Exploring Robust Neural Methods in Inductive Program Synthesis

Harry Smith
hs3061@columbia.edu
Columbia University
New York, New York

ABSTRACT

Inductive Program Synthesis (IPS) is an attractive goal for AI researchers as it provides a solution to the problem of getting programs to write code. Recent work has shown that neural networks are efficient tools for improving the performance of IPS tasks, and in this paper we formulate several approaches to IPS using state-of-the-art ML neural network architectures. In this work, we review the current field of neural architectures used in program synthesis and we further present a novel formulation for the introduction of Graph Neural Networks in search-guided synthesis methods..

1 INTRODUCTION

The ability for computers to create and understand human-readable source code is an attractive goal with profound implications for software engineering and security research. When trained properly, algorithms can take in natural text descriptions of problems to generate source code that solves them [10]. When machines can evaluate complex code bases, they are capable of automatically identifying otherwise opaque privacy bugs [16]. Along these lines, Inductive Program Synthesis (IPS) is the problem of taking input-output pairs and generating source code that, when executed on the provided inputs, produces the correct corresponding outputs. Researchers have formulated two major approaches to solving this problem: **(1)** "differential interpreters" which aim to learn the structure of the underlying program directly [6] and **(2)** search-oriented strategies which attempt to learn various properties of a program that are useful in guiding a search through the traversal of possible programs [2][8]. Currently, both approaches have their individual drawbacks: differential interpreters often fail to synthesize the correct program [6] whereas search-based approaches are limited in scope by prohibitively large spaces of possible programs.

In this paper, we propose an improvement to search oriented strategies. For our proposed improvement, we attempt to improve the search-based approach laid out in Balog et al.'s DeepCoder research [2], which uses a feed-forward neural network (FFNN) to learn which functions are likely to be consituents of a program that correctly maps the provided input-output pairs. Specifically, DeepCoder uses a FFNN to embed the input integers into vectors in a 20-dimensional space and combines the list of embedded integers with information about the input/output to perform its predictions. Our proposed improvement is to replace the simple FFNN with a Gated Graph Neural Network (GGNN) [9] that will provide a more intelligent embedding of the input/output pairs. Because the GGNN takes in representations of graphs as inputs, this architecture allows us to embed structural information about data types beyond numerical value. Balog et al. are able to infer probabilities of functions being part of candidate solutions using the FFNN alone and recent work has demonstrated that GGNNs can successfully learn a program's structure from the source code [1]; therefore, we anticipate that GGNNs are at least equally capable of estimating constituent function probability. While the change of the neural network architecture requires a slight reformulation of the problem established by Balog et al., we believe that the use of a GGNN will improve the accuracy of this function prediction task on the initial domain and allow DeepCoder to be extended to larger program spaces that include manipulations over more complex data structures.

2 CHOOSING THE RIGHT NN FOR IPS

While our novel contribution in this paper is towards the goal of improving DeepCoder's search-based strategy with GGNNs, we provide a review of other neural architectures which have shown promising results in the field of program synthesis.

Improving Generalization with Greater Attention

DeepCoder is an algorithm that searches for solutions consistent with a small set of input/output pairs. Balog et al. make no mention of the generalizability of the programs that DeepCoder produces [2]. In particular, the authors do not report the rate at which the programs produced are inconsistent with further input/output pairs generated by the same underlying ground-truth program. In general, generalizability is a large concern when learning from "programming-by-example", as a neural network might simply learn to the produce a lookup table for the provided input/output as its target program: a strategy that will always be *consistent* but very rarely *generalizable.* Although Balog et al.'s strategy of guiding search over a specific Domain Specific Language

(DSL) means that the internal AI is unable to define such lookup tables, the question of generalizability remains.

In Devlin et al.'s RobustFill paper, the authors perform a study about how the generalizability of synthesized programs compares with the level of complexity used in the AI that learns the structure of the underlying program [5]. Devlin et al. use increasingly complex Recurrent Neural Network (RNN) architectures to produce programs that solve provided input/output pairs. Specifically, Devlin et al. train four sequential LSTMs of the same size but use increasing levels of attention added among the same constituent pieces. In general, they find that increased attention in the LSTM results in programs that are more likely to generalize to additional input/output pairs. This suggests that DeepCoder's use of a single network with neither attention nor recurrence leaves significant room for improvement in the area of generalizability.

We note additionally that the authors' strategy for program synthesis uses a Beam Searching decoder to predict the next token of the program as each token is generated. This strategy differs somewhat from Balog et al.'s, which passes the results from its FFNN as a pseudo-heuristic for a separate search algorithm. Indeed, this strategy of Beam Search decoding appears to be somewhat more pervasive in recent literature [6][4]. Although Beam Search decoding shows promising results, the advantage of DeepCoder's separate guided search may be its speed. Devlin et al. cite an amortized cost of 0.3 seconds per synthesis on a GPU [5], while a version of DeepCoder generating a short satisfying program on a MacBookPro's CPU takes a similar amount of time.

Learning vs. Synthesizing the Program

Devlin et al. also explore how their attention-heavy LSTM is capable of learning to produce the proper outputs for a given set of inputs on its own without first synthesizing an additional program [5]. This line of work in program *induction* follows from Graves et al.'s 2014 work on Neural Turing Machines, which provided an early demonstration of the capability for neural networks to learn to mimic the behavior of an underlying program through training on input/output pairs [7]. Devlin et al. show that program induction results in similar generalizability performance to that of their most simply trained program synthesis tool [5].

These promising results indicate that both program induction and program synthesis are useful in "programming-by-example" tasks, although the applications of the two strategies diverge signficantly. Program induction never produces any source code, but the task of learning program representations has been shown to have utility in evaluating the correctness of existing source code [1]. The advantage of program synthesis' generation of source code motivates it as worthy area of study: indeed, it is advantageous to explore strategies that produce source code while there are still human programmers reading and using it.

Program Synthesis as a Sequence-to-Sequence Task

Sophisticated Neural Networks have been shown to be effective in machine translation tasks typical to NLP [13]. These tasks fall under the broad category of "Sequence to Sequence" transformations, and it is straightforward to formulate IPS as a problem in this perspective. In this formulation, IPS is the process of learning to map a set of n input/output pairs $\{X_i = ((I_1 \ldots I_k), O) \mid i \in [1 \ldots n]\}$ to a DSL program P, which itself is a sequence of tokens $t_1 \ldots t_m$. Sutskever et al. explain that while simpler neural architectures are unsuitable for Machine Translation tasks, networks with recurrence and/or attention like LSTMs succeed [13]. These results from NLP suggest that ISP might be successfully implemented within a differential interpreter context, using LSTMs with a corresponding decoder to turn example sequences directly into programs without the use of a separate search strategy.

Poloshukin and Skidanov test the performance of a sequence to sequence program synthesis model in their 2018 paper [11]. In particular, the authors attempt to synthesize programs from descriptions of program behavior rather than from explicit examples of input/output pairs. This task thus incorporates some additional challenge of parsing natural language but nevertheless has the advantage of having explicit directions for the resulting programs when compared to DeepCoder's programming-by-example task. Indeed, this may help to resolve the problem of deciding among equivalent programs [2] since the natural language descriptions of the programs can provide information about *how* they should be structured in addition to the baseline information about *what* they should do.

The authors find that an attentional sequence to sequence neural model outperforms Devlin et al's RobustFill model in generating programs that correctly satisfy test cases written based on the target programs, although Poloshukini and Skidanov do not fully explain how they generate input/output pairs that RobustFill expects [11]. The improvement in performance of the sequence model over RobustFill is dramatic—54% over 2.2%—although much of this difference could be attributed to the slight incompatibility of RobustFill for the task of program synthesis from natural language description.

Even better than the sequence to sequence model, however, is the authors' sequence to tree model. This model uses a decoder that creates an AST in the DSL of the paper; that is, the sequence to tree model outputs an abstract syntax tree that defines a program rather than the program itself. This trainable decoder gives the model an advantage in learning

the syntax of the DSL, making it more likely to generate a tree which represents a *valid* program [11]. Their sequence to tree model outperforms the sequence to sequence model 61% to 54%, all without the use of a separate searching algorithm.

Furthermore, in their comically named "Aint̀ Nobody Got Time for Coding" paper, Bednarek et al. create a further improvement over the sequence to tree model, reaching as high as 83% testing accuracy using their "Structure Aware Program Synthesis" (SAPS) model [3]. Each of these sequence to structure models suggest that the use of a differential interpreter alone for program synthesis is indeed "plausible" [3]. However, all of the models explored in Polosukhin and Skidanov's study are improved by the addition of a separate search algorithm. Indeed, their sequence to tree model generated a correct program 86% of the time for the testing dataset, beating out even Bednarek et al.'s intelligent SAPS model [11]. All of this work on differential interpreters suggests that sufficiently sophisticated neural models are effective for program synthesis, and that the simple addition of a search strategy on top of the neural models can serve to improve accuracy even when the search itself is limited to only 100 candidate programs [11].

Leveraging the Transformer Architecture

With the promising results from the previously mentioned attentional neural models, we suspect that the use of a Transformer architecture would provide further improvements in the field of program synthesis. The Transformer is a simple yet powerful architecure that provides groundbreaking performance without including any computationally expensive recurrence [15]. The Transformer has set new benchmarks in the field of Machine Translation, suggesting that it would be useful in the sequence to sequence interpretation of program synthesis that these recent studies have performed. Furthermore, the lack of recurrence in the Transformer provides it with significantly cheaper training costs when compared with other successful models in the area of Machine Translation. Indeed, recent improvements in the formulation of the Transformer model discovered through evolutionary architecture search have already improved the model's accuracies in Machine Translation while also shrinking the number of parameters in the model to the "mobile-friendly" size of seven million [12]. The availability of the Transformer in the TensorFlow extention "tensor2tensor" [14] makes this architecture appealing for further research in the task of program synthesis through the lens of sequence to sequence translation.

3 IMPROVING SEARCH WITH GGNNS

Graph Neural Networks

The architecture with which we propose to augment DeepCoder's FFNN is adapted closely from Li et al.'s 2016 work on Gated Graph Sequence Neural Networks (GGSNNs) [9]. We choose to incorporate into DeepCoder the simpler GGNN—an atomic unit of the GGSNN—which lacks the power to translate sequences to other sequences but is nonetheless able to provide graph-level summary outputs. In particular, the GGNN will **(1)** provide a context-sensitive embedding of the vertices in the input structure and **(2)** pass these embedded vertices through an additional pair of neural networks that serve to predict the likelihood of each function appearing in the correct program. This allows us to mirror the successful strategy of embedding into encoding into decoding that Balog et al. follow [2] while simultaneously adhering to the tested structure of a GGNN.

Input Format

Not surprisingly, any form of Graph Neural Network requires its input to be structured as a graph. Since DeepCoder's DSL recognizes only integers and arrays thereof, it is not difficult to provide a transformation of input/output pairs to graphs. First, we define a graph $\mathcal{G}$ to be a triple $(\mathcal{V}, \mathcal{E}, l_{\mathcal{V}})$, respectively a node set, an edge set, a label function from nodes to integers. In particular, this defines a directed graph with integer edge weights and nodes containing integer values. Second, we create a special "input" node v_{in}, and for each of the k arguments create a special "argument" node $(v_a^1 \dots v_a^k)$, with an edge from the "input" node to the first "argument" node, the first "argument" node to the second, and so on. Then, convert each argument to a series of nodes as follows. For a single integer m that is input argument i, create a node v and an edge (v_a^i, v) between "argument" node i and v and define $l_{\mathcal{V}}(v) = m$. For a list of integers $m_1 \dots m_n$ that is input argument i, create a start node α, and end node ω, and nodes $v_{m_1} \dots v_{m_n}$ for the integers of the list with $l_{\mathcal{V}}(v_{m_j}) = m_j$. Add edges to create a single directed path from α to ω with nodes $v_{m_1} \dots v_{m_n}$ connected in order along the way. The process is repeated again for the output, replacing the "input" node with an "output" node and "argument" nodes with a "result" node. Finally, for every node v representing an input integer (either alone or as part of a list), add an edge (v, v') to every node v' representing an output integer. An example of this conversion is outlined in Figure 1.

We note that this transformation takes a pair of lists (the inputs and the outputs to the program to be synthesized) and creates a single graph structure; thus, when discussing the data that is passed to the GGNN, we use only the term *input* to represent the structure of the original input/output pair. Although this conversion is somewhat verbose in the

case of single lists and integers, it is far simpler in the case of other inputs of interest like trees, heaps, or generic graphs. In these cases, it is only necessary to add the relevant marker nodes for inputs and outputs to the existing graph structure.

GGNN Architecture

We closely adapt the architecture that Li et al. lay out in their work. Thus, we must define a strategy for **node annotation**, the **propagation model**, and the **output model** [9].

Node Annotations. Node annotations are the means by which we encode information about the input vertices into the learning task. This mirrors the step in DeepCoder where information about input/output types are appended after the embedding step. For each vertex $v \in \mathcal{V}$, we define a vector $\mathbf{x}_v \in \mathbb{R}^6$ with the following properties:

- $\mathbf{x}_v^1 = 1$ if $v = v_{in}$ and 0 otherwise
- $\mathbf{x}_v^2 = 1$ if $v = v_{out}$ and 0 otherwise
- $\mathbf{x}_v^3 = 1$ if v is an *arg*node and 0 otherwise
- $\mathbf{x}_v^4 = 1$ if v is a node containing data that represents an integer NOT part of a list, and 0 otherwise
- $\mathbf{x}_v^5 = 1$ if v is is a node containing data that represents an integer that is part of a list, and 0 otherwise
- $\mathbf{x}_v^6 = l_{\mathcal{V}}(v)$ if v is a node representing an integer and 0 otherwise

Thus, each $\mathbf{x}_v$ is a vector with a positive entry in one or two locations, with zeros in all other locations. In order to make the dimensions of inputs consistent for passing into a neural network, it may be necessary to pad inputs with empty dummy vertices so that each input has the same size.

Propagation Model. GGNNs are recurrent networks which transfer information among nodes based on the edge connectivity of the underlying graph. Furthermore, they incorporate update and reset gates. Below is a series of equations which defines the computational graph for the GGNN when embedding a single vertex:

$$\mathbf{h}_v^{(1)} = [\mathbf{x}_v^\top, \mathbf{0}]^\top \tag{1}$$

$$\mathbf{a}_v^{(t)} = \mathbf{A}_v^\top [h_1^{(t-1)\top} \dots h_{|\mathcal{V}|}^{(t-1)\top}]^\top + \mathbf{b} \tag{2}$$

$$\mathbf{z}_v^{(t)} = \sigma\left(\mathbf{W}^{\mathrm{z}}\mathbf{a}_v^{(t)} + \mathbf{U}^{\mathrm{z}}\mathbf{h}_v^{(t-1)}\right) \tag{3}$$

$$\mathbf{r}_v^{(t)} = \sigma\left(\mathbf{W}^{\mathrm{r}}\mathbf{a}_v^{(t)} + \mathbf{U}^{\mathrm{r}}\mathbf{h}_v^{(t-1)}\right) \tag{4}$$

$$\widetilde{\mathbf{h}_v^{(t)}} = \tanh\left(\mathbf{W}\mathbf{a}_v^{(t)} + \mathbf{U}\left(\mathbf{r}_v^t \odot \mathbf{h}_v^{(t-1)}\right)\right) \tag{5}$$

$$\mathbf{h}_v^{(t)} = \left(1 - \mathbf{z}_v^{(t)}\right) \odot \mathbf{h}_v^{(t-1)} + \mathbf{z}_v^{(t)} \odot \widetilde{\mathbf{h}_v^{(t)}} \tag{6}$$

Equation (1) defines the initialization step, wherein the note annotation is concatenated with the zero vector. Equation (2) propagates the information from the representation of neighboring nodes. Equations (3) and (4) define the update gate and reset gates, respectively, while the final two equations (5) & (6) incorporate each gate into the representation for v at the tth timestep. σ refers to the logistic sigmoid function and $\odot$ is element-wise multiplication, as borrowed from Li et al. [9].

We highlight here that for all $v \in \mathcal{V}$ and all $t \in 1 \dots T$, $\mathbf{h}_v^{(t)} \in \mathbb{R}^E$, and we propose the initial parameter setting of $E = 26$. We propose this dimension because of Balog et al.'s choice to embed integers into $\mathbb{R}^{20}$ combined with the need to represent node annotations for the GGNN in $\mathbb{R}^6$. Further, we remark that each $\mathbf{A}_v \in \mathbb{R}^{E|\mathcal{V}| \times 2E}$ and represents information transfer between connected nodes. More explicitly, $\mathbf{A}_v = [\mathbf{A}_{v,OUT}, \mathbf{A}_{v,IN}]$ where $\mathbf{A}_{v,OUT}, \mathbf{A}_{v,IN} \in \mathbb{R}^{E|\mathcal{V}| \times E}$ and we requre that $\mathbf{A}_{v,OUT}$ has non-zero entries only in rows corresponding to vectors v' such that $(v, v') \in \mathcal{E}$ while $\mathbf{A}_{v,IN}$ has non-zero entries only in rows corresponding to vectors v' such that $(v', v) \in \mathcal{E}$. Li et al. are ambiguous about whether these $\mathbf{A}$ matrices are to be fixed or variable with learning. Fixing these matrices ensures that the transfer of information from node to node proceeds exactly with the connections present in the original graphs, whereas a learnt $\mathbf{A}$ allows the network to add direct transfer capability between vertices which are not connected by may be related.

Output Model. Li et al. suggest that, for graph-level outputs of GGNNs, one can define a representation vector as follows:

$$\mathbf{h}_{\mathcal{G}} = \tanh\left(\sum_{v \in \mathcal{V}} \sigma\left(i(\mathbf{h}_v^{(T)}, \mathbf{x}_v)\right) \odot \tanh\left(j(\mathbf{h}_v^{(T)}, \mathbf{x}_v)\right)\right)$$

In this equation, i, j refer to separate neural networks. Thus, $\sigma(i(\mathbf{h}_v^{(T)}, \mathbf{x}_v))$ allows for representation of attention for each node in the overall task for which j is the predictor. While we have highlighted that attention may be useful for improving the constituent function prediction task, our primary goal in proposing the GGNN architecture is to provide the means to encode basic structural relationships of the example data and to allow for DSLs that capture more complex data structures. Thus, since DeepCoder achieves success in constituent function prediction by simply concatenating the position-independent embeddings of the input/output integers and passing the result through a FFNN with no attention, we also propose defining a representation vector as follows:

$$\mathbf{h}_{\mathcal{G}}^{\text{concat}} = k(\mathbf{x}_1, \mathbf{h}_1^{(T)}, \dots \mathbf{x}_{|\mathcal{V}|}, \mathbf{h}_{|\mathcal{V}|}^{(T)}) \tag{7}$$

Here, we define k to be a neural network analagous to that of DeepCoder's Encoder + Decoder structures, a FFNN accepting an input vector $(\mathbf{x}_1, \mathbf{h}_1^{(T)}, \dots \mathbf{x}_{|\mathcal{V}|}, \mathbf{h}_{|\mathcal{V}|}^{(T)}) \in \mathbb{R}^{E|\mathcal{V}|}$ and producing an output $\mathbf{h}_{\mathcal{G}}^{\text{concat}} \in \mathbb{R}^L$, where L is the number of functions available in the chosen DSL[1]. Since a FFNN does not accept inputs of varying lengths, it is necessary to pad

[1]In the case of DeepCoder's DSL, $L = 34$

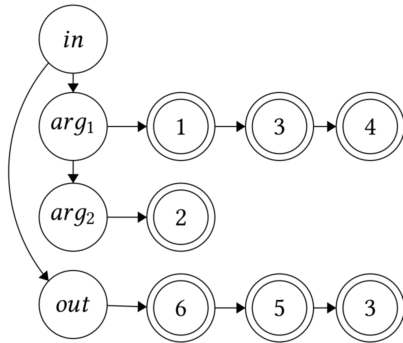

Figure 1: Graph transformation for inputs ([1, 3, 4]; 2) and output [6, 5, 3]. Doubled circles contain the input data and are the only nodes v for which $l_{\mathcal{V}}$ is defined.

the input with empty vertex vectors $\mathbf{0} \in \mathbb{R}^E$ to a maximum length.

4 DISCUSSION

In our review of the literature, we have summarized several strategies that authors are presently using to solve the problem of teaching machines how to learn to code. At the basic level, we have the distinction between program *induction* (learning a program within a neural architecture without producing source code) and program *synthesis* (generating source code that solves a given problem). We find that these strategies can provide comparable results with respect to the generalizability of the solutions they generate.

Additionally, we find that attention and/or recurrence are useful tools for the Neural Architectures used for program synthesis, finding a generally positive trend between increased attention and accuracy of results. This finding led us to propose the use of a Transformer in the task of IPS, since it appears to leverage high levels of attention in its structure to great effect in related tasks.

Finally, we provide a concrete adaptation of GGNNs for use in improving the performance, domain, and generalizability of DeepCoder. Further work will include an implementation of this adaptation into DeepCoder's architecture and a study of how this improves DeepCoder's resulting generalizability for programs in its original DSL. Additionally, we look to expand this DSL to allow for manipulations of more complex data structures like graphs, trees, and heaps using the improved structural embedding provided by GGNNs.

ACKNOWLEDGMENTS

Thanks for the interesting semester. This was the first experience I'd had in program analysis and it was fascinating to see how it could be fused with ML methods.

REFERENCES

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to Represent Programs with Graphs. (2017), 1–17. arXiv:1711.00740 http://arxiv.org/abs/1711.00740

[2] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. *CoRR* abs/1611.0 (2016). arXiv:1611.01989 http://arxiv.org/abs/1611.01989

[3] Jakub Bednarek, Karol Piaskowski, and Krzysztof Krawiec. 2018. Ain't Nobody Got Time For Coding: Structure-Aware Program Synthesis From Natural Language. (2018), 1–12. arXiv:1810.09717 http://arxiv.org/abs/1810.09717

[4] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. 2018. Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. (2018), 1–15. arXiv:1805.04276 http://arxiv.org/abs/1805.04276

[5] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. (2017). arXiv:1703.07469 http://arxiv.org/abs/1703.07469

[6] Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. 2016. Summary - TerpreT: A Probabilistic Programming Language for Program Induction. 1977 (2016). arXiv:1612.00817 http://arxiv.org/abs/1612.00817

[7] Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural Turing Machines. *CoRR* abs/1410.5 (2014). arXiv:1410.5401 http://arxiv.org/abs/1410.5401

[8] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Notices* 53, 4 (2018), 436–449. https://doi.org/10.1145/3296979.3192410

[9] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated Graph Sequence Neural Networks. 1 (2015), 1–20. arXiv:1511.05493 http://arxiv.org/abs/1511.05493

[10] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Andrew W Senior, Fumin Wang, and Phil Blunsom. 2016. Latent Predictor Networks for Code Generation. *CoRR* abs/1603.0 (2016). arXiv:1603.06744 http://arxiv.org/abs/1603.06744

[11] Illia Polosukhin and Alexander Skidanov. 2018. Neural Program Search: Solving Programming Tasks from Description and Examples. (2018), 1–11. arXiv:1802.04335 http://arxiv.org/abs/1802.04335

[12] David R. So, Chen Liang, and Quoc V. Le. 2019. The Evolved Transformer. (2019). arXiv:1901.11117 http://arxiv.org/abs/1901.11117

[13] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. (2014), 1–9. arXiv:1409.3215 http://arxiv.org/abs/1409.3215

[14] Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N Gomez, Stephan Gouws, Llion Jones, \Lukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. 2018. Tensor2Tensor for Neural Machine Translation. *CoRR* abs/1803.0 (2018). http://arxiv.org/abs/1803.07416

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. Nips (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762

[16] Jinman Zhao, Aws Albarghouthi, Vaibhav Rastogi, Somesh Jha, and Damien Octeau. 2018. Neural-augmented static analysis of Android communication. 342–353. https://doi.org/10.1145/3236024.3236066 arXiv:arXiv:1809.04059v1