Program Automation: Learning from Tests Defined in Unit Testing

Rihards Baranovskis, School of Electronics and Computer Science,
University of Southampton, Email: rb7e15@soton.ac.uk

***Abstract*—This paper reviews the latest research in machine learning powered software generation techniques. The statistical program synthesis and induction methodologies are explored, which may be used in generating software from given input/output examples. As well as differences between traditional non-statistical based approaches versus machine learning techniques are described, the current progress in the field discussed and potential use in test driven development is examined, where unit tests definitions can be used as predictor – target pairs in such machine learning problem.**

***Index Terms*—Program Learning, Statistical Program Synthesis, Statistical Program Induction, Unit Testing.**

I. Introduction

PROGRAM learning is one of the initial problems that Artificial Intelligence and Machine Learning has been trying to solve since the 70ties [1]. Originally, program learning implied creating formally assigned specifications of the desired program and then generating it from these definitions. These specifications were highly formal and often more complex than the programs themselves [2] [3]. Currently, traditional non-statistical, proof based and inductive synthesis systems commonly use input/output example pairs to specify program's behavior [4] [5], analogous to those used in assertions based unit tests. With the recent increase in popularity of neural networks, many developments have occurred in neural program synthesis and neural program induction, which are types of the statistical program learning methods and use machine learning rather than traditional deductive or combinatorial, search based methodologies [6] [7] [8] [9].

II. Non-Statistical Program Synthesis

This method generates program code based on user's predefined logical specification or behavior intent using non machine learning methods. Non-statistical synthesis ordinarily is approached in one of the following ways [10].

A. Deductive Synthesis

Deductive synthesis views program synthesis as a theorem proving problem [11]. It uses declarative logic specification to deduct a set of features from transformation rules, axioms, unification and mathematical induction to construct a program model. In the process of deduction, a proof is generated that the found model satisfies the predefined specification. This technique performs well with robust, domain specific logical specifications. Biggest drawback of such technique is that writing logical specification is a highly complex process.

B. Syntax-guided Synthesis

This technique uses parameterization in order to generate a user specific program from a specification written in a Domain Specific Language - DSL [12] [13]. Parameterization is exerted when generic synthesis algorithms are populated with parameters specific to a given DSL. Later, search algorithms are executed to find the fitting program generated by a parameterized generic synthesis algorithm. Using a limited DSL allows to reduce the space of search required, hence the complexity of the computation. Disadvantages of using such technique involve its reliance on Satisfiability Modulo Theory (SMT) based solvers, which quickly grow in complexity once grammar of a DSL grows. As result synthesis of such expressive DSL as CSS is already too complicated for this technique [10] [12].

C. Domain-specific inductive Synthesis

In inductive synthesis specification of a program is defined as a set of I/O examples, from which a more generic, domain specific [14] program code is induced. This technique dramatically simplifies the input required from the user, as result 'programming by example' like specification can be handled by an ordinary end user. The main difficulty of this technique is inflexibility, as it requires a unique synthesis algorithm for each domain specific application and DSL.

III. Statistical Program Synthesis

This method, as most of traditional synthesis systems, generates DLS program code, but instead of explicitly defining deduction, proof or search rules of how such code has to be generated, a statistical model is trained to substitute all [6] [7] or some parts of this logic [15]. Researches in [7] have implemented a novel neuro-symbolic program synthesis technique, which is based on Recursive-Reverse-Recursive Neural Network (R3NN) model. The task of this R3NN model was to synthesize Flashfill DSL, which is described in the methodologies performance section. In this model, I/O samples and the corresponding DSL program code have been pre-generated using program sampler and encoded and passed as input into the recurrent neural network, which is architected to construct the output DSL code. This model learns from the input examples and partial program tree derivations of DSL. Each node in the partial tree holds information about every other node it the global tree.

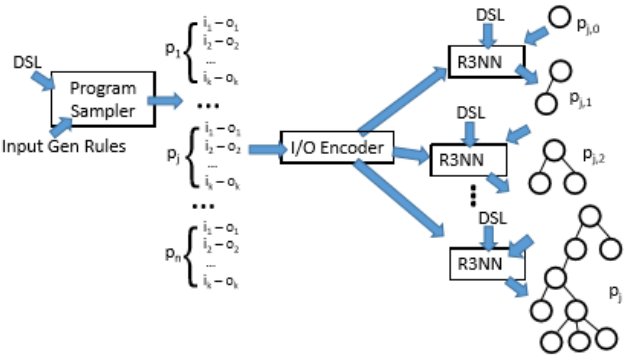

Fig. 1. Usage of R3NN in Program Synthesis. Reprinted from [7].

First, every symbol expansion rule in the DSL is vectorized. Then the model goes recursively, bottom-up through the partial tree nodes and assigns probabilities to the elements this vector. As result, when the root of the tree is reached, it holds information about the global constructed program tree. Later, the reverse-recursive pass happens and this tree construction information is assigned to each node in the tree. Surprisingly, this model performed only with 38% benchmark accuracy, when compared Flashfill performance [7].

IV. Statistical Program Induction

Statistical program induction, is a type of statistical program synthesis, that learns to induce a generic program from a given I/O examples pairs. Quite often the produced program representation is latent [7] [8] [16], which means that no code is being generated. As result, induction methods are often contrasted to the synthesis methods, where code is being produced [9]. Researchers at Google Brain [8] have produced neural network model, which uses augmented helper functions for arithmetic and logic operations. This architecture has been inspired by Neural Turing Machines [16], which is a self-programming Turing Machine. The Neural Programmer runs several steps of operations using recurrent neural networks as a learning controller, which is capable of remembering history of previous steps taken. In each step the neural network can select a portion of input data and perform an arithmetic or logic operation on that segment of data. In this manner it builds a sequence of operations that is performed on input data until it reaches the output, as seen in Figure 2.

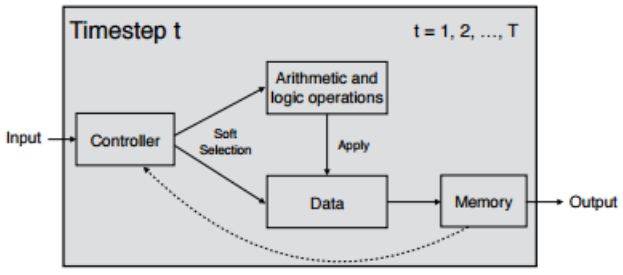

Fig. 2. Neural Programmer Architecture. Taken from [8].

The memory stores the intermediate results of data transformation. Using gradient descent back propagation, neural network can be trained to choose the right sequence of operations in order to achieve desired output. In this manner a generic latent program is induced. In this project, the inputs are strings with different human readable commands about a dataset. These are commands as "select fields from row E that are greater than 150" and expected results are the corresponding data from the dataset. Researchers have trained a model which achieves 100% test accuracy with 50,000 training samples, with added random noise [8].

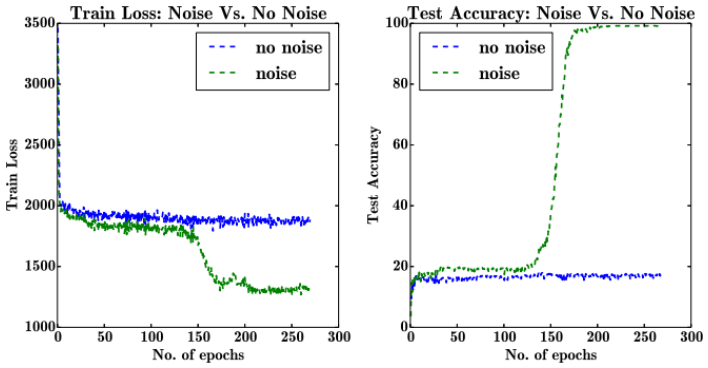

Fig. 3. Neural Programmer Test and Train Performance [8].

Another research at Google Brain [17] has produced a convolution gated recurrent unit based on a parallel GPU architecture, which addressed the problem of limited computational efficiency in Neural Turing Machines due to their sequential architecture. As result this parallel neural network was able to learn to perform non-trivial linear-algorithms such as long binary addition and multiplication with 100% test accuracy.

V. Program Learning Methodologies Performance

Problem Setup

In [9] researchers at MIT and Microsoft have compared DSL-based non statistical program synthesis with Neural Program Synthesis of DSL code and Neural Program Induction of latent programs. The functionality synthesized and induced consisted of reproducing Microsoft's Excels Flashfill functionality, which uses programming by example methodology to automate repetitive spreadsheet data transformation tasks [5]. Flashfill uses non statistical domain-specific inductive synthesis to generate DSL code in order to solve a unique string manipulation problem [10], such as given in Table 1.

TABLE I
FLASH FILL PROGRAM DEFINITION

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

Left column represents sample inputs and right sample outputs. Program P indicates the desirable program written in DSL. Taken from [9]

In case of a Neural Program Synthesis approach represent training input and P represents desired DSL output. As result performance of a synthesized program is assed with test data as . In case of Neural Program Induction, represent training input and desired training output, as result latent program model is tested with test samples.

Neural Program Learning Performance

For evaluating performance, a sequential RNN synthesis model and a search based inductive model were built, and then compared to the original Flashill program. Both of these models architectures differed from the ones described in [7] and [8]. Researchers in [9] have discovered that neural program synthesis achieves performance which is close to the Excel Flashill algorithm and outperform the model trained in [7] by 58%. The neural program induction, performs with far less accuracy with zero input noise. It had been proven to be the case that neural program synthesis and induction outperforms the original non machine learning based algorithm when input is noisy, which means that I/O testing samples had inconsistency errors of what program they expect to be generated [9].

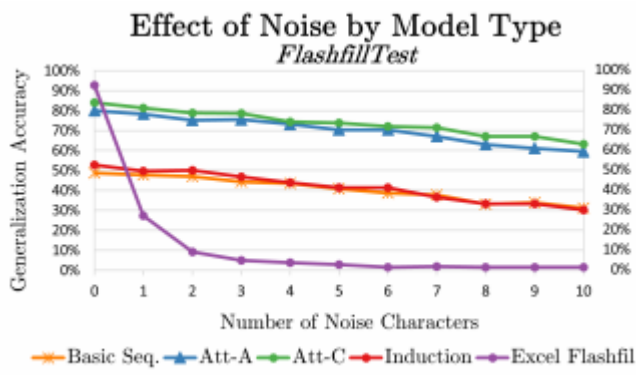

Fig. 4. Comparison of performance under noisy input. Att-* are attention RNNs, producing synthesised models. Recreated from [9].

VI. CONCLUSION

Current research indicates that highly accurate [6] [7] [8] [9], machine learning based models can be created for synthesizing and inducing software from input/output example pairs, close to ones defined in software unit testing. The biggest bottleneck of using neural network based learning systems is enormous demand for I/O pair samples in order to achieve near 100% performance [8]. This leads to a necessity of writing random samples generators, which can come closely to replicating functions that are actually being modeled. This problem can be potentially overcome by applying deep meta-reinforcement learning approach [18], which intends to generate generic meta-models for a specific application domain. If thirstiness of statistical program learning for I/O samples is reduced, current advances can lead to developing self-learning models, which can replace some parts of software development, when test driven development process is applied. Furthermore, since statistical learning generalizes a model better, when it is trained with noisy inputs, machine learning based approach has a potential of producing more robust generic programs than non-statistical methodologies.

REFERENCES

[1] Richard J. and Lee, Richard C. T. Prow Waldinger, "A step toward automatic program writing.," in *IJCAI*, 1969.

[2] X. Qian, "The deductive synthesis of database transactions," *ACM Transactions on Database Systems (TODS)*, vol. 18, no. 4, pp. 626-677, 1993.

[3] Richard Waldinger Zohar Manna, "Fundamentals of Deductive Program Synthesis," *EEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 18, no. 8, pp. 674-704, 1992.

[4] Peter-Michael Osera, David Walker, Steve Zdancewic Jonathan Frankle, "Example-Directed Synthesis: A Type-Theoretic Interpretation," *POPL*, pp. 802-815, January 2016.

[5] Rishabh, and Sumit Gulwani. Singh, "Transforming spreadsheet data types using examples," *ACM SIGPLAN Notices*, vol. 51, no. 1, 2016.

[6] X.V., Wang, C., Pang, D., Vu, K. and Ernst, M.D Lin, "Program Synthesis from Natural Language Using Recurrent Neural Networks," University of Washington Department of Computer Science and Engineering, Seattle, Technical Report UW-CSE-17-03-01, 2017.

[7] E., Mohamed, A.R., Singh, R., Li, L., Zhou, D. and Kohli, P. Parisotto, "Neuro-Symbolic Program Synthesis.," *arXiv preprint arXiv: 1611.01855*, 2016.

[8] Q.V. and Sutskever, I. Le, "Neural Programmer: Inducing Latent Programs with Gradient Descent," *arXiv preprint arXiv: 1511.04834*, 2016.

[9] J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.R. and Kohli, P. Devlin, "RobustFill: Neural Program Learning under Noisy I/O.," *arXiv preprint arXiv: 1703.07469*, 2017.

[10] O. and Gulwani, S. Polozov, "Flashmeta: A framework for inductive program synthesis.," *ACM SIGPLAN Notices*, vol. 10, no. 50, pp. 107-126, 2015.

[11] Z. Manna and R. Waldinger., "A deductive approach to program synthesis. ," *TOPLAS*, vol. 2, no. 1, pp. 90–121, 1980.

[12] R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. R. Alur, "Syntax-guided synthesis," *FMCAD*, pp. 1–17, 2013.

[13] A. and Klint, P. Van Deursen, "Domain-specific language design requires feature descriptions," *Journal of computing and information technolog*, vol. 10, no. 1, pp. 1-17, 2002.

[14] H. Lieberman, *Your wish is my command: Programming by example.*: Morgan Kaufmann, 2001.

[15] Matej, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Balog, "DeepCoder: Learning to Write Programs.," *arXiv preprint arXiv: 1611.01989*, 2016.

[16] Alex, Greg Wayne, and Ivo Danihelka Graves, "Neural turing machines," *arXiv preprint arXiv: 1410.5401*, 2014.

[17] Łukasz, and Ilya Sutskever Kaiser, "Neural gpus learn algorithms.," *arXiv preprint arXiv: 1511.08228*, 2015.

[18] J.X., Kurth-Nelson, Z., Tirumala, D., Soyer, H., Leibo, J.Z., Munos, R., Blundell, C., Kumaran, D. and Botvinick, M. Wang, "Learning to reinforcement learn," *arXiv preprint arXiv: 1611.05763*, 2016.