Neurosymbolic Programming: Overview and Case Study

Zhimin Zhao
Student ID number: 20311734
School of Computing, Queen's University
Kingston, ON, Canada
z.zhao@queensu.ca

I. Introduction

To save software developers from time-consuming but trivial efforts, researchers have proposed many approaches to generate source code automatically according to specification, i.e. program synthesis. Examples of the specification are input-output pairs that the program must satisfy or a piece of natural language description of the program. The correct program P takes input x generates output $y = P(x)$. We expect the synthesizer generates candidate program $\hat{P}$ where $\hat{y} = \hat{P}(x)$. In comparison, program induction makes no explicit use of program P since the model induces a latent representation of the program P to fit the input-output examples (x, y).

Traditional program synthesis primarily dealt with this paradigm since it is highly reliable and interpretable [1], [2]. However, the symbolic approach has many caveats. It remains challenging to capture the world entirely using rules and logic since many things are not even close to computable, let alone there is a little mechanism to handle sensory inputs for symbolism. Even if we model the investigated topic as some planning or theorem-proving, reasoning-based synthesis techniques suffer from prohibitively large search space in real-world problems, which can only scale so far. Furthermore, humans tend to make mistakes, and instruments tend to be inaccurate, so the collected data may have some noise. However, the symbolic approach is often vulnerable to noise and is effectively "broken" confronting ill-formedness [1]. Moreover, the symbolic approach depends on hand-crafted semantics to guide program synthesis. The semantics often requires human supervision and intervention to create matching specifications in a formal language [2].

With the lower cost of computing resources, researchers are embracing sub-symbolic approaches (machine learning etc.) to generate source code from noisy and unstructured specifications written in natural language [3], [4]. However, this methodology is far from perfect. First, machine learning models, particularly neural networks, are black boxes for most practical purposes. The modern neural architecture allows the composition of hundreds or even thousands of layers, rendering it impractical to know how and why they perform specific behaviours. Also, the training process is entirely data-driven. The model has to learn even the most basic human knowledge from data. Moreover, values of hidden variables change over time, often resulting in synthesized programs significantly different from reference implementations [1]. Also, neural networks map between continuous values by default. In deep learning, embedding is a mapping of a discrete variable to distributed representation, i.e. a vector of continuous numbers. However, unlike speech and images, program units (statement, block, function, module, etc.) are discrete. That leads to the closeness in values not reflecting the semantic relevance, degrading the task generalization for the learned embedding [5].

Recently, there have been researching efforts toward integrating symbolic reasoning and machine learning models under the umbrella term of neurosymbolic programming [6]. This composite computational cognitive model preserves the strengths of both symbolic and sub-symbolic AI while suppressing its weaknesses. The models are good at learning function semantics of the ingested programs, rather than a particular source code distribution. This paper provides a brief overview of recent studies on the topic and a case study for elaboration.

The remainder of this paper is structured as follows. Section II describes the preliminaries of symbolic and sub-symbolic artificial intelligence. Section III details the steps taken to retrieve the literature. Section IV presents the research results of our study. Section V discusses challenges, progress, and prospects. Section VI discusses the threats to the validity of our study and the means we used to alleviate these threats whenever possible. Section VII concludes our study and points directions for future work.

II. Background

This section introduces the background information of our study. Section II-A introduces preliminaries of symbolic and sub-symbolic AI, while Section II-B gives a brief review of program synthesis.

A. Symbolic and Sub-symbolic AI

a) Symbolic AI: is the collection of methods in AI research that rely on human-readable representations of knowledge, constraints, and reasoning [7]. Symbolic AI belongs to the category of hard computing, in which algorithms aim to find provably correct and optimal solutions to problems. Examples include search heuristics, genetic algorithms, inference engines, etc. Contrary to symbolic approaches, which

attempt to solve problems using a top-down methodology, sub-symbolic approaches use the bottom-up principle to gradually optimize the sub-symbols (parameters, embeddings, etc.) until it delivers the expected results. The sub-symbolic AI includes statistical learning approaches such as deep learning, Bayesian network, support vector machine, hidden Markov model, etc.

Before deep-diving into the neuro-symbolic specifics, we give a review of the preliminaries of machine learning in the following to make the study self-contained.

Machine Learning (ML) is a set of artificial intelligence (AI) techniques that aim to make decisions from data. ML is not a new concept but was popularized by Arthur Samuel in 1959 [8]. ML has three fundamental constituents:

- **Dataset**: is a collection of data instances that encode information about an object. The dataset is categorized into three types based on the purpose:
  - **Training dataset**: is the data used to learn the model.
  - **Validation dataset**: is the data used to evaluate and tune the model.
  - **Test dataset**: is the data used to test the model performance.
- **Algorithm**: is a set of instructions by which the model can learn from data. Every machine learning algorithm includes three key components:
  - **Representation**: is a form of knowledge learned from data.
  - **Evaluation**: is the way to measure the performance of a model. The evaluation metrics are often task-oriented. Examples include *precision-recall* for classification, *R square* for regression, and *average silhouette coefficient* for clustering, etc.
  - **Optimization**: is a search process to learn a satisfactory model from data.
- **Model**: is a program that can make decisions based on the knowledge learned from data. There are three main kinds of ML models based on learning paradigms:
  - **Supervised learning**: is an approach that learns a model with labelled input and output datasets.
  - **Unsupervised learning**: is an approach that learns a model with an unlabeled dataset and aims to understand its characteristics.
  - **Reinforcement learning**: is an approach that learns models from the sequential training dataset and aims to interact in a preset environment to maximize the rewards. For example, figure 1 shows a typical scenario: an agent takes actions in an environment; the observer interprets the action into a reward and a state representation; the agent receives the feedback and takes another action.

  There are also a few key concepts to the training of ML models:
  - **Pattern**: is the regularity and similarity in data.
  - **Feature**: is an observable and measurable property of data.
  - **Hyperparameter**: is a parameter used to control the learning process.

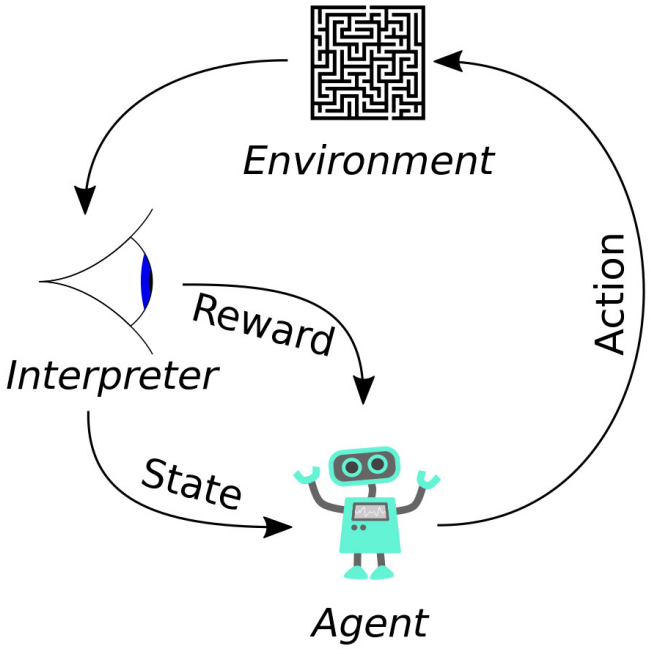

Fig. 1: A typical RL scenario from Wikipedia

  - **Epoch**: is a hyperparameter that represents an entire dataset that passes forward and backward through the neural network once.
  - **Batch/Mini-batch**: is the equally sized subset of the dataset over which the model parameters are updated in one iteration.
  - **Inference** is to calculate an output by running a data input into a model.

Deep learning (DL) is a subfield of machine learning, and neural networks make up its backbone. DL algorithms focus on extracting compact, numerical representations for sources of signal [9]. We list a few fundamental elements unique to a deep learning system below.

- **Artificial neural network** (ANN): is a multi-layered architecture of non-linear processing units for feature extraction and pattern recognition. ANN is inspired by the biological neurons, which fire under certain stimulation leading to a corresponding action performed by the body in response. For a typical feedforward neural network (FNN), as shown in Figure 4, we list the key elements for better understanding in the following.
  - **Neuron**: is mathematical function (Figure 2) that model the functionality of a biological neuron (Figure 3).
  - **Weight**: is a learnable parameter which represents the strength of neuron connection.
  - **Bias**: is a learnable parameter which transposes the product of weight and input.
  - **Activation function**: decides whether the neuron should be activated or not. Figure 5 shows that the sigmoid (for binary classification) or softmax (for multi-classification) non-linearity squashes real numbers to range $[0, 1]$.

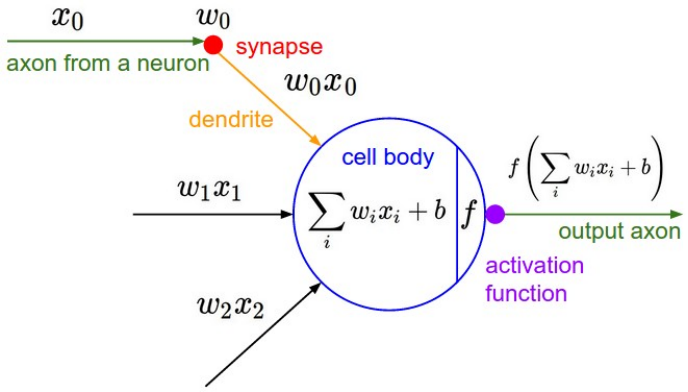

Fig. 2: Mathematical Model of Neuron [10]

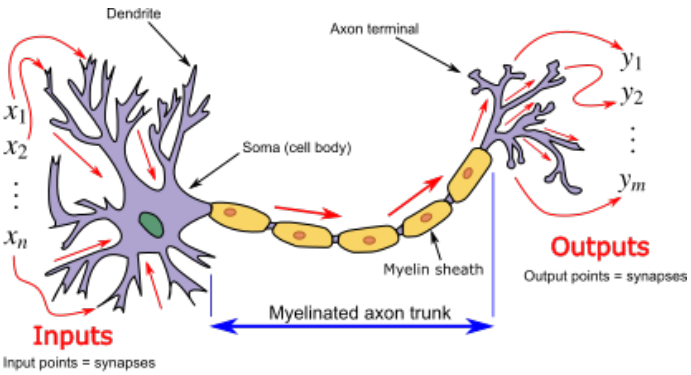

Fig. 3: Neuron and myelinated axon from Wikipedia

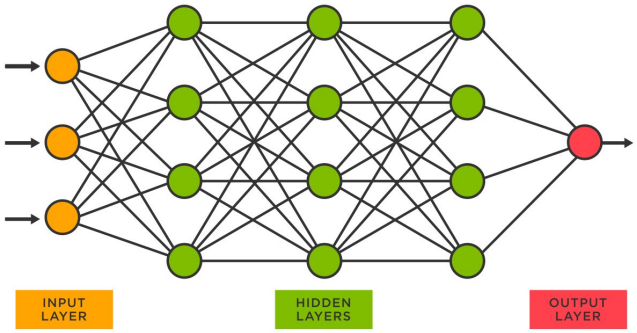

Fig. 4: A 4-layer FNN with 3 inputs, 3 hidden layers of 4 neurons each and 1 output layer.

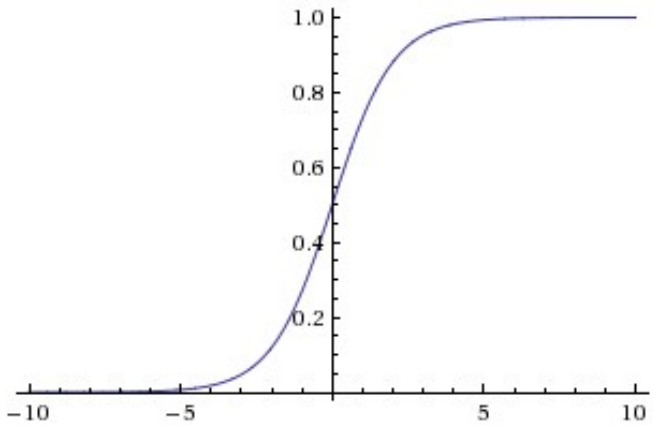

Fig. 5: Sigmoid Function from Wikipedia

– **Input layer**: represents dimensions of the input vector.
– **Hidden layer**: represents the intermediary nodes in which a set of weighted input plus the bias produces output through the activation function.
– **Output layer**: represents the output of the neural network.

We brief the following concepts in training a neural network.

– **Loss/Cost function**: is an evaluation metric of how well the neural network models the dataset.
– **Backpropagation**: is the process of calculating partial derivatives of the loss function concerning neural network parameters.
– **Gradient descent**: is an iterative optimization algorithm to adjust neural network parameters through backpropagation until the termination condition is reached.

There are common types of neural networks in the following.

– **Multi-layer perceptron** (MLP): is another name for FNN, which composes sequential layers of function compositions. MLP is the simplest form of neural network.
– **Convolutional neural network** (CNN): is a specialized neural network that uses convolution in place of matrix multiplication to mimic the human visual system. CNN is typically used for image classification and recognition. As shown in Figure 6, a typical CNN consists of the following components:

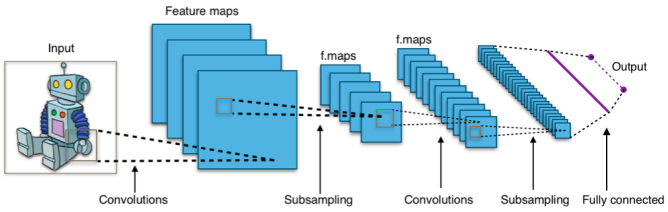

Fig. 6: A Typical CNN from Wikipedia

∗ **Filter/Kernel**: is a sliding matrix consisting of learnable weights to extract features from the images. We use filters to scan the images and generate feature maps afterwards.
∗ **Stride**: is the number of pixels that the filter moves on each step.
∗ **Padding**: is the addition of (typically) 0-valued pixels around the edges of an image.
∗ **Feature map**: is the output of one filter applied to the previous layer.
∗ **Convolution layer**: is the layer where filters are applied to the images.
∗ **Pooling**: is a form of non-linear subsampling. There are several functions to implement pooling.

For example, *max pooling* outputs the maximum for each sub-region partitioned from the images.

– **Recurrent neural network** (RNN): is a specialized neural network that can handle a variable-length sequential input. As shown in Figure 7, the information cycles through a loop to the hidden layer. RNN is typically used in speech recognition and natural language processing. We list some common RNN variants in the following.

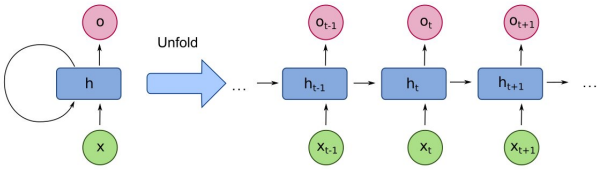

Fig. 7: Compressed (left) and unfolded (right) RNN from Wikipedia

∗ **Long short-term memory** (LSTM): uses three "gates", i.e. *forget gate input gate* and *output gate*, to control the data flow in and out the network. These gates serve as filters with each their own neural network. Studies [11] show that LSTM can memorize events that happened thousands or even millions of discrete time steps earlier.

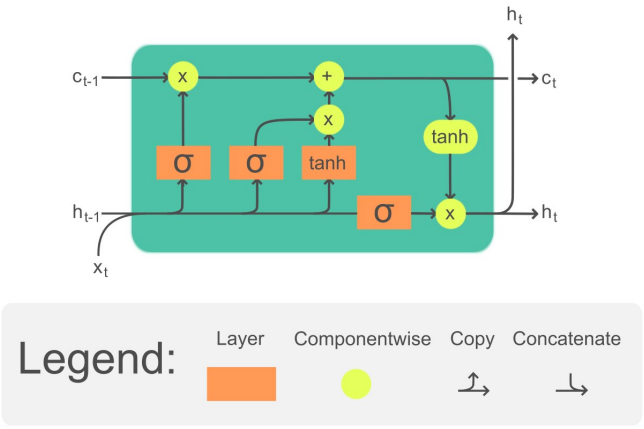

Fig. 8: LSTM Cell from Wikipedia

∗ **Gated recurrent unit** (GRU): use two "gate", i.e. *reset gate* and *update gate*, to control the data flow in and out the network. If the dataset is small then GRU is preferred.

– **Graph neural network** (GNN): is a specialized neural network that can perform inference on data described by graphs (Figure 10). Usually, GNN captures the dependency of the graph via message passing between nodes of the graph.

The following concepts are useful when we synthesize programs using a neural network.

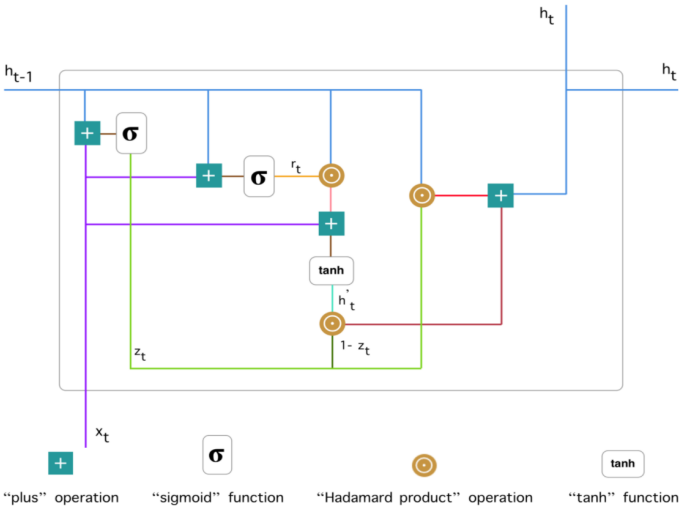

Fig. 9: Gated Recurrent Unit [12]

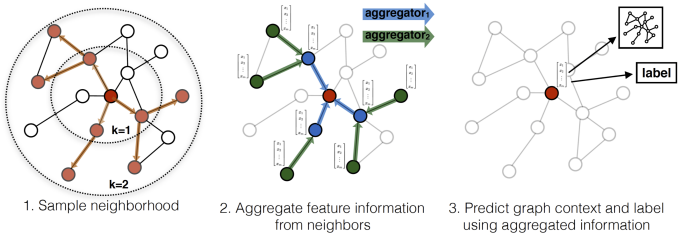

Fig. 10: Visual illustration of the GraphSAGE sample and aggregate approach [13]

– **Vocabulary**: represents the set of unique semantic units in the source code corpus.
– **Embedding**: is a low-dimensional continuous vector representation of a semantic unit. The embedding vector gets updated while training the neural network.
– **Attention**: is a mechanism that mimics cognitive attention by a weighted combination of all of the encoded input vectors. As shown in Figure 11, the effect strengthens some parts of the input data while weakening other parts. The model can selectively focus on the valuable information and learn the association between them with attention.

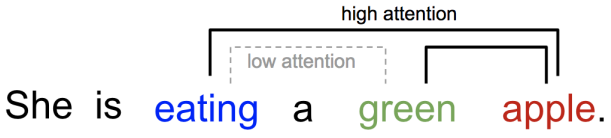

Fig. 11: Attention Mechanism for A Sentence

B. Program Synthesis at a Glance

There have been three waves of a technology shift in program synthesis according to Solar-Lezama's talk [1] in MAPL

[1] pldi20.sigplan.org/details/mapl-2020-papers/12/Neurosymbolic-Reasoning-and-the-Third-Wave-of-Program-Synthesis

2020.

The first wave of synthesis is based on deductive reasoning. Deductive reasoning focuses on progressing from general ideas to specific conclusions. For example, Church first proposed the circuit synthesis problem [14] in the 1960s. The idea is to take a formal specification and work backward, searching for a program that provably satisfies the specification. Some of these early synthesizers build on the successes of automated theorem provers, using them to construct a proof of the specification automatically, and then extract a program from that proof. Another branch of techniques tries to apply a comprehensive set of rewriting rules to transform the specification into an equivalent program.

The second wave of synthesis research focuses on inductive reasoning. Inductive reasoning focuses on progressing from specific ideas to general conclusions. Compared with deductive approaches, inductive synthesis does not demand the axiomatization of target languages or a complete formal specification. The most popular style is counterexample-guided inductive synthesis (CEGIS), pioneered by Sketch [15]. As shown in Figure 12, CEGIS first heuristically guesses a candidate program that works on partial inputs and then checks if the program works on all inputs. If not, we learn something from the counterexamples to inform future guesses and iterate the guess-check cycle. FlashFill [16] is a flagship success of the inductive approach, which has been yet made into a string transformation feature in Microsoft Excel. Notably, the idea of domain-specific language (DSL) underpins many inductive synthesis tools such as Sketch [15] or Rosette [17].

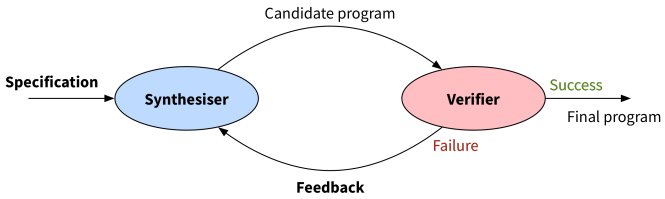

Fig. 12: Counterexample-guided inductive synthesis [18]

The third wave of program synthesis is driven by machine learning. Compared with previous approaches, data-driven specifications such as natural language documents and 3d print pictures bring synthesis techniques to more users without imposing a formality burden. Recently, large-scale language models have demonstrated an impressive ability to synthesize programs and are now able to complete easy to middle-level programming tasks. In 2020, Kevin Scott, CTO of Microsoft, showed the public a demo [19] of a *GPT-3* [20] variant model trained on repositories from Github. If you watch the broadcast and skip to 31 minutes, as shown in Figure 13, given only a function name like $compute_total_price$ and a short comment depicting what the function should do, the algorithm wrote the rest of the function in real-time. In February, DeepMind published AlphaCode [21], which can solve competitive programming exercises as well as an average human programmer (Top 54.3%). Another branch of approach focuses on training a model to guide existing synthesis techniques effectively. For instance, DeepCoder [22] uses a feedforward neural network with encoder-decoder architecture to guide the brute-force enumeration of candidate programs. These learned models effectively replace hand-tuned heuristics with ones learned from the dataset.

```
@dataclass
class Order:
    id: int
    items: List[Item]

    def compute_total_price(self, palindrome_discount=0.2):
        """
        Compute the total price and return it.
        Apply a discount to items whose names are palindromes.
        """
        total_price = 0
        for item in self.items:
            if is_palindrome(item.name):
                total_price += item.price * (1 - palindrome_discount)
            else:
                total_price += item.price
        return total_price
```

Fig. 13: Screenshot from the Demo [19]

III. Methodology

For the scope of this review and to maximize its validity, we follow a replicable approach to conduct our study based on the guidelines by [23].

A. Data Collection

The data collection process must identify primary studies related to neurosymbolic programming. We discuss the following aspects for this phase: search strings, search places, eligibility and exhaustion criteria. Notably, we identify a few relevant studies and do pilot research before running searches on all relevant sources.

Step 1: Search Place – We choose four places, i.e. *Google Scholar*, *IEEE Xplore*, *ACM digital library*, and *DBLP* to search potential primary studies.

Step 2: Search String – Our search string aims to capture all the information that can answer the three RQs. We develop the initial set of our search string in a brainstorming session. Then, we connect the various search string with Boolean (i.e. —, +) operators. To support the definition of our study scope, we evolve the search string iteratively via multiple rounds of pilot search to find different synonyms for specific topics. Finally, we augment our search with both forward and backward snowballing [24] on the papers found by keyword retrieval. After evaluating different options, we have defined the following search string:

- Neur* [2] | Program Synthesis
- Neurosymbolic Programming

B. Data Selection

Step 1: Quality control – We conduct multiple rounds of quality assessment for high-quality literature until we reach a

[2] "*" symbol represents derived words from the previous prefix.

closure. We refer to the form to keep only relevant resources for our study. Referring to Table I, we select literature for further analysis only when it satisfies all inclusion criteria and does not satisfy any of the exclusion criteria.

TABLE I: Eligibility Criteria for Literature

Type	Criterion
Inclusion	The literature discusses the general idea or one of the related aspects (capacity, workflow, component) of neurosymbolic programming.
	The literature proposes an approach, study, or tool/framework that targets operationalizing one of the ML components.
	The literature proposes a dataset or benchmark especially designed for neurosymbolic programming.
	The literature is published in English.
	Full text of the literature is accessible.
Exclusion	The literature contains duplicate content of a previously examined literature.

Notably, we leave out of the scope this paper models as suggested by [25], which cover Neural Turing Machines [26], Neural Stack Machines [27], Tensor Product Representation [28], Neural Programmer Interpreters [29], MAX-SAT [30], etc.

Step 2: Closure criteria – We continue adding the related literature that satisfies the criteria mentioned above until we reach a closure. We follow the suggestions in [23] to customize our closure criteria as below: 1

1) Theoretical saturation, i.e., when no new relevant results/concepts emerge from the search results anymore
2) Effort bounded, i.e., only consider the top N search hits; here, we use 100.
3) Evidence exhaustion, i.e., stop the research as early as the quality declines obviously.

Notably, we limit our search to the first 100 search hits and continue the search further if the hits on the last page still reveal additional relevant search results.

C. Data Analysis

We read through each piece of literature in this phase and selected one of the most representative studies for systematic analysis due to time constraints.

We release a replication package freely available online [3] to encourage the verification of our approach, data, and findings. It contains the complete list of sources, the bibliometrics and the categorization.

IV. Results

This section provides the results of our study. Section IV-A talks about the bibliometrics of our collection. Section IV-B gives a general overview of neurosymbolic programming. Section IV-C analyzes a representative technique, i.e. RobustFill [31], related to neurosymbolic programming.

A. Bibliometrics

As shown in Table II, we get 430,184 search results after applying search strings. After screening, we keep only 17 papers as these are the most relevant papers on neurosymbolic programming.

[3]bit.ly/3vysC8u

TABLE II: Literature Retrieval Statistics

Channel	Results	Filtered
Google Scholar	13,600	15
IEEE Xplore	1588	8
ACM digital library	414,928	11
DBLP	68	13
Total	430,184	17

Figure 14 shows the number of annual publications with time until Apr. 2022. We can see most of the literature (58.8%) appeared in 2018 and 2021, testifying to the emergence of a new domain of interest: *neurosymbolic programming*.

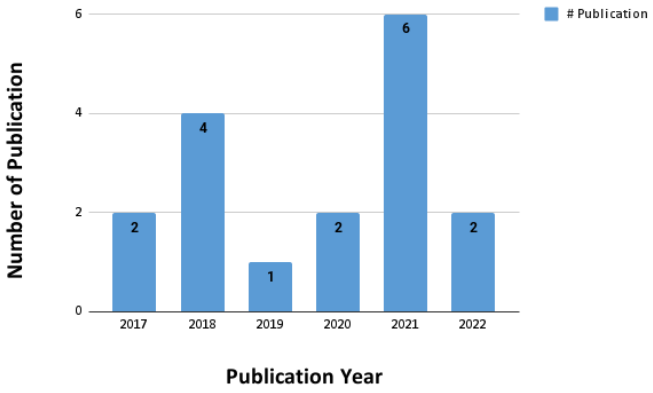

Fig. 14: Publications with time until Apr. 2022

Figure 15 shows the distribution of paper collection for different publication purposes in neurosymbolic programming. Most of the papers are published to introduce a new technique (88.2%), while others are published for an overview (11.8%).

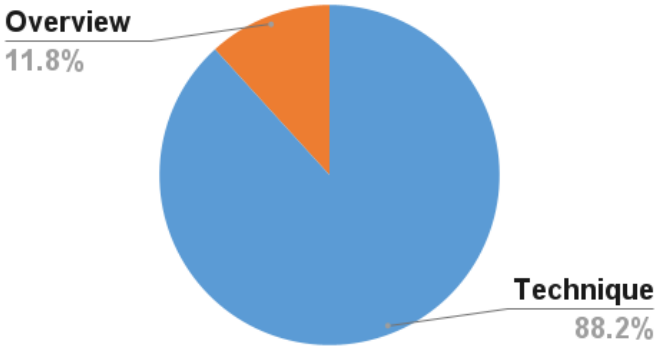

Fig. 15: Publications based on Purposes

B. Overview

We know the concept of neurosymbolic programming from Chaudhuri et al. [25]. Overall, neurosymbolic programming is a generalization of classic program synthesis. A neurosymbolic learning algorithm integrates deep learning approaches into symbolic frameworks or vice versa. Thus, the algorithm aims to synthesize a program that obeys hard constraints while optimizing the loss function. A neurosymbolic program can

be seen as a program that "uses neural components and either symbolic components or compositions" [25].

C. Case Study

Devlin et al. [31] propose a novel synthesis model named "RobustFill". RobustFill applies an RNN with the attention mechanism to encode variable-length sets of I/O examples. This model has achieved 92% accuracy on the dataset from FlashFill [32]. Particularly, RobustFill can handle noise in data (e.g. typos) without deteriorating much in performance. We elaborate on this paper from the perspective of the problem statement, dataset, model, and performance in the following.

Problem Statement: Given a set of input/output (I/O) string examples and a set of unpaired input strings, our goal is to generate the corresponding output strings. We assume at least one program exists for each pair set that will transform all of these examples correctly. In the program synthesis approach, we train a neurosymbolic model that inputs observed examples and synthesizes a program as output. In the program induction approach, we train a neurosymbolic model that takes input and generates output the same as observed examples.

Figure 16 shows an instance of the problem. Throughout this work, we refer to (I_j, O_j) as observed examples and (I_j^y, O_j^y) as assessment examples.

$I_1 =$ `January`	$O_1 =$ `jan`
$I_2 =$ `February`	$O_2 =$ `feb`
$I_3 =$ `March`	$O_3 =$ `mar`
$I_1^y =$ `April`	$O_1^y =$ `apr`
$I_2^y =$ `May`	$O_2^y =$ `may`
$P =$ `ToCase(Lower, SubStr(1,3))`	

Fig. 16: An Instance of Synthesis Problem

We extend the DSL in Parisotto et al. [33] with arbitrary constant strings, substring extraction, and nested expressions. Figure 17 shows the syntax of the DSL while Figure 18 presents the formal semantics. For example, we can define a substring expression using $SubStr(k_1, k_2)$ in which k_1 is the starting index while k_2 is the ending index. The negative indices denote positions counting from the right. We can also use $GetSpan(r_1, i_i, y_1, r_2, i_2, y_2)$ to return the substring between the i_1^{th} occurrence of regex r_1 and the i_2^{th} occurrence of regex r_2 in which y_1 and y_2 denote the starting and ending indices of the corresponding regex matches.

Dataset: We expand the FlashFill [32] dataset to train our model as follows:

1) We randomly sample programs from the DSL, up to 10 expressions.
2) We compute a set of heuristic requirements to ensure that sample programs are executable without throwing exceptions.
3) We generate input strings constrained by the requirements from a random sequence of ASCII characters.
4) We generate the output strings by executing sample programs on the input strings.

$$
\begin{aligned}
\text{Program } p &:= \texttt{Concat}(\texttt{e}_1, \texttt{e}_2, \texttt{e}_3, ...) \\
\text{Expression } e &:= f \mid n \mid n_1(n_2) \mid n(f) \mid \texttt{ConstStr(c)} \\
\text{Substring } f &:= \texttt{SubStr}(\texttt{k}_1, \texttt{k}_2) \\
&\mid \texttt{GetSpan}(\texttt{r}_1, \texttt{i}_1, \texttt{y}_1, \texttt{r}_2, \texttt{i}_2, \texttt{y}_2) \\
\text{Nesting } \texttt{n} &:= \texttt{GetToken}(\texttt{t}, \texttt{i}) \mid \texttt{ToCase}(\texttt{s}) \\
&\mid \texttt{Replace}(\delta_1, \delta_2) \mid \texttt{Trim}() \\
&\mid \texttt{GetUpto}(\texttt{r}) \mid \texttt{GetFrom}(\texttt{r}) \\
&\mid \texttt{GetFirst}(\texttt{t}, \texttt{i}) \mid \texttt{GetAll}(\texttt{t}) \\
\text{Regex } r &:= t_1 \mid \cdots \mid t_n \mid \delta_1 \mid \cdots \mid \delta_m \\
\text{Type } t &:= \texttt{Number} \mid \texttt{Word} \mid \texttt{Alphanum} \\
&\mid \texttt{AllCaps} \mid \texttt{PropCase} \mid \texttt{Lower} \\
&\mid \texttt{Digit} \mid \texttt{Char} \\
\text{Case } s &:= \texttt{Proper} \mid \texttt{AllCaps} \mid \texttt{Lower} \\
\text{Position } k &:= -100, -99, ..., 1, 2, ..., 100 \\
\text{Index } i &:= -5, -4, -3, -2, 1, 2, 3, 4, 5 \\
\text{Character } c &:= \texttt{A}-\texttt{Z}, \texttt{a}-\texttt{z}, \texttt{0}-\texttt{9}, \texttt{!?}, \texttt{@...} \\
\text{Delimiter } \delta &:= \texttt{\&, .?!@()[]\%\{\}/ :; \$\#"'} \\
\text{Boundary } y &:= \texttt{Start} \mid \texttt{End}
\end{aligned}
$$

Fig. 17: Syntax of the DSL for string transformations

$$
\begin{aligned}
[\![\texttt{Concat}(e_1, e_2, e_3, ...)]\!]_v &= \texttt{Concat}([\![e_1]\!]_v, [\![e_2]\!]_v, [\![e_3]\!]_v, ...) \\
[\![n_1(n_2)]\!]_v &= [\![n_1]\!]_{v_1}, \text{ where } v_1 = [\![n_2]\!]_v \\
[\![n(f)]\!]_v &= [\![n]\!]_{v_1}, \text{ where } v_1 = [\![f]\!]_v \\
[\![\texttt{ConstStr}(c)]\!]_v &= c \\
[\![\texttt{SubStr}(k_1, k_2)]\!]_v &= v[p_1..p_2], \text{ where} \\
&\quad p_1 = k_1 > 0 \; ? \; k_1 \; : \; \texttt{len(v)} + k_1 \\
&\quad p_2 = k_2 > 0 \; ? \; k_2 \; : \; \texttt{len(v)} + k_2 \\
[\![\texttt{GetSpan}(r_1, i_1, y_1, r_2, i_2, y_2)]\!]_v &= v[p_1..p_2] \text{ ,where} \\
&\quad p_1 = y_1 (\text{Start or End}) \text{ of } |i_1|^{\text{th}} \text{ match of } r_1 \text{ in } v \text{ from beginning (end if } i_i < 0) \\
&\quad p_2 = y_2 (\text{Start or End}) \text{ of } |i_2|^{\text{th}} \text{ match of } r_2 \text{ in } v \text{ from beginning (end if } i_2 < 0) \\
[\![\texttt{GetToken(t, i)}]\!]_v &= |i|^{\text{th}} \text{ match of } t \text{ in } v \text{ from beginning (end if } i < 0) \\
[\![\texttt{GetUpto(r)}]\!]_v &= v[0..i], \text{ where } i \text{ is the index of end of first match of } r \text{ in } v \text{ from beginning} \\
[\![\texttt{GetFrom(r)}]\!]_v &= v[j..len(v)], \text{ where j is the end of last match of } r \text{ in } v \text{ from end} \\
[\![\texttt{GetFirst(t, i)}]\!]_v &= \texttt{Concat}(s_1, \cdots, s_i), \text{ where } s_j \text{ denotes the } j^{\text{th}} \text{ match of } t \text{ in } v \\
[\![\texttt{GetAll(t)}]\!]_v &= \texttt{Concat}(s_1, \cdots, s_m), \text{ where } s_i \text{ denotes the } i^{\text{th}} \text{ match of } t \text{ in } v \text{ and } m \text{ denotes the total matches} \\
[\![\texttt{ToCase(s)}]\!]_v &= \texttt{ToCase}(s, v) \\
[\![\texttt{Trim()}]\!]_v &= \texttt{Trim}(v) \\
[\![\texttt{Replace}(\delta_1, \delta_2)]\!]_v &= \texttt{Replace}(v, \delta_1, \delta_2)
\end{aligned}
$$

Fig. 18: Semantics of the DSL for string transformations

We use FlashFillTest [32] to evaluate the trained model. It consists of 205 real-world examples collected from Microsoft Excel spreadsheets. Each test instance includes four observed and six assessment I/O examples. All assessment examples must be exactly correct for the model to get credit.

Model: We model program synthesis as a seq2seq generation task. As shown in Figure 19 [4], seq2seq is a sequential encoder-decoder model aiming to learn the latent representation of programs. Our goal is to maximize the log-likelihood of reference program sequence. In Figure 20, we explore and evaluate four increasing complex variants of attentional RNN [34] in the following.

- **Basic Seq-to-Seq** (baseline): The LSTM encoder ingests the observed I/O examples while the LSTM decoder spits

[4] medium.com/@t.schnake/a-formalization-of-a-simple-sequential-encoder-decoder-b31be

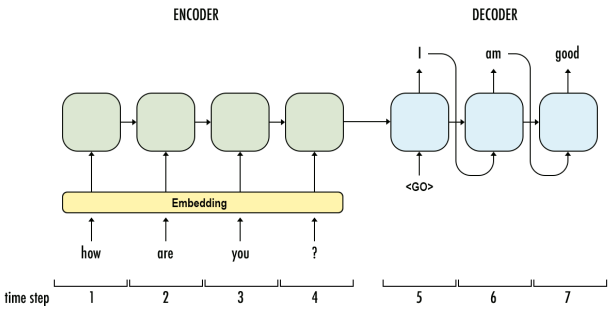

Fig. 19: Sequential Encoder-Decoder Model

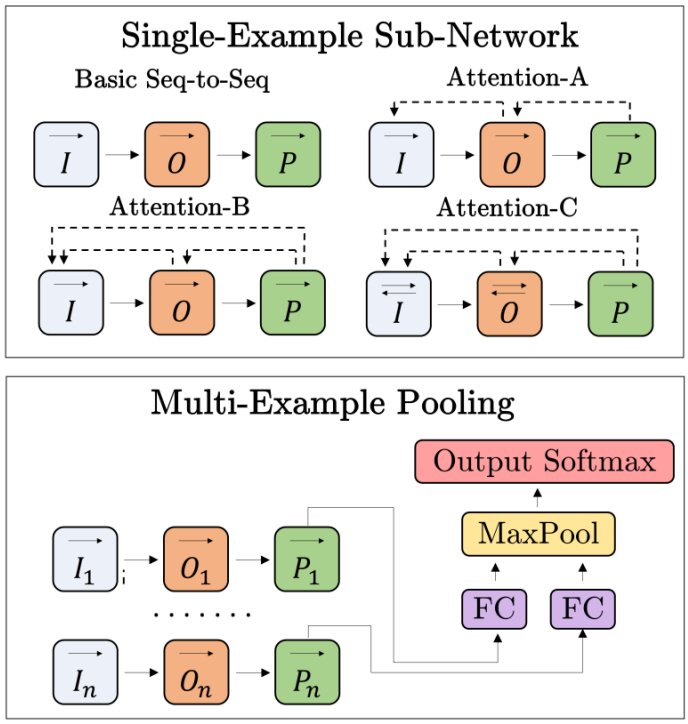

Fig. 20: Synthesis Architecture in which a dotted line indicates an attentional direction.

the program token by token. The final hidden state is fed as the initial hidden state of the next LSTM.

- **Attention-A**: O and P are attentional LSTMs, with O attending to I and P attending to O.
- **Attention-B**: I, O, and P are attentional LSTMs, with O attending to I and P attending to both I and O.
- **Attention-C**: Same as Attention-C, and I and O and bidirectional LSTMs (Figure 21 [5]) in which the sequential data feeds forward(past to future) and backwards (future to past) in both directions.

As shown in Figure 22, we also create an induction model, which resembles the architecture of Attention-A. Notably, an additional LSTM encodes I^y, and the decoder layer O^j uses double attention on O_j and I^y. A model only takes a single observed example as input and synthesizes a program as output in all cases. All inputs and outputs are fed at the character

[5]paperswithcode.com/method/bilstm

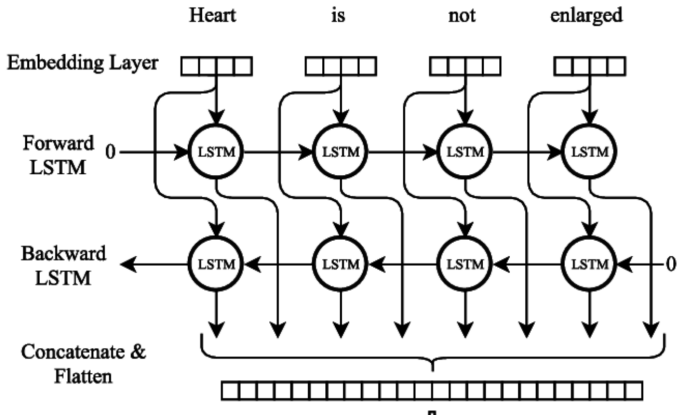

Fig. 21: Bidirectional LSTM [35]

level. The input to I and O are ASCII character embeddings whose vocabulary consists of 95 tokens. The inputs and outputs for P are the flattened source code whose vocabulary consists of 430 tokens, including all function names, parameter values, and special tokens.

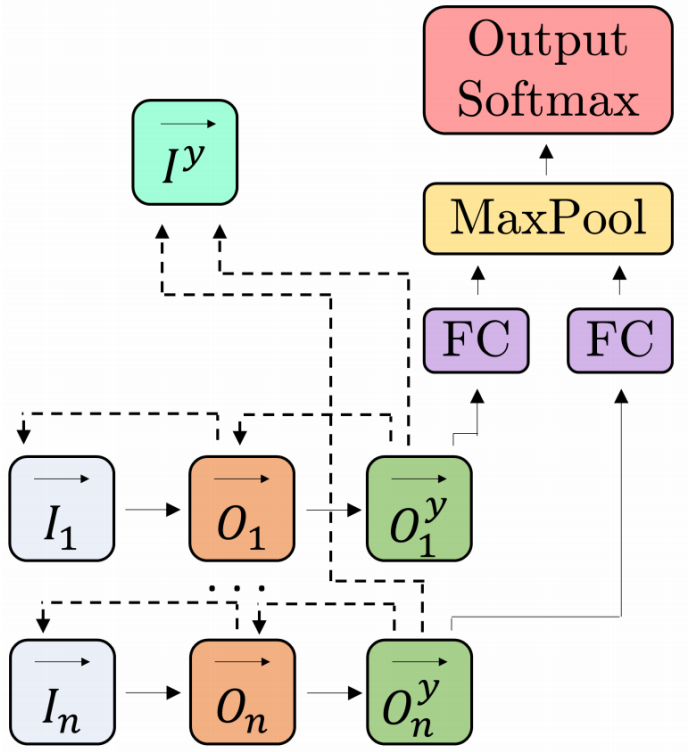

Fig. 22: Induction Network Architecture

We create a technique named “late pooling” to alleviate the internal conditional dependency between variable-length, unordered sets of I/O examples. Each I/O example has its own layer for I, O, and P. For P, the weights are shared across I/O examples. We pool the hidden states of $P_1, ..., P_n$ at each time step before feeding into a single output softmax layer. After pooling, we add another fully connected (FC) layer to enhance the classification.

Training: Table III shows the configuration of our training

model. We resample each minibatch once, so the training model is fed with 256 million random programs and 1024 million random I/O examples.

Attribute	Value
Size of layers	512
Size of embedding	128
Optimization method	Stochastic gradient descent [36], Gradient clipping [37]
Number of minibatch	2 million
Size of minibatch	128
Training period	24h
GPU	Titan X
Number of GPU	2

TABLE III

After the training, we resort to "DP-beam" [38], a variant of beam search, to decode the synthesized programs. DP-beam adds search constraints similar to the dynamic programming algorithm. If any resulting partial output string is not a prefix of the observed output string, we remove the partial program from the beam. DP-beam is effective since our DSL is primarily concatenative. Then, we check all k-best candidates against consistency with all observed examples but only select the program with the highest model score as the final output.

Performance: First, we evaluate the generalization accuracy, i.e. the percent of test instances for which the model has generated the correct output for assessment examples. As shown in Figure 23, all attentional variants outperform the baseline by a very large margin - 20% 30% improvement. The difference between the three attentional variants is smaller, but there is still a slight improvement in accuracy as the model becomes more complex.

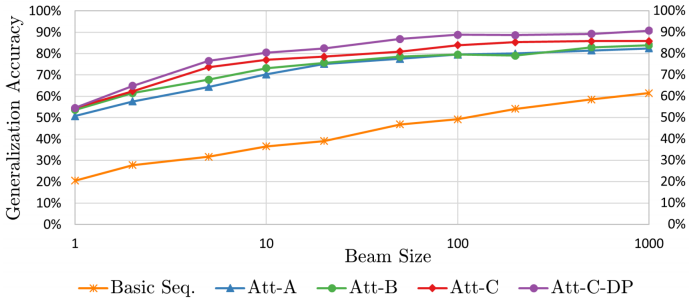

Fig. 23: Generalization Accuracy

As shown in Figure IV, we then compare our generalization accuracy with *Neuro-Symbolic Program Synthesis* [33], i.e. one of the strongest neural synthesis models for solving FlashFillTest. The accuracy improvement of 58% might be attributed to the following.

- Late pooling allows the integration of a multi-attention mechanism.
- The DSL is more expressive than that of [33].

Then, we compare the consistency and generalization accuracy using Attention-C, as shown in Figure 24. Notably, $Beam = 1$ decoding only synthesizes consistent output approximately 50%, implying that the latent function semantics learned by the model are not nearly perfect.

System	Beam	
	100	1000
Neuro-Symbolic Program Synthesis	23%	34%
Basic Seq-to-Seq	51%	56%
Attention-C	83%	86%
Attention-C-DP	89%	92%

TABLE IV

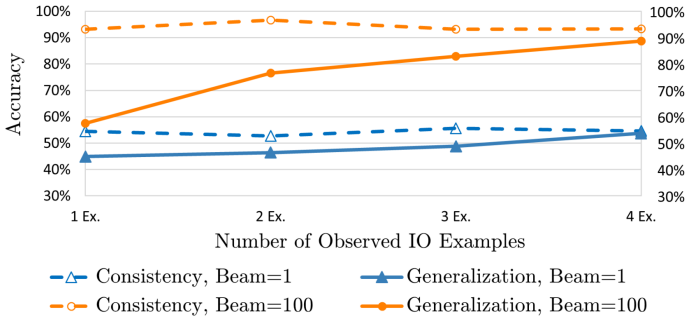

Fig. 24: Consistency vs. Generalization Accuracy Using Attention-C

Then we compare the synthesis models with the induction model in Figure 25. Notably, the induction model performs as good as the synthesis model with $Beam = 1$.

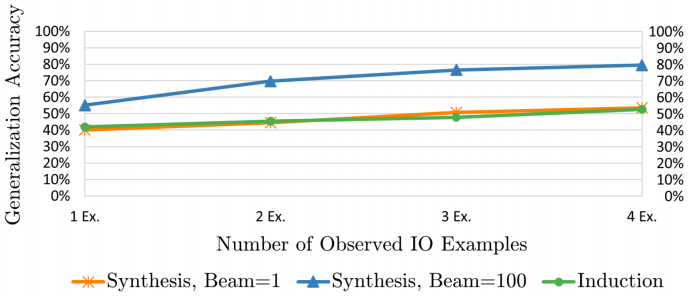

Fig. 25: Induction vs. Synthesis Using Attention-A and Standard Beam Search

As shown in Figure 26, we then evaluate the average-example accuracy, i.e. the percent of correct assessment examples averaged over all instances. As we see, synthesis models tend to be "all or nothing", while the induction model has a much higher chance of getting partially correct results. Notably, the induction model is better than synthesis models with $Beam = 1$ under the metric of average-example accuracy. However, synthesis models with $Beam = 100$ still beat the induction model by 10%.

Finally, we want to know the generalization accuracy of different models in the face of noise (e.g. typos). To achieve this, we synthetically inject the noise via two means:

- We apply character substitutions, deletions, or insertions with uniform random probability into the input or output strings.
- We choose the best program with character edit rate to the observed examples instead of an exact match. The standard beam is also applied rather than DP-Beam.

As shown in Figure 27, we compare our models with FlashFill [32], one of the best hand-engineered algorithms. Our

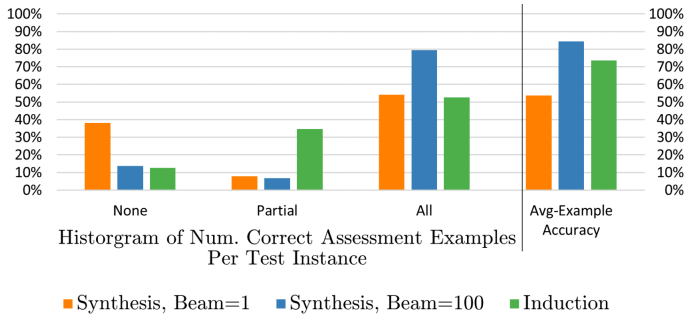

Fig. 26: Average-example Accuracy

models remain robust to moderate level of noise in the I/O examples, compared with FlashFill, which is "broken" for the slightest amount of noise.

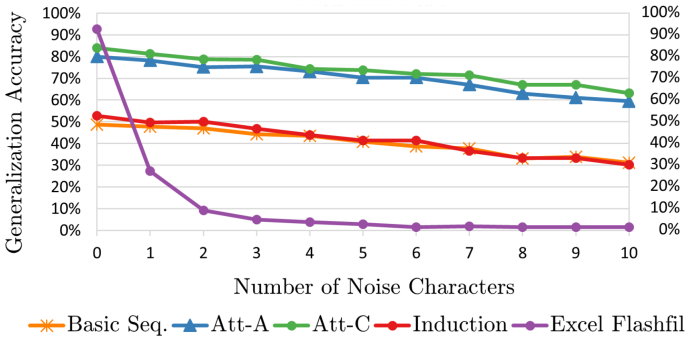

Fig. 27: Noise Effect by Model Type, all synthesis model use $Beam = 100$

V. Discussions

This section discusses the recurrent challenges in Section V-A and research opportunities in Section V-B on neurosymbolic programming.

A. Challenges

Based on our readings, we find a series of open issues related to neurosymbolic programming:

- Perhaps the most immediate challenge is scalability. Despite the sustained effort from the community, desired program architectures that can be automatically discovered through neurosymbolic programming tend to be small.
- Symbol grounding problem [39] is prevalent since neural and symbolic components can be entangled in nontrivial manners. It is tricky to align symbolic specifications with representations learned using neural networks.
- Ambiguity is a significant problem in the natural language specification. How do the users convey their goals both efficiently and unambiguously? If multiple programs satisfy the requirements, how can we tell which one the user actually wants? The training model can miss-interpret requirements and can synthesize absurd and erroneous programs.
- Contemporary data-driven synthesis techniques are not ready for the industry due to a lack of qualified data and efficient program representation.
- Since most software development involves working in the context of existing software systems, fixing bugs, optimizing code, and performing other kinds of maintenance tasks, how can we broaden the application of synthesis ideas to software development tasks beyond green-field software creation?

B. Opportunities

Based on our readings, we find a series of promising research directions following the challenges mentioned above:

- We need fundamental algorithmic innovation to refine representation learning for code structure.
- We need to design standards for symbolic requirements. A lenient DSL would complicate the search and encourage overfitting, while a confining DSL would prevent the disclosure of interesting models.
- we need to develop new NLP techniques to deal with natural language specifications.
- We need to build new codebases that incorporate necessary information, e.g. communication among developers, in an SDLC.

VI. Threats to Validity

Construct validity: We carefully refined the study methodology, including the study scope, goal, and protocol. We look into the literature to find out more about neurosymbolic programming, but the problem is that we have scattered sources of information in academia. Thus, we only focus on the most relevant papers and include them in our review. That may bias readers' perspectives on what neurosymbolic programming is. We mitigate this construct threat by reading even more papers until we reach a consensus to move on to the next step.

Internal validity: One threat to the internal validity might be that the collection and screening are subjective to individual opinions since there is only one author in this paper. We mitigate the internal threat by carefully double-checking until the results are good enough to be accepted.

Conclusion validity: We still have limited knowledge of neurosymbolic programming since such a field has substantially evolved since its born. Thus, our search strategy might exclude related works on the topic. Nevertheless, we alleviate the conclusion threat by comparing our search results with previous studies.

External validity: There are a few works in neurosymbolic programming, but we only select one for analysis. We mitigate external threats by picking the most representative research after reading extensively on the topic.

VII. Conclusion

This paper provides an overview of the challenges, progress, and prospects of neurosymbolic programming. We also want to give a case study to help the readers to understand the topic. In the future, we will add more depth and detail to expand our analysis to a systematic literature review.

REFERENCES

[1] N. Kant, "Recent advances in neural program synthesis," *ArXiv*, vol. abs/1802.02353, 2018.

[2] E. Kitzelmann, "Inductive programming: A survey of program synthesis techniques," in *AAIP*, 2009.

[3] Y. Li, D. H. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, Tom, Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de, M. d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, Alexey, Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de, Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," 2022.

[4] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," *ArXiv*, vol. abs/1611.01989, 2017.

[5] L. Mou, H. Zhou, and L. Li, "Discreteness in neural natural language processing," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019 - Tutorial Abstracts*, T. Baldwin and M. Carpuat, Eds. Association for Computational Linguistics, 2019. [Online]. Available: https://aclweb.org/anthology/papers/D/D19/D19-2004/

[6] A. S. d'Avila Garcez and L. Lamb, "Neurosymbolic ai: The 3rd wave," *ArXiv*, vol. abs/2012.05876, 2020.

[7] M. Garnelo and M. Shanahan, "Reconciling deep learning with symbolic artificial intelligence: representing objects and relations," *Current Opinion in Behavioral Sciences*, vol. 29, pp. 17–23, 2019.

[8] A. L. Samuel, "Machine learning," *The Technology Review*, vol. 62, no. 1, pp. 42–45, 1959.

[9] Y. Bengio, A. C. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, pp. 1798–1828, 2013.

[10] 2022. [Online]. Available: https://cs231n.github.io/neural-networks-1/

[11] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks : the official journal of the International Neural Network Society*, vol. 61, pp. 85–117, 2015.

[12] S. Kostadinov, "Understanding gru networks," Nov 2019. [Online]. Available: https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be

[13] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[14] A. Church, "Logic, arithmetic, and automata," *Journal of Symbolic Logic*, vol. 29, no. 4, 1964.

[15] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006, pp. 404–415.

[16] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," *ACM Sigplan Notices*, vol. 46, no. 1, pp. 317–330, 2011.

[17] E. Torlak and R. Bodik, "Growing solver-aided languages with rosette," in *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, 2013, pp. 135–152.

[18] J. Bornholt, "Program synthesis explained," 2022. [Online]. Available: https://www.cs.utexas.edu/~bornholt/post/synthesis-explained.html

[19] K. Scott, "Microsoft build 2020: Cto kevin scott on the ... - youtube," 2020. [Online]. Available: https://www.youtube.com/watch?v=eNhYTLWQFeg

[20] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[21] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago *et al.*, "Competition-level code generation with alphacode," *arXiv preprint arXiv:2203.07814*, 2022.

[22] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," *arXiv preprint arXiv:1611.01989*, 2016.

[23] V. Garousi, M. Felderer, and M. Mäntylä, "Guidelines for including the grey literature and conducting multivocal literature reviews in software engineering," *Information and Software Technology (IST)*, vol. 106, pp. 101–121, 2019.

[24] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2014, pp. 38:1–38:10.

[25] S. Chaudhuri, K. Ellis, O. Polozov, R. Singh, A. Solar-Lezama, Y. Yue *et al.*, *Neurosymbolic Programming*. Now Publishers, 2021.

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

[27] E. Grefenstette, K. M. Hermann, M. Suleyman, and P. Blunsom, "Learning to transduce with unbounded memory," *Advances in neural information processing systems*, vol. 28, 2015.

[28] P. Smolensky, M. Lee, X. He, W.-t. Yih, J. Gao, and L. Deng, "Basic reasoning with tensor product representations," *arXiv preprint arXiv:1601.02745*, 2016.

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

[30] P.-W. Wang, P. Donti, B. Wilder, and Z. Kolter, "Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver," in *International Conference on Machine Learning*. PMLR, 2019, pp. 6545–6554.

[31] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli, "Robustfill: Neural program learning under noisy i/o," in *International conference on machine learning*. PMLR, 2017, pp. 990–998.

[32] S. Gulwani, W. R. Harris, and R. Singh, "Spreadsheet data manipulation using examples," *Communications of the ACM*, vol. 55, no. 8, pp. 97–105, 2012.

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

[34] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[35] A. Graves, N. Jaitly, and A.-r. Mohamed, "Hybrid speech recognition with deep bidirectional lstm," in *2013 IEEE workshop on automatic speech recognition and understanding*. IEEE, 2013, pp. 273–278.

[36] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[37] J. Zhang, T. He, S. Sra, and A. Jadbabaie, "Why gradient clipping accelerates training: A theoretical justification for adaptivity," *arXiv: Optimization and Control*, 2020.

[38] H. Ney, D. Mergel, A. Noll, and A. Paeseler, "A data-driven organization of the dynamic programming beam search for continuous speech recognition," in *ICASSP'87. IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 12. IEEE, 1987, pp. 833–836.

[39] S. Harnad, "The symbol grounding problem," *Physica D: Nonlinear Phenomena*, vol. 42, no. 1-3, pp. 335–346, 1990.