Neuro-Symbolic Program Synthesis: Leveraging Syntactic Datasets and Active Learning

Oleg Dats

Ukrainian Catholic University, Lviv, Ukraine

Abstract. This research proposal focuses on Neuro-Symbolic Program Synthesis, specifically targeting the automation of software development using "programming by examples" [1].
The study aims to explore the utilization of syntactic datasets, investigate suitable Domain Specific Languages (DSL), incorporate active learning techniques, and leverage the Transformer architecture [2] for program synthesis tasks. The research aims to establish Scaling Laws for Neuro-Symbolic Language Models to understand the relationship between model complexity, training data, and solution space.
Neuro-symbolic computation, which aims to connect deep learning with logic, forms the foundation of our approach [3].

1 Introduction

1.1 Problem statement

The ambition to automate software development is deeply rooted in history. Alonzo Church was the first to identify the challenge of creating a circuit based on mathematical specifications [4]. In my studies, my focus will be on "programming by examples". This is a method where a program is produced using a collection of input-output examples given by the user.

1.2 Importance of a problem

1. The practical applications of program synthesis are vast and impactful, particularly in the industry. One such area is data wrangling, where the transformation of data from one format to another is crucial. This accessibility is essential for non-programmers who require data analysis and cleaning capabilities [5].
   Moreover, program synthesis offers various other possibilities, such as generating efficient code from examples, providing educational assistance by suggesting possible solutions, enabling system identification, and more.
2. In the quest for more intelligent and human-like artificial systems, program synthesis plays a crucial role. François Chollet highlighted current limitations in achieving this goal in "On the Measure of Intelligence" [6] . To address these shortcomings, he created the Abstraction and Reasoning Corpus

(ARC), which assesses a human-like manifestation of general fluid intelligence. It allows fair comparisons of general intelligence between AI systems and humans, aligning seamlessly with our methodologies. The ARC dataset contains image pairs with solutions represented as transformation programs. Another perspective, presented by Yann LeCun in [7], emphasizes the importance of World models - symbolic representations of intuitive physics and the world around us. Incorporating such representations is vital to achieving general artificial intelligence.

3. To further advance in this field, we aim to derive empirical laws that govern the relationship between the computation model class, size of solution space (maximal program size), amount of training data, size of the model, and value of the loss function. This line of research will draw inspiration from "Scaling Laws for Neural Language Models" [8] with an aim to transfer this approach from general purpose neural models to neuro-symbolic models. It will be termed Scaling Laws for Neuro-Symbolic Language Models.

Research has demonstrated that grammar induction is an NP-complete problem [9]. One interesting perspective is considering neural networks (NN) as a form of heuristic storage, enabling them to efficiently solve NP-complete problems in polynomial or even linear time with high probability. This idea is akin to how people use SAT solvers to tackle NP-complete problems effectively. Notably, SAT was the first problem proven to be NP-complete [Cook–Levin theorem], and subsequently, SAT solvers were developed to achieve significantly faster solutions for this challenging problem [10].

2 Current research progress and major challenges in program synthesis

2.1 Classical symbolic AI

There are four commonly used state-of-the-art (SotA) techniques in program synthesis [11]:

1. Enumerative search: This technique structures the hypothesis space and utilizes pruning to efficiently search for the desired program.
2. Constraint solving: Modern SMT (Satisfiability Modulo Theories) solvers are employed to handle program synthesis by solving the associated constraints.
3. Stochastic search: This approach learns a distribution over the space of programs within the hypothesis space, conditioned on the given specification. Sampled using algorithms like Metropolis-Hastings or genetic programming.
4. Deduction-based: This method involves a top-down search and applies a divide-and-conquer strategy to break down the synthesis problem into smaller, manageable subproblems.

Two main challenges faced in program synthesis are scaling for large program spaces and dealing with noise in the data. To reduce the search space, researchers have explored designing Domain Specific Languages (DSLs), but this approach is often time-consuming and requires domain experts [12].

2.2 Machine learning

In contrast, machine learning, particularly the Transformer architecture introduced in [2], has shown promise in handling noise and scaling efficiently for large NLP datasets. Researchers have started applying this approach to program synthesis tasks.

One successful approach is fine-tuning LLM models, such as Codex [13], where GPT3 is fine-tuned on a large codebase from public repositories on GitHub. Another approach, demonstrated in [14], involves carefully selecting a training set to achieve good results with smaller model sizes and less data.

However, this line of research is heavily reliant on hand-crafted datasets, leading to significant effort spent on searching for the right data and then cleaning and transformations. Furthermore, models trained with this approach often struggle with tasks from out-of-training distribution.

To address the limitation of depending on hand-crafted datasets and allow models to discover new algorithms, DeepMind formulated the program synthesis problem as a single-player game [16]. They applied an approach from AlphaZero [15] to teach models coding. Their novel model, AlphaDev [16], discovered new sorting and hashing algorithms. However, this method requires retraining the model from scratch for each new problem, making it computationally expensive.

Another line of research explores the use of synthetic data to train models. For example, in [17], neural networks are shown to learn syntactic constituency parsing with additional training data synthesized by annotating text using BerkeleyParser. In [18], NNs are trained to learn symbolic integration and solve differential equations by generating synthetic training data through grammar-based function sampling and solving using SymPy.

The next logical step in this line of research is to continue exploring the use of synthetic data and ask the main question: can this approach work effectively for program synthesis? If so, how large can the program search space be effectively handled?

3 Motivation

3.1 Open questions

Based on the "Current State of Research" section I am able to identify the following research questions that remain unanswered:

1. Can we augment NN training with synthetic data for programs and establish Scaling Laws for Neuro-Symbolic Language Models to demonstrate the benefits and limitations of this approach? If so, this could address the problem of insufficient data, given that sample sizes are typically not large [19]. The generation of additional samples during model learning could be the key.
2. How to leverage Active learning and uncertainty about model parameres to improve sampling of training data [20]?

3. The process of LLM fine-tuning relies on a Python code base. Is it challenging to sample lengthy and valid Python programs? Could we explore alternative programming languages that offer a simpler yet more accessible sampling syntax?
4. What is the correlation between a computational model's complexity and the performance achievable within it? Based on Chomsky's hierarchy: Context-free grammar holds more expressive power than Regular. For example: some Regular languages are not definable (programs are not realizable):

$$L = \{a^n b^n \mid n \geq 1\}$$

   Recursively enumerable (Turing complete) languages hold even more expressiveness. However, could such expressivity significantly impact performance metrics?
5. What part does DSL play? The practical utility of high-level languages with libraries (Python + Numpy) over low-level ones like Assembler is evident. Also, François Chollet emphasized the significance of DSL for ARC in [6]. Can empirical evidence confirm or refute this assertion?
6. Can DSL emerge during the training phase?
7. Can we have a dynamic training set distribution? As demonstrated in [18], poor generalization occurs for tasks beyond the training distribution. Can we alter the static distribution to a more adaptable one that adjusts according to the validation error? If the validation error decreases slowly, the training distribution is flawed, and the sampling algorithms should modify the shape.
8. Can we tackle ARC by training a model that can identify a program capable of transforming pairs of 2D images?

3.2 My motivation and competences to complete research

AI offers fascinating insights into how our own minds might operate. Edsger W. Dijkstra once famously remarked, "Computer science is no more about computers than astronomy is about telescopes." [21]

1. Love for coding since young, CS degree, started as a software engineer at 19, now the founder of an IT software development company for 9 years, 300 employees, eager to apply research in practice.
2. Proficient in modern programming languages, especially fond of functional programming and theoretical models like LC and SKI combinators.
3. Conducted personal research on NNs, focusing on the generalization problem. [22]
4. Explored ARC problem with Genetic Algorithms, looking to enhance results with deep learning.
5. Fascination with the connections between knowledge, philosophy (World of Ideas by Socrates) and Kolmogorov complexity theory, physics (Computational Universe), and biology (Model-Based Reinforcement Learning).
6. Aspirations: To direct scientific career toward neuro-symbolic program synthesis with applications across various fields.

4 Methodology

4.1 Research Design

The approach follows the spirit of [18], where the authors used a synthetic dataset of math formulas to train NNs to solve integration and differential equations.

The program synthesis problem can be viewed in a similar way as solving computational equations, analogous to solving differential equations to find an unknown function.

For instance, given a set of input-output pairs S = {(a1, b1), (a2, b2)}, the objective is to discover a program X such that (X(a1), X(a2)) = (b1, b2). To formalize this process, we can employ Higher-order rewriting systems, seeking a substitution for X that causes the Left term to reduce to the Right term and terminate [23].

I suggest the following extensions to already established methodology:

1. Extension to use programs: The methodology is extended to incorporate programs as part of the learning process.
2. Utilization of different models of computation: Various models of computation will be employed in the investigation.
3. New procedure for sampling training set: Instead of fixed sampling strategies, a flexible distribution will be used to match the hidden distribution of the test set. This is necessary because the true distribution of the programs is unknown, and a uniform distribution may not be suitable due to its broad nature.

For each model of computation, the following artefacts will be developed:

1. Grammar: A set of rules governing the structure and composition of a formal language will be defined.
2. Compiler: The creation of a compiler that translates strings into runnable programs will be accomplished.
3. Evaluation Procedure (Computation): A procedure that takes a program and input and produces the corresponding output will be established.

This setup enables the sampling, parsing, and execution of strings as programs.

Based on the "Church-Turing thesis" [24] the research will take into consideration two principal computational models. Each model allows for subsets of programs, with each subset being strictly larger than the previous, meaning it contains more programs:

1. Sequential Model. The hierarchy includes: Finite-state machines $\subset$ Pushdown automata $\subset$ Turing machines, proven by Pumping lemmas as discussed in [25]
2. Functional Model. SKI combinators. The hierarchy includes: $BCK \subset SKI : Typed \subset SKI$. Integration of types ensures program termination. This formalism enables testing how language design impacts performance, such as

using a minimal set of combinators $\{S, K, I\}$, or introducing additional combinators $\{B, T, ...\}$, resembling higher-level languages. It is worth noting that any combinator can be represented as a combination of $\{S, K\}$, making the set of programs the same even with the introduction of more combinators. The details of this theory are covered in[26].

4.2 Stage 1: Scaling Laws for Neuro-Symbolic Language Models

For each model of computation and its subsets, the following steps will be executed:

1. Define Grammar for Programs: A grammar will be established to govern the structure of programs.
2. Sample Expression from Tree: A procedure for sampling expressions from a tree contracted based on the defined grammar will be developed.
3. Determine Expression Tree Depths: Fix the depths of the expression trees and count the number of possible expressions. This metric will serve as a measure of task complexity, as it controls the size of the solution space.
4. Sample Test and Training Sets: Test and training sets will be sampled to assess the number of examples required to cover or approximate the sample space.
5. Choose Model Size: The model size will be chosen to understand the relationship between the required model size and the complexity of the solution space.
6. Train Model and Measure Results: The model will be trained, and its performance will be measured based on the selected metrics.

During this investigation, Scaling Laws for Neuro-Symbolic Language Models will be explored by varying models of computation, solution space complexity, training set size, and model size. The research aims to identify potential outcomes:

1. Consistent Performance: The model performs equally well across all models of computation.
2. Preferable Setup: There is a preferred configuration, either utilizing short languages like SKI or employing over 100 specialized combinators.

4.3 Stage 2: Encoding Bayes over synthetic training set

As a subsequent move, my intention is to study the encoding of Bayes over the distribution space of training programs. Previous findings indicate that knowing an approximate solution length aids in the productive constraint of training programs, as illustrated in [14]. There's a need to formulate additional conditions because the space for solutions expands exponentially. There are two possible methods to encode these Bayes:

1. Implement fixed Bayes, utilizing type constraints[27]. For instance, $P : \text{List} \rightarrow \text{List}$ restricts programs that manipulate lists.

2. Deploy adaptable Bayes. Harness Reinforcement learning to dynamically revise probabilities over the sample space. The reward correlates with the proficiency of NN training using this sample space. This is akin to how AlphaZero modifies its value function over the decision tree based on game outcomes. In our context, the agent is better of if the sampling strategy is beneficial, evidenced by the decrease in test loss.

4.4 Stage 3: Leverage Active Learning

Active Learning is a technique wherein an algorithm actively selects the most impactful and relevant data points [28]. This approach is potentially beneficial for our circumstances, where we are generating syntactic data. My goal is to examine the applicability and productivity of this methodology. Nevertheless, the central issue is to evaluate its scalability.

4.5 Stage 4: Test on public datasets

In the concluding phase, we aim to cultivate a syntactic dataset to train NN and achieve SotA results on public datasets:

1. Primarily Basic Programming Problems: 1,000 Python programming problems obtained via crowdsourcing, intended to be solvable by beginner programmers.
2. Branch out to specific types: 2d images. Perform tests on ARC.

4.6 Feasibility and Resources

The proposed research is deemed plausible due to the possibility to regulate the size of the sample space by setting a maximum depth for the tree of potential programs. The training of the NN can be executed with a single GPU, making it reasonably manageable in terms of computational resources.

The problem will be limited to a completely "automated" approach, but it can potentially be expanded by involving humans in the evaluation process for both the generated code and the program's computed output.

5 Summary

The application of program synthesis in production systems and its significance in building general AI makes it an exciting and rewarding area for a career in science. The task of creating a program to meet specific criteria is a longstanding challenge in Computer Science. However, I remain hopeful that deep learning can offer innovative solutions to this old problem. As Hamming would say: "A new way of attacking the problem." [29].

Training NN on evolving syntactic datasets can facilitate autonomous learning of programming without the need for massive pre-filtered datasets. Active

learning strategies can reduce dataset size by enabling the selection of relevant examples, and specialized DSL can further speed up this process. My aspiration is that such an approach will achieve SotA performance on programming tasks and will also be capable of working with 2D images to solve the ARC problem.

Currently, neural networks operate as a "black box" the inner workings of which are often unclear. However, by compelling neural networks to generate programs that can explain data, we can illuminate some of these inner processes. This will enhance both the generalization capability and interpretability of these models in a broader context.

References

1. A Menon. A Machine Learning Framework for Programming by Example.
2. Ashish Vaswani et al. Attention is All you Need.
3. Md Kamruzzaman Saker et al. Neuro-Symbolic Artificial Intelligence.
4. A Church. Applications of recursive arithmetic to the problem of circuit synthesis.
5. S Gulwani. Automating string processing in spreadsheets using input-output examples.
6. F Chollet. On the measure of intelligence.
7. Yann LeCun et al. Deep learning, reinforcement learning, and world models.
8. J Kaplan. Scaling Laws for Neural Language Models.
9. M Charikar et al. The smallest grammar problem.
10. S Alouneh et al. A comprehensive study and analysis on SAT-solvers: advances, usages and achievements.
11. S Gulwani et al. Program synthesis.
12. S Gulwani et al. Inductive programming meets the real world.
13. M Chen et al. Evaluating large language models trained on code.
14. S Gunasekar et al. Textbooks Are All You Need.
15. D Silver et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm.
16. DJ Mankowitz et al. Faster sorting algorithms discovered using deep reinforcement learning.
17. O Vinyals et al. Grammar as a foreign language.
18. G Lample, F Charton. Deep learning for symbolic mathematics.
19. N Gelman. N is never large.
20. Z Ghahramani. Probabilistic machine learning and artificial intelligence.
21. EW Dijkstra. A discipline of programming.
22. O Dats. Agent adaptation to a changing environment, the final project in the "Machine Learning" course at UKU Data Science.
23. F Baader, T Nipkow. Term rewriting and all that.
24. BJ Copeland. The church-turing thesis.
25. M Sipser. Introduction to the Theory of Computation.
26. JR Hindley, JP Seldin. Lambda-calculus and Combinators, an Introduction.
27. JR Hindley. Basic Simple Type Theory.
28. B Settles. Active Learning Literature Survey.
29. RR Hamming. The Art of Doing Science and Engineering.