Neurosymbolic Programming

Swarat Chaudhuri[1], Kevin Ellis[2], Oleksandr Polozov[3], Rishabh Singh[4], Armando Solar-Lezama[5] and Yisong Yue[6]

[1] *The University of Texas at Austin; swarat@cs.utexas.edu*
[2] *Cornell University; kellis@cornell.edu*
[3] *Google; Work authored while at Microsoft Research; polozov@google.com*
[4] *Google; rising@google.com*
[5] *Massachusetts Institute of Technology (MIT); asolar@csail.mit.edu*
[6] *The California Institute of Technology (Caltech); yyue@caltech.edu*

ABSTRACT

We survey recent work on *neurosymbolic programming*, an emerging area that bridges the areas of deep learning and program synthesis. Like in classic machine learning, the goal here is to learn functions from data. However, these functions are represented as *programs* that can use neural modules in addition to symbolic primitives and are induced using a combination of symbolic search and gradient-based optimization.

Neurosymbolic programming can offer multiple advantages over end-to-end deep learning. Programs can sometimes naturally represent long-horizon, procedural tasks that are difficult to perform using deep networks. Neurosymbolic representations are also, commonly, easier to interpret and formally verify than neural networks. The restrictions of a programming language can serve as a form of regularization and lead to more generalizable and data-efficient

Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama and Yisong Yue (2021), “Neurosymbolic Programming”, Foundations and Trends® in Programming Languages: Vol. 7, No. 3, pp 158–243. DOI: 10.1561/2500000049.

learning. Compositional programming abstractions can also be a natural way of reusing learned modules across learning tasks.

In this monograph, we illustrate these potential benefits with concrete examples from recent work on neurosymbolic programming. We also categorize the main ways in which symbolic and neural learning techniques come together in this area. We conclude with a discussion of the open technical challenges in the field.

1

Introduction

The last decade has seen breathtaking progress in the field of deep learning. Every few months, the media is abuzz with tales of a deep neural network conquering yet another milestone in artificial intelligence (AI). Deep learning systems such as AlphaGo (Silver *et al.*, 2016), the deep reinforcement learning agent that defeated a world champion in the game of Go, and GPT-3 (Brown *et al.*, 2020), the neural language model with 175 billion trainable parameters and the ability to generate stunningly human-like text, are now part of folklore.

At the same time, concerns remain about the use of deep neural networks in real-world problem domains (Marcus and Davis, 2019). In applications such as autonomous robotics and the natural sciences, it is important for learning models to be *interpretable*. However, neural networks are black boxes for most practical purposes. While modern neural networks are obtained through the composition of many layers, it is nearly impossible to assign responsibility for a network's capabilities to specific layers. This makes it difficult to *reuse* components of a network in the way that is possible for traditional, human-written software. Finally, the training process for neural networks is entirely data-driven and must learn even the most basic forms of human-held knowledge from data. Such training usually takes very large volumes of data. Also, commonly, their results depend on low-level facets of the training environment, and the networks they learn can fare poorly on inputs that fall even slightly outside the training distribution.

More fundamentally, as Bengio argues in his Turing lecture (Bengio, 2019), deep learning primarily automates what Kahneman (Kahneman, 2011) calls System 1 thinking: intuitive, fast, and unconscious pattern recognition. However, human cognition also includes System 2 thinking, which is slow, logical, and algorithmic. Bengio points out that AI systems of the future would need to also automate such thought.

Of course, the symbolic AI tradition, which dominated AI for several decades, had invested heavily on automating System 2 thought. This tradition modeled the world using symbolic rules and logical assertions and used symbolic methods like heuristic search and automated deduction to construct inferences and plans. Unfortunately, capturing the world entirely using rules and logic proved difficult, not least because it was difficult to model uncertain and ambiguous knowledge in such notations. Also, classical symbolic methods did not have a mechanism to handle sensory inputs. Finally, discrete reasoning is an NP-hard problem, and algorithms for tasks such as planning and theorem-proving could only scale so far. As a result, bottom-up, data-driven methods ended up eclipsing these methods.

However, as integrating System 1 and System 2 thought becomes more important, an emerging body of work on *neurosymbolic machine learning* is seeking to couple classical symbolic algorithms with contemporary deep learning techniques. The resulting whole is often greater than the sum of its parts. For example, the neural component of such a method can help the method's symbolic component scale better, by *guiding* the latter's discrete decisions. The neural component can also handle lower-level sensory processing and allow the symbolic algorithm to operate on perceptual inputs. Conversely, the symbolic component can often serve as a *regularizer* that helps the neural component learn better, and provide a level of interpretability and trust that would be impossible with a purely neural method.

In particular, there is an emerging body of work on neurosymbolic learning that lies at the interface of deep learning and *program* *synthesis* (Gulwani *et al.*, 2017). We refer to this literature as *Neurosymbolic Programming*. The purpose of this monograph is to survey recent developments in this area.

1.1 What is Neurosymbolic Programming?

Neurosymbolic programming, as we define the term, is a generalization of classic program synthesis. The goal in program synthesis is to automatically discover programs from high-level task specifications. Traditionally, the specifications are hard logical constraints, for example, tests that need to be satisfied exactly, pre-postcondition pairs, or temporal logic formulas. The programs are structured, symbolic terms that follow the syntax of a domain-specific language (DSL). The discovery of programs is based on a combination of combinatorial search and automated reasoning (Gulwani *et al.*, 2017).

By contrast, programs in neurosymbolic programming can have both neural and symbolic elements. The synthesis objective may include hard constraints like in classic program synthesis. However, neurosymbolic programming also incorporates into the picture the standard objective of machine learning: finding a function that fits a given dataset *approximately* and *generalizes* to unseen inputs.

Now we delineate the boundaries of neurosymbolic programming more precisely. Let us define a *symbolic component* as a function that comes with a symbolic implementation, or at least, a (possibly incomplete) symbolic specification of its functionality. In contrast, a *neural component* is a (typically over-)parameterized, differentiable blackbox function that does not have an a priori specification.

Composition is a fundamental operation in both traditional programming and deep learning. However, there is a key difference between composition in the two settings. In traditional programming, composition requires that certain requirements hold at the interface of the components being composed. No such requirement exists at the interface of different layers in deep learning. Let us designate the former kind of composition as *symbolic*. We consider a *neurosymbolic program* to be a program that uses neural components and either symbolic components or symbolic compositions.

A *neurosymbolic learning algorithm* is a mechanism for program synthesis that uses deep representations and gradient-based optimization as well as symbolic methods such as search and automated deduction. Such an algorithm must discover the program's discrete structure, or *architecture*. In addition, the algorithm must discover the program's real-valued parameters (for example, parameters of the program's neural modules), if any. The task specification that directs this search could include hard constraints like in classic program synthesis. Commonly, however, it also includes a quantitative loss function derived from, for example, labeled data or reward functions. The goal of the algorithm is to find a program that optimizes the loss while obeying the hard constraints.

The sets of methods that target neurosymbolic programs and use neurosymbolic learning algorithms are overlapping but not identical. In particular, there are methods that use neurosymbolic algorithms to discover symbolic programs, and methods that synthesize neurosymbolic programs using purely symbolic or purely neural methods. In this paper, we take a broad perspective and consider *neurosymbolic programming* to be the study of the *union* of the two sets.

We leave out of the scope of this paper models such as Neural Turing Machines (Graves *et al.*, 2014) and Neural Stack Machines (Grefenstette *et al.*, 2015). These models are inspired by classic models of programming, such as Turing Machines and stack machines, and have certain program-like capabilities (for example, Neural Turing Machines can perform reads and writes to a differentiable memory). However, these models are not learned using neurosymbolic algorithms. Also, they do not impose human-comprehensible specifications on the interfaces between model components and cannot be naturally decomposed into high-level modules.

Neither do we consider Tensor Product Representations (Smolensky *et al.*, 2016), which provide a symbolically inspired inductive bias in neural networks, to be an example of neurosymbolic programming. This is because such a model cannot be naturally interpreted as a program even when the network implements a programmatic task. Also outside our scope are Neural Programmer Interpreters (Reed and De Freitas, 2015), which are recurrent neural networks that learn to execute programs, and neural models for combinatorial tasks such as MAX-SAT solving (Wang *et al.*, 2019). That said, all of these models are closely related to neurosymbolic programming, and future work may integrate them with approaches that we discuss in this monograph.

1.2 Benefits Over Deep Learning

Neurosymbolic programming has multiple potential benefits over end-to-end deep learning. In general, by virtue of their modularity and use of symbolic primitives, neurosymbolic programs are closer to human-written code than deep networks. Because of this, neurosymbolic programming can provide a means for interpretable learning, especially when the learning algorithm being used is biased towards models that are "more symbolic" and structurally simpler.

Symbolic abstractions can also simplify the automated *analysis* of models. Over the last few years, there have been many efforts on algorithmic analysis of the safety and robustness of neural networks (Katz *et al.*, 2017; Gehr *et al.*, 2018; Anderson *et al.*, 2019). To a significant extent, these methods are adaptations of methods for quality assurance of traditional software. Unfortunately, analyzing even simple properties of deep neural networks is NP-hard, and scaling these analysis methods to real-world neural networks is difficult. In contrast, recent work on certifiable learning reduces (Anderson *et al.*, 2020) the safety and robustness analysis of certain specialized neurosymbolic models to the analysis of its (simpler, more compact) symbolic components. The latter task can be done relatively easily with existing software analysis techniques.

Neurosymbolic programming gives human users a mechanism to guide the learning process, and this can lead to more reliable learning. For example, in recent work in the reinforcement learning setting (Cheng *et al.*, 2019), a (deep) learning algorithm is given a human-provided function (program) that performs a task, albeit not optimally, and tasked with improving the performance of this program. The resulting learning process has lower statistical variance than one in which the task is entirely learned from data. If the prior is of reasonable quality, this strategy can lead to greater overall performance as well.

Also, a key advantage of high-level programming abstractions is that they tend to be compositional, i.e., allow the structured creation of larger programs using smaller modules. This enables a principled mechanism of *transferring* knowledge (Valkov *et al.*, 2018) across learning tasks: one can train a model in one task and reuse it in another task in the way a human programmer would use a library module. This mechanism is a generalization of a popular family of transfer learning techniques in deep learning, which commonly reuse network layers with frozen weights across tasks (Yosinski *et al.*, 2014).

Finally, higher-level programming abstractions can reduce the supervision effort needed for learning. In supervised learning settings such as image classification, one requires human users to label training inputs, and this can get expensive. In contrast, in the *data programming* paradigm (Ratner *et al.*, 2016), the user writes *labeling programs* that can automatically produce labels for inputs. Such programs are easy to write or automatically synthesize in many domains (Zhan *et al.*, 2020; Sun *et al.*, 2020). When they are available, they can drastically reduce the cost of learning.

1.3 Why Now?

The idea of combining neural and symbolic methods has a long history in AI research (Sun and Alexandre, 2013; Garcez *et al.*, 2002). In *knowledge-based neural networks*, an early example of a neurosymbolic model (Towell *et al.*, 1990), a set of hand-written symbolic rules were compiled into a neural network, which is then refined using data. It was shown that such neural networks are more data-efficient and tend to generalize better than classic neural networks. There were complementary efforts that extracted symbolic models, such as rules (Towell and Shavlik, 1993) and finite automata (Giles *et al.*, 1992), out of neural networks, essentially performing a form of program synthesis. Unifying these two strands of work, Shavlik, 1994 proposed a general learning framework in which an initial neural network, constructed using symbolic knowledge, is refined using data, and new symbolic knowledge is extracted from this refined network. These ideas were embodied, and taken further, in the Connectionist Inductive Learning and Logic

Programming system (Garcez and Zaverucha, 1999), which integrated logic programming and neural networks. These approaches certainly fit the category of neurosymbolic programming as defined in this paper.

While the first wave of neurosymbolic programming produced many interesting ideas, the practical impact of this line of work was limited. However, the current moment feels especially appropriate for resurrecting this area. For one, AI is increasingly deployed in real-world problems in which safety, reliability, and interpretability are important, and there is growing awareness about the limitations of pure deep learning in these problems. This opens up a window of opportunity for neurosymbolic methods. (Indeed, as we describe in Section 1.5, neurosymbolic methods are already making inroads into these tasks.)

Second, we now have access to much larger datasets and computational power than we did in the 1990s. This fact was key to the revival of neural networks as a research area, and it can help neurosymbolic programming as well. Finally, in the recent past, there has been significant progress on symbolic program synthesis, and new ways of coupling gradient-based and combinatorial search have emerged. As we show in this monograph, a new wave of research on neurosymbolic programming is already beginning to build on this progress.

1.4 Algorithmic Approaches

The fundamental challenge in neurosymbolic programming is that here, one must search through a combinatorial, and quickly exploding, space of program architectures. Worse, for each architecture explored in such a search, one must often perform high-dimensional continuous optimization to find optimal parameters for the neural modules appearing in the architecture. Nevertheless, over the last few years, researchers have discovered multiple new lines of attack on this problem, and some unifying themes are beginning to emerge in this area.

For example, one set of methods for neurosymbolic programming uses a neural network to *learn to synthesize programs*, i.e., direct a search process over program architectures (Balog *et al.*, 2016; Murali *et al.*, 2018). The network is trained using metalearning, from data that relates a set of tasks to programs that solve the tasks. Once a program architecture is generated, its lower-level parameters can be found using neural or symbolic methods.

A second category of methods, which we call *learning to specify* (Ellis *et al.*, 2018b), determine how to generalize incomplete or ambiguous task specifications to more complete specifications. These complete specifications are then used to direct a program synthesis process. A third category uses *neural relaxations* of a nonsmooth set of programs (Shah *et al.*, 2020). This space could consist of programs with completely different architectures. However, since a program is ultimately a representation of a function, the parameters of a neural network can (approximately) represent it. A final body of methods goes in the other direction, *distilling* a smooth neural function (Verma *et al.*, 2018) into a discrete program whose behavior approximately matches the network's. We discuss all of these methods in more depth in Section 2 and Section 4.

1.5 Applications

The algorithmic innovations sketched above are already beginning to impact real-world applications. Now we sketch some of these applications. Given the increased deployment of machine learning in domains in which trust and procedural reasoning are important, we expect many more such applications to emerge in the coming years.

Scientific Discovery. Building learning algorithms that discover new scientific hypotheses and guide experiments is a grand challenge in AI. Such algorithms must respect constraints known to hold in the world and produce outputs that scientists can interpret. This makes neurosymbolic programming a natural fit to this space.

As a concrete example, Cranmer *et al.* (2020) propose a method for *symbolic regression* — the automatic discovery of symbolic equations from data — and apply it to a task in cosmology (dark matter prediction). Also, several recent efforts use neurosymbolic programming in behavior analysis of lab animals. For instance, Sun *et al.* (2020) use a neurosymbolic method to embed videos of lab animals into lower-dimensional representations. Shah *et al.* (2020) use neurosymbolic programming to classify sequential animal behaviors. Zhan *et al.* (2021)

use neurosymbolic representation learning for interpretable clustering of such behaviors. Tjandrasuwita *et al.* (2021) learn interpretable programs that describe divergences between different human experts annotating behaviors.

Programming Systems. There has been significant recent interest in machine-learning-based assistants for software developers. Purely neural tools often struggle with understanding the complex, logical semantics of software, which are sensitive to even the smallest changes in syntax. Neurosymbolic programming is a natural way to overcome this issue given that symbolic methods have long been used successfully in program analysis.

For example, the Bayou system (Murali *et al.*, 2018) automatically completes Java methods given a few keywords that appear in the method. The Patois system (Shin *et al.*, 2019a) uses neurosymbolic programming to semantically parse text into code. Ellis *et al.* (2018b) simplifies graphics programming with a method to synthesize graphics code from a given picture.

Dialog Systems. Task-oriented dialog systems assist users with specific goals through a natural language interface. As digital assistants, they facilitate travel booking, database question answering, scheduling, and much more. The key challenge of task-oriented dialog is *state tracking* — identifying the user's intent and parameters in each dialog act, and using them to drive the system's actions. Fundamentally, dialog state is an intermediate symbolic representation that depends on complex, high-dimensional semantic context, namely dialog history and the underlying knowledge base or API. Thereby, neurosymbolic programming is a natural choice for modeling dialog state, successfully applied in many domains. For example, Andreas *et al.* (2020) design a calendar assistant in which scheduling actions, dialog corrections, and exceptions are represented as compositional programs, synthesized by neurosymbolic models in context.

Process Automation. The field of *robotic process automation (RPA)* aims to automate procedural GUI workflows to facilitate business digi-

The version of record is available at: http://dx.doi.org/10.1561/2500000049

tization and software testing. RPA agents interact with Web browsers, GUI applications, and APIs to accomplish the user's parameterized tasks. They are typically pretrained for each task using natural language commands, UI-grounded demonstrations, task completion rewards, or some combination thereof.

In RPA, the agent's state and action spaces are enormous – the current screen or Web page defines the state and the action space includes all possible interactions with its elements. Learning a robust and interpretable RPA agent is challenging even from grounded demonstrations as supervision. Instead, recent approaches leverage neurosymbolic programming and model the agent as a neurosymbolic task program. For example, Srivastava *et al.* (2020) combine neural language modeling with inductive program synthesis Gulwani *et al.*, 2017 to learn a generative model of programs that both guarantees consistency with the demonstrations and optimizes natural language alignment.

Robotics and Control. When designing policies or controllers for autonomous embodied systems, factors such as safety and data efficiency become paramount. For both low-level control and high-level planning problems, the standard practice has been to leverage symbolic domain knowledge (e.g., the governing equations of motion for the system, or an automaton representation of the high-level states) to design structured models that have certifiable guarantees, good generalization, or both (*e.g.*, Verma *et al.* (2019)). An emerging research direction is to automatically learn or discover the structure of the symbolic knowledge (*e.g.*, Xu *et al.* (2018)), which can be viewed as an instance of neurosymbolic programming.

1.6 Roadmap

The rest of this monograph is organized as follows. In Section 2, we give an overview of the landscape of recent research on neurosymbolic programming. Section 3 describes in some depth the main motivating goals for research in this area, along with concrete examples of how recent research is addressing these goals. In Section 4, we discuss some of the common themes in learning algorithms for neurosymbolic programs.

We conclude with a discussion of future challenges in the area in Section 5.

2

The Landscape of Neurosymbolic Programming

Research on neurosymbolic programming has two dimensions:

- The study of *neurosymbolic learning algorithms*, i.e., mechanisms to discover symbolic code or neurosymbolic programs using a combination of deep learning and symbolic program synthesis.

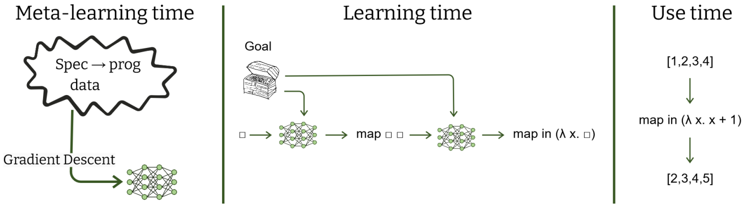
- The study of *neurosymbolic program representations*, obtained through the composition of symbolic code and neural modules in complex ways.

Over the past few years, the community has come up with a variety of innovations along both dimensions. In this section, we provide an overview of the landscape of this research.

2.1 Neurosymbolic Learning Algorithms

At the heart of neurosymbolic programming are a body of methods that search over symbolic program architectures using a combination of neural and symbolic techniques. Most of these ideas were originally presented in the context of learning symbolic code; however, they also apply to searching over programs with neural components. Now we sketch the main categories that these methods can be divided into.

2.1.1 Learning to Synthesize



Perhaps the most well-established paradigm for neurosymbolic program discovery is *Learning to synthesize*. The goal here is to generate programs from a high-level specification which may include both hard constraints — any requirement for which it is possible to unambiguously check whether it is satisfied or not — and soft constraints. Hard constraints can include input/output examples as well as formal specifications that describe properties the code must satisfy on all possible inputs. The soft constraints may include artifacts such as natural language specifications. Unlike the hard constraints, the soft constraints can only be interpreted in a probabilistic way, by drawing statistical correlations in the training data between the constraints and the generated code.

More formally, at learning time, the user is expected to provide *evidence* v consisting of hard constraints and soft constraints $v = (\varphi_h, \varphi_s)$. Given this evidence, the goal is to find the most likely program e that satisfies the hard constraints—indicate by the predicate $\varphi_h(e)$.

The starting point for this approach is a corpus of solved synthesis problems. Each of these includes the hard and soft constraints that the user would be expected to provide as inputs, together with the output program that the synthesizer would be expected to produce $D = \{(v_i, e_i)\}$.

During the *meta-learning phase*, the system learns how to map from a given specification to a program in a previously specified language which may include structure constraints restricting the set of valid programs. During the *learning phase*, the resulting trained synthesizer is then fed the particular specification for which the user wants to synthesize a program, and the synthesizer produces a program in return.

A standard way to formalize this approach is in terms of conditional probabilities. In this setup, one assumes that there is a conditional distribution over programs that depends on a body of evidence v, and a set of parameters ω. So for any program e, one can compute the probability of that program given the parameters and the evidence as $\mathbf{Pr}[e|v, \omega]$. This conditional distribution is generally represented as a neural network, where the parameters ω correspond to the parameters of the network.

During the meta-learning phase, the goal is to discover the optimal parameters such that

$$\omega^* = \arg\max_{\omega} \sum_{(v_i, e_i) \in D} \mathbf{Pr}[e_i|v_i, \omega]$$

Once the optimal parameters have been discovered, they can be applied during the use phase to find the most likely program given some particular evidence v.

$$e^* = \arg\max_{e \in \mathcal{L}} \mathbf{Pr}[e|v, \omega^*]$$

If the evidence includes both hard and soft constraints $v = (\varphi_h, \varphi_s)$, then it is not enough to find the most likely program given the learned parameters ω^*, it is also necessary to make sure the returned program satisfies the hard constraints

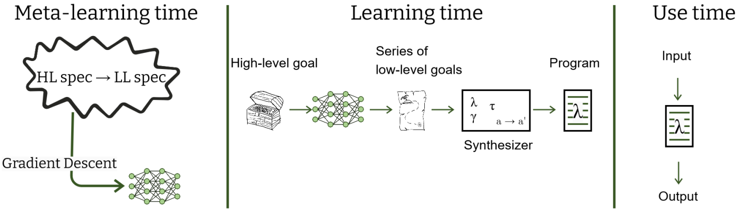
$$e^* = \arg\max_{e \in \mathcal{L} \ \wedge \ \varphi_h(e)} \mathbf{Pr}[e|v, \omega^*]$$

An important observation about the formula above is that while the formula seems to imply that one should first identify all the programs that satisfy the hard constraint and then select among them the one with the highest probability, existing algorithms can apply more sophisticated search techniques to sample the most likely programs from $\mathbf{Pr}[\cdot|v, \omega^*]$ and then check them against the hard constraints.

Many of the approaches proposed for this paradigm seek to emulate the ways in which humans engage in cognitively demanding tasks which usually require combining intuitive knowledge—“muscle memory“—with more formal knowledge. An early instance of this paradigm was DeepCoder (Balog *et al.*, 2016). In this system, the distribution $\mathbf{Pr}[\cdot|v, \omega^*]$

had a very simple form where it would simply determine what components could be part of the program and would give a zero probability to any program that had disallowed components. Another early example was Bayou (Murali *et al.*, 2018), which generated code snippets demonstrating the use of an API from soft constraints containing information about the APIs to use and hard constraints that required the resulting code to typecheck. Yet another example of this paradigm is RobustFill (Devlin *et al.*, 2017), where the conditional distribution is a more sophisticated attention model. For more details on these methods, see Section 4.3.

2.1.2 Learning to Specify



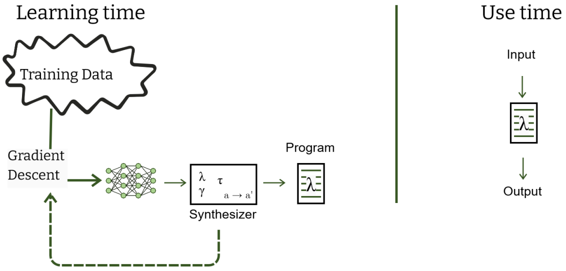
The paradigm of *learning to specify* is closely related to learning-to-synthesize. Here, the user also wants to generate symbolic code from a high-level specification, but it aims to address a different problem. Learning to specify aims to address a mismatch between the specification artifacts available, and the function that we want to produce. For example, an early instance of Learning to Specify was by Ellis *et al.* (2018b), who aimed to synthesize symbolic code from hand-drawings. In this case, there was a mismatch between the formal specifications expected by the symbolic synthesis and the hand drawings that the user would provide during the use phase.

The Learning to specify approach was to pre-train a neural network during the meta-learning phase that could map from hand drawings to a set of hard constraints that could then be used to drive a symbolic synthesis.

The same approach could also be followed switching the roles of gradient-based learning and symbolic synthesis. For example, suppose the goal is to train a neural network, which normally requires large amounts of labeled data, but only a small number of training labels are available. There is some work (Ratner *et al.*, 2016) that uses (manually written) *labeling programs* to automatically generate labels, enabling the application of deep learning. Recent work has begun to explore ways to use symbolic program synthesis to automatically generate such labeling programs. For more details on these methods, see Section 4.8.

One important aspect of meta-learning in Learning to specify systems is that they do not require a corpus of synthesis problems mapping specifications to programs, since the component that is meta-learned, the *specification translator*, does not have to be able to generate full programs, only specifications. So you only need either a corpus of high-level and low-level specifications, or a corpus of high-level specifications and an oracle that can tell whether the low-level specifications match the intent of the high-level ones.

2.1.3 Distillation



Ultimately, neural networks and symbolic code are just two different representations of functions from inputs to outputs (with different expressiveness). A recent set of methods go back and forth between these two kinds of representations as part of a neurosymbolic learning process.

In particular, one body of methods are based on *distillation* of neural networks into symbolic code. This approach is similar to learning-to-specify in that it aims to address a mismatch between the specification available and the desired artifact. Unlike learning-to-specify, however, distillation does not rely on a meta-learning phase to train a specification translator. Instead, the approach is to generate a neural network (or even a program with neural modules) that is a good match to the available specification and then to extract from that a symbolic program whose behavior approximates the neural function.

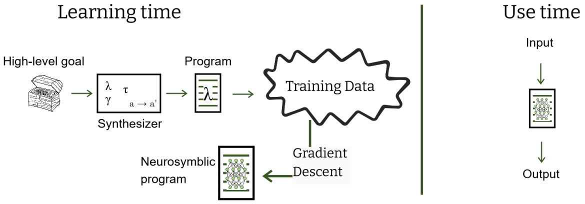
More formally, given a task φ, the first step is to learn a model f that solves the task, and then derive a symbolic program e^* that most closely approximates the learned model

$$e^* = \arg\min_{e \in \mathcal{L}} Dist(e, f)$$

One potential problem with this simple approach is that there may be many neural models f that solve the task almost as well, but not all of them can be approximated equally well with a symbolic program. Some variations of this approach address this by introducing a feedback loop, where the symbolic program is then used to introduce a regularization penalty that is used to find a new neural model that is more likely to be representable as a program, and the process is repeated with that new model.

An early example of Symbolic Distillation is Programatically Interpretable Reinforcement Learning (Pirl) by Verma et al. (Verma *et al.*, 2018), where the goal was to learn symbolic code that could control a race car given a simulator environment for the car. Deep reinforcement learning can do a god job of learning a controller for the car given this environment, but producing symbolic code from a complex simulation environment is beyond the capabilities of traditional program synthesizers. Pirl approaches this difficulty by first using deep reinforcement learning to produce a neural representation of the controller from the simulation environment, and then distilling this neural network into symbolic code using imitation learning. For more details on this and related methods, see Section 4.4.

2.1.4 Relaxation



Another body of methods take the opposite path of *relaxing* symbolic programs into neural networks. The motivation here is that while one needs combinatorial methods to search over symbolic code, the neural relaxations can be trained using more scalable gradient-based techniques. For example, PROPEL (Verma *et al.*, 2019), a successor to the PIRL approach, utilizes this perspective. Here, a purely symbolic controller, discovered using a distillation step, is relaxed into a controller with neural modules (in principle, this program could be just a single neural network). The relaxed controller is then trained further using deep reinforcement learning. The NEAR (Shah *et al.*, 2020) method applies a neural relaxation to direct a discrete search over program architectures. A followup effort by Cui and Zhu (2021) goes further, using a relaxation to reduce the entire architecture search problem to a continuous optimization task. For more details on these methods, see Section 4.5.

2.1.5 Beyond the basic categories

The approaches we have described so far in this section represent broad classes of neurosymbolic algorithms. Later in the paper, we provide details of concrete algorithms in each of these categories.

It is important to note that many practical algorithms can incorporate elements from many of these categories. For example, an algorithm could use learning to-synthesize techniques to produce a symbolic program, followed by a relaxation step to replace some elements of the program with symbolic components. An algorithm could also use a learning-to-synthesize approach to make distillation more efficient. Section 4 provides more concrete examples of these combinations.

2.2 Neurosymbolic Representations

Now we discuss program representations that weave together neural components and symbolic code. The holy grail for research on neurosymbolic programming would be a general learning algorithm that could efficiently generate arbitrary combinations of neural components and symbolic code. However, such an algorithm has not been developed, and seems hard given the joint numerical and combinatorial optimization that would be required. However, the community has had important successes with a few specialized approaches tailored to particular combinations of program structure and neural components.

2.2.1 Symbolic-after-Neural Pipeline

This is a particular combination where the inputs are preprocessed by a collection of neural components, and the results are then passed to symbolic code, generally either a functional or logic program.

For example, consider a program meant to detect whether an image has a cat on top of a table. Such a program could be written by relying on neural components to identify cats and tables, combined into a symbolic program that checks that both elements are present in the image and that the cat is indeed on top of the table.

The HOUDINI approach (Valkov *et al.*, 2018) uses this idea to perform structured computations over natural inputs, for example, to count the number of cars in a photograph. The neural components can do a good job identifying cars, but learning to count is something more effectively done via symbolic computation. Murali and Madhusudan (2019) have also explored similar ideas in a different context.

In general, this approach requires some pre-training of the neural components, which then must be further optimized for the particular context where they will be applied in the program. See Section 4.6 for more details.

2.2.2 Neural-after-Symbolic Pipeline

It is also possible to set up a pipeline in the opposite direction, with inputs being fed first through a set of synthesized program components, and the results then fed to a neural network.

With older machine learning methods, which relied heavily on feature engineering in order to produce good results, this was a way to support the discovery of new features. For example, back in 2012, Cheung *et al.* (2012) showed that you could improve upon a simple linear classifier by leveraging synthesized features.

With deep learning, there is less of a need for feature engineering, but there are still contexts where deep learning benefits from pre-processing the input. This creates an opportunity in cases where this pre-processing involves the kind of text or data-structure manipulations that program synthesis techniques excel at.

2.2.3 Neural Module Networks

Neural Module Networks (Andreas *et al.*, 2016b) constitute another neurosymbolic paradigm where a collection of neural modules are treated as components in a programming language which can be composed together to provide high-level functionality.

In the original work by Andreas *et al.* (2016b), the program-like composition was extracted directly from a text specification through traditional parsing, and the modular components are learned by training from data on the end-to-end behavior of different compositions. The idea has been expanded in more recent work that aims to learn not just the modules, but also how to compose them. For example, the work on dynamic neural module networks (Andreas *et al.*, 2016a) learns a *network layout predictor* that determines how the modules will be composed. The starting point for this composition is still provided by a traditional parser, but there is a neural network that learns to score the possible parses to select the best composition. More recently, Alet *et al.* (2018a) have explored *Modular meta-learning*. In this approach, there is a meta-learning phase where the system learns a set of models and their composition to solve a given set of tasks, and then the models are frozen, so that the use-phase just looks like a traditional synthesis problem where the goal is to search for a composition of the existing modules that solves the task.

2.2.4 Algebraic Compositions

Another way of combining neural and symbolic programs is to allow them to run independently and then aggregate their results using an algebraic operator $\oplus$. The resulting hybrid program h can be described as

$$h(s) = e(s) \oplus f(s)$$

For example, one of the simplest instantiations of the operator $\oplus$ is vector addition. One way to interpret this instantiation is that the result of the neural program is added as a correction to the result of the symbolic program. This idea originally appeared in the CORE-RL approach to reinforcement learning Cheng *et al.* (2019). This effort considers policies that additively combine a symbolic program and a neural network, and shows that the resulting policy preserves some of the guarantees of the symbolic program. One weakness of the approach is that the symbolic component of the policy is static and human-written. The PROPEL approach (Verma *et al.*, 2019), mentioned earlier, overcomes this weakness by allowing the symbolic component of the policy to be automatically synthesized (using a combination of distillation, relaxation, and gradient-based learning).

2.2.5 Neurally Accelerated Symbolic Programs

There is strong precedent for using machine learning to help guide the execution of a program in ways that do not affect its functional correctness but help it run more efficiently. For example, in the SAT community, there is work dating back to the mid 2000s that used machine learning to set the parameters of a SAT solver that control its internal heuristics (Hutter *et al.*, 2007). We do not consider such applications on their own as an instance of neurosymbolic programming, but they can be if parts of the program itself are synthesized as well.

More formally, the goal is to synthesize a neurosymbolic program h, that is a composition of a neural network f (interpreted as a performance-enhancing heuristic) and a symbolic program e,

$$h(x) = e(x, f)$$

where the symbolic program $e(x, f)$ will invoke the neural network f internally, but for any input x, its output is guaranteed to be correct regardless of the values provided by those invocations of h. An important special case is when f is invoked only once on the input x, so h can be represented as

$$h(x) = e(x, f(x))$$

There is a range of possibilities for this approach depending on how much work is done by e relative to the neural heuristic f. For example, at one extreme, a neural heuristic is simply fine-tuning some parameters in otherwise symbolic code, like the parameters in a SAT solver that would otherwise just be hard-coded to some value that happens to work well for most benchmarks. At the other extreme, the oracle is actually doing most of the work, and the symbolic code is acting simply as a monitor, intervening only when it sees the oracle performing a bad action.

It is important to note that the line separating this paradigm from algebraic composition can be blurry in some cases. For example, consider the recent work on safe learned controllers (Zhu *et al.*, 2019; Fulton and Platzer, 2018; Anderson *et al.*, 2020). The basic idea is to combine a learned controller with a synthesized function that computes a set of safe actions at each state. If the action proposed by the controller is in the set of safe actions, it is simply taken. If it is not, then an action from the safe set is taken instead.

Such a system can be seen as synthesizing a neural guided programs where for a given state s, $a = f(s)$ computes an action a, and then $e(x, a)$ computes a set of safe actions and returns a if it is in that set or its own safe action otherwise, so the program is guaranteed to take a safe action regardless of the value a. On the other hand, it can also be seen as a form of algebraic composition

$$h(s) = e(x) \oplus f(x)$$

Where the program returns its set of safe actions, and then the combination operator $\oplus$ is an if-then-else combinator that chooses between the result of the neural policy or one of the safe actions.

3

Motivating Goals

Now that we have given an overview of neurosymbolic programming, we elaborate further on the main motivating goals for research in this space. We also present some concrete examples of research driven by these goals.

3.1 Generalization and Sample Efficiency

Generalization and sample efficiency are foundational concepts in machine learning. Generalization refers to the ability of a learned system to predict well on new test cases outside of its training set (under a suitable definition of "well"). Sample efficiency refers to the number of training samples needed to reach a target learning goal (e.g., being able to generalize well). In most practical settings, the two notions are virtually synonymous with each other.

Historically, generalization and sample efficiency have largely been considered when evaluating performance (as measured by a standard loss function) on test examples sampled *independently and identically distributed* (i.i.d.) from the same distribution that generated the training examples, i.e., the so-called i.i.d. setting. In other words, how many training examples (sampled i.i.d. from the data distribution) does one need to train a predictor that achieves a target accuracy on unseen examples drawn i.i.d. from the data distribution?

A canonical theoretical framework to study this setting is the PAC ("Probably Approximately Correct") learning framework (Valiant, 1984). The basic premise is to study the following quantity

$$\mathbf{Pr}\left(\forall P \in \mathcal{L}: \left|R(P) - \hat{R}_n(P)\right| \leq \epsilon\right) \geq 1 - \delta, \tag{3.1}$$

where $\hat{R}_n$ is the *empirical risk* (with n denoting the number of training examples), R is the true population risk (expected loss over the true test distribution), ϵ quantifies "approximately", and δ quantifies "probably". A guarantee of this nature typically expresses n as a function ϵ and $1/\delta$, e.g., $n = O([\texttt{complexity}(\mathcal{L}) + \log(1/\delta)]/\epsilon^2)$, for some measure of the complexity (or capacity) of the hypothesis class $\mathcal{L}$. In other words, the more complex $\mathcal{L}$ is, then the more training data is required in order to generalize from the empirical training set to the true population distribution. In this context, the study of regularization and inductive bias in improving generalization and sample efficiency is well studied, and one can think of neurosymbolic architectures are providing a form of regularization.

More recently, there has been increased interest in studying generalization (and sample efficiency) when the target or test domain is not the same data distribution as the training domain (Bickel *et al.*, 2009; Dudík *et al.*, 2014; Chen *et al.*, 2016; Beery *et al.*, 2018; Taori *et al.*, 2020). For instance, in sequential decision making scenarios (such as those typically solved using reinforcement learning), the training environments may be substantially different from the test environments (James *et al.*, 2019; Liu *et al.*, 2020). Other examples include (adversarially) perturbed inputs from training examples, e.g., visually imperceptible modifications to the pixel values of an image can change a prediction of a cow to that of airplane. The growing interest in this area reflect the increasing recognition that real-world learning systems will encounter test examples that are not sampled i.i.d. from the training distribution. One can also relate the study of generalization and sample efficiency to transfer learning (i.e., efficiently generalizing to new domains) (Wang and Schneider, 2014), discussed further in the next section. In all the above cases, the use of neurosymbolic primitives offers the possibility of greatly improved generalization power and sample efficiency. In particular, the function class that one learns over is a neurosymbolic one, that ideally provides useful inductive biases to improve generalization and sample efficiency.

For example, the CoreRL (Cheng *et al.*, 2019) and Propel (Verma *et al.*, 2019) approaches to reinforcement learning are examples of how neurosymbolic programs can aid generalization. In these methods, one considers policies that are neurosymbolic programs of the form $h(s) = e(s) \oplus f(s)$, where e is a symbolic program and f is a neural network. The symbolic component of such a program is expressed in a human-written DSL and therefore captures a potentially useful inductive bias. The papers demonstrate the value of this bias theoretically — by showing that it can lead to lower learning variance — as well as empirically.

Zhan *et al.* (2021) demonstrate the generalization benefits of neurosymbolic programming in unsupervised representation learning. The goal here is to simultaneously learn an encoder and a decoder that map data to latent vector representations and back. The encoder has a neural module and a symbolic module, each of which constructs part of the learned representation. This approach is shown to outperform purely neural approaches at efficiently learning latent spaces that are semantically meaningful, i.e., separated into well-defined clusters consisting of encodings of semantically similar inputs.

A final category in this topic is programmatic supervision (Ratner *et al.*, 2016; Zhan *et al.*, 2020), which is closely related to the concepts of self-supervision (Hendrycks *et al.*, 2019; Chen *et al.*, 2020a) and data augmentation (Taylor and Nitschke, 2017). Self-supervision and data augmentation have become popular ways to improve sample efficiency of representation learning. In self-supervision, the basic idea is use automatically generated auxiliary tasks as a form of supervision. Common examples arising in images include inpainting, and denoising. Using such trained representations can often lead to significant annotation savings on the down-stream prediction task (e.g., object classification), thus leading to improved sample efficiency. In data augmentation, the basic idea is to procedurally create modified variants of training examples (e.g., rotating an image) in a way that does not change the supervised label. More recently, the concept of programmatically generated (self-)supervision tasks has proven effective (Sun *et al.*, 2021). In other words, the program provides supervised guidance to a (deep) learning system, rather than directly constraining the learning system by using a neurosymbolic function class.

In most methods of this sort, the programs are hand-written rather than automatically synthesized. However, recent work has begun to integrate program synthesis with such programmatic supervision. We will elaborate more on this topic in Section 4.8.

3.2 Transfer and Abstraction

Transfer learning systems aim to distill useful knowledge from a corpus of training tasks, which can then be generalized to testing tasks—ideally, even if those testing tasks are not drawn from the training distribution. Within neurosymbolic programming, transfer learning schemes typically work by first discovering reusable modules. These modules can be either small neural nets or fragments of symbolic code, which are induced given a training corpus of synthesis problems. They then transfer these reusable modules to test problems by deploying them in the test-time program synthesizer.

For instance, the Houdini (Valkov *et al.*, 2018) system learns neural modules which are shared across synthesis tasks. It takes as input a collection of synthesis problems, each of which it solves by finding a program with symbolic control flow operating over neural modules. At the end state of learning, the system produces neurosymbolic programs solving each synthesis task, *and also* outputs the set of neural modules which can be transferred for solving further tasks in the future. As a concrete example, Houdini can solve a problem involving detecting digits in images, for which it trains a reusable neural module. Later, it can transfer this neural module to perform a task such as counting how many times a given digit occurs in a list of images–which interviews both programmatic, procedural reasoning with low-level perception.

Ideally, such modules should correspond to reusable high-level abstractions. In the Houdini example, the system reused learned object detectors (Valkov *et al.*, 2018). In a purely symbolic functional programming domain, a transfer learning system might learn and reuse recursive control flow patterns (Ellis *et al.*, 2020). In a semantic parsing domain, the system might learn and reuse chunks of code which execute common English phrases (Shin *et al.*, 2019a). Thus, transfer learning via these abstractions can be an end goal to itself: at the end of learning, such systems will have constructed reusable, executable abstractions which can be used for virtually any downstream neural programming task. In principle these abstractions could also be reused by human programmers, although to to the best of our knowledge this human-facing application has not been explored yet to date.

While inducing transferable abstractions can be an end to itself, it also connects to and supports other goals of neural programming. It can aid sample efficiency, particularly when the reused modules are neural networks, because the reused modules do not need to be learned from scratch. This boost to sample efficiency was analyzed by *e.g.* Liang *et al.* (2010), which showed that learning reusable subtrees of text editing programs causes those programs to generalize better to held-out inputs.

This kind of transfer learning can also reduce computational complexity by reducing the depth of search. For symbolic abstractions, this works because each abstraction is a fragment of code, so using it implicitly calls many other pieces of code, effectively taking multiple steps at once through the symbolic search space. For neural modules, this works because the module does not need to be retrained from scratch, effectively reusing statistical knowledge distilled from prior training data. By reducing both the search depth and the sample complexity, this approach is then conceptually related to the 'options' framework within reinforcement learning (Sutton *et al.*, 1999).

Transferring abstractions can also help interpretability by discovering more compact ways of expressing program solutions. This connection to interpretability was noted by *e.g.* Ellis *et al.* (2020) and Dechter *et al.* (2013), which showed that many learned abstractions correspond to human-understandable functions, such as functional programming idioms like *map* and *fold*, and Boolean operations like XOR *etc.* Figure 3.1 (top) shows example output of one such system inducing transferable programs for physics equations. Intriguingly, the same benefits to interpretability seem to hold for transferring modular neural networks.

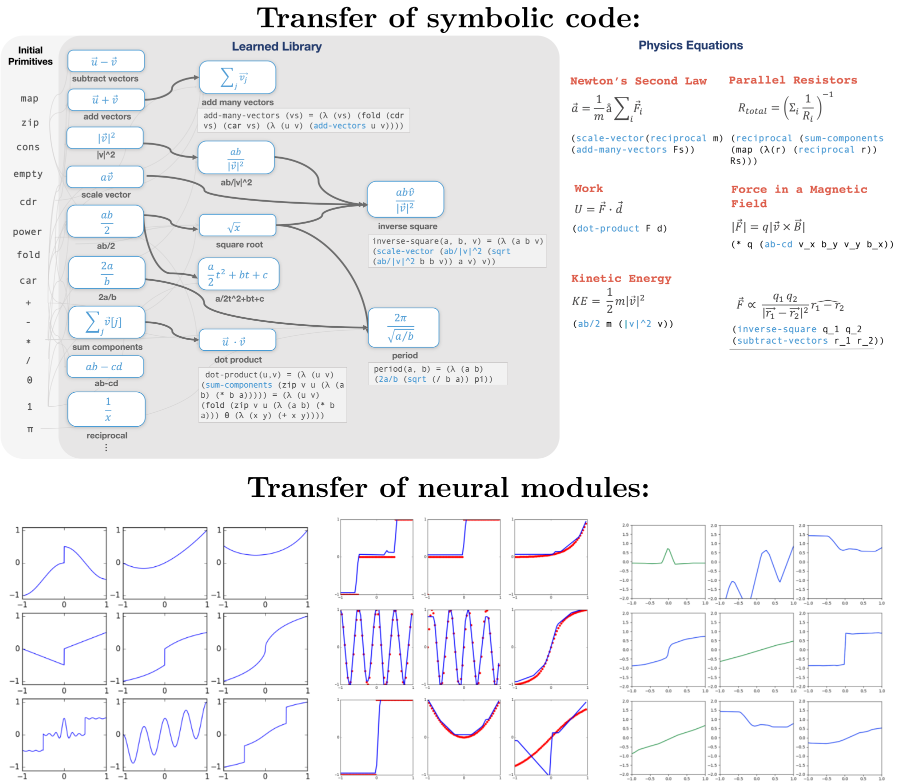
This is demonstrated by Alet *et al.* (2018b) (Figure 3.1 bottom), which learns neural modules for symbolic regression and other domains such as human pose estimation. Without supervision, this system learns (on symbolic regression) interpretable basis functions such as sinusoids, while learning (on human pose estimation) canonical pose-components that are independent of the human doing the posing.

3.3 Interpretability

With increasing development and deployment of machine learning techniques in various safety-critical applications, the *interpretability* of these models is becoming ever so important. Interpretable models not only allow for a better understanding of the logic of the learnt function, but are also extremely helpful for debugging and verifying their functionality.

Several approaches have been proposed recently for interpreting machine learning models. One common approach is to attribute importance to different features for each prediction. Integrated Gradients (Sundararajan *et al.*, 2017) (IG) uses an axiomatic approach to design an attribution technique that computes a feature attribution for the model prediction on an input x by accumulating the gradients at all points along the path from a baseline input x' to the input x. These attributions can then be used to construct a saliency map for interpreting importance of different features for a prediction. TCAV (Kim *et al.*, 2018) (Testing with Concept Activation Vectors) is a technique to calculate the degree to which a user-defined high-level concept (trained as a concept vector) influences the model's prediction for a given class, allowing to understand the model prediction in terms of high-level concepts. Some approaches use symbolic solvers such as SMT solvers for identifying minimal input regions that are critical for a model prediction (Zhang *et al.*, 2018; Sahoo *et al.*, 2020).

Unlike these approaches that compute input feature attribution, high-level programs (symbolic or neurosymbolic) present an alternate approach to interpret the learned models. Such programs can represent learned functions in terms of compositions of abstract primitives, which makes them more interpretable compared to traditional deep learning models. This structured representation of learned functions can be



Transfer of symbolic code:

```
add-many-vectors (vs) = (λ (vs) (fold (cdr
vs) (car vs) (λ (u v) (add-vectors u v))))
```

```
inverse-square(a, b, v) = (λ (a b v)
(scale-vector (ab/|v|^2 (sqrt
(ab/|v|^2 b b v)) a v) v))
```

```
dot-product(u,v) = (λ (u v)
(sum-components (zip v u (λ (a
b) (* b a))))) = (λ (u v)
(fold (zip v u (λ (a b) (* b
a))) 0 (λ (x y) (+ x y))))
```

```
period(a, b) = (λ (a b)
(2a/b (sqrt (/ b a)) pi))
```

```
(scale-vector(reciprocal m)
(add-many-vectors Fs))
```

```
(reciprocal (sum-components
(map (λ(r) (reciprocal r))
Rs)))
```

```
(dot-product F d)
```

```
(* q (ab-cd v_x b_y v_y b_x))
```

```
(ab/2 m (|v|^2 v))
```

```
(inverse-square q_1 q_2
(subtract-vectors r_1 r_2))
```

Transfer of neural modules:

Figure 3.1: Examples outputs of transfer learning, both for symbolic code (**top**) and for neural modules (**bottom**). At **top** we show the output of DreamCoder (from Ellis *et al.*, 2020) when trained to synthesize symbolic code. The system is tasked with synthesizing programs fitting random data simulated from physics equations such as those shown at the right. To do this, it starts with a functional programming language (left), but does not start with vector algebra. In synthesizing these programs it assembles a library of reusable procedures (middle). These reusable procedures include basic vector algebra and common patterns found in physics equations, which can then be *transferred* when solving new synthesis tasks in the future. **Bottom**: Example output of Modular Metalearning (taken from Alet *et al.*, 2018b). This system is tasked with synthesizing neurosymbolic programs fitting real-valued curves such as those shown at the left. Each of these training functions is built from a small number of basis functions. The middle shows neural modules learned by this system, with the closest basis function plotted in red. They have a high degree of agreement with the original ground truth basis functions that generated the data. Contrast with baseline (right): Modular Metalearning correctly discovers the true basis, which is both more interpretable and more useful for transfer.

```
if ($s[\texttt{tangle}] < 0.011$ and $s[\texttt{tangle}] > -0.011$)
        then $\mathrm{PID}_{\texttt{rpm},0.45,3.54,0.03,53.39}(s)$ else $\mathrm{PID}_{\texttt{rpm},0.39,3.54,0.03,53.39}(s)$
```

Figure 3.2: A learned programmatic policy for acceleration, automatically discovered by Propel (Verma *et al.*, 2019). This program invokes PID controllers $\mathrm{PID}_{j,\theta_S,\theta_P,\theta_I,\theta_D}$, where j identifies one of 29 sensors that provide inputs to the controller, θ_S is the target value (setpoint) that the controller aims for, and θ_P, θ_I, and θ_D are respectively the real-valued coefficients of the proportional, integral, and derivative terms in the controller. Note that the program only uses the sensors `tangle` and `rpm`. The `tangle` sensor, which measures the position of the car relative to the track axis, is used to decide which controller to use ($s[\texttt{tangle}]$ represents the most recent reading from sensor `tangle`). Note that only the `rpm` sensor is needed to calculate the acceleration. The program has the following interpretation: *If the car aligned with the center of the track, then aim for a high acceleration value, otherwise aim for a lower one.*

considered similar to writing programs in a high-level language, whose functionality can be understood by understanding the semantics of the underlying language primitives. This representation also opens up the possibility of using decades of research in program verification and debugging tools to verify various properties about the learnt functions.

Pirl (Verma *et al.*, 2018) and Propel (Verma *et al.*, 2019) are two neurosymbolic programming approaches motivated by interpretability. These methods address the reinforcement learning (RL) setting, in which the goal is to discover a policy that maximizes an agent's long-term reward. In mainstream deep RL, these policies are represented as neural networks. In contrast, policies in Pirl and Propel are required to have a parsimonious representation in a domain-specific language (DSL) that is assumed to be restricted enough to ensure interpretability.

Concretely, Pirl and Propel consider spaces of functional programs in which classic controllers — for example, Proportional-Integral-Derivative or PID controllers — can be used as "library modules". Programs from these spaces are discovered using a combination of deep RL and symbolic program synthesis (for more details, see Section 4.4). Both methods are evaluated in an RL domain in which the goal is to drive a car around a race track. Figure 3.2 shows a program for controlling the accelerator of this car that was automatically discovered using Propel.

$$\begin{aligned}
&\mathbf{map}(\lambda x.\ \mathbf{if}\ \mathit{DistAffine}_{[.0217];-.2785}(x)\\
&\qquad \mathbf{then}\ \mathit{AccAffine}_{[-.0007,.0055,.0051,-.0025];3.7426}(x)\\
&\qquad \mathbf{else}\ \mathit{DistAffine}_{[-.2143];1.822)}(x))
\end{aligned}$$

Figure 3.3: A program, learned using the NEAR approach (Shah *et al.*, 2020), for classifying a "sniff" action between two mice in the CRIM13 dataset. Here, *DistAffine* and *AccAffine* are functions that first select the parts of the input that represent distance and acceleration measurements, respectively, and then apply affine transformations to the resulting vectors. In the parameters (subscripts) of these functions, the brackets contain the weight vectors for the affine transformation, and the succeeding values are the biases. The program 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.*

The NEAR approach (Shah *et al.*, 2020) uses symbolic programs as interpretable classifiers for sequential behavior data, and discovers these programs using a combination of heuristic search and neural relaxations of discrete sets of programs (see Section 4.5). Figure 3.3 shows an example classifier for mice behavior, in the CRIM13 dataset, discovered by this approach.

There is also a line of work that combines deep learning and *symbolic regression*, i.e., the discovery of symbolic equations from data, with the motivation of interpretability. These methods are a form of neurosymbolic programming because equations are a simple kind of programs. An interesting example of such work is AI Feynman (Udrescu and Tegmark, 2020), an approach that repeatedly applies a series of tactics, including neural network training and enumeration over a space of symbolic expressions, to discover symbolic equations from data. For another example, Cranmer *et al.* (2020) give a method for learning equations that first trains a graph neural network on a dataset, and then distills various components of this network into symbolic subexpressions of the overall equation. This method is applied to, among other things, data from a gravitational simulation of the universe, and used to recover an interpretable equation that determines the excess amount of matter in "dark matter halos".

3.4 Safety

Safety is a central concern in recent machine learning research. There is for multiple reasons. First, machine learning is now routinely deployed in real-world applications in which failure has high human and financial costs. Second, deep models are now known to fail badly on inputs that are even slightly outside the training data distribution. Third, deep networks cannot be analyzed using the mechanisms that programmers traditionally use to reason about the software that they write. Instead, methods that provide *automatic assurance* of models are required, and the design of such methods is a challenging open problem.

The problem of safe machine learning comes in two flavors. In the first, one requires the final learned function to satisfy a constraint (a *requirement*) over the function's inputs and outputs. Recent work has considered many specific forms of such requirements, including robustness to adversarial perturbations, fairness, differential privacy, avoidance of "bad" states, and reachability of "good" states. In general, we can pose the task of learning modulo such requirements as a constrained optimization problem

$$f^* = \arg\min_f J(f) \quad \text{s.t.} \quad f \in \mathcal{S}, \tag{3.2}$$

where f ranges over learnable functions (hypotheses), $\mathcal{S}$ is the set of functions that meet the safety requirement, and $J(f)$ is the loss function that the learning algorithm seeks to minimize.

The second formulation applies to scenarios that are especially safety-critical, for example, when the learning algorithm is deployed on a physical robot whose actions can have real-world consequences. Here, every hypothesis that the learner considers during training is required to obey the safety requirements.

In either case, one needs a mechanism for determining the extent to which specific hypotheses are safe. In most of the safe learning literature, this mechanism is statistical, i.e., based on estimation using samples from a model's input space. While principled and algorithmically simple, such mechanisms often need intractably many samples to give strong guarantees. More critically, they fail to guarantee safe behavior on adversarial inputs coming from outside the training distribution.

Automatic formal verification is an increasingly popular response to this challenge. Here, a statistical model (in particular, a neural network) is viewed as a program in a low-level language. Classic formal methods such as abstract interpretation and bounded model checking are now used to prove or falsify this program's requirements. Unfortunately, scalability is a basic challenge with these methods. Symbolic verification tends to only be efficient for programs that are small or have a compositional structure (enabling the verification process to be easily decomposed into subproblems). Neural networks do not satisfy either property, and verification of state-of-the-art neural networks for complex tasks remains a faraway goal. Invoking such verifiers from within the learning loop, as would be required if one desires safety during training, is simply infeasible.

An important benefit of programmatic models is that they make this verification problem more tractable. As described in Section 3.3, it is sometimes possible to learn purely symbolic code that also have a high performance in the learning objective. Such programs tend to be parsimonious and compositional, and this makes them easier to verify. Also, a high-level programming language facilitates *abstraction*. For example, suppose that, given a program with neural and symbolic modules, we replace the decisions made by the neural component with nondeterministic choice. We can now use standard formal methods to determine whether the resulting symbolic program is safe under all possible instantiations of this choice. If it is, then the original neurosymbolic program is safe as well.

Safety at Convergence. So far, most work on formally verified program learning has happened in the RL setting. The goal in these efforts is to learn programmatic policies that come with formal proofs of safety. In particular, the Pirl (Verma *et al.*, 2018) and Propel (Verma *et al.*, 2019) approaches for programmatic RL (mentioned in Section 3.3) consider formal verification of the learned program. Also, the closely related Viper (Bastani *et al.*, 2018) approach learns decision tree policies using neurosymbolic methods with the primary objective of formal verification. Properties considered here include:

- *Smoothness:* Small perturbations to the (continuous) inputs of a policy do not substantially change its (continuous) outputs (Verma *et al.*, 2018; Verma *et al.*, 2019).

- *Boundedness:* Assuming the inputs of a policy are within a reasonable bound, the policy outputs (continuous) are bounded as well (Verma *et al.*, 2018; Verma *et al.*, 2019).

- *Stability:* The learned policy is provably stable (Bastani *et al.*, 2018).

- *Functional correctness:* The learned policy for a game (specifically, Pong) leads to provable victory under reasonable assumptions about competing players (Bastani *et al.*, 2018).

Pirl, Viper, and Propel perform verification in a post-learning phase. A problem with this strategy is there is no obvious course of action is provided if the learned policy fails the requirements. One could discard the policy and learn a new one, but there is no guarantee that the newly learned policy will satisfy the requirement either. In response to this challenge, Zhu *et al.* (2019) give a method for incorporating a safety constraint into the program synthesis process. More details on this method are available in Section 4.4.

Zhu et al.'s paper has a second interesting dimension. Extending prior work on *shielded, or simplex, control* (Alshiekh *et al.*, 2018), the paper extends its safety guarantees to a certain class of neurosymbolic policies. These policies have a neural module $g(x)$ and a provably safe symbolic module $f(x)$ (a *safety shield*). By default, the neural component $g(x)$ is executed; however, before actually executing this module, a predicate known as a *safety monitor*, is used to check if its execution can lead to a potentially unsafe state. If so, the shield $f(x)$ executed instead. The overall policy $h(x)$ can be described in code as

$$h(x) = \textbf{if } (\psi(x, f(x))) \textbf{ then } f(x) \textbf{ else } g(x). \tag{3.3}$$

Zhu et al. give a way to synthesize the monitor and the shield together such that $h(x)$ is safe no matter what $g(x)$ is. Importantly, this construction does not require explicit formal verification of the neural network $g(x)$. Thus, the approach can be seen to establish safety through an *abstraction* of the neurosymbolic program that is ultimately deployed on the agent.

Safety during Training. *Safe exploration*, or the problem of ensuring safety during learning, is a well-studied problem in RL (Garcıa and Fernández, 2015; Achiam *et al.*, 2017). Most approaches to the problem estimate the safety of policies statistically (Moldovan and Abbeel, 2012; Chow *et al.*, 2018; Achiam *et al.*, 2017; Chow *et al.*, 2018). However, there are also recent efforts (Alshiekh *et al.*, 2018; Fulton and Platzer, 2018; Anderson *et al.*, 2020) that attack the problem using formal methods.

Broadly, these approaches apply the idea of shielded control during the training process. Specifically, Alshiekh *et al.* (2018) and Fulton and Platzer (2018) construct spaces of formally verified shields before the learning process starts. During the training process, a monitor predicate observes the learner, forbidding all actions that cannot result from one of these safe policies. If the learner is about to take a forbidden action, a shield is executed instead.

One issue with these methods is that the shields and safety monitors are constructed a priori and do not improve as learning progresses. The recent Revel framework (Anderson *et al.*, 2020) addresses this weaknesses using by maintaining a neurosymbolic policy representation as in Equation 3.3, but allowing updates to both the neural and the symbolic components of the policy as training progresses. See Section 4.4 for more details.

3.5 Procedural Reasoning

We say that an AI system exhibits *procedural reasoning* when it accomplishes multi-step tasks that involve program-like abstraction mechanisms such as looping, functional abstractions, and symbolic data structures. Procedural reasoning is one of the key challenges of contemporary AI systems, as evidenced by their poor *inductive generalization* on a wide range of procedural tasks such as database question answering and embodied instruction following (Keysers *et al.*, 2020). Neurosymbolic programs offer a task representation that inductively generalizes *by construction*, by the virtue of retaining many of its task parameters in explicit symbolic form. In addition, they represent a form of explicit System-2 reasoning as a logical and algorithmic procedure (Kahneman, 2011). This has made neurosymbolic programming effective as a means to solve algorithmic tasks such as question answering (Amizadeh *et al.*, 2020) or learn representations of human-made semi-structured objects such as graphical illustrations (Young *et al.*, 2019).

Question answering is a quintessential reasoning task – it requires an AI system to translate a natural language question into an interaction plan over an underlying knowledge base (*e.g.* a relational database, a knowledge graph, an image or its scene graph) that retrieves or computes the answer to the question. While purely-neural approaches that emit a symbolic program (*e.g.* SQL for a relational database or SPARQL for a knowledge graph) are effective, they often fail to inductively generalize to different parameterizations of the same question or exhibit other *compositional generalization* (Keysers *et al.*, 2020). They also require limited symbolic DSLs for interacting with non-discrete KB modalities such as images and video.

Neural module networks (Andreas *et al.*, 2016b; Hu *et al.*, 2017; Liu *et al.*, 2019a) are the first neurosymbolic approach to visual question answering. Their key idea is to represent each visual operation in the target program (*e.g.* `Find[apple]`) as a neural module with learnable parameters, which compose into an end-to-end differentiable program to answer a question. Figure 3.4 shows an example.

Question answering via NMNs typically involves two stages. In the first, semantic parsing maps the question into a program, which includes the required operators and their symbolic parameters. In the second, each operator is replaced with its neural module representation, and the resulting neurosymbolic program is executed to obtain the answer. The two phases are connected via a discrete decision, which complicates end-to-end training of neural modules. They can be either pre-trained in a separate phase using supervised programs (Vedantam *et al.*, 2019; Amizadeh *et al.*, 2020) or trained end-to-end using gradient estimators such as Gumbel-Softmax (Liu *et al.*, 2019a).

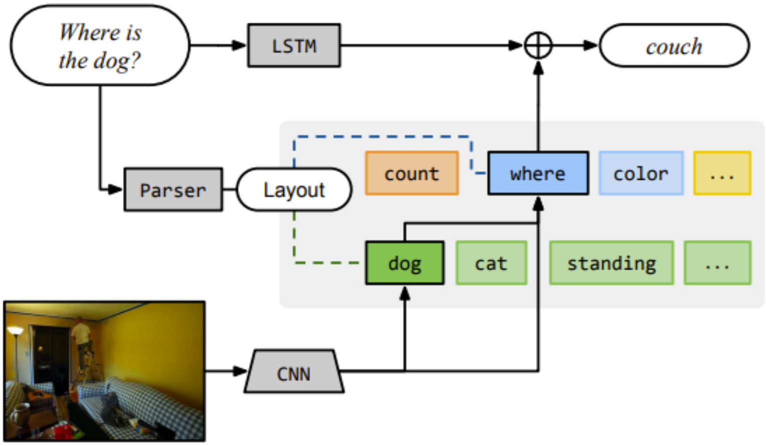


Figure 3.4: An example neural module network for visual question answering (by Andreas *et al.* (2016b)). The neurosymbolic program `where[dog]` is composed of a neural module `where`, implemented as a trainable CNN f_θ, and its symbolic parameter `dog`. The symbolic architecture of this program can be either inferred from dependency tree parsing, or synthesized.

TAPAS (Herzig *et al.*, 2020) instead follows the Neural-λ pipeline (Section 2.2.1) by training a neural semantic parsing model that culminates in a single layer of symbolic program operators such as `COUNT`. These operators allow a soft differentiable estimator, which makes the model end-to-end differentiable. At training time, TAPAS optimizes the expected semantic parsing result by computing a probability distribution over all operators and averaging the corresponding operator estimations.

Neurosymbolic reasoning is also effective to elicit inductive generalization in other reasoning domains. Chen *et al.* (2019) develop a Neural Symbolic Reader (NeRd) architecture for reading comprehension, *i.e.* question answering over natural language passages. Its neurosymbolic programmatic representation of the question combines symbolic or mathematical operators such as `COUNT` or `ARGMAX` with pointer-based neural operators for selecting spans out of the passage based on given keys. Similarly to other NMN systems, NeRd is weakly supervised and trained using hard EM with thresholding. Thanks to its explicit reasoning capabilities, NeRd significantly outperforms neural baselines on the challenging DROP (Dua *et al.*, 2019) and MathQA (Amini *et al.*, 2019) benchmarks that require mathematical and multi-step inference.

4

Learning Algorithms

In this section, we discuss in more depth the main categories of learning algorithms in neurosymbolic programming.

4.1 Symbolic Search

In symbolic program synthesis, the core learning algorithms that have enabled numerous successful applications are *enumerative*, *solver-driven*, *deductive*, and *evolutionary* program search (Gulwani *et al.*, 2017). Given a domain-specific language $\mathcal{L}$ and a specification φ, the goal of program search is to discover a program expression $e \in \mathcal{L}$ that satisfies φ (written $e \models \varphi$). The specification φ can take many forms, including pre-postconditions, types, input-output examples, or equivalent programs.

Gulwani *et al.* (2017) present an extensive overview of symbolic program synthesis techniques. In this section, we will outline them at a high level, along with representative examples of their usage in neurosymbolic programming. Symbolic program synthesis proves effective in a multitude of neurosymbolic applications, typically either as a backbone search process for *symbolic program sketches* (program architectures), or as a *neurosymbolic pipeline stage* with a specification φ deduced neurally.

4.1.1 Enumerative search

Enumerative search (Udupa *et al.*, 2013) follows the grammar of the DSL $\mathcal{L}$ and iteratively constructs candidate programs e from smaller ones to bigger ones until it finds a program that satisfies the spec φ. It maintains a cache of previously examined program expressions, which are used as constituents to construct bigger candidate expressions, following the principle of dynamic programming. The search can either follow the grammar *top-down* (from its root to its leaves), or *bottom-up* (from its leaves to its root). A top-down search grows partial program expressions by iteratively expanding their holes into appropriate grammar productions (*e.g.* `PLUS(??, ??)` → `PLUS(??, VAR(`x`))`). A bottom-up search grows smaller program expressions by assembling them as constituents into larger program expressions (*e.g.* `VAR(`x`)` and `CONST(1)` → `PLUS(CONST(1), VAR(`x`))`). Enumerative search sometimes uses the specification to drive exploration but typically only uses it to validate candidate programs.

In neurosymbolic programming, enumerative search is often used to drive program synthesis for neurosymbolic programs identically to its use for synthesis of symbolic code. The transfer learning tool Houdini (Valkov *et al.*, 2018), discussed in detail in Section 4.6, uses enumerative search to iterate over strongly-typed neurosymbolic program sketches, whose parameters are then learned using gradient descent (Section 4.2).

Formally, Houdini first uses the training set $\mathcal{D}$ to identify a *type signature* τ_0 for the desired neurosymbolic program e. Its language of types, shown in Figure 4.1, uses the idioms of functional programming such as *algebraic data types (ADTs)* and higher-order combinators, to strongly-type any neurosymbolic program expressible as a composition of feed-forward layers, graph convolutions, or recurrent networks. Strong typing of expressions and sub-expressions allows Houdini to efficiently enumerate programs that compose to conform to τ_0, eliminating ill-typed expressions during the search. Each full program candidate is a neurosymbolic program composing a mixture of pre-trained neural modules from a given library $\mathcal{L}$ and un-trained neural modules. Since every such program candidate is fully differentiable, the parameters of its fresh modules can be learned with gradient descent. The overall optimization process of HOUDINI thus alternates between enumerating the space of appropriately-typed programs and recording the trained performance of each candidate program until convergence to the desired performance.

$$\begin{array}{rlrl} \text{Type } \tau & ::= A \mid TT \mid ADT & \text{Atom } A & ::= \texttt{bool} \mid \texttt{real} \\ \text{Tensor } TT & ::= \mathsf{Tensor}\,\langle A \rangle\,[m_1][m_2]\ldots[m_k] & ADT & ::= A \mid \alpha\,\langle ADT \rangle \\ \text{Functor } F & ::= ADT \mid F_1 \to F_2 & \alpha & ::= \texttt{list} \mid \texttt{graph} \end{array}$$

$$\begin{array}{l} \textbf{Programs over a library } \mathcal{L}: \\ P ::= f_w :: \tau \mid P_1 \circ P_2 \mid \mathbf{map}_\alpha\ P \mid \mathbf{fold}_\alpha\ P \mid \mathbf{conv}_\alpha\ P \end{array}$$

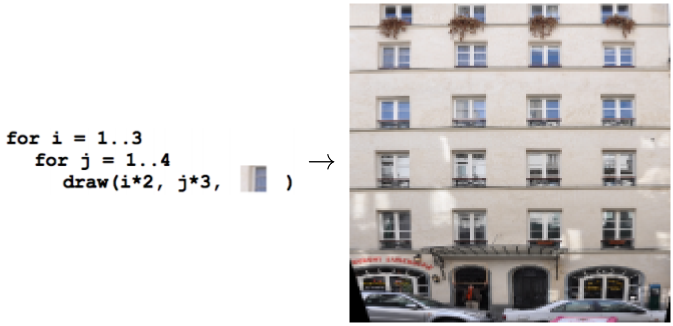
Figure 4.1: *Top:* The grammar of types for neurosymbolic programs of HOUDINI. *Bottom:* The grammar of HOUDINI programs over a library $\mathcal{L}$ of neural modules. Here $f_w :: \tau$ are library modules of type τ with learned parameters w; **map**, **fold**, **conv** are higher-order combinators whose arguments are neurosymbolic lambda functions; $m_1, \ldots, m_k$ are integers defining a tensor's shape.

Domain-specific enumerative search has also been applied to neurosymbolic *generative models*, *e.g.* by Young *et al.* (2019) and Mao *et al.* (2019). Such a model generates a semi-structured 2D image or 3D scene via a neurosymbolic composition of programmatic operators and image/scene generative networks, *e.g.* as in Young *et al.* (2019).

Figure 4.2 shows an image with repetitive structure and a learned program that generates this image. The architecture of the program is generated using a domain-specific program synthesis algorithm over the grammar of 2D loops with an enumerative grid search component for their parameters.

4.1.2 Deductive and Solver-driven search

Deductive search (Manna and Waldinger, 1971; Polozov and Gulwani, 2015) is an extension of top-down enumerative search that recursively decomposes the spec φ into smaller sub-specs $\{\varphi_j\}_j$ s.t. (a) the solution to the synthesis problem $e \models \varphi$ can be reconstructed out of the sub-



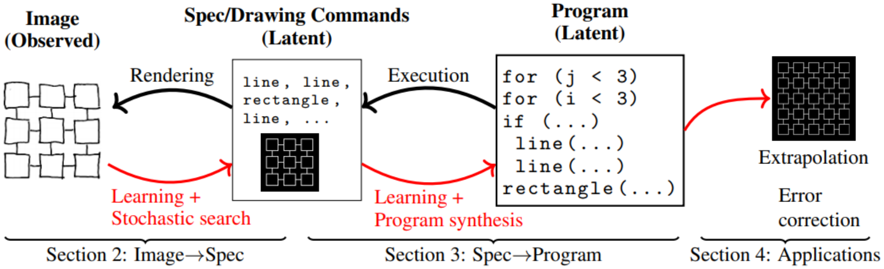
```
for i = 1..3
  for j = 1..4
    draw(i*2, j*3,      )
```

Figure 4.2: Programmatic generative models for repetitive structures (figure from Young *et al.* (2019)).

problem solutions $e_j \models \varphi_j$, and (b) the sub-problems φ_j can be solved via recursive application of deductive search. Typically φ_j represent either constraints on the subexpressions of the desired program or sub-constraints of the original spec φ. Deductive search is often augmented with neural or probabilistic guiding policies that prioritize some subtasks φ_j in the search (see Section 4.3).

Solver-driven search (Solar-Lezama *et al.*, 2006; Jha *et al.*, 2010; Torlak and Bodik, 2013) translates the space of programs $\mathcal{L}$ and the spec φ into a joint formal constraint for a SAT or SMT solver such as Z3 (Moura and Bjørner, 2008). A satisfying assignment of variables in the constraint corresponds to a single program $e \in \mathcal{L}$ s.t. $e \models \varphi$ by construction. High-level frameworks and constraint languages often help produce an efficient encoding of the desired constraint. For example, Sketch (Solar-Lezama, 2009) translates a C-like program with syntactically limited holes (which act as the spec φ) into a SAT/SMT constraint. Rosette (Torlak and Bodik, 2013) is a framework embedded into the Racket virtual machine that translates a subset of the Racket language into an SMT constraint using symbolic execution. In all cases, the semantics of the program execution and the spec must be expressible in some SMT theory (Barrett *et al.*, 2010).

Most applications of deductive and solver-driven search use neurosymbolic *learning algorithms* to synthesize symbolic code. This includes neural-guided search, described in Section 4.3, and neurosymbolic



```
line, line,
rectangle,
line, ...
```

```
for (j < 3)
for (i < 3)
if (...)
  line(...)
  line(...)
rectangle(...)
```

Figure 4.3: Learning of generative programs for 2D graphics images (figure adapted from Ellis *et al.* (2018b)). Black arrows represent a top-down generative model: Program→Spec→Image. Red arrows represent a neurosymbolic inference pipeline: Image→Spec→Program.

pipelines, described in Section 2.2.1. Neural-guided search (Ellis *et al.*, 2020; Kalyan *et al.*, 2018; Parisotto *et al.*, 2017; Ji *et al.*, 2020) uses a probabilistic or neural policy $\pi\colon \varphi \mapsto \mathbb{R}^n$ to prioritize branches of the program search tree characterized by their sub-task specifications φ_j. Neurosymbolic pipelines, such as works by Ellis *et al.* (2018b) and Chen *et al.* (2020b), process the input using a neural component to learn a specification that is passed to a symbolic program synthesis components (with possible feedback). For example, Ellis *et al.* (2018b) synthesize generative programs for semi-structured 2D drawings by first neurally parsing the drawings into atomic strokes and shapes, and then using those as a specification φ for a solver-driven program synthesis algorithm (see Figure 4.3). While an application of such hybrid pipeline to learn a program for a given input constitutes neurosymbolic learning, the learned program typically comprises of symbolic components only.

A notable exception is DeepProbLog (Manhaeve *et al.*, 2018), which integrates neural components into the ProbLog (De Raedt *et al.*, 2007) probabilistic logic programming system. In ProbLog, atomic facts $q(t_1, \ldots, t_m) :: p$ are annotated with their probabilities p. When such facts appear as outputs of neural modules, the modules can be interpreted as predicates in the ProbLog system and seamlessly integrated into the inference process of logical programming. For example, a neural module $M_w\colon \mathbb{R}^{28\times 28} \to [0,1]^{\times 10}$ classifying MNIST digits can be interpreted as a neural predicate emitting probabilities for its associated facts `digit(`$\vec{v}$`, `d`)`, where $\vec{v} \in \mathbb{R}^{28\times 28}$ is a feature vector for a MNIST image, and $d \in [0, \ldots, 9]$. Formally, such neural predicate defines an *annotated disjunction*

$$nn(M_w, \vec{v}, [0, \ldots, 9]) :: \texttt{digit(}\vec{v}\texttt{, 0)}; \ldots; \texttt{digit(}\vec{v}\texttt{, 9).}$$

where the notation $nn(M_w, \vec{v}, d)$ indicates that the associated probability for a disjunct `digit(`$\vec{v}$`, `d`)` is obtained by evaluating the network $M_w(\vec{v})$ and retrieving its d^{th} normalized probability from the output layer.

Learning of the parameters w of these predicates happens jointly with a logical inference process. Specifically, during learning, a DeepProbLog program is automatically annotated with *gradient semiring tuples* $(p, \nabla p)$. Here p is each fact's probability, and ∇p is the probability's partial derivative w.r.t. the program's parameters w. This allows using the ProbLog inference mechanism to compute the gradient of the desired loss through the ground program. The gradient is then passed directly to an auto-differentiation framework to optimize the parameters w, iterating until convergence. The desired loss ensures that the user's *query statement* in a logic programming system is satisfied above a certain threshold of probability.

4.1.3 Evolutionary Search

Evolutionary methods are a popular paradigm in classical program synthesis (Krawiec, 2016; Le Goues *et al.*, 2011). These methods iteratively update a pool of candidate programs written in a DSL $\mathcal{L}$. A *fitness function* derived from the specification of the synthesis problem is used to filter this pool. Mutation and crossover operations mimicking biological evolution are used to grow the pool.

Evolutionary methods are also a popular choice for architecture search for neural networks (Liu *et al.*, 2021). Deep learning approaches that use such architecture search can be seen as a form of neurosymbolic program learning, as any neural architecture can be represented as a program. That said, most existing work on neural architecture search only aims to automate low-level design decisions such as the width of convolutional filters, the details of the gating mechanism, and so on. The use of such a restricted, low-level DSL sets these methods apart from most methods that we consider in this monograph.

Recent work by Real *et al.* (2020) significantly extends the mandate of neural architecture search and brings it closer to general program synthesis. Here, the goal is to discover entire machine learning algorithms, expressed in a generic programmatic notation, that perform well on a given set of tasks. Evolutionary methods are used to solve this problem. It is shown that the resulting framework can rediscover contemporary neural architectures and algorithmic techniques, for example, bilinear interactions, normalized gradients, and weight averaging. Also, the synthesis method is shown to adapt to different task types — for instance, in settings with limited data, dropout emerges as a top candidate.

4.2 Gradient-Based Search

One of the distinguishing features of deep learning is the use of stochastic gradient descent as the underlying search mechanism, so it is natural to ask whether the same mechanism can be used to search for programs. This can be particularly valuable when the goal is to discover hybrid programs that include both program-like features—branches, loops, recursion and potentially even complex data-types—but also some numerical computation, which may even include small or not-so-small neural networks.

There is a whole space of solutions that have attempted this in different forms, but they all involve the use of gradient-based optimization at synthesis time. This stands in contrast to neural-guided search techniques, which perform gradient-based learning at meta-learning time. Also, the program returned to the user is the direct result of the optimization process, unlike the Distillation approaches where the gradient-based learning is just one step in the process of searching for a discrete program.

There are two major constraints that all the techniques in this space have to cope with. First, the space of programs is fundamentally discrete, so searching this space through continuous optimization necessarily requires us to come up with a continuous approximation for this search space; we can refer to this as structural discontinuity. And second, the programs themselves may have discrete behavior, especially if they involve conditionals. So even if the discrete architecture of the program is already given, one may still need a continuous approximation of the program in order to search for numerical parameters within the program; we refer to this second kind of discontinuity as parameter discontinuity.

To illustrate this issue, consider the program shown below, which encodes a space of programs as a parametric program. The structural discontinuity is illustrated by the parameter α, which represents a discrete choice between adding or multiplying at every iteration of the loop. The parameter β, by contrast represents a continuous choice, but the output of the overall program is still discontinuous on this choice β; in the strictest sense, the gradient for the output t with respect to β is either zero or undefined for every point in the space, even after fixing the choice of α.

```
$t = 0$;
while($x > y$){
    if($\alpha$){$x = x * \beta$}
    else{$x = x + \beta$}
    $t = t + 1$;
}
return $t$;
```

The challenge in using gradient search on programs is to cope with both of these types of discontinuities.

Program smoothing, introduced by Chaudhuri and Solar-Lezama (2010), was one of the early responses to these challenges. The idea behind that work was to represent the overall search space as a parametric program similar to the one shown above, but to reduce the structural discontinuity to parameter discontinuity, for example, by replacing the code `if`(α) above with `if`($\alpha > 0$) so that α is now a continuous parameter, but it still has a discontinuous effect on the function. The parameter discontinuity is then handled by symbolically approximating the effect of running the program on a Gaussian distribution of inputs and then computing the expected value at the output.

Chaudhuri *et al.* (2014) used a similar approach to synthesize programs that could be verified to satisfy a set of probabilistic assertions. Instead of smoothing the execution of the program itself, this paper smoothed execution of an abstract interpretation of the program, and in that way was able to find parameters that allowed the program to verify with that abstract interpretation.

Subsequent work on differentiable program interpreters (Gaunt *et al.*, 2016; Riedel *et al.*, 2016; Gaunt *et al.*, 2017) connected continuous relaxations with automatic differentiation. These papers gave frameworks for writing parameterized programs that use high-level constructs for, among other things, looping, branching, and procedure calls. These programs are interpreted using a differentiable abstract machine. Program parameters are discovered via backpropagation through this abstract machine.

In principle, the parameters in these programming frameworks could dictate a choice between discrete program statements. In practice, however, the frameworks require detailed, hand-written templates, and general search over discrete program architectures is not allowed. This sets these methods apart from most of the other methods considered in this monograph.

The DiffLog (Si *et al.*, 2019) approach to inductive logic programming is an example of how gradients can be used to search over program architectures. The goal in this work is to learn or select a subset of rules that explains a dataset, starting from a large, predefined set of Datalog rules. DiffLog's learning algorithm relaxes this discrete search problem into a continuous problem in which rules have real-valued weights, and the goal is to find an optimal weight assignment. Once this continuous problem is solved, the algorithm uses certain properties of Datalog to recover the discrete-valued target program.

The general approach of synthesizing programs using gradient descent works well for simple problems. However, on more complex tasks, gradient-based search becomes more challenging. The issue is that differentiability is just one element that makes deep neural networks effective, and that there are two other elements that smoothed programs tend to lack. The first one is overparameterization. Unlike a neural network, a smoothed program will often have a lot of structure and relatively few parameters, leading to an optimization landscape with many local minima. The second one is what may be termed "derivative engineering"; many well known neural architectures such as LSTMs and residual networks owe their existence to the fact that more naïve architectures can end up with gradients that either vanish or have numerical instabilities. When taking derivatives over smoothed programs, a user must ensure that such vanishing gradients or numerical instabilities do not arise, and this can be challenging.

A recent method, dPads (Cui and Zhu, 2021), tries to address the overparameterization issue by coupling gradient-based search with neural relaxations. The method's goal is to synthesize both the architectures and the parameters of symbolic programs. To this end, it follows the lead of recent work on gradient-based architecture search for neural networks Liu *et al.*, 2019b and defines a continuous relaxation of the space of programs. This relaxation involves neural elements with many parameters. We elaborate on this method further in Section 4.5.

Hamiltonian Monte-Carlo (HMC; see Neal (2011)) methods, which use gradients to guide a sampling process, form another promising direction. HMC has been popular in probabilistic programming languages such as Stan (Carpenter *et al.*, 2017) and Pyro (Bingham *et al.*, 2019) where one cares not just about finding a single optimal solution but finding a representative sample over some prior conditioned by some constraints. The probabilistic programming language Omega combines program smoothing with HMC in an algorithm called *predicate exchange* that allows it to condition the search on complex boolean predicates (Tavares *et al.*, 2019).

4.3 Learning to Synthesize

A key challenge in generating symbolic programs is the large combinatorial search space of all possible discrete programs. One common approach for tackling this large search space is to *learn to synthesize*, i.e., to learn, from data, neural policies that generate probability distributions over the space of programs. These distributions can then be used to guide the combinatorial search to efficiently identify the desired program.

Let $\mathcal{L}$ denote a domain-specific language (DSL) for a given domain D. In neural program synthesis, the goal is to find a program $P \in \mathcal{L}$ that satisfies a given specification φ. Quite often, the specification is defined using a set of input-output examples $\varphi \equiv \{(I_j, o_j)\}_j$, such that the desired program P produces the corresponding output when executed on the input, i.e. $\forall j : \; P(I_j) = o_j$. However, in certain cases,

```
a ← [int]               An input-output example:
b ← FILTER (<0) a       Input:
c ← MAP (*4) b          [-17, -3, 4, 11, 0, -5, -9, 13, 6, 6, -8, 11]
d ← SORT c              Output:
e ← REVERSE d           [-12, -20, -32, -36, -68]
```

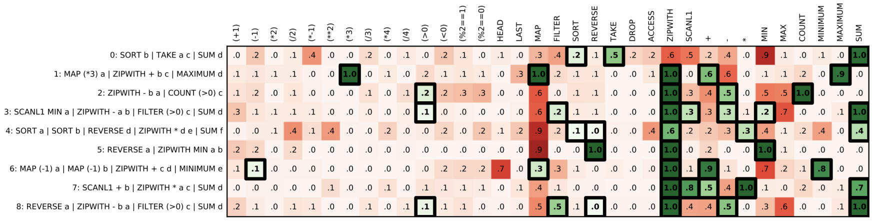
Figure 4.4: An example array program in the DeepCoder DSL together with the corresponding input-output example from Balog *et al.* (2016). The goal here is to automatically synthesize the program on the left given the input-output example.

the specification could also be more ambiguous, for example, a set of keywords (Murali *et al.*, 2018).

DeepCoder (Balog *et al.*, 2016) is a neural program synthesis system that first learns a probability distribution over the DSL operators given the input-output example specification, and then uses the learnt distribution to guide an enumerative breadth-first or depth-first search over DSL programs. Formally, it trains a policy $\pi : \varphi \rightarrow \mathbb{R}^n$, where n denotes the number of operators in $\mathcal{L}$. Given a specification φ at test time, it calls the policy $\pi(\varphi)$ to return a distribution over DSL operators and assigns a probability score to each operator. It then uses the probability scores to assign scores to different programs $P \in \mathcal{L}$ and uses a greedy algorithm to explore the search space to find a program that is consistent with the specifcation.

An example task for synthesizing an array manipulation program considered by DeepCoder is shown in Figure 4.4. The DSL consists of first-order functions such as `HEAD`, `TAIL` etc. as well as higher-order functions such as `MAP, FOLD` etc. where only certain pre-defined lambda functions can be used. Using large amounts of synthetic training data of randomly sampled programs in the DSL, DeepCoder trains a policy that predicts probabilities of functions in the DSL given an input-output example. Some example predictions are shown in Figure 4.5. Finally, it uses these predicted probabilities to guide an enumerative depth-first or a breadth-first search over DSL functions to generate a program that is consistent with a given input-output example.

In contrast to DeepCoder that only learns the initial distribution over DSL operators, Neuro-symbolic program synthesis (Parisotto *et al.*, 2017) learns a distribution over programs to guide every step of the



	(+1)	(-1)	(*2)	(/2)	(*-1)	(**2)	(*3)	(/3)	(*4)	(/4)	(>0)	(<0)	(%2==1)	(%2==0)	HEAD	LAST
0: SORT b | TAKE a c | SUM d	.0	.2	.0	.1	.4	.0	.0	.2	.0	.1	.0	.2	.1	.0	.1	.0
1: MAP (*3) a | ZIPWITH + b c | MAXIMUM d	.1	.1	.1	.1	.0	.0	1.0	.0	.1	.0	.2	.1	.1	.1	.0	.3
2: ZIPWITH - b a | COUNT (>0) c	.1	.2	.0	.1	.0	.0	.0	.1	.0	.1	.2	.2	.3	.3	.0	.0
3: SCANL1 MIN a | ZIPWITH - a b | FILTER (>0) c | SUM d	.3	.1	.1	.1	.1	.0	.0	.0	.0	.0	.1	.0	.0	.0	.0	.0
4: SORT a | SORT b | REVERSE d | ZIPWITH * d e | SUM f	.0	.0	.1	.4	.1	.4	.0	.0	.2	.0	.0	.2	.0	.2	.1	.2
5: REVERSE a | ZIPWITH MIN a b	.2	.2	.0	.2	.0	.0	.0	.0	.0	.0	.0	.0	.0	.0	.0	.0
6: MAP (-1) a | MAP (-1) b | ZIPWITH + c d | MINIMUM e	.1	.1	.0	.0	.0	.0	.0	.0	.0	.0	.0	.2	.2	.2	.7	.0
7: SCANL1 + b | ZIPWITH * a c | SUM d	.0	.0	.0	.0	.0	.1	.0	.0	.1	.0	.1	.1	.1	.1	.0	.1
8: REVERSE a | ZIPWITH - b a | FILTER (>0) c | SUM d	.2	.1	.0	.1	.1	.0	.0	.1	.0	.1	.1	.1	.1	.0	.0	.0

	MAP	FILTER	SORT	REVERSE	TAKE	DROP	ACCESS	ZIPWITH	SCANL1	+	-	*	MIN	MAX	COUNT	MINIMUM	MAXIMUM	SUM
0: SORT b | TAKE a c | SUM d	.3	.4	.2	.1	.5	.2	.2	.6	.5	.2	.4	.0	.9	.1	.0	.1	.0	1.0
1: MAP (*3) a | ZIPWITH + b c | MAXIMUM d	1.0	.2	.1	.1	.0	.0	.1	1.0	.0	.6	.6	.0	.1	.1	.2	.0	.9	.0
2: ZIPWITH - b a | COUNT (>0) c	.6	.0	.1	.1	.0	.0	.0	1.0	.3	.4	.5	.0	.5	.5	1.0	.0	.0	.0
3: SCANL1 MIN a | ZIPWITH - a b | FILTER (>0) c | SUM d	.6	.2	.1	.1	.0	.0	.0	1.0	.3	.3	.3	.1	.2	.7	.0	.0	.1	1.0
4: SORT a | SORT b | REVERSE d | ZIPWITH * d e | SUM f	.9	.2	.1	.0	.0	.0	.4	.6	.2	.2	.3	.3	.4	.1	.2	.4	.0	.4
5: REVERSE a | ZIPWITH MIN a b	.9	.0	.0	1.0	.0	.0	.0	1.0	.0	.2	.0	.0	1.0	.1	.0	.0	.0	.0
6: MAP (-1) a | MAP (-1) b | ZIPWITH + c d | MINIMUM e	.3	.3	.1	.0	.0	.0	.0	1.0	.1	.9	.1	.0	.7	.2	.1	.8	.0	.0
7: SCANL1 + b | ZIPWITH * a c | SUM d	.4	.1	.0	.0	.0	.0	.0	1.0	.8	.5	.4	1.0	.1	.0	.2	.0	.1	.7
8: REVERSE a | ZIPWITH - b a | FILTER (>0) c | SUM d	.5	.5	.1	.0	.0	.0	.0	1.0	.4	.4	.5	.0	.3	.6	.0	.0	.1	1.0

Figure 4.5: The probabilities of different functions in the DSL predicted by the trained neural network given the input-output example for different example tasks (one row per task) taken from Balog *et al.* (2016). The synthesis algorithm then uses these probabilities to guide an enumerative breadth-first or depth-first search.

program search as shown in Figure 4.6. More concretely, it trains a policy $\pi : (p, \varphi) \to \mathbb{R}^{|c(p)|}$, where p denotes a partial program in $\mathcal{L}$ and $c(p)$ denotes all possible completions of the partial program p given the grammar of the DSL. It presents an architecture called R3NN (Reverse-Recursive-Reverse neural network) to encode partial program trees. Similar to recursive neural networks, it first performs a bottom-up pass over a program tree to compute distributed representations of the tree nodes starting from the leaf nodes. It then perform a top-down pass, where it updates the node representations using a new set of weight matrices by conditioning on parent nodes. Finally, it uses the updated leaf node representations to generate a distribution over promising non-terminal nodes to expand next for the search.

RobustFill (Devlin *et al.*, 2017) uses seq2set2seq models using attention to learn a distribution over complete programs. This approach trains a policy $\pi : \varphi \to \mathbb{R}^{(n \times s)}$, where n denotes the vocabulary size of the DSL and s denotes the length of the target program. The policy first uses bi-directional LSTMs to encode each input and output strings for each example individually, and then uses a decoder LSTM to generate latent program embeddings for each example while attending over the example embeddings. Finally, it uses a max pooling over the latent embeddings to generate a distribution over program sequence, where beam search can be used to generate programs from the learnt distribution.

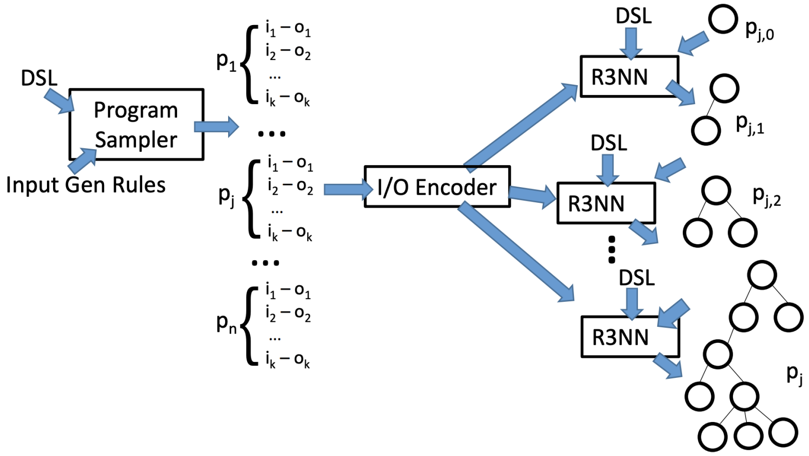


Figure 4.6: Parisotto *et al.* (2017) trains a policy to guide the program search at each intermediate step. It uses R3NN architecture to learn a distribution over next expansion of a partial program given the input-output example and the current partial program. The search terminates when there are no non-terminal nodes left to expand in the partial program.

Bayou (Murali *et al.*, 2018) learns an encoder-decoder, trained using variational methods, that maps specifications to distributions over syntactic abstractions, or *sketches*, of programs. Sketches generated from this distribution are concretized into complete symbolic programs using classic methods for program synthesis (Alur *et al.*, 2015).

Neural-Guided Deductive Search (NGDS) (Kalyan *et al.*, 2018) uses a hybrid search approach where the policy is used to guide a deductive synthesis search. The deductive search (Polozov and Gulwani, 2015) recursively decomposes the original synthesis task φ into smaller sub-tasks $\{\varphi_j\}_j$ such that same deductive principles can be further applied to the sub-problems φ_j. NGDS learns a policy $\pi : \varphi \to \mathbb{R}^k$, where k denotes the number of sub-problems at a given search depth to prioritize selection of promising sub-problems amongst all possible decompositions.

A common theme for many of these learning-to-synthesize approaches is using synthetic datasets of programs and specifications to train the guiding policy. To generate such training datasets, one common methodology is to first sample random programs $P \in \mathcal{L}$ and then use pre-conditions of P to sample a set of random inputs $I \equiv \{I_j\}_j$ such that they are valid inputs for the program. The program P is then executed on the inputs to generate corresponding outputs $O \equiv \{o_j\}_j$. Generating datasets using such sampling methods can potentially introduce some biases and some recent techniques have been proposed to generate more diverse and balanced datasets for training the guiding policies (Shin *et al.*, 2019b).

The neural search policies used in the learning-to-synthesize paradigm are usually trained using supervised learning on the synthetically generated specification and program pairs. One issue with such supervised learning approaches is that they do not capture cases when there are multiple possible correct programs that are consistent with a given specification. Some techniques (Bunel *et al.*, 2018; Chen *et al.*, 2020b) use reinforcement learning to alleviate this issue. The central challenge in these methods is that the reward landscape here is extremely sparse, as "bad" programs vastly outnumber the "good" ones. This makes it difficult to train an RL policy directly for learning the search policy. To overcome this problem, Bunel *et al.* (2018) first train a supervised model for program generation. Then they use the Reinforce algorithm to fine-tune this distribution by sampling programs from the distribution and giving a positive reward for a program that, when executed, produces the desired output. The approach by Chen *et al.* (2020b), applicable to tasks in which the search objective is to satisfy a hard constraint, uses automated deduction to prune out subspaces of undesirable programs as the search progresses. One interesting aspect of this approach is that the deduction algorithm can give feedback not only about programs that can be generated using the current search policy, but also programs that are "out-of-distribution" (i.e., produced by search actions that have not been explored so far). The method uses an off-policy RL algorithm (Levine and Koltun, 2013) to make use of such feedback.

4.4 Distillation

In machine learning, *knowledge distillation* Hinton *et al.* (2015) refers to methods that transfer the knowledge stored within a trained, high-capacity model (the *teacher*) to a lower-capacity model (the *student*). For example, the teacher could be a large neural network, and the student could be a decision tree or a small network. The traditional reason for distillation was to execute learning models on platforms with limited computational resources. However, in the recent past, distillation has also emerged as a key tool for learning symbolic and neurosymbolic programs.

In distillation-based approaches to program learning, one starts with a neural or neurosymbolic teacher function g that is learned using gradient-based optimization. The core distillation operation is to synthesize a fully symbolic program, with the same domain and range as g, that maximally imitates the input-output behavior of g. In versions of the problem that arise in safe or verified learning (Section 3.4), the symbolic program is additionally required to satisfy a correctness constraint. In all cases, the symbolic program can then be used to construct a new teacher that can be updated and distilled again.

Formally, let us assume a space $\mathcal{F}$ of symbolic programs (satisfying relevant syntactic and semantic constraints), a distribution D over function inputs that we can statistically estimate, and a distance function δ over the space of function outputs. The problem of *distilling into programs* can be can be stated as

$$f^* = \arg\min_{f \in \mathcal{F}} \mathbf{E}_{x \in D}\ \delta(f(x), g(x)) \tag{4.1}$$

For distillation to be profitable, the above problem must be easier than the end-to-end program learning problem, and the teacher g should be easy to construct. These assumptions hold in the setting of reinforcement learning (RL), which is where most distillation-based program learning has been applied so far.

Specifically, Pirl (Verma *et al.*, 2018), Viper (Bastani *et al.*, 2018), and Propel (Verma *et al.*, 2019), described earlier in this paper, are three distillation-based approaches to RL. The goal in all of these methods is to learn a fully symbolic, programmatic policy that maximizes an

RL agent's long-term reward. Pirl and Viper approach this objective through a two-step process. Both methods first learn a neural policy that performs well in the task objective using a standard deep RL algorithm. This neural network is then distilled into a symbolic program using *online imitation learning*, a family of methods for learning policies that behaviorally mimic a given oracle policy. Specifically, Pirl and Viper use variants of Dagger, an imitation learning algorithm that grows a dataset of input-output pairs through repeated queries to the oracle. Distillation amounts to finding a program that fits this dataset, i.e., an instance of *supervised program learning*.

The subsequent Propel approach (Verma *et al.*, 2019) incorporates the distillation procedure inside an iterative training loop. Specifically, the framework considers neurosymbolic policies $h(x)$ of the form $f(x) + g(x)$, where $f(x)$ is a neural policy and $g(x)$ is a symbolic policy expressible in a DSL. The set of symbolic policies is viewed as forming a proper subset Π of the set $\mathcal{H}$ of neurosymbolic policies, and the discovery of an optimal symbolic policy is framed as a constrained optimization task. The learning process, an instance of *approximate functional mirror descent*, starts with an initial, suboptimal neurosymbolic policy. In each learning step, it iterates between:

1. Applying a series of gradient updates ($\mathit{Update}_{\mathcal{F}}$) to the current neurosymbolic policy. Importantly, these updates are based on the gradient of the neural component of the policy, and is thus an *approximation* to the true gradient in the $\mathcal{H}$-space.

2. Using an operation $\mathit{Project}_{\Pi}$ to distill this neurosymbolic policy into a symbolic policy, which is then lifted back into the neurosymbolic space $\mathcal{H}$.

Thus, both the neural and the symbolic components of the program are learned from data through a synergistic process. The approach is visually depicted in Figure 4.7 from Verma *et al.* (2019).

Several recent efforts extend the above ideas to settings involving a formal safety property. In particular, a method by Zhu *et al.* (2019) (previously alluded to in Section 3.4) uses a combination of imitation learning and *counterexample-guide inductive synthesis* to distill a neural

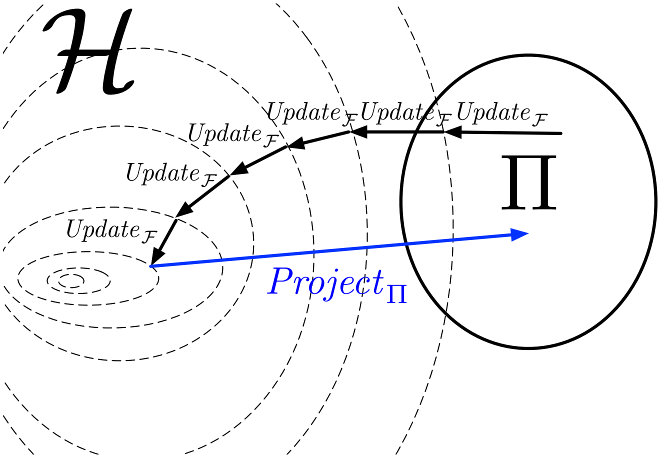


Figure 4.7: The Propel algorithm (figure from Verma *et al.* (2019)).

policy into a provably safe symbolic policy, without direct verification of neural networks. The Revel approach (Anderson *et al.*, 2020) for formally verified exploration is an extension of Propel in which the neurosymbolic policies have the form

$$h(x) = \textbf{if } (\psi(x, f(x)) \textbf{ then } f(x) \textbf{ else } g(x).$$

where $f(x)$ is a shield (a safe symbolic policy), $g(x)$ is a potentially unsafe neural policy, and ψ is a symbolic monitor predicate. Each learning iteration consists of an update to the neural component of the current neurosymbolic policy, a distillation step that produces a new monitor and shield, and a lifting operation that constructs a new neurosymbolic policy. The main difference from Propel is that each of these steps maintains the invariant of provable safety.

Cranmer *et al.* (2020) offer an example of distillation-based program learning outside the RL setting. The goal here is symbolic regression, i.e., the automatic discovery symbolic mathematical equations from labeled data; however, the method can be extended to more general programs as well. The algorithm in this work first trains a sparse graph neural network on the input dataset. Next, it uses genetic programming to distill components of this network into explicit symbolic expressions. Collectively, these expressions describe physical relationships between objects that the data describes.

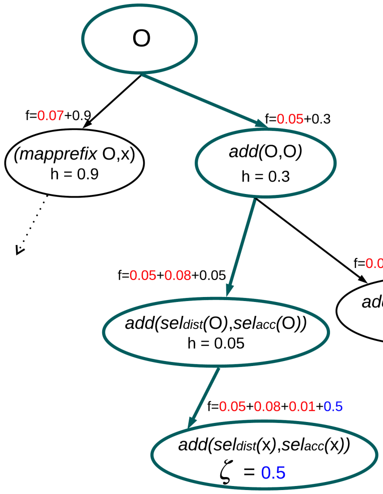


Figure 4.8: An example of program learning formulated as informed search, from Shah *et al.* (2020). Nodes represent incomplete program architectures, constructed in a top-down manner. The numbers in red represent static “structural costs” for the use of various program constructs. The values of the neural heuristic are in black, prediction errors ζ are in blue, $\circ$ refers to a “hole” in a partial architecture, and the path to a goal node returned by Near is in teal.

4.5 Neural Relaxations

Another class of learning algorithms is based on *neural relaxations* of programs. These methods are motivated by the insight that one can interpret programs as a constrained or regularized subspace of functions learnable by deep neural networks.

Formally, we can think of the space of programs $\mathcal{L} \subset \mathcal{H}$ as a subset of the space of functions learnable by deep neural networks $\mathcal{H}$. Given a training loss function C, we can then write the learning optimization problem as

$$\arg\min_{P \in \mathcal{L} \subset \mathcal{H}} C(P). \tag{4.2}$$

The goal then is to learn a program that minimizes C (e.g., has small classification error). While learning over $\mathcal{L}$ directly suffers from combinatorial intractability issues, learning over (subsets of) $\mathcal{H}$ is straightforward using modern deep learning techniques. The key idea of neural relaxations then is to leverage the fact that we can effectively learn (or search) over $\mathcal{H}$ in order to effectively learn over $\mathcal{L}$.

The Propel approach (Verma *et al.*, 2019) from Section 4.4 is partly based on the idea of neural relaxations. As explained earlier, Propel views (4.2) through the lens of constrained mirror descent: in any iteration of the learning algorithm, one takes an unconstrained update step into $\mathcal{H}$ space (via deep learning), then projects back to the $\mathcal{L}$ space (through distillation into a fully symbolic program using methods discussed in Section 4.4), then *lifts* the distilled into the $\mathcal{H}$ space. This last step can be viewed as applying a relaxation.

The Near (neural admissible relaxations) approach (Shah *et al.*, 2020) uses neural relaxations to guide a combinatorial search over program architectures. The method views a neural relaxation as an admissible heuristic, which can be used to instantiate informed search algorithms, such as A*-search, over discrete program architectures. The key difference between this approach and other neural-guided search methods discussed in Section 4.3 is that, rather than training a neural net to predict good search directions, the neural relaxation is directly trained to find a high-performing neural net that is functionally similar to a program.

Specifically, Near considers a search procedure in which program architectures are constructed top-down (see Figure 4.8). This procedure can be seen to traverse a program derivation graph whose nodes are partial and complete program architectures. To apply informed search methods such as A* to this graph, one requires an admissible heuristic that lower bounds the best error achievable from an internal node in the graph (a partial program architecture). A trivial lower bound is 0, but in that case A*-search reduces to breadth-first search which suffers from high combinatorial cost. If the heuristic is perfectly tight (i.e., the lower bound is tight), the A*-search reduces to Dijkstra's shortest path algorithm (which runs in polynomial time). In Near, each incomplete program architecture is relaxed by substituting its "holes" (missing sub-structures) by neural networks. This relaxation is then trained end to end. Because deep neural networks can be expected to perform at least as well as the best program completion, the training loss on the deep neural network is lower bound on the loss of the best program, hence it is an admissible heuristic.

Subsequent work by Cui and Zhu (2021) extends the ideas in Near to develop an end-to-end differentiable method for program synthesis. In this method, called dPads, the entire program derivation graph is modeled as a differentiable program. That is, at each node in the search graph, the categorical choice between different edges out of the node (corresponding to the application of different grammar rules to the partial architecture at the node) is relaxed into to a softmax with trainable weights. The gradient updates for these softmax weights are guided by a lower-level signal from the neural relaxations for the nodes to which the edges lead. The optimization problem resulting from this formulation can be challenging to solve. However, the paper proposes several heuristic techniques, such as iteratively unfolding the graph and sharing nodes between partial architectures, that somewhat simplify the problem.

4.6 Transfer of Neural Modules

In general, the aim of transfer learning for abstraction is to induce reusable modules–either neural, symbolic, or both–which can be reused when solving new synthesis tasks (Section 3.2); here we focus on transfer of neural modules. They operate via a "pre-training" phase, where this inventory of neural modules is constructed, followed by a "deployment" or test-time phase when this inventory is used to solve testing problems. This deployment phase can use neural programming algorithms such as the search algorithms discussed in Section 4.3.

During the "pre-training" phase, an inventory of small neural networks are composed to solve training synthesis tasks in tandem with training the weights of each network. The size of this bank of neural modules can either be fixed ahead of time–such as in modular meta-learning (Alet *et al.*, 2018b) and memoised wake-sleep (Hewitt *et al.*, 2020)–or grow in size after each training experience, such as in Houdini (Valkov *et al.*, 2018). At the test time or "deployment" phase, these neural modules can be used wholesale when synthesizing solutions to test problems, and so not need massive amounts of data to learn at test time. The programmatic, compositional reuse of these neural modules supports strong forms a generalization. For example, HOUDINI bootstraps neural networks for recognizing digits, which then, when combined with arithmetic and control flow primitives, allow it to calculate the sum of a list of digits represented as hand-drawn images.

At a high level, we formalize such approaches as follows. Let the library $\mathcal{I}$ be a set of neural networks, and let $\mathit{Progs}(\mathcal{I})$ be the space of symbolic programs that can call out to members of $\mathcal{I}$; referencing the pseudocode in Sec. 2.2.1, the primitives `catPos` and `tablePos` would be members of $\mathcal{I}$, and the whole expression would be a member of $\mathit{Progs}(\mathcal{I})$. We also have a collection of N training specifications with loss functions $\{C_n(\cdot)\}_{n=1}^N$. At train-time we synthesize N programs $\{h_n\}_{n=1}^N$ minimizing the loss for each spec, *and* induce the reusable library $\mathcal{I}$. For instance, in a batched setting where all specs are available at once, we could minimize the summed losses

$$\begin{aligned} \mathcal{I}^* &= \arg\min_{\mathcal{I}} \sum_{n=1}^{N} \arg\min_{h_n \in \mathit{Progs}(\mathcal{I})} C_n(h_n) \\ h_n &= \arg\min_{h_n \in \mathit{Progs}(\mathcal{I})} C_n(h_n) \end{aligned} \tag{4.3}$$

HOUDINI instead works through an incremental, lifelong learning setting where a sequence of libraries $\{\mathcal{I}_n\}_{n=1}^N$ are constructed sequentially while synthesizing each program $h_n \in \mathit{Progs}(\mathcal{I}_n)$. Each program h_n is symbolic code that can call out to neural modules in $\mathcal{I}_n$ as well as new neural modules, whose parameters are written θ_n. Writing $h_n(\cdot|\theta_n)$ for the function computed by symbolic code h_n with its new neural module weights set to θ_n, HOUDINI iterates

$$h_n(\cdot|\theta_n) = \arg\min_{\substack{\theta_n \text{ a weight vector} \\ h_n \in \mathit{Progs}(\mathcal{I}_n)}} C_n(h_n(\cdot|\theta))$$

$$\mathcal{I}_n = \mathcal{I}_{n-1} \cup \{\text{neural modules in } h_{n-1}(\cdot|\theta_n)\}$$

where the optimization

$$\arg\min_{\substack{\theta_n \text{ a weight vector} \\ h_n \in \mathit{Progs}(\mathcal{I}_n)}} C_n(h_n(\cdot|\theta_n))$$

works by an outer loop search over discrete program structures, and an inner loop of gradient descent to optimize θ_n. This inner loop requires that the program and loss be differentiable with respect to the module parameters.

Houdini freezes the weights of its neural modules, but contemporary meta-learning approaches (Finn *et al.*, 2017) adapt pre-trained weights to each new task. The modular meta-learning (MoMA) algorithm introduced by Alet *et al.*, 2018b combines Houdini-style pre-trained modules with this weight adaptation. Writing Θ for the weights of all of the neural modules in $\mathcal{I}$, and $h|_{\Theta'}$ for the expression h with weights assigned to Θ', MoMA tries to solve for

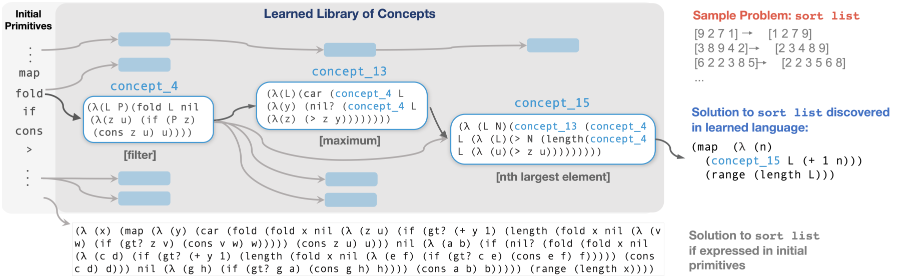
$$\Theta^* = \arg\min_{\Theta} \sum_{n=1}^{N} \arg\min_{h_n \in Progs(\mathcal{I})} C_n(h_n|_{\Theta_n(\Theta, h_n)})$$

$$\Theta_n(\Theta, h) = \Theta - \eta \nabla_{\Theta} C_n(h|_{\Theta}), \text{ where } \eta \text{ is a learning rate}$$

(Compare with Eq. 4.3: we have replaced $\mathcal{I}$ with Θ because the library is defined solely in terms of the weights of each of its constituent neural modules.) This optimization is difficult and so MoMA alternates between using simulated annealing to infer a collection of low-loss h_n's, and then taking gradient steps on Θ.

4.7 Transferring Symbolic Code

A symbolic transfer learning approach is to induce a library of reusable symbolic code, typically by compressing out reused syntactic patterns across a training corpus of programs. Figure 4.9 illustrates an example output of one such system. Here, the system is initialized with basic primitives for list processing and arithmetic–`map`, `fold`, etc.–and is given a corpus of around 100 training problems, such as sorting a list, or counting the number of even numbers in a list. It tackles these problems by jointly synthesizing code satisfying each training spec, while also building a library of reused functions that prove useful while solving the training corpus–and which should transfer to similar problems in the future. It builds this library by examining the source code of synthesized programs and mining them for reusable templates. Figure 4.10 illustrates



```
(λ(L P)(fold L nil
 (λ(z u) (if (P z)
  (cons z u) u))))
```

```
(λ(L)(car (concept_4 L
(λ(y) (nil? (concept_4 L
(λ(z) (> z y))))))))
```

```
(λ (L N)(concept_13 (concept_4
L (λ (L)(> N (length(concept_4
L (λ (u)(> z u)))))))))
```

```
(map  (λ (n)
  (concept_15 L (+ 1 n)))
  (range (length L)))
```

```
(λ (x) (map (λ (y) (car (fold (fold x nil (λ (z u) (if (gt? (+ y 1) (length (fold x nil (λ (v
w) (if (gt? z v) (cons v w) w))))) (cons z u) u))) nil (λ (a b) (if (nil? (fold (fold x nil
(λ (c d) (if (gt? (+ y 1) (length (fold x nil (λ (e f) (if (gt? c e) (cons e f) f))))) (cons
c d) d))) nil (λ (g h) (if (gt? g a) (cons g h) h)))) (cons a b) b))))) (range (length x))))
```

Figure 4.9: Adapted from Ellis *et al.*, 2020. This example system transfers symbolic code by learning a reusable library of expressions (middle, 'Learned Library') which it uses to synthesize programs solving training specs, such as the input-outputs for 'sort list' (right, topmost). Before learning the system has rudimentary primitives such as those shown at the left. Over its learning trajectory it defines new functions that build on top of these (learned call graph notated with arrows). Using the learned library, problems such as list sorting can be solved with relatively small amounts of code (rightmost, middle). In principle these problems are solvable without the learned library, but would be prohibitively lengthy in many cases (bottommost).

a special case of this library learning where the reused template is a syntactic fragment of the synthesized programs.

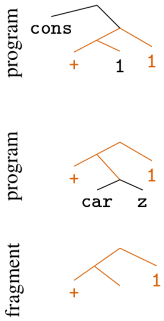
Transferring symbolic code also opens up the possibility of supervised learning: learning the library by mining repeated code schemas from ground-truth, human-written symbolic code. At the same time, unsupervised approaches–which learn only from specifications–remain viable by learning a library on the programs the system itself discovers. A supervised system inputs a set of programs $\{e_n\}_{n=1}^N$ and outputs a set $\mathcal{I}$ of symbolic expressions, each of which is a reusable function. An unsupervised system instead inputs a set of specs $\{\varphi_n\}_{n=1}^N$. For example, if we think of the library as producing a prior over expressions, supervised maximum a posteriori learning of the library corresponds to Eq. 4.4; unsupervised algorithms interleave an update like that of Eq. 4.4 with a program synthesis step like that of Eq. 4.5

solve for library, supervising on programs:

$$\mathcal{I}^* = \arg\max_{\mathcal{I}} \mathbf{Pr}(\mathcal{I}) \prod_n \mathbf{Pr}(e_n|\mathcal{I}) \tag{4.4}$$

optional, unsupervised only: alternate with imputing programs:

$$e_n = \arg\max_{\substack{e: \\ e \models \varphi_n}} \mathbf{Pr}(e|\mathcal{I}^*) \tag{4.5}$$



Example synthesized programs	Proposed library expression

`(λ (ℓ) (map (λ (x) (index x ℓ))`
`(range (- (length ℓ) 1))))`
`(λ (ℓ) (map (λ (x) (index x ℓ))`
`(range (+ 1 1))))`
	`(map (λ (x) (index x ℓ))`
	`(range α))`

`(λ (s) (map (λ (x)`
`(if (= x '.') '-' x))) s)`
`(λ (s) (map (λ (x)`
`(if (= x '-') ',' x))) s)`
	`(λ (s) (map (λ (x)`
	`(if (= x α) β x))) s)`

Figure 4.10: Left: syntax trees of two programs sharing common structure, highlighted in orange, from which the system in Ellis *et al.*, 2018a extracts a fragment (reused syntactic template) and adds it to the library (bottom). **Right:** actual programs, and proposed fragments that (top) slice from the beginning of a list or (bottom) perform character substitutions.

As examples of *unsupervised* approaches, the EC^2 (Ellis *et al.*, 2018a) algorithm alternates between (1) searching over a space of possible libraries to find the library which best encodes the programs found during the synthesis step of Eq. 4.4 (equivalent to this maximum a posteriori formulation); then (2) updating $\{e_n\}_{n=1}^N$ with this new library according to Eq. 4.5; and (3) additionally training a neural program synthesis network (Sec. 4.3) which guides search during synthesis, aiding the optimization in Eq. 4.5. It also finds a set of programs solving each task, which helps if there is ambiguity as to the correct program. There is one such set for each of the N tasks, collectively written $\{\mathcal{B}_n\}_{n=1}^N$. Using this notation, EC^2 updates its library by marginalizing over programs found solving each task

$$\mathcal{I} \leftarrow \arg\max_{\mathcal{I}} \; \mathbf{Pr}(\mathcal{I}) \prod_n \sum_{e\in\mathcal{B}_n} \mathbf{Pr}(e|\mathcal{I}) \tag{4.6}$$

Its neural network guides the search during its program synthesis step, by predicting a spec-conditioned distribution over programs which is then enumerated from. Writing $Q(e|\varphi)$ for the distribution predicted by the neural network, the synthesis phase is

$$\mathcal{B}_n \leftarrow \mathop{\arg\operatorname{top}\mathrm{k}}_{\substack{e\sim Q(\cdot|\phi_n)\\ e\models\varphi_n}} \; \mathbf{Pr}(e|\mathcal{I}) \tag{4.7}$$

where k, number of programs we keep around for each task, is small (typically $k = 5$). Above, the notation $e \sim Q(\cdot|\phi_n)$ should be read as the expression being drawn from the distribution predicted by the neural network, but algorithmically EC^2 works by enumerating programs in decreasing order according to $Q(\cdot|\phi_n)$, and scoring them according to $\mathbb{1}\{e \models \varphi\}\,\mathbf{Pr}(e|\mathcal{I})$.

Last, EC^2 trains its neural network. The network learns to map a spec to a distribution over programs likely to solve it. This network is trained on programs found during the synthesis step (*i.e.* programs in $\{\mathcal{B}_n\}$ given the corresponding specs $\{\varphi_n\}$). Additionally however, we can use the current library to generate random programs and train on those too. As the library learns to put more probability mass on realistic programs, this randomly generated training data will grow more useful for learning:

$$\begin{aligned} Q \leftarrow \arg\min_{Q}\ & \mathbf{E}_\varphi\left[\mathrm{KL}(\mathbf{Pr}(\cdot|\mathcal{I},\varphi)\ ||\ Q(\cdot|\varphi))\right] \qquad (4.8)\\ \approx \arg\min_{Q}\ & \underbrace{\mathbf{E}_n\left[\mathrm{KL}(\mathbf{Pr}(\cdot|\mathcal{I},\varphi_n, e\in\mathcal{B}_n)\ ||\ Q(\cdot|\varphi_n))\right]}_{\text{train on programs found solving tasks}} + \\ & \underbrace{\mathbf{E}_{e\sim\mathbf{Pr}(\cdot|\mathcal{I})}\left[\log Q(e|\varphi_e)\right]}_{\text{train on randomly generated programs}} \quad \text{, for a } \varphi_e \text{ where } e\models\varphi_e \end{aligned}$$

This iterative scheme allows a kind of bootstrap learning that compensates for the lack of supervision. The easiest synthesis problems are solvable without the learned library, and ideally should expose transferable abstractions which allow harder synthesis problems to be solved. Over successive iterations, the learned library can build new abstractions on top of those learned earlier by calling out to them, much as human software engineers build more sophisticated libraries by calling out to code in more basic libraries (Ellis *et al.*, 2020). Thus even the pretraining phase hinges on transfer of abstractions across synthesis problems.

A complementary work in this space, PATOIS (Shin *et al.*, 2019a), combines library learning with semantic parsing. This work shows that training a neural semantic parser is more effective when it parses into expressions built using a learned library. They give a supervised library learning system which optimizes for libraries whose components are both *frequent* and *informative*. Learning driven only by frequency yields trivial components and learning driven only by informativeness yields non-reusable components.

To balance both objectives, the *idiom mining* process of PATOIS optimizes for high *coverage* as well as high *cross-entropy gain* of each idiom. Coverage measures the extent to which a candidate library component is used in the dataset ($\frac{1}{n}\sum_n \mathbb{1}\left\{\ell \in e_n\right\}$, where ℓ is a candidate library component). Cross-entropy gain measures a component's expressivity, defined as an average log-ratio of its probability under the inferred grammar over the probability yielded by the basic probabilistic grammar of the underlying language: $\frac{1}{\mathsf{size}(\ell)} \log \frac{\mathbf{Pr}(\ell|\mathcal{I})}{\mathbf{Pr}(\ell)}$, where $\mathsf{size}(\cdot)$ measures the size of a syntax tree, and $\mathbf{Pr}(\ell)$ is a description-length prior over library components.

Discussion. A key link between this symbolic library learning and neural programming is that, once we have this transferable library, we can train a neural program synthesis model to map between task specifications and a distribution over programs built from the library that are likely to solve that spec. Indeed, the MWS algorithm Hewitt *et al.*, 2020 performs both operations in tandem, jointly training a bank of neural modules together with a neural program synthesizer that learns to wire up these modules conditioned on a particular spec. PATIOS accomplishes an analogous integration by conditioning the neural synthesizer on natural language.

While transfer can support some forms of strong generalization (see Valkov *et al.*, 2018), the training examples should require concepts that are, at minimum, useful for solving later tasks. Dechter *et al.*, 2013 also shows that unsupervised methods require a spectrum of training task difficulty: otherwise, there is too large a gap between easy and hard problems, preventing the algorithm's bootstrapping dynamics. HOUDINI exploits this intuition further through its explicit curriculum of training problems.

4.8 Programmatic Weak Supervision

A completely different way to use programs is for them to generate training signals for conventional machine learning. This idea is common known as generating so-called "pseudo-labels" that can serve as proxies for supervised labels. Pseudo-labels have been used most prominently in the context of self-training (Daumé III, 2008; Sachan and Xing, 2018), whereby a model trained a small labeled dataset is used to generate labels on the larger unlabeled dataset. A recent interesting development is to instead use programs to generate the pseudo-labels instead to train the conventional machine learning approach (e.g., a handcrafted "labeling" function that automatically generates labels to train a deep neural network) (Ratner *et al.*, 2016; Zhan *et al.*, 2020; Sun *et al.*, 2020), which can lead to orders-of-magnitude savings in human effort compared to annotation.

The basic setting is as follows. For now, suppose we are given a program P that was hand-crafted by a domain expert. Then the goal is to learn a neural representation h to optimize

$$\sum_{x} \left[C_{supervised}(h(x), P(x)) + C_{unsupervised}(h, x) \right], \tag{4.9}$$

where $C_{supervised}$ quantifies the ability of $h(x)$ to predict $P(x)$, and $C_{unsupervised}$ is a standard unsupervised learning objective such as autoencoding (Kingma and Welling, 2013). The goal is for the program P to provide pseudo-labels that can pre-train the neural representation h to quickly learn from only a few ground-truth supervised labels.

The bulk of existing work has relied on hand-craft programmatic labeling functions, as embodied in the above example, but there is also interest in automatically learning such programs as well (Iyer *et al.*, 2019). In this latter case, the resulting learning problem becomes a a cyclical one, whereby the conventional (deep) learning component is providing a learning signal to the program learning component, and vice versa. The key intuition is that the inductive bias of the programming language is encouraging the program to learn or distill specific patterns from the conventional machine learning model (i.e., we do not know exactly what the right labels should be a priori, but they should satisfy certain structure as encoded by our programming language). In this case one would alternate between learning h using P as in (4.9), and learning P using h and the few ground truth labels. Figure 4.11 shows a depiction of this workflow from Iyer *et al.*, 2019.

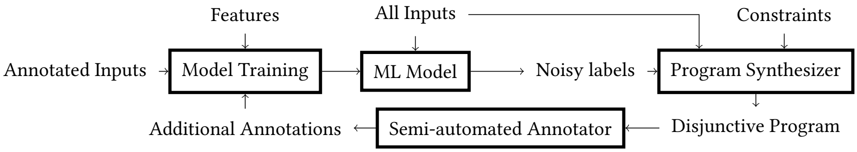


Figure 4.11: An example from Iyer *et al.*, 2019 on using programs to provide weak supervision for conventional machine learning. In this example, there is a cyclical process where the predictions of the conventional learning approach act as noisy labels that is "cleaned" using the programmatic structure. The program is then used as a semi-automated annotator (which is cleaned or augmented by human annotators) to further improve training of the conventional learning model.

5

Conclusion

In this monograph, we have discussed *neurosymbolic programming*, an emerging branch of computer science that brings together ideas from deep learning and program synthesis. The central ideas in this area are to use a combination of neural and symbolic primitives and to cast the learning problem as a form of program synthesis. Recent research has identified multiple distinct ways of bringing together statistical and symbolic techniques to solve this problem.

Now we summarize some of the challenges that research in this area needs to address in the short to medium term.

Scalability. Perhaps the most immediate challenge in neurosymbolic programming is scalability. Searching over discrete program architectures is a combinatorially hard problem, and despite sustained effort from the program synthesis community, program architectures that can be automatically discovered through search tend to be small. In contrast, end-to-end deep learning now scales to models with hundreds of billions of parameters.

One response to this challenge is to use high-performance computing. GPUs were a key contributor to the deep learning revolution. Likewise, methods for large-scale distribution and parallelization on specialized hardware could go a long way in scaling neurosymbolic programming. Prior work on parallelization of combinatorial problems such as SAT-solving (Hamadi and Wintersteiger, 2012; Dal Palù *et al.*, 2015), model counting (Fichte *et al.*, 2019), and model checking (Lerda and Sisto, 1999; Barnat *et al.*, 2010) can form a starting point for research on such methods.

However, better implementations can only go so far; we also need fundamental algorithmic innovation. One hope in neurosymbolic programming is that continuous methods and neural representations can massively accelerate the generation of programs. In this monograph, we have discussed multiple examples of such methods. However, these efforts only scrape the surface of what is possible.

A key perspective in neurosymbolic programming is that program synthesis and deep learning are part of a continuum, rather than discrete categories. In particular, several approaches in this monograph allow programs to be “largely neural”, i.e., describe the composition of a small number of neural modules. Such programs can describe complex functions with a small amount of symbolic code; thus, algorithms for learning these programs can be largely gradient-based. Naturally, this ease of learning requires us to trade off some of the benefits of symbolic models. Future research should systematically explore these tradeoffs.

Specification Neurosymbolic programming assumes that one can symbolically specify known facts about the world. However, such specification can be nontrivial. For example, DSLs for program learning need to be carefully designed. An overly restrictive DSL would prevent the discovery of interesting models. An overly permissive DSL would make search too difficult and also encourage overfitting. The recent DreamCoder project has explored neurosymbolic approaches to learn the DSL from data (Ellis *et al.*, 2020), but more research into this area is needed to fully address the challenges of hand-crafting DSLs.

The same goes for semantic constraints, in particular, requirements for safe neurosymbolic learning. There needs be sustained effort on designing “canonical” symbolic requirements that can constrain and guide learning in application domains. There are some efforts along these lines (Karimi and Duggirala, 2020; Dennis *et al.*, 2016), but much more can be done.

An important question in this kind of specification design is how to align symbolic specifications with representations learned using neural methods. For example, a DSL may use symbols such as `pedestrian`, `is-walking`, and `velocity`; however, associating these symbols to neural representations computed by a neurosymbolic program may be nontrivial. This *symbol grounding* problem (Harnad, 1990; Mooney, 2008) is well-established in many areas of AI. However, it is especially tricky in neurosymbolic programming, where symbolic and neural components can be interleaved in complex ways.

Applications Deep learning started making waves when neural methods convincingly outperformed the state of the art in real-world applications in speech processing, vision, natural language processing, and control. Similarly, for neurosymbolic programming to realize its promise, methods in this area must clearly outperform the state of the art in at least some high-impact applications. In Section 1.5, we have sketched some promising candidates for such applications. However, these efforts are only a start. There is a distance to go before practitioners in even the domains we have noted widely adopt neurosymbolic programming; also, there may be “killer apps” for neurosymbolic methods that have not been identified yet. The pursuit of such applications needs to be among the highest priorities of researchers in the area.

References

Achiam, J., D. Held, A. Tamar, and P. Abbeel. (2017). "Constrained Policy Optimization". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017*. Ed. by D. Precup and Y. W. Teh. Vol. 70. *Proceedings of Machine Learning Research*. PMLR. 22–31.

Alet, F., T. Lozano-Perez, and L. P. Kaelbling. (2018a). "Modular meta-learning". In: *Proceedings of The 2nd Conference on Robot Learning*. Ed. by A. Billard, A. Dragan, J. Peters, and J. Morimoto. Vol. 87. *Proceedings of Machine Learning Research*. PMLR. 856–868. URL: http://proceedings.mlr.press/v87/alet18a.html.

Alet, F., T. Lozano-Pérez, and L. P. Kaelbling. (2018b). "Modular meta-learning". *Conference on Robot Learning*.

Alshiekh, M., R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu. (2018). "Safe Reinforcement Learning via Shielding". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18)*. Ed. by S. A. McIlraith and K. Q. Weinberger. AAAI Press. 2669–2678.

Alur, R., 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. (2015). "Syntax-Guided Synthesis". In: *Dependable Software Systems Engineering*. 1–25. DOI: 10.3233/978-1-61499-495-4-1.

Amini, A., S. Gabriel, P. Lin, R. Koncel-Kedziorski, Y. Choi, and H. Hajishirzi. (2019). "MathQA: Towards interpretable math word problem solving with operation-based formalisms". *arXiv preprint arXiv:1905.13319*.

Amizadeh, S., H. Palangi, O. Polozov, Y. Huang, and K. Koishida. (2020). "Neuro-Symbolic Visual Reasoning: Disentangling "Visual" from "Reasoning"". In: *International Conference on Machine Learning*.

Anderson, G., S. Pailoor, I. Dillig, and S. Chaudhuri. (2019). "Optimization and abstraction: a synergistic approach for analyzing neural network robustness". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 731–744.

Anderson, G., A. Verma, I. Dillig, and S. Chaudhuri. (2020). "Neurosymbolic Reinforcement Learning with Formally Verified Exploration". In: *Neural Information Processesing Systems (NeurIPS)*.

Andreas, J., J. Bufe, D. Burkett, C. Chen, J. Clausman, J. Crawford, K. Crim, J. DeLoach, L. Dorner, J. Eisner, and et al. (2020). "Task-Oriented Dialogue as Dataflow Synthesis". *Transactions of the Association for Computational Linguistics*. 8(Dec.): 556–571.

Andreas, J., M. Rohrbach, T. Darrell, and D. Klein. (2016a). "Learning to Compose Neural Networks for Question Answering". In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. San Diego, California. 1545–1554.

Andreas, J., M. Rohrbach, T. Darrell, and D. Klein. (2016b). "Neural module networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 39–48.

Balog, M., A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. (2016). "DeepCoder: Learning to Write Programs". *CoRR*. abs/1611.01989. arXiv: 1611.01989.

Barnat, J., L. Brim, M. Češka, and P. Ročkai. (2010). "Divine: Parallel distributed model checker". In: *Ninth international workshop on parallel and distributed methods in verification, and second international workshop on high performance computational systems biology*. IEEE. 4–7.

Barrett, C., P. Fontaine, and C. Tinelli. (2010). *The SMT-LIB Standard Version 2.6*.

Bastani, O., Y. Pu, and A. Solar-Lezama. (2018). "Verifiable Reinforcement Learning via Policy Extraction". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. 2499–2509.

Beery, S., G. Van Horn, and P. Perona. (2018). "Recognition in Terra Incognita". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 456–473.

Bengio, Y. (2019). "From System 1 Deep Learning to System 2 Deep Learning". Neural Information Processing Systems.

Bickel, S., M. Brückner, and T. Scheffer. (2009). "Discriminative learning under covariate shift." *Journal of Machine Learning Research*. 10(9).

Bingham, E., J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. (2019). "Pyro: Deep Universal Probabilistic Programming". *J. Mach. Learn. Res.* 20(1): 973–978. ISSN: 1532-4435.

Brown, T. B., B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. (2020). "Language Models are Few-Shot Learners". *CoRR*. abs/2005.14165. arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.

Bunel, R., M. J. Hausknecht, J. Devlin, R. Singh, and P. Kohli. (2018). "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.

Carpenter, B., A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell. (2017). "Stan: A probabilistic programming language". *Journal of statistical software*. 76(1).

Chaudhuri, S., M. Clochard, and A. Solar-Lezama. (2014). "Bridging boolean and quantitative synthesis using smoothed proof search". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 207–220.

Chaudhuri, S. and A. Solar-Lezama. (2010). "Smooth interpretation". In: *PLDI*. 279–291.

Chen, T., S. Kornblith, M. Norouzi, and G. Hinton. (2020a). "A simple framework for contrastive learning of visual representations". In: *International Conference on Machine Learning*.

Chen, X., M. Monfort, A. Liu, and B. D. Ziebart. (2016). "Robust covariate shift regression". In: *Artificial Intelligence and Statistics*. 1270–1279.

Chen, X., C. Liang, A. W. Yu, D. Zhou, D. Song, and Q. V. Le. (2019). "Neural symbolic reader: Scalable integration of distributed and symbolic representations for reading comprehension". In: *International Conference on Learning Representations*.

Chen, Y., C. Wang, O. Bastani, I. Dillig, and Y. Feng. (2020b). "Program Synthesis Using Deduction-Guided Reinforcement Learning". In: *International Conference on Computer Aided Verification*. Springer. 587–610.

Cheng, R., A. Verma, G. Orosz, S. Chaudhuri, Y. Yue, and J. W. Burdick. (2019). "Control regularization for reduced variance reinforcement learning". In: *ICML*.

Cheung, A., A. Solar-Lezama, and S. Madden. (2012). "Using program synthesis for social recommendations". In: *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*. Ed. by X. Chen, G. Lebanon, H. Wang, and M. J. Zaki. ACM. 1732–1736.

Chow, Y., O. Nachum, E. Duenez-Guzman, and M. Ghavamzadeh. (2018). "A Lyapunov-based approach to safe reinforcement learning". In: *Advances in neural information processing systems*. 8092–8101.

Cranmer, M., A. Sanchez-Gonzalez, P. Battaglia, R. Xu, K. Cranmer, D. Spergel, and S. Ho. (2020). "Discovering symbolic models from deep learning with inductive biases". In: *Neural Information Processing Systems*.

Cui, G. and H. Zhu. (2021). "Differentiable Synthesis of Program Architectures". *Advances in Neural Information Processing Systems*. 34.

Dal Palù, A., A. Dovier, A. Formisano, and E. Pontelli. (2015). "Cud@ sat: Sat solving on gpus". *Journal of Experimental & Theoretical Artificial Intelligence*. 27(3): 293–316.

Daumé III, H. (2008). "Cross-Task Knowledge-Constrained Self Training". In: *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*. 680–688.

De Raedt, L., A. Kimmig, and H. Toivonen. (2007). "ProbLog: A Probabilistic Prolog and Its Application in Link Discovery." In: *IJCAI*. Vol. 7. Hyderabad. 2462–2467.

Dechter, E., J. Malmaud, R. P. Adams, and J. B. Tenenbaum. (2013). "Bootstrap learning via modular concept discovery". In: *Proceedings of the International Joint Conference on Artificial Intelligence*. AAAI Press/International Joint Conferences on Artificial Intelligence.

Dennis, L., M. Fisher, M. Slavkovik, and M. Webster. (2016). "Formal verification of ethical choices in autonomous systems". *Robotics and Autonomous Systems*. 77: 1–14.

Devlin, J., J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli. (2017). "RobustFill: Neural Program Learning under Noisy I/O". In: *ICML*.

Dua, D., Y. Wang, P. Dasigi, G. Stanovsky, S. Singh, and M. Gardner. (2019). "DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs". *arXiv preprint arXiv:1903.00161*.

Dudík, M., D. Erhan, J. Langford, L. Li, *et al.* (2014). "Doubly robust policy evaluation and optimization". *Statistical Science*. 29(4): 485–511.

Ellis, K., L. Morales, M. Sablé-Meyer, A. Solar-Lezama, and J. Tenenbaum. (2018a). "Learning libraries of subroutines for neurally–guided Bayesian program induction". In: *Advances in Neural Information Processing Systems*. 7805–7815.

Ellis, K., D. Ritchie, A. Solar-Lezama, and J. Tenenbaum. (2018b). "Learning to infer graphics programs from hand-drawn images". In: *Advances in neural information processing systems*. 6059–6068.

Ellis, K., C. Wong, M. Nye, M. Sable-Meyer, L. Cary, L. Morales, L. Hewitt, A. Solar-Lezama, and J. B. Tenenbaum. (2020). "Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning". *arXiv preprint arXiv:2006.08381*.

Fichte, J. K., M. Hecher, and M. Zisser. (2019). "An improved GPU-based SAT model counter". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 491–509.

Finn, C., P. Abbeel, and S. Levine. (2017). "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks". In: *ICML*.

Fulton, N. and A. Platzer. (2018). "Safe reinforcement learning via formal methods: Toward safe control through proof and learning". In: *Thirty-Second AAAI Conference on Artificial Intelligence*.

Garcez, A. S. d., K. B. Broda, and D. M. Gabbay. (2002). *Neural-symbolic learning systems: foundations and applications*. Springer Science & Business Media.

Garcez, A. S. d'Avila and G. Zaverucha. (1999). "The Connectionist Inductive Learning and Logic Programming System". *Appl. Intell.* 11(1): 59–77. DOI: 10.1023/A:1008328630915.

Garcıa, J. and F. Fernández. (2015). "A comprehensive survey on safe reinforcement learning". *Journal of Machine Learning Research*. 16(1): 1437–1480.

Gaunt, A. L., M. Brockschmidt, R. Singh, N. Kushman, P. Kohli, J. Taylor, and D. Tarlow. (2016). "Terpret: A probabilistic programming language for program induction". *arXiv preprint arXiv:1608.04428*.

Gaunt, A. L., M. Brockschmidt, N. Kushman, and D. Tarlow. (2017). "Differentiable Programs with Neural Libraries". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. 1213–1222.

Gehr, T., M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. (2018). "AI^2: Safety and robustness certification of neural networks with abstract interpretation". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 3–18.

Giles, C. L., C. B. Miller, D. Chen, H.-H. Chen, G.-Z. Sun, and Y.-C. Lee. (1992). “Learning and extracting finite state automata with second-order recurrent neural networks”. *Neural Computation*. 4(3): 393–405.

Graves, A., G. Wayne, and I. Danihelka. (2014). “Neural turing machines”. *arXiv preprint arXiv:1410.5401*.

Grefenstette, E., K. M. Hermann, M. Suleyman, and P. Blunsom. (2015). “Learning to transduce with unbounded memory”. *Advances in neural information processing systems*. 28: 1828–1836.

Gulwani, S., O. Polozov, and R. Singh. (2017). “Program synthesis”. *Foundations and Trends in Programming Languages*. 4(1-2): 1–119.

Hamadi, Y. and C. Wintersteiger. (2012). “Seven challenges in parallel SAT solving”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 26. No. 1.

Harnad, S. (1990). “The symbol grounding problem”. *Physica D: Nonlinear Phenomena*. 42(1-3): 335–346.

Hendrycks, D., M. Mazeika, S. Kadavath, and D. Song. (2019). “Using self-supervised learning can improve model robustness and uncertainty”. In: *Advances in Neural Information Processing Systems*. 15663–15674.

Herzig, J., P. K. Nowak, T. Müller, F. Piccinno, and J. M. Eisenschlos. (2020). “TAPAS: Weakly Supervised Table Parsing via Pre-training”. *arXiv preprint arXiv:2004.02349*.

Hewitt, L. B., T. A. Le, and J. B. Tenenbaum. (2020). “Learning to learn generative programs with Memoised Wake-Sleep”. *UAI*.

Hinton, G. E., O. Vinyals, and J. Dean. (2015). “Distilling the Knowledge in a Neural Network”. *CoRR*. abs/1503.02531. arXiv: 1503.02531. URL: http://arxiv.org/abs/1503.02531.

Hu, R., J. Andreas, M. Rohrbach, T. Darrell, and K. Saenko. (2017). “Learning to reason: End-to-end module networks for visual question answering”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 804–813.

Hutter, F., D. Babic, H. H. Hoos, and A. J. Hu. (2007). "Boosting Verification by Automatic Tuning of Decision Procedures". In: *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings.* IEEE Computer Society. 27–34.

Iyer, A., M. Jonnalagedda, S. Parthasarathy, A. Radhakrishna, and S. K. Rajamani. (2019). "Synthesis and machine learning for heterogeneous extraction". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 301–315.

James, S., P. Wohlhart, M. Kalakrishnan, D. Kalashnikov, A. Irpan, J. Ibarz, S. Levine, R. Hadsell, and K. Bousmalis. (2019). "Sim-to-real via sim-to-sim: Data-efficient robotic grasping via randomized-to-canonical adaptation networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 12627–12637.

Jha, S., S. Gulwani, S. A. Seshia, and A. Tiwari. (2010). "Oracle-guided component-based program synthesis". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1. ICSE '10.* Cape Town, South Africa: ACM. 215–224.

Ji, R., Y. Sun, Y. Xiong, and Z. Hu. (2020). "Guiding Dynamic Programing via Structural Probability for Accelerating Programming by Example". *Proc. ACM Program. Lang.* 4(OOPSLA).

Kahneman, D. (2011). *Thinking, fast and slow.* Macmillan.

Kalyan, A., A. Mohta, O. Polozov, D. Batra, P. Jain, and S. Gulwani. (2018). "Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings.*

Karimi, A. and P. S. Duggirala. (2020). "Formalizing traffic rules for uncontrolled intersections". In: *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS).* IEEE. 41–50.

Katz, G., C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. (2017). "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks". *CoRR.* abs/1702.01135.

Keysers, D., N. Schärli, N. Scales, H. Buisman, D. Furrer, S. Kashubin, N. Momchev, D. Sinopalnikov, L. Stafiniak, T. Tihon, D. Tsarkov, X. Wang, M. van Zee, and O. Bousquet. (2020). "Measuring Compositional Generalization: A Comprehensive Method on Realistic Data". In: *International Conference on Learning Representations*.

Kim, B., M. Wattenberg, J. Gilmer, C. J. Cai, J. Wexler, F. B. Viégas, and R. Sayres. (2018). "Interpretability Beyond Feature Attribution: Quantitative Testing with Concept Activation Vectors (TCAV)". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by J. G. Dy and A. Krause. Vol. 80. *Proceedings of Machine Learning Research*. PMLR. 2673–2682.

Kingma, D. P. and M. Welling. (2013). "Auto-encoding variational Bayes". *arXiv preprint arXiv:1312.6114*.

Krawiec, K. (2016). *Behavioral program synthesis with genetic programming*. Vol. 618. Springer.

Le Goues, C., T. Nguyen, S. Forrest, and W. Weimer. (2011). "Genprog: A generic method for automatic software repair". *IEEE transactions on software engineering*. 38(1): 54–72.

Lerda, F. and R. Sisto. (1999). "Distributed-memory model checking with SPIN". In: *International SPIN Workshop on Model Checking of Software*. Springer. 22–39.

Levine, S. and V. Koltun. (2013). "Guided policy search". In: *International Conference on Machine Learning*. 1–9.

Liang, P., M. I. Jordan, and D. Klein. (2010). "Learning programs: A hierarchical Bayesian approach". In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 639–646.

Liu, A., G. Shi, S.-J. Chung, A. Anandkumar, and Y. Yue. (2020). "Robust regression for safe exploration in control". In: *Learning for Dynamics and Control*. PMLR. 608–619.

Liu, D., H. Zhang, F. Wu, and Z.-J. Zha. (2019a). "Learning to Assemble Neural Module Tree Networks for Visual Grounding". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*.

Liu, H., K. Simonyan, and Y. Yang. (2019b). “DARTS: Differentiable Architecture Search”. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. URL: https://openreview.net/forum?id=S1eYHoC5FX.

Liu, Y., Y. Sun, B. Xue, M. Zhang, G. G. Yen, and K. C. Tan. (2021). “A survey on evolutionary neural architecture search”. *IEEE Transactions on Neural Networks and Learning Systems*.

Manhaeve, R., S. Dumancic, A. Kimmig, T. Demeester, and L. De Raedt. (2018). “DeepProbLog: Neural probabilistic logic programming”. *Advances in Neural Information Processing Systems*. 31: 3749–3759.

Manna, Z. and R. J. Waldinger. (1971). “Toward Automatic Program Synthesis”. *Communications of the ACM*. 14(3): 151–165.

Mao, J., X. Zhang, Y. Li, W. T. Freeman, J. B. Tenenbaum, and J. Wu. (2019). “Program-guided image manipulators”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 4030–4039.

Marcus, G. and E. Davis. (2019). *Rebooting AI: Building artificial intelligence we can trust*. Pantheon.

Moldovan, T. M. and P. Abbeel. (2012). “Safe Exploration in Markov Decision Processes”. In: *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*. icml.cc / Omnipress. URL: http://icml.cc/2012/papers/838.pdf.

Mooney, R. J. (2008). “Learning to Connect Language and Perception.” In: *AAAI*. 1598–1601.

Moura, L. M. de and N. Bjørner. (2008). “Z3: An Efficient SMT Solver”. In: *TACAS*. 337–340.

Murali, A. and P. Madhusudan. (2019). “Augmenting Neural Nets with Symbolic Synthesis: Applications to Few-Shot Learning”. *CoRR*. abs/1907.05878. arXiv: 1907.05878. URL: http://arxiv.org/abs/1907.05878.

Murali, V., L. Qi, S. Chaudhuri, and C. Jermaine. (2018). “Neural Sketch Learning for Conditional Program Generation”. *ICLR*.

Neal, R. M. (2011). “MCMC Using Hamiltonian Dynamics”. In: *Handbook of Markov Chain Monte Carlo*. CRC Press. Chap. 5. DOI: 10.1201/b10905-7.

Parisotto, E., A. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli. (2017). "Neuro-Symbolic Program Synthesis". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Polozov, O. and S. Gulwani. (2015). "FlashMeta: a framework for inductive program synthesis". In: *ACM SIGPLAN Notices*. Vol. 50. No. 10. ACM. 107–126.

Ratner, A. J., C. M. De Sa, S. Wu, D. Selsam, and C. Ré. (2016). "Data programming: Creating large training sets, quickly". In: *Advances in neural information processing systems*. 3567–3575.

Real, E., C. Liang, D. So, and Q. Le. (2020). "Automl-zero: Evolving machine learning algorithms from scratch". In: *International Conference on Machine Learning*. PMLR. 8007–8019.

Reed, S. and N. De Freitas. (2015). "Neural programmer-interpreters". *arXiv preprint arXiv:1511.06279*.

Riedel, S., M. Bosnjak, and T. Rocktäschel. (2016). "Programming with a differentiable Forth interpreter". *CoRR, abs/1605.06640*.

Sachan, M. and E. Xing. (2018). "Self-training for jointly learning to ask and answer questions". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. 629–640.

Sahoo, S. S., S. Venugopalan, L. Li, R. Singh, and P. F. Riley. (2020). "Scaling Symbolic Methods using Gradients for Neural Model Explanation". *CoRR*. abs/2006.16322. URL: https://arxiv.org/abs/2006.16322.

Shah, A., E. Zhan, J. J. Sun, A. Verma, Y. Yue, and S. Chaudhuri. (2020). "Learning Differentiable Programs with Admissible Neural Heuristics". In: *Advances in Neural Information Processing Systems*.

Shavlik, J. W. (1994). "Combining symbolic and neural learning". *Machine Learning*. 14(3): 321–331.

Shin, R., M. Allamanis, M. Brockschmidt, and O. Polozov. (2019a). "Program synthesis and semantic parsing with learned code idioms". In: *Advances in Neural Information Processing Systems*. 10825–10835.

Shin, R., N. Kant, K. Gupta, C. Bender, B. Trabucco, R. Singh, and D. Song. (2019b). "Synthetic Datasets for Neural Program Synthesis". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. URL: https://openreview.net/forum?id=ryeOSnAqYm.

Si, X., M. Raghothaman, K. Heo, and M. Naik. (2019). "Synthesizing datalog programs using numerical relaxation". In: *IJCAI*.

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. (2016). "Mastering the Game of Go with Deep Neural Networks and Tree Search". *Nature*. 529(7587): 484–489. DOI: 10.1038/nature16961.

Smolensky, P., M. Lee, X. He, W. Yih, J. Gao, and L. Deng. (2016). "Basic Reasoning with Tensor Product Representations". *CoRR*. abs/1601.02745. arXiv: 1601.02745. URL: http://arxiv.org/abs/1601.02745.

Solar-Lezama, A. (2009). "The Sketching Approach to Program Synthesis". In: *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*. 4–13. DOI: 10.1007/978-3-642-10672-9_3.

Solar-Lezama, A., L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. (2006). "Combinatorial sketching for finite programs". In: *ASPLOS*. 404–415.

Srivastava, S., O. Polozov, N. Jojic, and C. Meek. (2020). "Learning Web-based Procedures by Reasoning over Explanations and Demonstrations in Context". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 7652–7662.

Sun, J. J., A. Kennedy, E. Zhan, D. J. Anderson, Y. Yue, and P. Perona. (2021). "Task programming: Learning data efficient behavior representations". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2876–2885.

Sun, J. J., A. Kennedy, E. Zhan, Y. Yue, and P. Perona. (2020). "Task Programming: Learning Data Efficient Behavior Representations". *arXiv preprint arXiv:2011.13917*.

Sun, R. and F. Alexandre. (2013). *Connectionist-symbolic integration: From unified to hybrid approaches*. Psychology Press.

Sundararajan, M., A. Taly, and Q. Yan. (2017). "Axiomatic Attribution for Deep Networks". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by D. Precup and Y. W. Teh. Vol. 70. *Proceedings of Machine Learning Research*. PMLR. 3319–3328.

Sutton, R. S., D. Precup, and S. Singh. (1999). "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning". *Artificial intelligence*. 112(1-2): 181–211.

Taori, R., A. Dave, V. Shankar, N. Carlini, B. Recht, and L. Schmidt. (2020). "Measuring robustness to natural distribution shifts in image classification". *Advances in Neural Information Processing Systems*. 33.

Tavares, Z., J. Burroni, E. Minasyan, A. Solar-Lezama, and R. Ranganath. (2019). "Predicate Exchange: Inference with Declarative Knowledge". In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by K. Chaudhuri and R. Salakhutdinov. Vol. 97. *Proceedings of Machine Learning Research*. PMLR. 6186–6195. URL: http://proceedings.mlr.press/v97/tavares19a.html.

Taylor, L. and G. Nitschke. (2017). "Improving deep learning using generic data augmentation". *arXiv preprint arXiv:1708.06020*.

Tjandrasuwita, M., J. J. Sun, A. Kennedy, S. Chaudhuri, and Y. Yue. (2021). "Interpreting Expert Annotation Differences in Animal Behavior". *CoRR*. abs/2106.06114. arXiv: 2106.06114. URL: https://arxiv.org/abs/2106.06114.

Torlak, E. and R. Bodik. (2013). "Growing solver-aided languages with Rosette". In: *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM. 135–152.

Towell, G. G. and J. W. Shavlik. (1993). "Extracting refined rules from knowledge-based neural networks". *Machine learning*. 13(1): 71–101.

Towell, G. G., J. W. Shavlik, and M. O. Noordewier. (1990). "Refinement of approximate domain theories by knowledge-based neural networks". In: *Proceedings of the eighth National conference on Artificial intelligence*. Vol. 861866. Boston, MA.

Udrescu, S.-M. and M. Tegmark. (2020). "AI Feynman: A physics-inspired method for symbolic regression". *Science Advances*. 6(16): eaay2631.

Udupa, A., A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur. (2013). "TRANSIT: specifying protocols with concolic snippets". *ACM SIGPLAN Notices*. 48(6): 287–296.

Valiant, L. G. (1984). "A theory of the learnable". *Communications of the ACM*. 27(11): 1134–1142.

Valkov, L., D. Chaudhari, A. Srivastava, C. Sutton, and S. Chaudhuri. (2018). "HOUDINI: Lifelong Learning as Program Synthesis". In: *Advances in Neural Information Processing Systems*. 8701–8712.

Vedantam, R., K. Desai, S. Lee, M. Rohrbach, D. Batra, and D. Parikh. (2019). "Probabilistic Neural-symbolic Models for Interpretable Visual Question Answering". In: *ICML*.

Verma, A., H. M. Le, Y. Yue, and S. Chaudhuri. (2019). "Imitation-Projected Programmatic Reinforcement Learning". In: *Neural Information Processing Systems (NeurIPS)*.

Verma, A., V. Murali, R. Singh, P. Kohli, and S. Chaudhuri. (2018). "Programmatically Interpretable Reinforcement Learning". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. 5052–5061.

Wang, P., P. L. Donti, B. Wilder, and J. Z. Kolter. (2019). "SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver". In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. 6545–6554.

Wang, X. and J. Schneider. (2014). "Flexible transfer learning under support and model shift". *Advances in Neural Information Processing Systems*. 27: 1898–1906.

Xu, D., S. Nair, Y. Zhu, J. Gao, A. Garg, L. Fei-Fei, and S. Savarese. (2018). "Neural task programming: Learning to generalize across hierarchical tasks". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 3795–3802.

Yosinski, J., J. Clune, Y. Bengio, and H. Lipson. (2014). "How transferable are features in deep neural networks?" In: *Advances in neural information processing systems*. 3320–3328.

Young, H., O. Bastani, and M. Naik. (2019). "Learning neurosymbolic generative models via program synthesis". *arXiv preprint arXiv:1901.08565*.

Zhan, E., J. J. Sun, A. Kennedy, Y. Yue, and S. Chaudhuri. (2021). "Unsupervised Learning of Neurosymbolic Encoders". *CoRR*. abs/2107.13132. arXiv: 2107.13132. URL: https://arxiv.org/abs/2107.13132.

Zhan, E., A. Tseng, Y. Yue, A. Swaminathan, and M. Hausknecht. (2020). "Learning Calibratable Policies using Programmatic Style-Consistency". In: *International Conference on Machine Learning*. PMLR. 11001–11011.

Zhang, X., A. Solar-Lezama, and R. Singh. (2018). "Interpreting Neural Network Judgments via Minimal, Stable, and Symbolic Corrections". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. 4879–4890.

Zhu, H., Z. Xiong, S. Magill, and S. Jagannathan. (2019). "An inductive synthesis framework for verifiable reinforcement learning". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by K. S. McKinley and K. Fisher. ACM. 686–701.