5], 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 F1 score, accuracy and execution time for all methods. We also report results achieved using a highly expressive RNN baseline which provides an upper bound on F1-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 F1 scores and using a much less time consumption. dPads also achieves competitive accuracy with NEAR on Crim13. On the Basketball dataset, although dPads achieves a bit higher F1 score, the architectures searched 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 F1 score and higher accuracy.

Table 1: Experiment results. All results are reported as average over five random seeds. Cost of time are set as minute.

	Crim13-sniff			Fly-vs-fly			Bball-ballhandler			Sk152-10 actions		
	F1	Acc.	Time	F1	Acc.	Time	F1	Acc.	Time	F1	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**

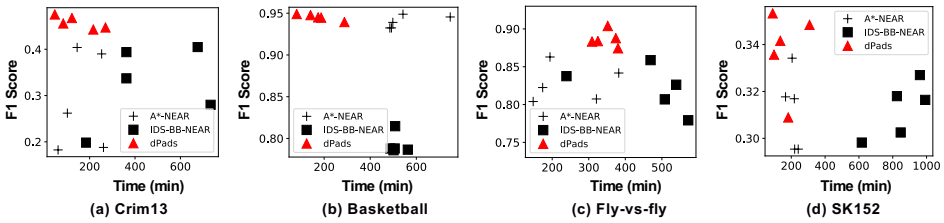

Figure 5: Results on 5 random seeds over Crim13, SK152, Fly-vs-fly and Basketball datasets. x axis refers to cost of time, recorded in minute. y axis indicate F1 score.

We visualize the results of dPads and NEAR in terms of F1 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 F1 scores with less computation and is closer to the RNN baseline.

Although the RNN baseline provides better performance, dPads learns program that are more interpretable. Fig. 6 shows the best programs synthesized by dPads on Crim13 and SK152 (among all the 5 random seeds). The programmatic classifier for Crim13 has a very simple architecture and achieves a high F1 score 0.475 (only 0.006 less than the RNN result). The program is highly human-readable and can be interpreted as follows: 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 F1 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.

Ablation study. We 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) using the Crim13 and SK152 datasets. We set N to be 1, 2, 3 respectively and study how dPads responds to these changes. Table 2 summarizes the average results of F1 score, accuracy and time cost, and the standard deviations of these results. When $N = 1$, dPads synthesizes programs directly from an optimized program derivation graph without further exploiting a variant of A* search. There is a significant decrease in time consumption compared with $N = 2$, however, dPads in this condition achieves less F1 scores and the results have a high variance, which suggests that architecture weights learned using only differentiable search overfit to sub-optimal programs. dPads gets similar F1 scores on both the two datasets when setting $N = 3$ compared to $N = 2$ but consumes more time. This confirms that leaving a large search space to the discrete graph search algorithm is time-expensive.

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 [4, 21–28]. There exist efforts that extend this line of research to program synthesis from noisy data [1, 2, 29–33]. These approaches either require a detailed hand-written program template or simply enumerate the discrete 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 [34, 5] that enumerate the space of program architectures prioritizing search directions with feedback from machine learning models.

```
Map(
 Multiply(
  PositionSelect_θ1(x_t),DistanceSelect_θ2(x_t))) x
```

(a) Crim13

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

(b) SK152

Figure 6: Synthesized Programs for the Crim13 and SK152 Datasets.

Table 2: Results of Ablation Study.

	N	**dPads**				
		F1	Acc.	Time	Std. F1	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
Sk152-10 actions	1	.272	.627	50.85	.020	.024
	2	.458	.812	147.87	.017	.017
	3	.450	.811	441.12	.011	.010

Specifically, Lee et al. [34] uses a probabilistic model (trained from a synthesis problem corpus) to guide an A* search over discrete program syntax trees and NEAR [5] uses neural networks to approximate missing expressions in a partial program whose F1 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 conduct program architecture search in a continuous relaxation of the discrete space of language grammar rules and only apply graph search algorithms in a much reduced search space after relaxation to address the gradient bias problem [9] in architecture search tasks.

Differentiable Architecture Search. Neural architecture search has attracted much interest as a promising approach to automate deep learning tasks [35–38]. Various search strategies have been developed including reinforcement learning and evuoluationary search. Particularly, our program architecture synthesis algorithm is inspired by DARTS [8]. 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. P-DARTs [9] and I-DARTs [10] further improve architecture search efficiency and accuracy. 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 expressions in a program architecture. By relaxing the discrete search space of language grammar rules with iterative unfolding of program derivation graphs and node sharing, 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 a context-free 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 iterative graph unfolding and node sharing, 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. While these benefits can usefully direct the learning process, the programming biases in a DSL may leave opportunities to open up a new line of 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.

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 M Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. *arXiv preprint arXiv:1907.05431*, 2019.

[3] 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. URL `https://proceedings.neurips.cc/paper/2018/hash/edc27f139c3b4e4bb29d1cdbc45663f9-Abstract.html`.

[4] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. *arXiv preprint arXiv:1906.04604*, 2019.

[5] Ameesh Shah, Eric Zhan, Jennifer J Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. Learning differentiable programs with admissible neural heuristics. *arXiv preprint arXiv:2007.12101*, 2020.

[6] Judea Pearl. Heuristics: intelligent search strategies for computer problem solving. 1984.

[7] 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. URL `https://proceedings.neurips.cc/paper/2013/hash/8ce6790cc6a94e65f17f908f462fae85-Abstract.html`.

[8] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[9] 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.

[10] 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.

[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. URL `https://doi.org/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] 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. URL `https://doi.org/10.1016/0004-3702(74)90014-9`.

[15] 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.

[16] 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`.

[17] 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.

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

[19] 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.

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

[21] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[22] 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.

[23] 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. URL `https://openreview.net/forum?id=rJ0JwFcex`.

[24] 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. URL `https://openreview.net/forum?id=HkfXMz-Ab`.

[25] 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. URL `https://openreview.net/forum?id=H1Xw62kRZ`.

[26] Amit Zohar and Lior Wolf. Automatic program synthesis of long programs with a learned garbage collector. *arXiv preprint arXiv:1809.04682*, 2018.

[27] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

[28] 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. URL `http://proceedings.mlr.press/v80/ganin18a.html`.

[29] 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. URL `https://proceedings.neurips.cc/paper/2018/hash/6788076842014c83cedadbe6b0ba0314-Abstract.html`.

[30] 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. URL `https://proceedings.neurips.cc/paper/2015/hash/b73dfe25b4b8714c029b37a6ad3006fa-Abstract.html`.

[31] 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.

[32] 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. URL `http://proceedings.mlr.press/v70/gaunt17a.html`.

[33] 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. URL `http://arxiv.org/abs/1608.04428`.

[34] 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. URL `https://doi.org/10.1145/3192366.3192410`.

[35] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[36] 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.

[37] 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.

[38] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

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]