RICE UNIVERSITY

Differentiable Program Learning with an Admissible Neural Heuristic

By

Ameesh Shah

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE

Swarat Chaudhuri
Swarat Chaudhuri (Aug 6, 2020 10:00 CDT)

Swarat Chaudhuri (Director)

Associate Professor of Computer Science,
University of Texas at Austin

Christopher Jermaine
Christopher Jermaine (Aug 10, 2020 15:47 CDT)

Chris Jermaine (Chair)

Professor of Computer Science, Rice
University

[illegible]

Anshumali Shrivastava

Assistant Professor of Electrical Engineering,
Computer Science, and Statistics, Rice
University

Konstantinos Mamouras
Konstantinos Mamouras (Aug 3, 2020 10:22 CDT)

Konstantinos Mamouras

Assistant Professor of Computer Science,
Rice University

HOUSTON, TEXAS

August 2020

RICE UNIVERSITY

Differentiable Program Learning with an Admissible Neural Heuristic

by

Ameesh Shah

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Chris Jermaine, Chair
Professor of Computer Science

Swarat Chaudhuri, Director
Associate Professor of Computer Science

Anshumali Shrivastava
Assistant Professor of Electrical
Engineering, Computer Science, and
Statistics

Konstantinos Mamouras
Assistant Professor of Computer Science

Houston, Texas

August, 2020

ABSTRACT

Differentiable Program Learning with an Admissible Neural Heuristic

by

Ameesh Shah

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

Acknowledgments

First and foremost, I would like to thank my advisor, Prof. Swarat Chaudhuri, for his immense support, mentorship, and advice over the course of my undergraduate and master's programs. Swarat is the reason I am pursuing further education and I am extremely thankful to have him as a role model. I'd also like to thank Prof. Ankit Patel for his mentorship during my undergraduate career. Prof. Richard Baraniuk and Prof. Chris Jermaine were also extremely valuable mentors and I'd like to thank them for their help.

Beyond Rice, I want to thank Prof. Yisong Yue at CalTech and his students, Eric Zhan and Jennifer Sun, for their tremendous help in this work. I'd also like to thank Abhinav Verma for his mentorship. Lastly, I'd like to thank Spencer Chang and Alex Hwang for their continuous feedback and advice throughout my research.

This thesis is directly adapted from a paper titled *Learning Differentiable Programs with Admissible Neural Heuristics* that I led and worked on with my co-authors, Eric Zhan, Jennifer Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. This work is available as a preprint on *arXiv*.

Contents

Illustrations

Tables

Chapter 1

Introduction

In the past decade, the entire paradigm of computational research has exploded with the advent of usable and effective machine learning techniques for scientists. State of the art Machine learning tools, such as deep learning models, have allowed for state of the art performances in a variety of domains, which has enabled us to reach new heights in tasks like translation, object recognition, and robotics [1]. In research labs across both industry and academia, researchers are continuing to create more and more complex models that can produce and imitate intelligent behaviors. As a result, the number of parameters that these models contain are often on the order of millions [2].

With modern machine learning models becoming so computationally complex, a number of critical issues are now surfacing that have yet to be addressed by the research community at large. One of the foremost issues in this space is that of *explainability* and *interpretability*. As a concrete example of how modern machine learning models can operate in often inexplicable ways, consider the work done in [3], which identifies that an adversarial agent can fool highly performant deep learning models, causing these models to fail on seemingly obvious examples in a highly uninterpretable and confusing way. These sorts of failures stem from the notion of deep learning models as *black boxes*; that is, models that do not have a human-interpretable methodology that underlines their decision-making policy.

The relative opaqueness and black-box nature of deep learning models becomes an increasingly important issue in environments where trustworthiness and interpretability are critical. Environments such as autonomous vehicle operation or precision medicine are domains where machine learning cannot be faithfully deployed without a thorough understanding of how the models work and where failures may occur. This is also true in fields of research within the natural sciences, such as behavioral classification in biology, which is a particular problem that will be addressed in this thesis. For machine learning models that can accurately perform a task like behavior classification to be accepted by a group of explanation-driven scientists, the methodology of the model requires a certain level of interpretability.

A number of recent efforts have aimed to create new interpretations of deep learning models [4], reduce the complexity of such models [2], or formally verify these models [5]. These approaches, while important steps in demystifying the black-box nature of deep learning models, do not propose an end-to-end interpretable solution for canonical machine learning tasks.

1.1 Program Learning

Recently, an emerging body of work has proposed the use of *program synthesis* as an approach to interpretable end-to-end machine learning. The methods here learn functions represented as programs in symbolic, domain-specific programming languages languages (DSLs) [6, 7, 8, 9, 10]. Such programs are more interpretable than neural networks and more expressive than linear models and decision trees. As such, methods in this space are appropriate for domains such as the natural sciences, where solutions to nontrivial learning problems must provide a high level of interpretability. The constraints that the DSL imposes through the functions and primitives that constitute the programming language can also serve as a form of regularization and

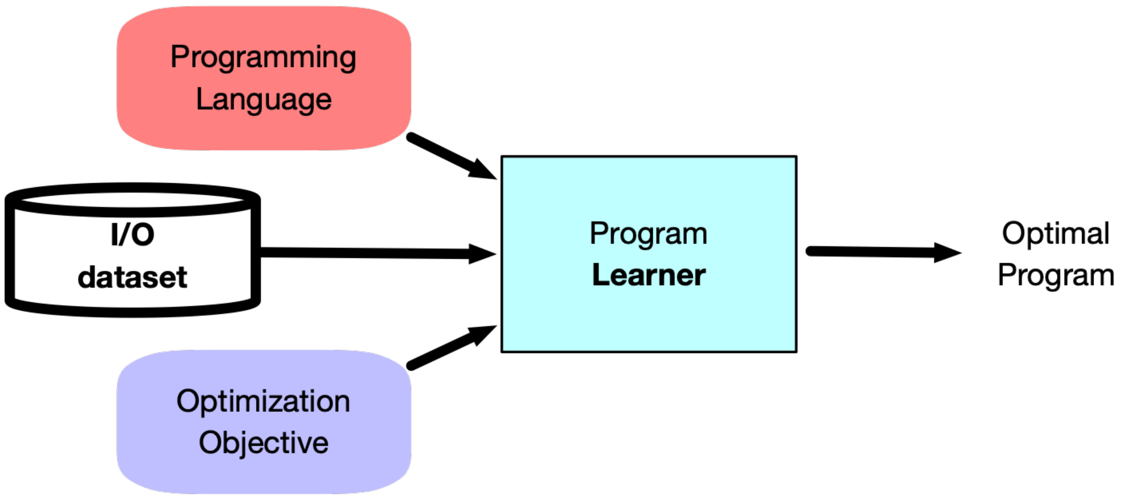

Figure 1.1 : A visual representation of the program learning paradigm.

allow for more reliable learning. In figure 1.1, we provide program learning framework diagrammatically. Recent work has shown that learned programmatic solutions can be more data-efficient, interpretable, and robust than state-of-the-art models. In this thesis, interpretability will remain the focus: we will aim to learn programs that are performant with respect to the original learning task, while remaining appropriately performant and interpretable.

In particular, we study how to learn *differentiable* programs, which use structured, symbolic primitives to compose a set of parameterized, differentiable modules. Unlike many traditional programming languages, that use the standard semantics of control-flow conditionality, differentiable programs define a space of programs that can be executed in an end-to-end differentiable manner. These differentiable programs have recently attracted much interest due to their ability to leverage the complementary advantages of programming language abstractions and differentiable learning. For example, recent work has used such programs to compactly describe modular neural networks that operate over rich, recursive data types [8]. The work done in [8] shows that complex tasks can be solved by using a combination of standard functional program semantics and deep learning models defined as library functions. By ensuring that the programs that call such deep learning functions are themselves differentiable, the entire resulting programs remains differentiable as well.

To learn a differentiable program, one needs to induce the program's "architecture" while simultaneously optimizing the parameters of the program's modules. This co-design task is difficult because the space of architectures is combinatorial and explodes rapidly. This is a central challenge for researchers in program synthesis: as programs scale in complexity, how can large swaths of the program space be explored and exploited efficiently? Prior work has approached this challenge using methods ranging from greedy enumeration, Monte Carlo sampling, and evolutionary algorithms [9, 8, 11]. These methods, while effective in a number of program synthesis settings, still face the challenge of scale in the search for a proper program architecture. In the setting of program learning, in particular, we identify that more fully exploiting the structure of the underlying combinatorial search problem will allow us to accelerate search through a given program space.

More specifically, in this thesis, we show that the differentiability of programs opens up a new line of attack on this search problem. A standard strategy for combinatorial optimization, which we pursue in this paper, is to exploit (ideally fairly tight) continuous relaxations of the search space. Optimization in the relaxed space is typically easier and can efficiently guide search algorithms towards good or optimal solutions. In our case of program learning, we propose to use various classes of neural networks as relaxations of instantiated partial program architectures. In order to pursue this line of attack, we frame our problem as searching through a graph, in which nodes encode program architectures with missing expressions, and paths encode top-down program derivations. For each partial architecture u encountered during this search, the relaxation amounts to substituting the unknown part of u with a type-corresponding neural network with free parameters. Because the greater space of programs are end-to-end differentiable, this network can be trained on the problem's end-to-end loss. If the space of neural networks is an (approximate) proper relaxation of the space of programs (and training identifies a near-optimum neural network), then we reason that the resulting training loss for the relaxation can be viewed as an admissible heuristic. So long as this training comes close to a global optimum, the heuristic $h(u)$ comes close to underestimating the cost to go at u. This means that the heuristic h is (approximately) *admissible*.

We instantiate our approach, called NEAR (abbreviation for ***Ne****ural* ***A****dmissible* ***R****elaxation*), on top of two informed search algorithms: A* and an iteratively deepened depth-first search that uses a heuristic to direct branching as well as branch-and-bound pruning (IDS-BB). These informed search algorithms are used frequently in graph-search literature and have provable guarantees on completeness and optimality under the condition of an admissible heuristic, which we expand upon in the context of our approximately admissible heuristic. We evaluate these search algorithms in the task of learning programmatic classifiers in three behavior classification applications for sequences of agent actions, in two animal biology applications and a sports analytics application. We show that these algorithms substantially outperform state-of-the-art methods for program learning in terms of both performance accuracy and efficiency. More importantly, we show that our strategies are able to learn classifier programs that bear natural interpretations and are close to commonly used neural models in accuracy.

So far as we know, this is the first approach to exploit the differentiability of a programming language in program synthesis.

1.2 Structure of this Thesis and Contributions

This thesis will be structured as follows:

In section 1, the overall field of machine learning is surveyed at a high level, and interpretability and explainability are identified as key issues for researchers to tackle. The concept of *program learning*, or program synthesis for machine learning, is introduced, and key challenges with program learning are discussed, primarily the challenge of exploring a broad space of programs when such a space explodes combinatorially with the scale of increasingly complex programs. We propose Near as a solution and briefly outline the important technical details of the strategy. Lastly, we establish the context of how our strategy will be used and mention the experimental domains that we will use to show the effectiveness of Near in comparison to state-of-the-art program learning and machine learning strategies.

In section 2, we establish the context in which this thesis work is rooted. The problem of program synthesis with a domain-specific language is defined and specific works that demonstrate the effectiveness of this methodology are offered. We follow by referencing work on neural program induction, which is captured in our admissible heuristic based on neural relaxations of programs. We identify key works and explain how this thesis distinguishes itself from prior efforts. Lastly, we discuss works dealing with discrete structure search using relaxations, which is closely related to the problem of program learning. The challenges that prevent the works identified in this section from being more general-purpose are mentioned.

In section 3, the problem of *Programmatic Sequence Classification* is formally defined in both the contexts of canonical machine learning and program synthesis. A bilevel optimization problem is introduced to balance a learned program's architecture and parameters, balancing performance and accuracy with parsimony and simplicity.

Upon defining the problem statement, we offer a template functional programming language that maintains end-to-end differentiability for any instantiated program in the language. We detail the provided functions in the language, as well as the ability for a user to provide their own functions to incorporate further domain-specific knowledge for a given task.

In section 4, the problem of program learning (more specifically, programmatic sequence classification) is formulated as an instance of top-down graph search. We explain how existing methods may lack effectiveness in this setting due to the combinatorial explosion of program space as we scale up in program complexity. We offer a solution in the form of a heuristic, titled Near, that utilizes continuous relaxations of program space modeled by neural networks. We explain how Near can be used in our graph search formulation, and instantiate two informed search methods that can utilize our heuristic. Additionally, we discuss the *approximate* admissibility of Near, and we outline proofs on the resulting approximate optimality of the program search process as a consequence of using an approximately admissible heuristic.

In section 5, we concretize the usage of Near-based strategies by providing three experimental domains for sequence classification tasks. Programs are learned to classify behaviors in these domains, two of which are rooted in the natural sciences and one of which is rooted in sports analytics. We explain the experimental setup for each domain, and demonstrate that the informed search strategies proposed in this work yield more accurate learned programs than existing state-of-the-art program learning strategies. We also show drastic improvement in the efficiency at which these learned programs are found. In comparison to deep neural networks, we find that our learned programs achieve close competitive accuracy. We perform auxiliary analyses to show the tradeoff between program simplicity and accuracy using our designed optimization function.

Lastly, in section 6, we discuss the overall findings from this thesis and evaluate directions for future work. We close by explaining the broader impact of program learning, and how a greater focus on interpretable machine learning can advance highly important areas of science and research.

Chapter 2

Related Work

2.1 DSL-based Program Synthesis

The well-defined problem of Program Synthesis is traditionally defined as the process automatically searching for a program in a given programming language that satisfies a given formal specification, most often a logical constraint that dictates the input/output behavior of a program [12]. The programming languages which make up the space of programs that a synthesizer will search through are denoted as Domain-Specific Languages, or *DSLs*. Domain-Specific Languages are widely used across problems in program synthesis due to the added benefits of both *inductive bias* as well as *reduced program space complexity*. Consider the task proposed in [13], where string-manipulations problems are approached from a program synthesis standpoint. If a general-purpose programming language was used as the backdrop for this synthesis problem, a number of issues would immediately arise: First, given that the problem is focused specifically on string transformations, other primitive types in a general-purpose language would be unnecessary. To include these primitives as options would make the space of programs larger without any possibility for that enlargement leading to additional solutions to the problem. This same issue extends to any functions or methods that do not operate over strings. The second problem is that of domain specificity: if a general-purpose programming language is provided with little additional functionality, any inductive bias towards the domain of problems must be automatically learned by the synthesizer.

Now, let's consider the Domain-Specific Language presented in [13]. This DSL directly addresses the previous two issues with a general purpose language. In order to deal with the specificity of the problem at hand, the language operates only over string type primitives, and only offers limited functionality to operate over these strings. This limited operation allows for the definition of *subtypes* within the greater category of strings, such as alphanumeric characters or other symbols. With an expressive string-based type system, the language also allows for the incorporation of inductive bias by way of string-transformation functions. These functions, such as the creation of regular expressions (regexes) or changing the case of letters, provide useful domain-specific use that would otherwise be challenging to implement with a general purpose language.

There is a large body of research on synthesis of programs from DSLs. In most of these methods, the goal is to find a program that satisfies a hard logical constraint. Foundational work such as [14] defines this problem of logical constraint solving via program synthesis by formally outlining the conditions for logical satisfaction, the solution space of programs, and methods of synthesis. This work was followed upon by [15], where the program synthesis paradigm was extended to use input-output (IO) examples that could guide candidate solutions that did (or did not) satisfy the existing set of examples. The groundwork laid by [14] and [15] allowed for a breadth of follow-up work: in [16], *program sketches* are introduced as a way to allow for program synthesis to learn low-level functionality in a given high-level programmatic structure. Other work, such as in [17, 18], exploit knowledge about a given space of programs, such as type signatures, to reduce the complexity of the program search space at a given step during the synthesis process. In this work, we will further exploit the structure of the program space, in a continuous setting, in order to accelerate program search.

To further improve upon methods of program synthesis, many recent methods in this area have used statistical models to guide the synthesis process. Some methods, such as [19, 20, 6] use a model learned over existing program synthesis examples to guide a search through program space. Others, such as [21, 7, 22, 13, 23], use learned models over a dataset of program synthesis instances as the primary means of synthesis. In particular, Lee et al. [24] use a probabilistic model to guide an A* search over programs. Most of these models (including the one in Lee et al. [24]) are trained using corpora of synthesis problems and corresponding solutions, which are not available in our setting. There is also category of methods based on reinforcement learning (RL) [23, 25]. Unlike NEAR, these methods do not directly exploit the structure of the search space. Combining them with our approach presents a number of opportunities for future work, which will be addressed in section 6.

Beyond synthesizing programs to solve logical constraint problems, a number of recent work focuses on learning programs that optimize a quantitative objective [26, 27, 9, 7, 28, 10]. We are aware of only one program synthesis effort that explicitly targets the synthesis of differentiable programs [28]. However, unlike in NEAR, the combinatorial search in that work neither receives neural guidance nor does it exploit the programs' differentiability.

2.2 Neural Program Induction

Just as NEAT induces programs to complete partial program architectures, similar efforts have tackled *neural program induction*, where a neural network is trained to generate a program's outputs by learning a latent representation of the target program within its parameters. The literature on *neural program induction* (NPI) [29, 30, 31, 32] develops methods to learn neural networks that can perform procedural (program-like) tasks, typically using architectures augmented with differentiable memory. These efforts have shown that neural program induction can be more data-efficient and robust than purely neural methods in both discrete and continuous settings [33, 34] compared neural program induction to neural program synthesis on a variety of discrete tasks [13, 19, 35]. Others have used neural networks augmented with differentiable modules to induce programmatic behavior [29, 30, 31, 32]. Our approach differs from these methods in that its final output is a symbolic program. However, since our heuristic, NEAR, involves using neural approximations of programs, our work can be seen as repeatedly performing NPI as the program is being produced. While we have so far used classical feedforward and recurrent architectures to implement our neural heuristics, future work could use richer models from the NPI literature to this end.

2.3 Discrete Structure Search using Relaxations.

The problem of program learning, framed as a search through program space, bears similarities with the problems of searching over neural architectures and the structure of graphical models. The problem of discrete structure search has become increasingly prominent in the current age of deep learning, where it is highly expensive to manually design an optimal neural architecture for a deep learning model. In early deep learning works, these neural architectures were not optimized, and design was done on largely a trial-and-error basis. However, recent work has emerged to *learn* the precise connections and architectures of these neural models. This work is rooted in discrete structure search that precedes neural learning methods.

The key difference between these efforts and ours is that the design space in our problem is much richer, making the methods in prior work difficult to apply. Some prior works have used various types of relaxations to solve these problems [36, 37, 38, 39, 40]. Specifically, the A* lasso approach for learning sparse Bayesian networks [40] uses a dense network to construct admissible heuristics [36, 37]. These works on structure search have also relied upon using a single function class that encapsulates the optimal solution; for example, composition of softmaxes over all possible candidate operations between a fixed set of nodes [36]. In particular, Darts [36] uses a composition of softmaxes over all possible candidate operations between a fixed set of nodes that constitute a neural architecture, and the heuristics in the A* lasso method come from a single, simple function class. This prior work draws a number of similarities to our problem, in that both aim to optimize over bilevel optimization problems, where both a discrete structure and continuous parameters within that structure must be learned in conjunction with one another. Another similarity between the Darts work and our efforts lies in that both works train the bilevel optimization problem end-to-end, opting to train both levels of optimization simulataneously as opposed to fixing a level and then training the other level with respect to that fixed solution. However, a number of differences distinguish our work from these previous endeavors. In our setting of program learning, there is no fixed bound on the number of expressions in a program. Moreover, different sets of operations can be available at different points of synthesis, and the input and output type of the heuristic (and therefore, its architecture) can vary based on the part of the program derived so far. As an example, there are a different number of functions (and a different set of functions) available for choosing an expression that operates over sequences, as opposed to the options when choosing an expression that operates over atomic values. This limitation in being able to standardize the search space at an arbitrary point during synthesis severely limits the types of relaxations we can apply. This increase in complexity also prevents an RL-based search such as [38], since assumptions such as fixed-size action space are not guaranteed.

Another distinction is that our approach leverages multiple (neural) function classes to build our relaxations. Prior efforts in differentiable model architecture search, such as neural architecture search, use a single parameterized function class as the learning model for which an optimal solution represents the optimal architecture structure [36, 37, 40].

Chapter 3

Problem Statement: Programmatic Sequence Classification

3.1 Definitions

In this work, we formulate the problem of program learning as a two-pronged optimization problem. More specifically, in our domain-specific language (DSL), we view a program as a pair (α, θ), where α is a discrete *(program) architecture* and θ is a vector of real-valued parameters. The *program architecture* represents the syntax of the program itself: expressions in our domain-specific language are composed, and along with their corresponding parameters, a fully instantiated program is formed. The architecture, which we denote as α, is generated using a *context-free grammar*.

The usage of a context-free grammar is a standard practice in the program synthesis literature with respect to the definition of domain-specific programming languages [41]. The context-free grammar consists of a set of rules $X \rightarrow \sigma_1 \ldots \sigma_k$, where X is a *nonterminal* and $\sigma_1, \ldots, \sigma_k$ are either nonterminals or *terminals*. A nonterminal stands for a missing subexpression; a terminal is a symbol that can actually appear in a program's code. At a given point during synthesis, a nonterminal that exists in a production can be *expanded*, or replaced, by any rule that replaces the nonterminal (on the left-hand side of a rule) with a sequence of terminal and nonterminal symbols (that appear on the right-hand side of a rule.) The grammar starts with an initial nonterminal, then iteratively applies the rules to produce a series of *partial architectures*:

sentences made from one or more nonterminals and zero or more terminals. The process continues until there are no nonterminals left. These finalized programs can then be executed with respect to inputs to produce the set of corresponding outputs.

We refer to such fully parameterized program architectures as *complete architectures*. The *semantics* of the architecture α is given by a function $[\![\alpha]\!](x, \theta)$, defined by rules that are fixed for the DSL. We require this function to be differentiable in θ.

Under this definition of the program learning problem, we aim to find an $((\alpha, \theta))$ that will perform optimally on some learning-based objective (often framed in terms of a loss function.) Moreso, however, we aim for these learned programs to be *simple*, that is, for learned programs to not exceed a level of complexity that prevents them from being human-interpretable. In order to account for this simplicity, we define a *structural cost* for program architectures: Let each rule r in the DSL grammar have a non-negative real cost $s(r)$. The structural cost of α is $s(\alpha) = \sum_{r \in \mathcal{R}(\alpha)} s(r)$, where $\mathcal{R}(\alpha)$ is the multiset of rules used to create α. By introducing this structural cost function, we allow for a user to define their own standards of complexity, and we impose this simplicity on the program learner as an additional factor that learned programs must be optimized with respect to.

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

$$(\alpha^*, \theta^*) = \arg\min_{(\alpha,\theta)}(s(\alpha) + \zeta(\alpha, \theta)). \tag{3.1}$$

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

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

Figure 3.1 : Grammar of DSL for sequence classification. Here, x, c, $\oplus$, and $\oplus_\theta$ represent inputs, constants, basic algebraic operations, and parameterized library functions, respectively. $\textbf{sel}_S$ returns a vector consisting of a subset S of the dimensions of an input x.

3.2 Program Learning for Sequence Classification.

Although the paradigm of program learning we introduce is broadly applicable across machine learning domains, we specifically study it in the sequence classification context, using the canonical definiton of one-to-many or many-to-many sequence classification for machine learning, as specified in [42]. In figure 3.1, we present our context-free domain-specific language for *programmatic sequence classification* in the standard Backus-Naur form [43].. Like many others DSLs for program synthesis [17, 19, 28], our DSL is purely functional. Our usage of a functional language allows for the end-to-end differentiability of programs, due to the fact that the composition of differentiable functions remains differentiable (provided that every function defined in our DSL is also differentiable). Programs in this DSL operate over two data types: real vectors and sequences of real vectors. We assume a simple type system that makes sure that these types are used consistently. We now elaborate on each function specified in our language and reason about the differentiability of each:

- **Map.** The *Map* function is a higher-order function that follows its standard semantics in functional languages, as outlined in [28]. *Map* is, by definition, differentiable if the function passed as input to *Map* is differentiable as well.

- **Fold.** The *Fold* function, like *Map*, is also a higher-order function that follows the semantics outlined in [28]. Note that our provided *Fold* function does not take an accumulator $Atom$ as input; for ease of interpretability, we initialize the accumulator in *Fold* to the zero vector at the beginning of execution on an input. Like *Map*, *Fold* is differentiable if and only if the function that is passed to it as input is differentiable as well.

- **MapPrefixes.** The *MapPrefixes* function combines the *Map* function with the a function that converts an ordered list to a list of ordered subsets of that list. This prefixing function alone is not differentiable; thus, we introduce *MapPrefixes* as a differentiable approximation of the original *Prefix* function composed with *Map*. In *MapPrefixes*, we apply the function passed as argument to the subset with a single element, then again to the subset with two elements, and so on, in a recurrent fashion.

- **IfThenElse (ITE).** The IfThenElse function defined in our language follows the traditional semantics of the if-then-else programming construct. This construct is not inherently differentiable, and so we introduce a smoothed approximation of conditional logic. When we compute a value in our "If" statement, that value is then smoothed and bounded within $[0, 1]$ by the sigmoid function σ. The result of our If-Then-Else is then a linear combination of the values computed in the "Then" and "Else" clauses based on our bounded and smoothed "If" value.

$$\textbf{map}(\textbf{if } \textit{DistAffine}_{[.0217];-.2785}(x)$$

$$\textbf{then } \textit{AccAffine}_{[-.0007,.0055,.0051,-.0025];3.7426}(x) \textbf{ else } \textit{DistAffine}_{[-.2143];1.822)}(x)$$

Figure 3.2 : Synthesized program classifying a "sniff" action between two mice in the CRIM13 dataset.

More specifically, we define this function as:

$$[\![\textbf{if } \alpha_1 > 0 \textbf{ then } \alpha_2 \textbf{ else } \alpha_3]\!](x, (\theta_1, \theta_2, \theta_3))$$
$$= \sigma(\beta \cdot [\![\alpha_1]\!](x, \theta_1)) \cdot [\![\alpha_2]\!](x, \theta_2) + (1 - \sigma(\beta \cdot [\![\alpha_1]\!](x, \theta_1))) \cdot [\![\alpha_3]\!](x, \theta_3). \tag{3.2}$$

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

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

In order to indeed demonstrate that our DSL can produce interpretable programs, figures 3.2 and 3.3 show two programs synthesized by our learning procedure using our DSL with libraries of domain-specific affine transformations. Both programs offer an interpretation in their respecitve domains, while offering respectable performance against an RNN baseline. We provide interpretations for each program as follows: In figure 3.2, *DistAffine* and *AccAffine* are functions that first select the parts of the

$\mathbf{map}(\mathbf{multiply}($

$\quad\mathbf{add}(\mathit{OffenseAffine}(x), \mathit{BallAffine}(x)), \mathbf{add}(\mathit{OffenseAffine}(x), \mathit{BallAffine}(x)))$

Figure 3.3 : Synthesized program classifying the ballhandler for basketball.

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

In 3.3, *OffenseAffine()* and *BallAffine()* are parameterized affine transformations over the XY-coordinates of the offensive players and the ball (see the appendix for full parameters). **multiply** and **add** are computed element-wise. The program structure can be interpreted as computing the Euclidean norm/distance between the offensive players and the ball and suggests that this quantity can be important for determining the ballhandler. On a set of learned parameters (not shown), this program achieves an accuracy of 0.905 (vs. 0.945 for an RNN baseline).

Chapter 4

Accelerated Program Learning with Near

In order to synthesize complex programs such as those given in figures 3.2 and 3.3 in an efficient manner, we propose an accelerated method of differentiable program search.

Existing methods of search through a given program space that are compatible with the differentiable program learning problem rely on simple yet often-effective techniques for finding performant and parsimonious programs. *Top-Down Enumeration* relies on synthesizing programs in increasing order of complexity, evaluating each in order to find a minimally complex program that satisfies a desired threshold of performance [9]. However, if the solution program is nontrivially complex, exhaustive top-down enumeration will scale poorly, requiring every less-complex program to be synthesized and evaluated in order before reaching the solution. Another commonly used top-down synthesis method, Monte-Carlo search, involves sampling expressions based on randomly *rolling-out*, or expanding, a partial program architecture until a fully complete program is reached. This program is done an arbitrary number of times, and the performances of those rolled-out programs are used in combination to evaluate the partial program in question. In cases where rolling-out and evaluating partial programs is expensive, or when a partial program may have a large number of possible complete descendants, Monte-Carlo search becomes inefficient.

In contrast to the previously described methods, there also exist a class of *bottom-up* synthesis methods that start with a number of fully synthesized programs, and iteratively perturbs and modifies these initial programs until a large swath of the program search space has been explored. One of the most effective and popular bottom-up methods involves *genetic algorithms*, which select top-performing programs from an initial population, cross-over, and mutate these programs to produce new generations of diverse programs. Genetic algorithms have been used in prior differentiable program synthesis work [28]; however, these methods rely on a diverse initial population of programs, which is challenging to guarantee when the defined program space scales in size and complexity.

In this section, we propose a novel method of program learning that exploits the differentiability of program architectures during search by using neural relaxations in partial programs. We first describe the program learning search problem in the context of graph search, and then propose our method as a neural heuristic function that accelerates the program search process. We reason that our proposed heuristic is *approximately* admissible, and provide theoretical bounds on the optimality of an approximately admissible heuristic. We describe two informed graph search algorithms that utilize the heuristic to improve the efficiency of the overall program search process.

4.1 Program Learning as Graph Search

We first formulate our program learning problem as a form of graph search. Like many other efforts on program synthesis [17, 44, 28], The search derives program architectures in a top-down fashion: it begins with the *empty* architecture, generates a series of partial architectures following the DSL grammar, and terminates when a complete architecture is derived.

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

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

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

- Edges are directed and capture single-step applications of rules of the DSL. Edges can be divided into: (i) *internal edges* (u, u') between partial architectures u and u', and (ii) *goal edges* (u, α) between partial architecture u and complete architecture α. An internal edge (u, u') exists if one can obtain u' by substituting a nonterminal in u following a rule of the DSL. A goal edge (u, α) exists if we can complete u into α by applying a rule of the DSL.

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

A path in the graph $\mathcal{G}$ is defined as usual, as a sequence of nodes $u_1, \ldots, u_k$ such that there is an edge (u_i, u_{i+1}) for each $i \in \{1, \ldots, k-1\}$. The cost of a path is the sum of the costs of these edges. Our goal is to discover a least-cost path from the source u_0 to some goal node α^*. Given that path costs represent both the structural and objective costs of a learned program, by this graph's construction, α^* is an optimal solution to our learning problem in Eq. (3.1).

4.2 Neural Relaxations as Admissible Heuristics

The main challenge in our search problem lies mainly in that *partial programs* and the intermediate nodes that represent them cannot be evaluated with respect to

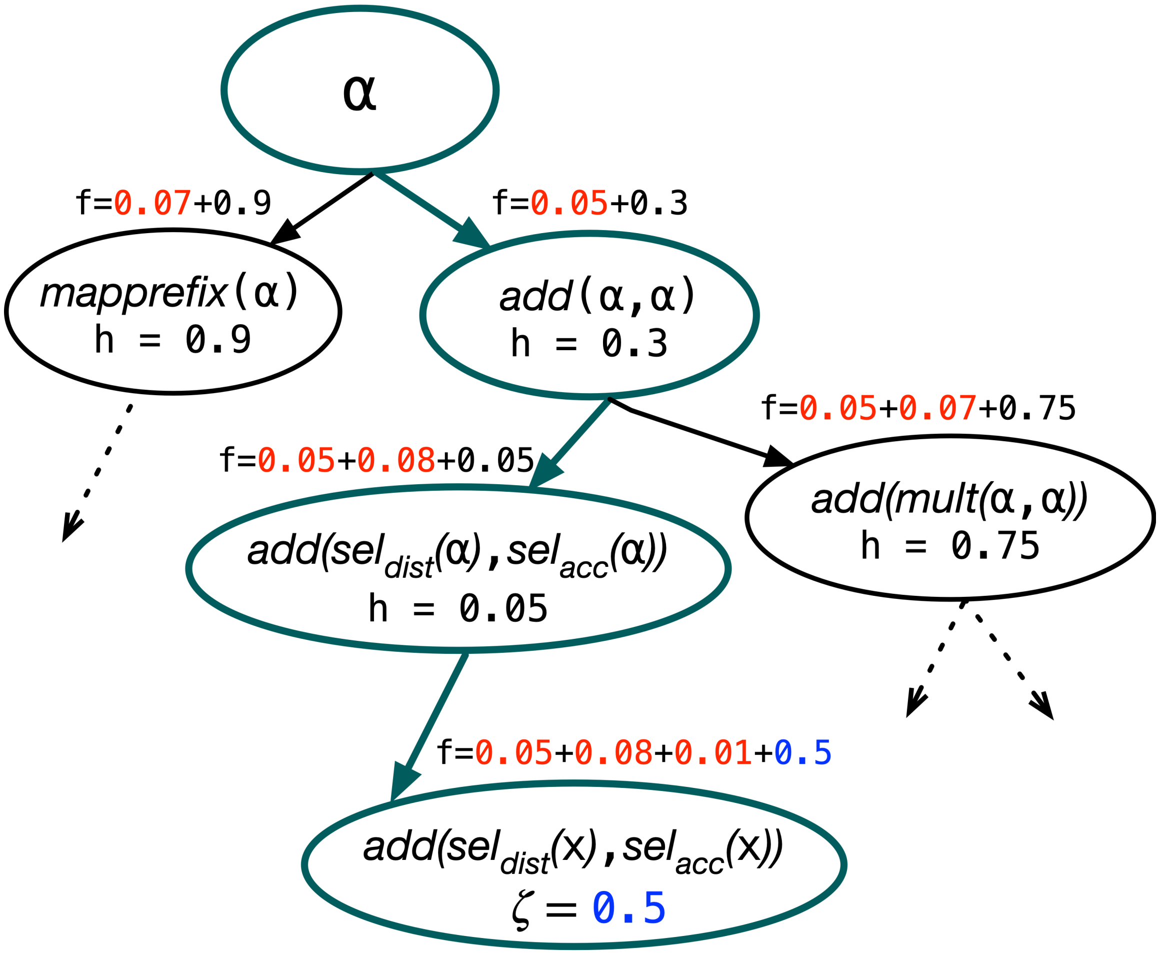

Figure 4.1 : An example of program learning formulated as graph search. Structural costs are in red, heuristic values are in black, and prediction errors ζ are in blue. O refers to a nonterminal in a partial architecture, and the path to a goal node returned by A*-Near search is in teal.

performance. That is, the rich information that our goal edges contain is only accessible when a path has been explored until the end, making uninformed searches in this program space intractable. A heuristic function $h(u)$ that can predict the value of choices made at nodes u encountered early in the search can help with this difficulty. If such a heuristic is *admissible* — i.e., underestimates the cost-to-go along the eventual minimum-cost path through a certain node — it enables the use of informed search strategies such as A* and branch-and-bound while guaranteeing the optimality of the minimum-cost path found. Our Near approach (abbreviation for ***Ne****ural* ***A****dmissible* ***R****elaxation*) uses neural relaxations of program space to construct a heuristic that is *approximately* admissible, or some ϵ-close to being admissible.

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

$$J(u) = \min_{\alpha_1,\ldots,\alpha_k,\theta_u,\theta}((s(u[\alpha_1, \ldots, \alpha_k] - s(u)) + \zeta(u[\alpha_1, \ldots, \alpha_k], (\theta_u, \theta)) \tag{4.1}$$

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

To compute a heuristic cost $h(u)$ for a partial architecture u encountered during search, we substitute the nonterminals in u with neural networks parameterized by ω. These networks are *type-correct* — for example, if a nonterminal is supposed to generate subexpressions whose inputs are sequences, then the neural network used in its place is recurrent. We show an example of Near used in a program learning-graph search formulation in Figure 4.1.

We view the neurosymbolic programs resulting from this substitution as tuples $(u, (\theta_u, \omega))$. We define a semantics for such programs by extending our DSL's semantics, and lift the function ζ to assign costs $\zeta(u, (\theta_u, \omega))$ to such programs. The heuristic cost for u is now given by:

$$h(u) = \min_{w,\theta} \zeta(u, (\theta_u, \omega)). \tag{4.2}$$

As $\zeta(u, (\theta_u, \omega))$ is now end-to-end differentiable in ω and θ_u, we can compute $h(u)$ using gradient-based optimization techniques, such as gradient descent, as we would with complete program architectures during evaluation.

ϵ-Admissibility. It is well-known that in principle, neural networks are *universal function approximators*, when provided sufficient parameterization [45]. The neural-network based relaxation we use as our heuristic should then, in principle, be able to replicate the behavior of any architecture that would eventually replace such a network. However, in practice, the neural networks that we use may only form an approximate relaxation of the space of completions and parameters of architectures, due to time and resource constraints. Further, the training of these networks are not guaranteed to reach global optima. To account for these errors, we consider an approximate notion of admissibility and reason about our graph search problem through this lens. Many such notions have been considered in the past [46, 47, 48]; here, we follow a definition used by Harris [46]. For a fixed constant $\epsilon > 0$, let an *ϵ-admissible heuristic* be a function $h^*(u)$ over architectures such that $h^*(u) \leq J(u) + \epsilon$ for all u. Now consider any completion $u[\alpha_1, \ldots, \alpha_k]$ of an architecture u. As neural networks with adequate capacity are universal function approximators, there exist parameters ω^* for our neurosymbolic program such that for all $u, \alpha_1, \ldots, \alpha_k$, θ_u, and θ:

$$\zeta(u, (\theta_u, \omega^*)) \leq \zeta(u[\alpha_1, \ldots, \alpha_k], (\theta_u, \theta)) + \epsilon. \tag{4.3}$$

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

$$\begin{aligned} h(u) &\leq \min_{\alpha_1,\ldots,\alpha_k,\theta_u,\theta} \zeta(u[\alpha_1,\ldots,\alpha_k],(\theta_u,\theta)) + \epsilon \\ &\leq \min_{\alpha_1,\ldots,\alpha_k,\theta_u,\theta} \zeta(u[\alpha_1,\ldots,\alpha_k],(\theta_u,\theta)) + (s(u[\alpha_1,\ldots,\alpha_k]) - s(u)) + \epsilon = J(u) + \epsilon. \end{aligned} \tag{4.4}$$

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

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

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

4.3 Integrating NEAR with Graph Search Algorithms

The NEAR approach can be used in conjunction with any heuristic search algorithm [49] over architectures. Specifically, we have integrated NEAR with two classic graph search algorithms: A^* [47] and an iteratively deepened depth-first search with branch-and-bound pruning (IDS-BB). Both algorithms maintain a *search frontier* by computing an *f-score* for each node: $f(u) = g(u) + h(u)$, where $g(u)$ is the incurred path cost from the source node u_0 to the current node u, and $h(u)$ is a heuristic estimate of the

Algorithm 1: A* Search

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

Figure 4.2 : The A* search algorithm.

cost-to-go from node u. Additionally, IDS-BB prunes nodes from the frontier that have a higher f-score than the minimum path cost to a goal node found so far.

In Algorithm 1, we provide the pseudocode for the A* algorithm. This A* algorithm follows the standard formulation of A* search [47], where a priority queue is maintained as a frontier to select the next node for exploration during search. In Algorithm 2, we provide the pseudocode for the IDS-BB algorithm. This algorithm is a Heuristic-Guided Depth-First Search with three key characteristics: (1) the search depth is iteratively increased; (2) the search is ordered using a function $f(u)$ as in A*, and (3) Branch-and-Bound is used to prune unprofitable parts of the search space. We find that the use of iterative deepening in the program learning setting is useful in that it prioritizes searching shallower and less parsimonious programs early on in the search process.

ϵ-Optimality. An important property of a search algorithm is *optimality*: when multiple solutions (paths to complete programs) exist, the algorithm will find an optimal solution (the complete program with the lowest path cost). Both A* and

IDS-BB are optimal given admissible heuristics. An argument by Harris [46] shows that under heuristics that are ϵ-admissible, such as in the case of NEAR, the algorithms return solutions that at most an additive constant ϵ away from the optimal solution. Let C^* denote the optimal path cost in our graph $\mathcal{G}$, and let $h(u)$ be an ϵ-admissible heuristic (Eq. (4.4)). Suppose IDS-BB or A* returns a goal node α_G that does not have the optimal path cost C^*. Then there must exist a node u_O on the frontier that lies along the optimal path and has yet to be expanded. This lets us establish an upper bound on the path cost of α_G:

$$g(\alpha_G) = f(\alpha_G) \leq f(u_O) = g(u_O) + h(u_O) \leq g(u_O) + J(u_O) + \epsilon \leq C^* + \epsilon. \tag{4.5}$$

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

$$f(u_p^*) = g(u_p) + J(u_p) \geq f(u_p) = g(u_p) + h(u_p) + \epsilon > C = g(u_p) + h(u_p) > C - \epsilon \tag{4.6}$$

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

4.4 Implementation details on NEAR algorithms

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

In our experiments, we show that using these approximative hyperparameters allows for an acclerated search while maintaining strong empirical results with our NEAR-guided search algorithms. In chapter 5, we present the hyperparameter choices made for each experimental domain.

Algorithm 2: Iterative Deepening Depth-First-Search

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

Figure 4.3 : The Iterative Deepening Depth-First-Search Algorithm

Chapter 5

Experimental Design and Results

In order to show the effectiveness of NEAR-guided search for program learning, we instantiate the programmatic sequence classification problem in three separate domains, and use these domains as the test beds for our experiments. We start by providing details for each experimental domain. We then discuss experimental results that show that NEAR-guided search strategies can perform comparably to deep learning models trained on the same task. We also show that NEAR-guided methods are both more performant and efficient that existing state-of-the-art methods for program learning.

5.1 Datasets for Sequence Classification

For all datasets presented below, we augment the base DSL in Figure 3.1 with domain-specific library functions that include 1) learned affine transformations over a subset of the original set of features provided, and 2) sliding window feature-averaging functions.

5.1.1 CRIM13.

The *CRIM13* dataset The CRIM13 dataset studies the social behavior of a pair of mice annotated each frame by behavior experts [50] at 25Hz. The interaction between a resident mouse and an intruder mouse, which is introduced to the cage of the resident, is recorded. Each mice is tracked by one keypoint and a 19 dimensional feature vector based on this tracking data is provided at each frame. The feature vector consists of features such as velocity, acceleration, distance between mice, angle and angle changes. Our task in this domain is *sequence classification*: we classify each frame with a behavior label from CRIM13. Every frame is labelled with one of 12 actions, or “other". The “other" class corresponds to cases where no action of interest is occurring. Here, we focus on two binary classification tasks: other vs. rest, and sniff vs. rest. The first task, other vs. rest, corresponds to labeling whether there is an action of interest in the frame. The second task, sniff vs. rest, corresponds to whether the resident mouse is sniffing any part of the intruder mouse. These two tasks are chosen such that the RNN baseline has reasonable performance only using the tracked keypoint features of the mice. We split the train set in [50] at the video level into our train and validation set, and we present test set results on the same set as [50]. Each video is split into sequences of 100 frames. There are 12404 training trajectories, 3077 validation trajectories, and 2953 test trajectories.

Training details of CRIM13 baselines. All CRIM13 baselines training uses the Adam [51] optimizer and cross-entropy loss. In the loss for sniff vs. rest, the sniff class is weighted by 1.5. Each synthesis baseline was run on an Intel 2.2-GHz Xeon CPU with 4 cores, equipped with an NVIDIA Tesla P100 GPU with 3584 CUDA cores.

5.1.2 Fly-vs.-Fly.

The *Fly-vs.-Fly* dataset [52] tracks a pair of flies and their actions as they interact in different contexts. Each timestep is represented by a 53-dimensional feature vector including 17 features outlining the fly’s position and orientation along with 36 position-invariant features, such as linear and angular velocities. Our task in this domain is that of *bout-level classification*, where we are tasked to classify a given trajectory of timesteps to a corresponding single action taking place. Of the three datasets within *Fly-vs.-Fly*, we use the *Aggression* and *Boy-meets-Boy* datasets and classify trajectories over the 7 labeled actions displaying aggressive, threatening, and nonthreatening behaviors in these two datasets. We omit the use of the *Courtship* dataset for our classification task, primarily due to the heavily skewed trajectories in this dataset that vary highly in length and action type from the *Aggression* and *Boy-meets-Boy* datasets. Full details on these datasets, as well as where to download them, can be found in [52]. To ensure a desired balance in our training set, we limit the length of trajectories to 300 timesteps, and break up trajectories that exceed this length into separate trajectories with the same action label for data augmentation. Our training dataset has 5339 trajectories, our validation set has 594 trajectories, and our test set has 1048 trajectories. The average length of a trajectory is 42.06 timesteps.

Training details of Fly-v.-Fly baselines. For all of our program synthesis baselines , we used the Adam [51] optimizer and cross-entropy loss. Each synthesis baseline was run on an Intel 4.9-GHz i7 CPU with 8 cores, equipped with an NVIDIA RTX 2070 GPU w/ 2304 CUDA cores.

5.1.3 Basketball.

We use a subset of the basketball dataset from [53] that tracks the movements of professional basketball players. Each trajectory is of length 25 and contains the xy-positions of 5 offensive players, 5 defensive players, and the ball (22 features per frame). We aim to learn programs that can predict which offensive player has the ball (the "ballhandler") or whether the ball is being passed. The basketball data tracks player positions (xy-coordinates on court) from real professional games. We used the processed version from [53], which includes trajectories over 8 seconds (3 Hz in our case of sequence length 25) centered on the left half-court. Among the offensive and defensive teams, players are ordered based on their relative positions. Labels for the ballhandler were extracted with a labeling function written by a domain expert. In total, we have 18,000 trajectories for training, 2801 for validation, and 2693 for test. See Table 5.3 for full details of this dataset.

Training details of Basketball baselines. All Basketball experiments use Adam [51] and optimize cross-entropy loss. Each synthesis baseline was run on an Intel 3.6-GHz i7-7700 CPU with 4 cores, equipped with an NVIDIA GTX 1080 Ti GPU with 3584 CUDA cores.

Details discussed in this section regarding each dataset are presented in table 5.3.

5.2 Overview of Baseline Program Learning Strategies

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

Top-down enumeration. We synthesize and evaluate complete programs in order of increasing complexity measured using the structural cost $s(\alpha)$. This strategy is widely employed in program learning contexts [28, 9, 10] and is provably complete. Since our graph $\mathcal{G}$ grows infinitely, our implementation is akin to breadth-first search up to a specified depth.

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

Genetic algorithm. We follow the formulation in Valkov et al. [28]. In our genetic algorithm, crossover, selection, and mutation operations evolve a population of programs over a number of generations until a predetermined number of programs have been trained. The crossover and mutation operations only occur when the resulting program is guaranteed to be type-safe.

In tables 5.4, 5.5, 5.6, 5.7, and 5.8, we present the hyperparameters used in our implementation for all baselines.

For all baseline algorithms, as well as A*-Near and Ids-bb-Near, model parameters (θ) were learned with the training set, whereas program architectures (α) were evaluated using the performance on the validation set. Additionally, all baselines (including Near algorithms) used F1-score [54] error as the evaluation objective ζ by which programs were chosen. To account for class imbalances, F1-scoring is commonly used as an evaluation metric in behavioral classification domains, such as those considered in our work [52, 50].

5.3 Experimental Results

5.3.1 Performance of learned programs.

Table 5.1 shows the performance results on the test sets of our program learning algorithms, averaged over 3 seeds. We also provide the standard deviations of our 3 seeds in 5.2 for further detail. The same structural cost function $s(\alpha)$ is used for all algorithms, but varies across domains. We find that the Near-guided search algorithms consistently outperform other baselines in F1-score while accuracy is

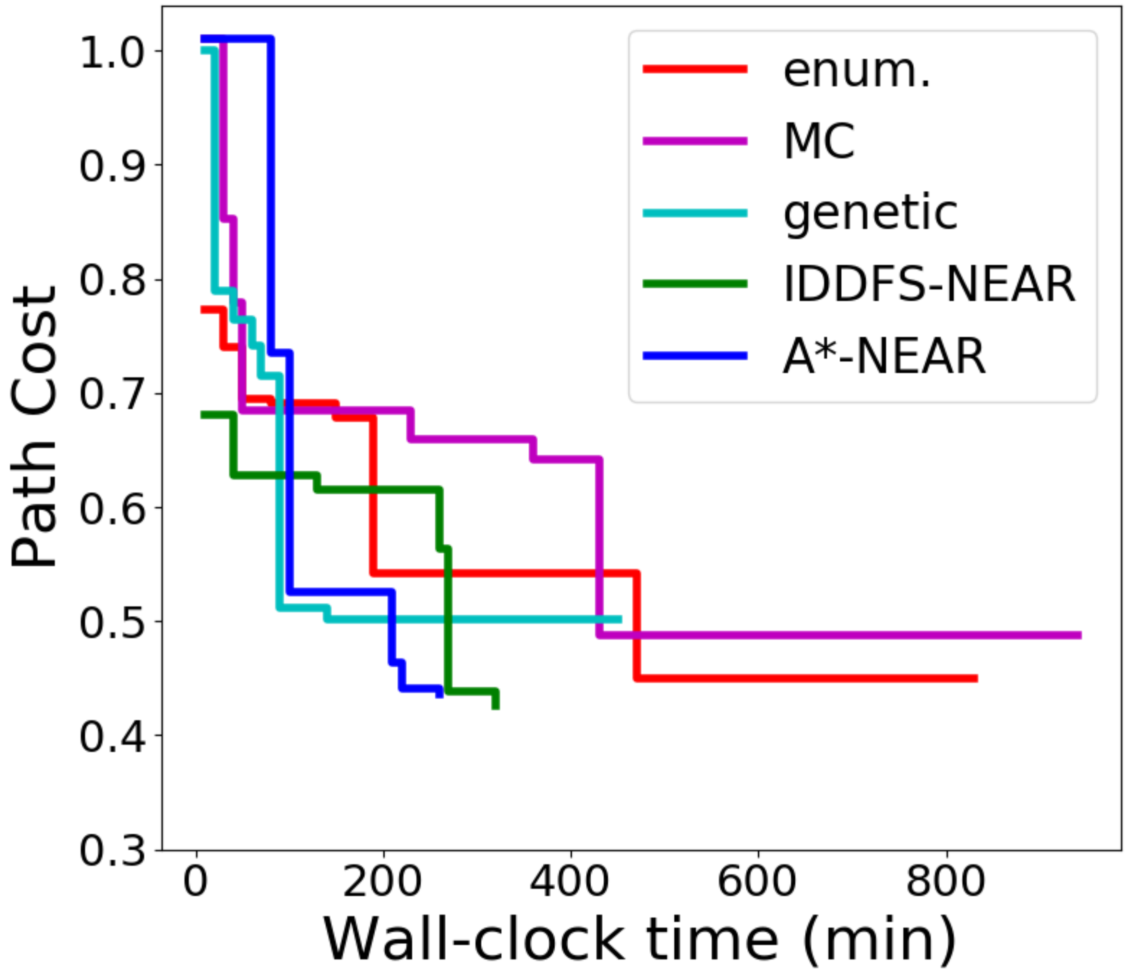

Figure 5.1 : CRIM13-sniff minimum path costs found over time. This plot shows the median minimum path cost to a goal node found at a given time, across 3 trials (for trials that terminate first, we extend the plots so the median remains monotonic). A*-Near (blue) and Ids-bb-Near (green) will often find a goal node with a smaller path cost, or find one of similar performance but much faster.

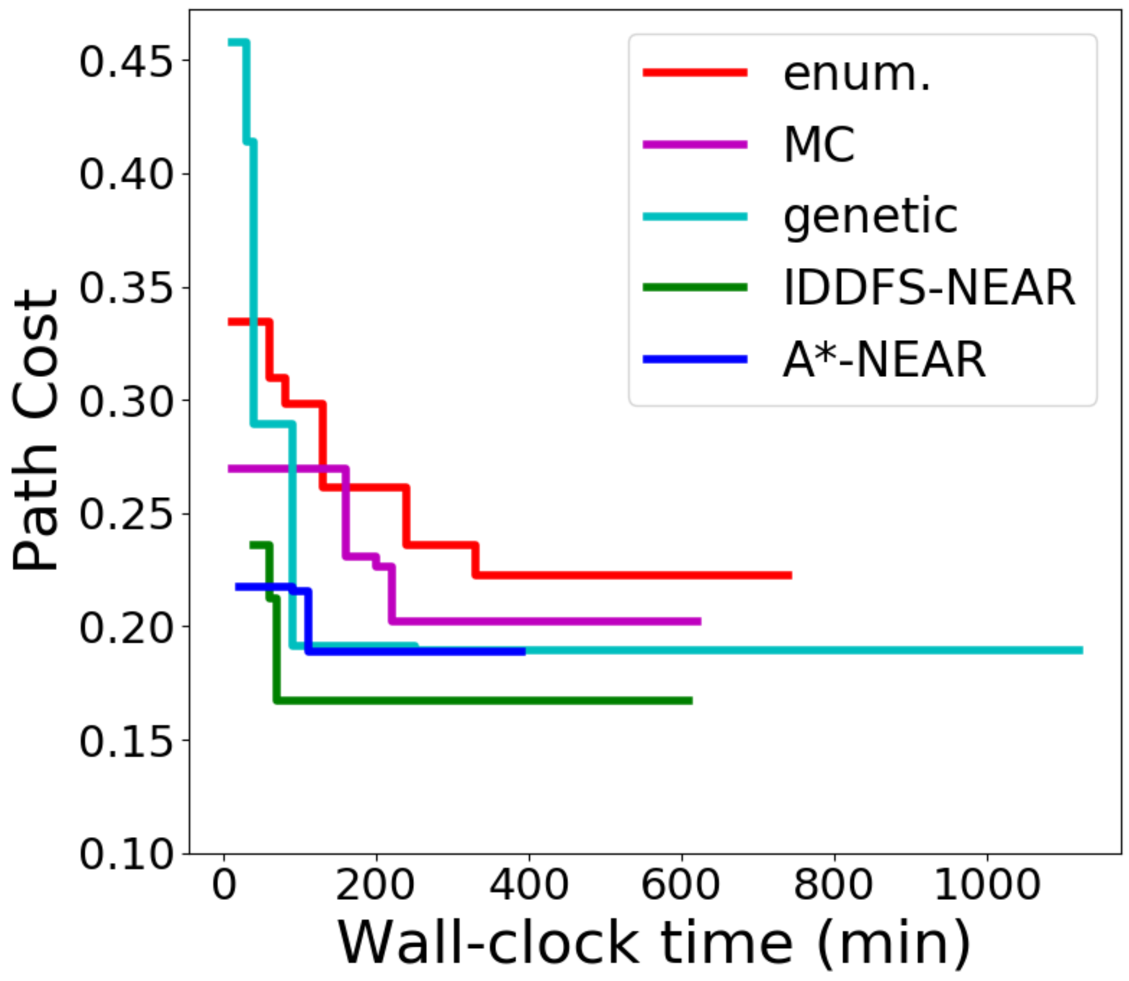

Figure 5.2 : Fly-vs.-Fly minimum path costs found over time.

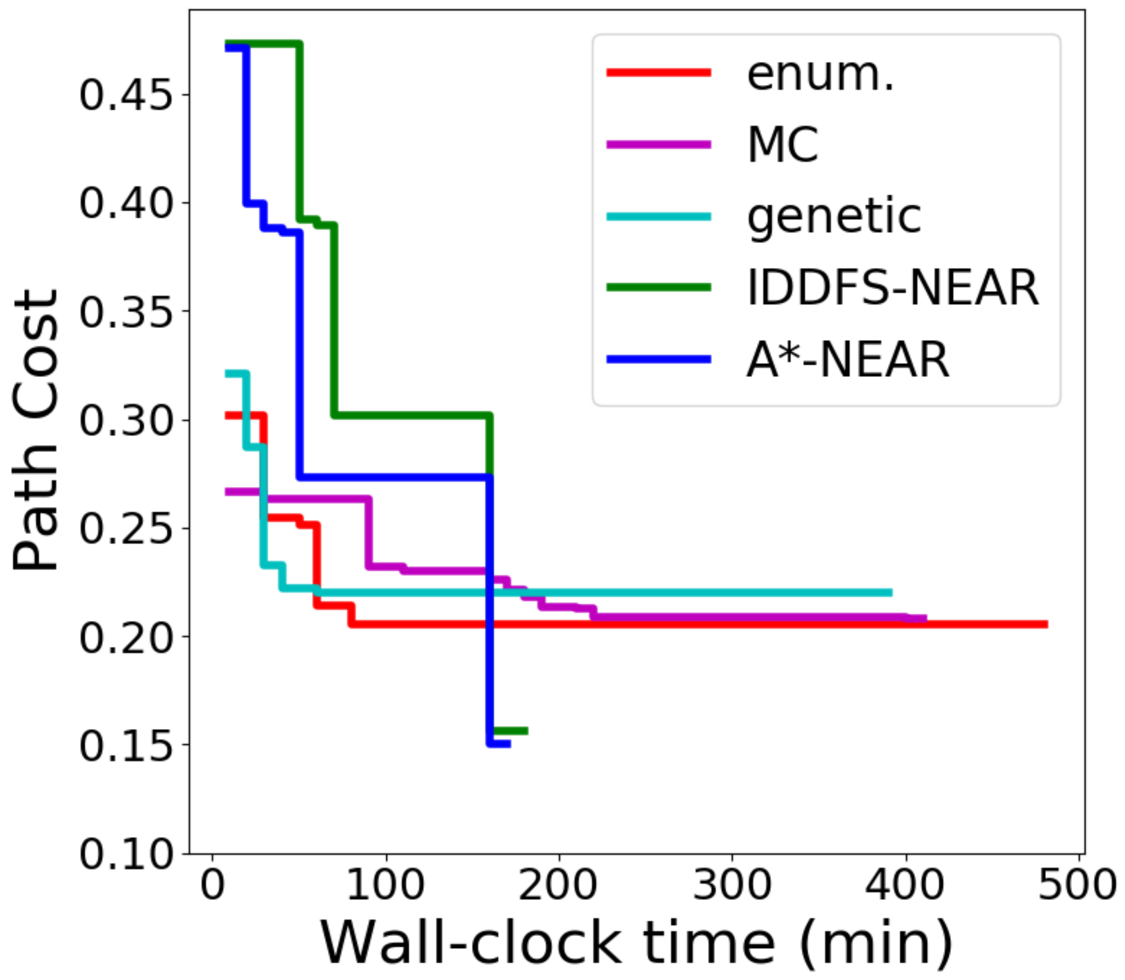

Figure 5.3 : Bball-ballhandler minimum path costs found over time.

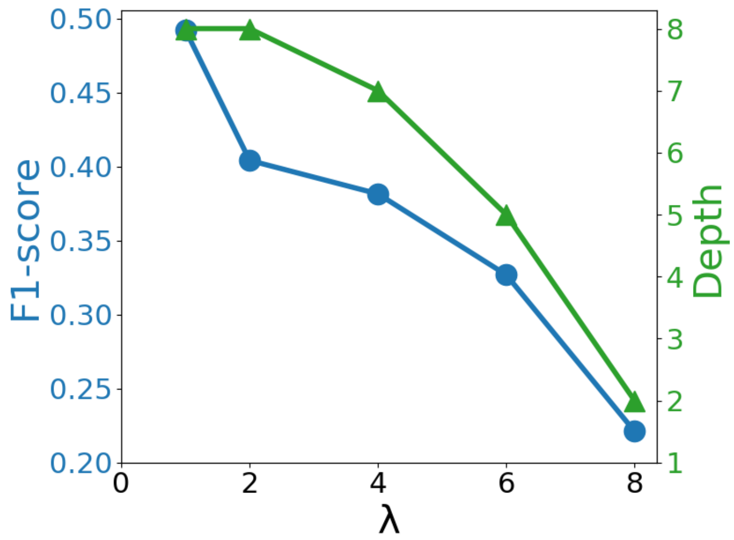

Figure 5.4 : CRIM13-sniff penalty variation.

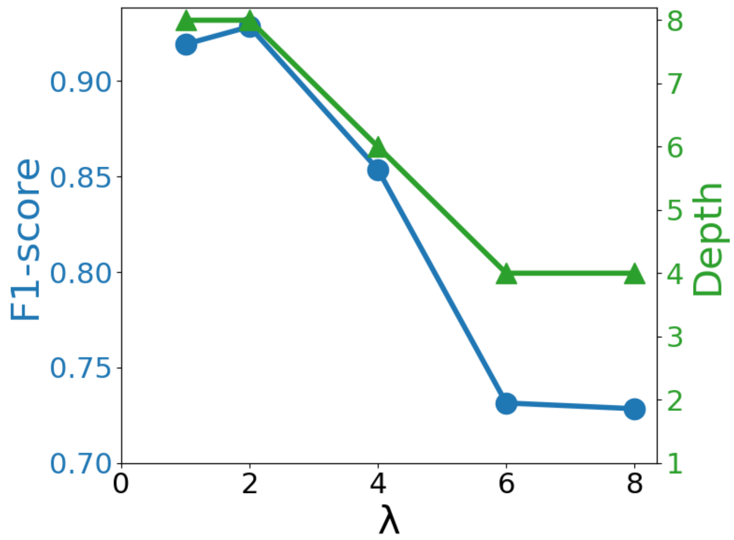

Figure 5.5 : Bball-ballhandler penalty variation.

Figure 5.6 : As we increase λ in Eq. (5.1), we observe that A*-NEAR will learn programs with decreasing program depth and also decreasing F1-score. This highlights that we can use λ to control the trade-off between structural cost and performance.

	CRIM13-sniff			CRIM13-other			Fly-vs.-Fly			Bball-ballhandler		
	Acc.	F1	Dep.	Acc.	F1	Dep.	Acc.	F1	Dep.	Acc.	F1	Dep.
Enum.	.851	.221	3	.707	.762	2	.819	.863	2	.844	.857	6.3
MC	.843	.281	7	.630	.715	1	.833	.852	4	.841	.853	6
Genetic	.829	.181	1.7	.727	.768	3	.850	.868	6	.843	.853	6.7
IDS-BB-NEAR	.831	.452	7.5	.704	.760	8.7	.876	.892	4	.889	.903	8
A*-NEAR	.826	.448	7.7	.723	.770	7.7	.872	.885	4	.906	.918	8
RNN	.889	.481	-	.756	.785	-	.963	.964	-	.945	.950	-

Table 5.1 : Mean accuracy, F1-score, and program depth of learned programs (3 trials). Programs found using our NEAR algorithms consistently achieve better F1-score than baselines and match more closely to the RNN's performance. Our algorithms are also able to search and find programs of much greater depth than the baselines. Experiment hyperparameters are included in the appendix.

comparable (note that our ζ does not include accuracy). Furthermore, NEAR-guided search algorithms are capable are finding deeper and more complex programs that can offer non-trivial interpretations, such as the ones shown in Figures 3.2 and 3.3. Lastly, we verify that our learned programs are comparable with highly expressive RNNs, and see that there is at most a 10% drop in F1-score when using NEAR-guided search algorithms with our DSL.

	CRIM13-sniff			**CRIM13-other**			**Fly-vs.-Fly**			**Bball-ballhandler**		
	Acc.	F1	Dep.	Acc.	F1	Dep.	Acc.	F1	Dep.	Acc.	F1	Dep.
Enum.	.024	.105	1	.036	.011	1	.013	.012	0	.009	.009	0.6
MC	.013	.127	1.7	.088	.031	0.6	.028	.018	2	.012	.012	0.6
Genetic	.003	.015	0.6	.005	.004	1.7	.028	.030	1	.016	.019	0.6
IDDFS-NEAR	.024	.022	0.6	.024	.016	0.6	.023	.016	0	.006	.006	0
A*-NEAR	.009	.068	1	.012	.002	1.5	.003	.004	0	.034	.034	0
RNN	.008	.019	-	.005	.002	-	.006	.005	-	.001	.001	-

Table 5.2 : Standard Deviations of accuracy, F1-score, and program depth of learned programs (3 trials).

5.3.2 Efficiency of NEAR-guided graph search.

Figures 5.3, 5.2, and 5.1 track the progress of each program learning algorithm during search by following the median best path cost (Eq. (3.1)) at a given time across 3 independent trials. For times where only 2 trials are active (i.e. one trial had already terminated), we report the average. Algorithms for each domain were run on the same machine to ensure consistency, and each non-NEAR baseline was set up such to have at least as much time as our NEAR-guided algorithms for their search procedures (see Appendix). We observe that NEAR-guided search algorithms are able to find low-cost solutions more efficiently than existing baselines, while maintaining an overall shorter running time.

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

Table 5.3 : Dataset details.

5.3.3 Cost-performance trade-off.

We can also consider a modification of our objective in Eq. (3.1) that allows us to use a hyperparameter λ to control the trade-off between structural cost and performance:

$$(\alpha^*, \theta^*) = \arg\min_{(\alpha,\theta)}(\lambda \cdot s(\alpha) + \zeta(\alpha, \theta)). \tag{5.1}$$

To visualize this trade-off, we run A*-Near with the modified objective Eq. (5.1) for various values of λ. Note that $\lambda = 1$ is equivalent to our experiments in Table 5.1. Figure 5.6 shows that for the *Basketball* and *CRIM13* datasets, as we increase λ, which puts more weight on the structural cost, the resulting programs found by A*-Near search have decreasing F1-scores but are also more shallow. This confirms our expectations that we can control the trade-off between structural cost and performance, which allows users of Near-guided search algorithms to adjust to their preferences. Unlike the other two experimental domains, the most performant programs learned in *Fly-vs.-Fly* were relatively shallow, so we omitted this domain as the trade-off showed little change in program depth.

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

Table 5.4 : Hyperparameters for constructing graph $\mathcal{G}$.

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

Table 5.5 : Training hyperparameters for RNN baseline.

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

Table 5.6 : Training hyperparameters for all program learning algorithms. The # neural epochs hyperparameter refers only to the number of epochs that neural program approximations were trained in NEAR strategies.

	A*-Near	Ids-bb-Near			
	frontier size	frontier size	init. depth	depth bias	perf. mult.
CRIM13-sniff	8	8	5	0.95	0.975
CRIM13-other	8	8	5	1.3*	0.975
Fly-vs.Fly	10	10	4	0.9	0.95
Bball-ballhander	400	30	3	1.0	1.0

Table 5.7 : Additional hyperparameters for A*-Near and Ids-bb-Near. The depth bias value for CRIM13-other used a slightly different implementation (see codebase for details.)

	MC	Enum.	Genetic					
	# samps	max # prog.	popu. size	select. size	# gens	total # evals	mut. prob.	enum. depth
CRIM13-sniff	50	300	15	8	20	100	0.1	5
CRIM13-other	50	300	15	8	20	100	0.1	5
Fly-vs.Fly	25	100	20	10	10	10	0.1	6
Bball-ballhander	150	1200	100	50	10	1000	0.01	7

Table 5.8 : Additional hyperparameters for other program learning baselines

Chapter 6

Conclusions

6.1 Discussion and Future Work

In this thesis, the paradigm of *program learning* was presented as a burgeoning new approach to creating more robust, interpretable solutions to machine learning problems than those of state-of-the-art black-box learning models. Despite the great promise of program learning, we identified a number of issues with the approach, such as its inefficiency at scale and the lack of optimized learning strategies available for programs in prior domain-specific languages. In order to address these issues, we presented a novel graph search approach to learning differentiable programs. We explained how in this problem formulation, as well as other common formulations of program learning problems, existing top-down and bottom-up methods become highly inefficient as the program space scales in size and complexity.

In order to create a more efficient method of program search through our defined space of programs, we leverage a novel construction of an admissible heuristic using neural relaxations to efficiently search over program architectures within this graph space. This heuristic exploits the differentiability of the programs within our space, and allows for freely parameterized neural networks to approximate the behavior of program architectures in an underestimating fashion, making this heuristic (approximately) admissible. We discuss the guarantees that can be made with such an approximately admissible heuristic, and we outline how this heuristic, Near, can be leveraged in well-known informed search algorithms, primarily A* and Iterative-Deepening Depth-First-Search with Branch-and-Bound. Our experiments showed that programs learned using our Near-guided algorithms can have competitive performance with respect to deep neural models, and that our search-based learning procedure substantially outperforms conventional program learning approaches in both performance and efficiency.

There are many directions for future work. One direction is to extend the approach to richer DSLs and neural heuristic architectures, for example, those suited to reinforcement learning [10] and generative modeling [55]. The meta-approach of using neural relaxations to approximate program behavior allows for a flexibility of usage that can be exploited across a variety of problem domains within machine learning. Another is to combine Near with classical program synthesis methods based on symbolic reasoning. A third is to more tightly integrate with real-world applications to evaluate the interpretability of learned programs. This closer integration with specific domains will likely require both domain expertise as well as a technical understanding of how to express domain knowledge in the form of a differentiable function that is usable by our learning approach.

6.2 Program Learning's Broader Impact

As discussed in Section 1, *interpretability* is a key motivator for research on program learning. Programmatic models can be better than neural models at explaining causal relationships in data and exposing underlying biases that may have been learned without explicit direction, while offering greater expressiveness and performance than shallower models (e.g., linear or logistic classifiers). For this reason, progress on program learning can allow for more widespread use of machine learning in fields, such as healthcare and the natural sciences, where safety and accountability to humans are critical.

Program learning efforts such as ours also allow for incorporation of inductive bias, which can highly influence the semantic meaning of learned programs depending on the functions provided by the user. In fields such as autonomous driving, rules and regulations that are otherwise challenging for purely neural models to learn can be easily encoded by users into program learning solutions. However, this bias can just as easily be exploited by users who desire specific outcomes from interpretable programmatic solutions. Ultimately, users of program learning methods must ensure that any incorporated inductive bias will not lead to unfair or misleading programs.

Bibliography

[1] Q. Zhang, L. T. Yang, Z. Chen, and P. Li, "A survey on deep learning for big data," *Inf. Fusion*, vol. 42, pp. 146–157, 2018.

[2] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Training pruned neural networks," *CoRR*, vol. abs/1803.03635, 2018.

[3] D. Tsipras, S. Santurkar, L. Engstrom, A. Turner, and A. Madry, "Robustness may be at odds with accuracy," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, OpenReview.net, 2019.

[4] J. J. Michalenko, A. Shah, A. Verma, R. G. Baraniuk, S. Chaudhuri, and A. B. Patel, "Representing formal languages: A comparison between finite automata and recurrent neural networks," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, OpenReview.net, 2019.

[5] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in *Computer Aided Verification*, pp. 97–117, 2017.

[6] K. Ellis, A. Solar-Lezama, and J. Tenenbaum, "Sampling for bayesian program learning," in *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10,*

2016, Barcelona, Spain (D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, eds.), pp. 1289–1297, 2016.

[7] K. Ellis, D. Ritchie, A. Solar-Lezama, and J. Tenenbaum, "Learning to infer graphics programs from hand-drawn images," in *Advances in Neural Information Processing Systems*, pp. 6059–6068, 2018.

[8] L. Valkov, D. Chaudhari, A. Srivastava, C. Sutton, and S. Chaudhuri, "Houdini: Lifelong learning as program synthesis," in *Advances in Neural Information Processing Systems*, pp. 8687–8698, 2018.

[9] A. Verma, V. Murali, R. Singh, P. Kohli, and S. Chaudhuri, "Programmatically interpretable reinforcement learning," in *International Conference on Machine Learning*, pp. 5052–5061, 2018.

[10] A. Verma, H. M. Le, Y. Yue, and S. Chaudhuri, "Imitation-projected programmatic reinforcement learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[11] K. Ellis, M. I. Nye, Y. Pu, F. Sosa, J. Tenenbaum, and A. Solar-Lezama, "Write, execute, assess: Program synthesis with a REPL," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada* (H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, eds.), pp. 9165–9174, 2019.

[12] S. Gulwani, A. Polozov, and R. Singh, *Program Synthesis*, vol. 4. NOW, August 2017.

[13] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli, "Robustfill: Neural program learning under noisy I/O," *CoRR*, vol. abs/1703.07469, 2017.

[14] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Dependable Software Systems Engineering*, pp. 1–25, 2015.

[15] A. Solar-Lezama, "The sketching approach to program synthesis," in *Programming Languages and Systems*, pp. 4–13, Springer, 2009.

[16] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS*, pp. 404–415, 2006.

[17] J. K. Feser, S. Chaudhuri, and I. Dillig, "Synthesizing data structure transformations from input-output examples," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pp. 229–239, 2015.

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

[19] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track*

Proceedings, 2017.

[20] X. Chen, C. Liu, and D. 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.

[21] E. Parisotto, A.-r. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli, "Neuro-symbolic program synthesis," *arXiv preprint arXiv:1611.01855*, 2016.

[22] V. Murali, S. Chaudhuri, and C. Jermaine, "Neural sketch learning for conditional program generation," in *ICLR*, 2018.

[23] Y. Ganin, T. Kulkarni, I. Babuschkin, S. M. A. Eslami, and O. Vinyals, "Synthesizing programs for images using reinforced adversarial learning," *CoRR*, vol. abs/1804.01118, 2018.

[24] W. Lee, K. Heo, R. Alur, and M. Naik, "Accelerating search-based program synthesis using learned probabilistic models," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pp. 436–449, 2018.

[25] R. Bunel, M. Hausknecht, J. Devlin, R. Singh, and P. Kohli, "Leveraging grammar and reinforcement learning for neural program synthesis," *arXiv preprint arXiv:1805.04276*, 2018.

[26] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum, "Human-level concept learning through probabilistic program induction," *Science*, vol. 350, no. 6266, pp. 1332–1338, 2015.

[27] K. Ellis, A. Solar-Lezama, and J. B. Tenenbaum, "Unsupervised learning by program synthesis," in *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pp. 973–981, 2015.

[28] L. Valkov, D. Chaudhari, A. Srivastava, C. A. Sutton, and S. Chaudhuri, "HOUDINI: lifelong learning as program synthesis," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pp. 8701–8712, 2018.

[29] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *arXiv preprint arXiv:1410.5401*, 2014.

[30] S. Reed and N. De Freitas, "Neural programmer-interpreters," *arXiv preprint arXiv:1511.06279*, 2015.

[31] K. Kurach, M. Andrychowicz, and I. Sutskever, "Neural random-access machines," *arXiv preprint arXiv:1511.06392*, 2015.

[32] A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra, and T. P. Lillicrap, "Meta-learning with memory-augmented neural networks," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016* (M. Balcan and K. Q. Weinberger, eds.), vol. 48 of *JMLR Workshop and Conference Proceedings*, pp. 1842–1850, JMLR.org, 2016.

[33] J. Devlin, R. Bunel, R. Singh, M. J. Hausknecht, and P. Kohli, "Neural program meta-induction," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December*

2017, Long Beach, CA, USA (I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, eds.), pp. 2080–2088, 2017.

[34] A. Saha, G. A. Ansari, A. Laddha, K. Sankaranarayanan, and S. Chakrabarti, "Complex program induction for querying knowledge bases in the absence of gold programs," *Trans. Assoc. Comput. Linguistics*, vol. 7, pp. 185–200, 2019.

[35] S. Sun, H. Noh, S. Somasundaram, and J. J. Lim, "Neural program synthesis from diverse demonstration videos," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018* (J. G. Dy and A. Krause, eds.), vol. 80 of *Proceedings of Machine Learning Research*, pp. 4797–4806, PMLR, 2018.

[36] H. Liu, K. Simonyan, and Y. 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.

[37] R. Shin, C. Packer, and D. Song, "Differentiable neural network architecture search," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*, OpenReview.net, 2018.

[38] B. Zoph and Q. 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.

[39] E. Real, A. Aggarwal, Y. Huang, and Q. 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, pp. 4780–4789, AAAI Press, 2019.

[40] J. Xiang and S. 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* (C. J. C. Burges, L. Bottou, Z. Ghahramani, and K. Q. Weinberger, eds.), pp. 2418–2426, 2013.

[41] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to automata theory, languages, and computation, 3rd Edition.* Pearson international edition, Addison-Wesley, 2007.

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

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

[44] A. Albarghouthi, S. Gulwani, and Z. Kincaid, "Recursive program synthesis," in *International conference on computer aided verification*, pp. 934–950, Springer, 2013.

[45] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang, "The expressive power of neural networks: A view from the width," in *Advances in Neural Information Processing*

Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA (I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, eds.), pp. 6231–6239, 2017.

[46] L. R. Harris, "The heuristic search under conditions of error," *Artificial Intelligence*, vol. 5, no. 3, pp. 217–234, 1974.

[47] J. Pearl, "Heuristics: Intelligent search strategies for computer problem solving," *Addision Wesley*, 1984.

[48] R. A. Valenzano, S. J. Arfaee, J. Thayer, R. Stern, and N. R. Sturtevant, "Using alternative suboptimality bounds in heuristic search," in *Twenty-Third International Conference on Automated Planning and Scheduling*, 2013.

[49] S. Russell and P. Norvig, "Artificial intelligence: a modern approach," 2002.

[50] X. P. Burgos-Artizzu, P. Dollár, D. Lin, D. J. Anderson, and P. Perona, "Social behavior recognition in continuous video," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1322–1329, IEEE, 2012.

[51] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[52] E. Eyjolfsdottir, S. Branson, X. P. Burgos-Artizzu, E. D. Hoopfer, J. Schor, D. J. Anderson, and P. Perona, "Detecting social actions of fruit flies," in *European Conference on Computer Vision*, pp. 772–787, Springer, 2014.

[53] Y. Yue, P. Lucey, P. Carr, A. Bialkowski, and I. Matthews, "Learning fine-grained spatial models for dynamic sports play prediction," in *2014 IEEE international*

conference on data mining, pp. 670–679, IEEE, 2014.

[54] Y. Sasaki, “The truth of the f-measure,” *Teach Tutor Master*, 01 2007.

[55] D. Ritchie, A. Thomas, P. Hanrahan, and N. Goodman, “Neurally-guided procedural models: Amortized inference for procedural graphics programs using neural networks,” in *Advances in neural information processing systems*, pp. 622–630, 2016.