Usage of Machine Learning Algorithms in Modelling Software Functions from Tests Defined in Unit Testing.

**Abstract – This paper reviews the latest research in machine learning powered software generation techniques. The statistical program synthesis and induction methodologies are explored, which are 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.**

**Introduction**

Program learning is one of the initial problems that Artificial Intelligence and Machine Learning has been trying to solve since the 70ties (Waldinger, 1969). Originally program learning implied creating formally assigned specifications of the desired program and generating it from them. Often these specifications were more complex than the programs themselves. Currently, most traditional non-statistical, rules based, deductive synthesis systems use input/output pairs to specify program’s behaviour (Devlin, 2017), analogue 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, statistical modelling rather than traditional combinatorial search problem (Polozov, 2015) (Devlin, 2017).

**Non-statistical Program Synthesis**

This method generates program code based on user’s predefined logical specification or intent using non machine learning methods. Non-statistical synthesis can be approached in one of the following techniques (Polozov, 2015).

1. *Deductive Synthesis*

Deductive synthesis views program synthesis as a theorem proving problem (Waldinger., 1980). 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.

1. *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 . DSL is language designed to work in a specific application domain and is providing expressive grammar necessary to work effectively in such domain . 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 parameterised 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 that it relies 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 .

1. *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 (Lieberman, 2001) 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 unique synthesis algorithm for each domain specific application and DSL.

**Statistical Program Synthesis**

This method, as same as most of traditional synthesis systems, generates DLS program code, but instead of explicitly defining deduction, induction or search rules of how such code has to be generated, a statistical model is trained to substitute this logic. In (Parisotto, 2016) 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 synthesise 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, thereby assigns probabilities to the different non-terminals in partial tree derivations of the DSL syntax, which guides the search for the full solution tree as seen in **Figure 1**. This architecture allows producing a vector representation of every possible symbol and expansion rule in the DSL grammar, which then builds up a global tree representation from these vectors in each node of the tree using reverse-recursive pass.

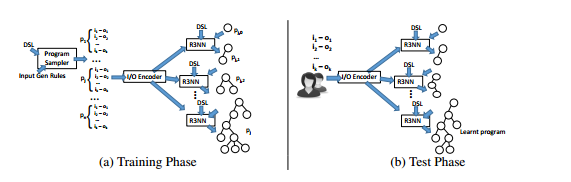


Figure 1 - Usage of R3NN in Program Synthesis

**Statistical Program Induction**

Statistical program induction, on the contrary program synthesis, learns to induce a latent representation of a program from a given I/O examples pairs and does not produce program code. Researchers at Google Brain (Le, 2016) have produced neural network model, which uses augmented helper functions for arithmetic and logic operations. The neural programmer runs several steps of operations using recurrent neural networks. 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**. 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 input are strings with different human readable commands about a dataset, such as selection “select fields from row E that are greater than 150” and expected results is the corresponding data from the dataset.

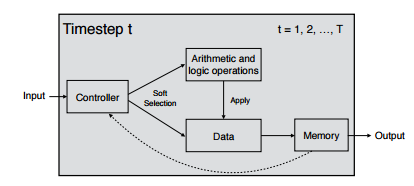
****

Figure 2 - Neural Programmer Architecture

**Program Learning Methodologies Performance**

In (Devlin, 2017) 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 that was synthesized and induced consisted of reproducing Microsoft’s Excels FlashFill functionality, which essentially uses programming by example methodology to automate repetitive spreadsheet data transformation tasks (Gulwani, 2012). Flashfill uses non statistical domain-specific inductive synthesis to generate DSL code in order to solve a unique transformation problem (Polozov, 2015), such as given in **Table 1**. In case of a Neural Program Synthesis approach represents training input and represents desired DSL output. As result performance of a synthesised program is assed with test data as. In case of Neural Program Induction, represents training input and desired training output, as result latent program model is tested with test samples.

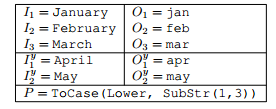


Table 1 - Flashfill example program definition

1. *Replicating Flashfill with Neural Program Synthesis*
2. *Replicating Flashfill with Neural Program Induction*
3. *Performance of Neural Program Learning*

Researchers in (Devlin, 2017) have discovered that though neural program synthesis achieves performance which is close to the Excel Flashill algorithm and the neural program induction performs with far less accuracy with zero input noise. It turned out that neural program synthesis outperforms the original algorithm when input is noisy, which means that I/O testing samples had inconsistency errors of what program they expect to be generated.

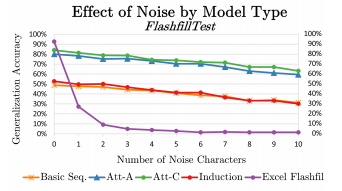


Figure 3 - Performance results comparison

**Conclusion**

Current research indicates that highly accurate, machine learning based models can be created for synthesizing and inducing software from input/output pairs, analogue to ones defined in software unit testing. The biggest bottleneck of using neural network based learning systems is high demand for I/O pair samples. This leads to a necessity of writing random samples generators, which can come closely to replicating functions that are actually being modelled. If thirstiness of statistical program learning for I/O samples is overcome, current advances can lead to developing self-learning models, which can replace some parts of software development, when test driven development process is applied.

**References**

Devlin, J..U.J..B.S..S.R..M.A.R.a.K.P., 2017. RobustFill: Neural Program Learning under Noisy I/O. 1703.07469(arXiv preprint arXiv).

Gulwani, S., 2011. Automating String Processing in Spreadsheetsusing Input-Output Examples. In *Principles of Programming Languages (POPL)*., 2011.

Gulwani, S.H.W.R.a.S.R., 2012. Spreadsheet data manipulation using examples.. In *Communications of the ACM*., 2012.

Le, Q.V.a.S.I., 2016. Neural Programmer: Inducing Latent Programs with Gradient Descent.

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

Parisotto, E..M.A.R..S.R..L.L..Z.D.a.K.P., 2016. Neuro-Symbolic Program Synthesis. (arXiv preprint arXiv:1611.01855).

Polozov, O.a.G.S., 2015. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 10(50), pp.107-26.

R. Alur, R.B.G.J.M.M.M.M.R.S.A.S.R.S.A.S.-L.E.T.a.A.U., 2013. Syntax-guided synthesis. *FMCAD*, pp.1– 17.

Van Deursen, A.a.K.P., 2002. Domain-specific language design requires feature descriptions. *Journal of computing and information technolog*, 10(1), pp.1-17.

Waldinger., Z.M.a.R., 1980. A deductive approach to program synthesis.. *TOPLAS*, 2(1), pp.90–121.

Waldinger, R.J.a.L.R.C.T.P., 1969. A step toward automatic program writing. In *IJCAI*., 1969.