TOKEN PAIRING TO IMPROVE NEURAL PROGRAM SYNTHESIS MODELS

Walden Yan
MIT PRIMES
Avon High School
Avon, CT 06001, USA
waldenyan20@gmail.com

William S. Moses
MIT CSAIL
Cambridge, MA 02139, USA
wmoses@mit.edu

ABSTRACT

In neural program synthesis (NPS), a network is trained to output or aid in the output of code that satisfies a given program specification. In our work, we make modifications upon the simple sequence-to-sequence (Seq2Seq) LSTM model. Extending the most successful techniques from previous works, we guide a beam search with an encoder-decoder scheme augmented with attention mechanisms and a specialized syntax layer. But one of the withstanding difficulties of NPS is the implicit tree structure of programs, which makes it inherently more difficult for linearly-structured models. To address this, we experiment with a novel technique we call *token pairing*. Our model is trained and evaluated on AlgoLisp, a dataset of English description-to-code programming problems paired with example solutions and test cases on which to evaluate programs. We also create a new interpreter for AlgoLisp that fixes the bugs present in the builtin executor. In the end, our model achieves 99.24% accuracy at evaluation, which greatly improves on the previous state-of-the-art of 95.80% while using fewer of parameters.

1 INTRODUCTION

One of the most exciting ideas in the field of artificial intelligence is the prospect of creating an AI that can write code. The most obvious benefit is that it would accelerate software development. This not only has great economic value but also great intellectual value as it could help quickly verify or attain theoretical results in many fields like computer science, mathematics, and physics. Additionally, a programmer AI could be used in the production of automated personal assistants that, if unable to perform a certain task, can simply program themselves to do said task. Of course, there are additional possibilities, but the authors will leave that to the reader's imagination.

This challenge to create a program that can generate code is known as program synthesis. Using traditional rule-based approaches, this proves to be a very difficult task. But with the recent success of deep learning models, especially in natural language (NL) applications (Radford et al., 2019) and symbolic mathematics (Lample & Charton, 2019), it has become more realistic that a program, aided by a neural network, could handle the task of program synthesis. Previous works have done just that, giving birth to the field of neural program synthesis (NPS).

But before tackling the "synthesis" part of program synthesis, it must first be decided how a desired program should be specified. One approach would be to use input/output (I/O) pairs and have the model infer the underlying computation (Devlin et al., 2017b; Balog et al., 2016). There are a couple of issues with this method. First, there are infinite programs that could fit the sample I/O pairs. Hence, this approach fails to guarantee that a program is actually matching the desired behavior beyond the sample test cases. Second, for complex programs, it is impractical to infer the computation based on test data. Another approach would be to use some sort of formal specification of the computation. However, formal specifications can be quite difficult to attain. Furthermore, it is sometimes just as hard to create a formal specification than implement the program. Finally, there is the option to use a natural language like English to describe what the program should do. For a long time, this ran into the issue of being very unstructured, but this is no longer a barrier with the recent success of machine learning models in NL processing (Radford et al., 2019; Puri et al., 2018).

In this work, we focus on NL specifications. Similar to translation problems, the baseline model is a sequence-to-sequence (Seq2Seq) encoder-decoder scheme using long-short-term-memory (LSTM) recurrent neural networks (Hochreiter & Schmidhuber, 1997). The whole model is then used to guide a beam search for the most likely programs. Other works have already made many improvements on this. One of the first was Polosukhin & Skidanov (2018), which uses a doubly-recurrent neural network to decode the program as a tree structure (Seq2Tree). Another contribution of Polosukhin & Skidanov (2018) is their now public dataset, AlgoLisp, which consists of 100,000 English text description programming tasks with corresponding test cases and example solutions written in a Lisp-inspired domain specific language (DSL). This dataset is used by several other works (Bednarek et al., 2018; Nye et al., 2019; Chen et al., 2019) and is also used in our work.

The first of these additional works on AlgoLisp, Bednarek et al. (2018), introduced SAPS, a modified Seq2Tree model with a novel signal propagation scheme and attention mechanism. Then, Nye et al. (2019) offered SketchAdapt, a Seq2Seq model that has the capability of letting an enumerative search take over at specified points. Finally, Chen et al. (2019) gave TP-N2F, a Seq2Seq model based on tensor product representations. Although it did not use the AlgoLisp dataset, it is also worth mentioning Bunel et al. (2018) as it introduced the use of a trained syntax layer.

1.1 Our Contribution

Although models are improving, all previous work has faced the issue of dealing with the implicit tree structure of programs. Some choose to use a model that explicitly decodes a tree (Polosukhin & Skidanov, 2018; Bednarek et al., 2018), but the issue with these is that the complexity of the recurrence structure makes them difficult to train effectively. Furthermore, linear models have been studied for much longer than tree models and have thus been refined over time. But simply using a linear model has obvious issues as the unnatural settings makes the task more difficult. Consider a program that at some point requires performing a sum over a list:

```
...(reduce (code for calculating the list) 0 +) ...
```

The list itself may take quite a few tokens to calculate. Hence, the three tokens that specify the summation, `reduce`, `0`, and `+`, might appear far apart despite them being grouped together logically. A linear model will need to remember that it has to place those two extra tokens. Furthermore, there are likely multiple levels in the program where this situation occurs, amplifying the issue.

To ease this difficulty for linear models, we propose a novel technique, *token pairing*. This combines tokens that are usually logically coupled into one convenient package for the model to use, despite how far apart they may appear in the final program. This allows us to apply a linear model very successfully, and with the addition of attention mechanisms and a specialized syntax layer, we are able to achieve new state-of-the-art performance on the AlgoLisp dataset.

While exploring some inconsistencies in the AlgoLisp dataset, we also found a bug in the builtin code executor. To fix it, we create a new interpreter that we hope to make public with the rest of our code.

1.2 Other Related Work

While much of the work mentioned earlier was on creating better models, some work has also been on done on using different paradigms to train NPS models. Bunel et al. (2018) explores the use of a reinforcement learning paradigm. That work also used a different program synthesis task, the Karel dataset, created by Devlin et al. (2017a). Another notable example, Abolafia et al. (2018) created a training framework based on a priority queue that only requires a reward function during training.

An area of research similar to NPS is that of differentiable compilers. The earliest work was Graves et al. (2014) which created the Neural Turing Machine (NTM), a neural-network analog of a turing machine on differentiable memory that could learn to perform a task through gradient descent. Later, Graves et al. (2016) introduced an improvement on the NTM that was able to perform well on an impressive number of unique tasks and even demonstrate problem-solving abilities.

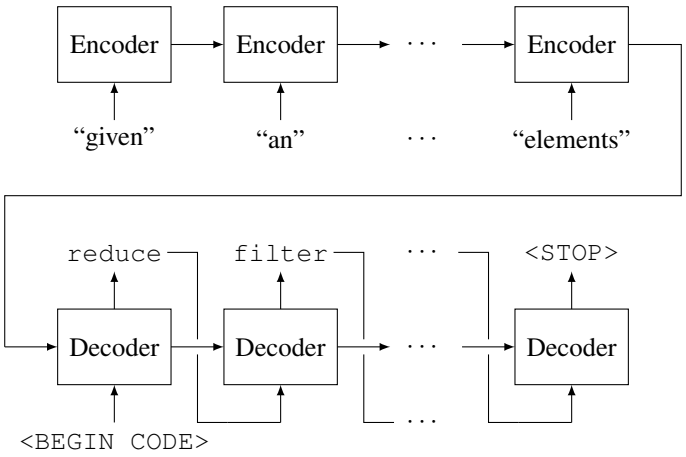

Figure 1: Bare Seq2Seq LSTM model setup at prediction time (no beam search).

2 General Approach

The task of neural program synthesis can nearly be treated as a translation problem in the sense that a sequence of tokens is being converted to another sequence of tokens. The way that we approach this is to try to construct the output sequence token by token. To do this, we train a model that can to take the input sequence and a prefix of program tokens and provide a probability distribution for the next token. After deciding which token to use next, it is added to the existing prefix and the process repeats.

More formally, using this model, we can assign a probability to any sequence of output tokens $\tilde{y}_1, \tilde{y}_2, \ldots, \tilde{y}_m$:

$$\prod_{i=1}^{m} P\left(\tilde{y}_i | x_1, \ldots, x_l; \tilde{y}_1, \ldots, \tilde{y}_{i-1}; \Theta\right)$$

where $x_1, \ldots, x_l$ is the sequence of input tokens and P is a conditional probability modeled by the neural network with parameters Θ.

Then comes the task of finding the sequence of tokens that maximizes that probability. The simplest approach would be to repeatedly select the next most likely token (Note that there is a specialized `<STOP>` token to indicate when the output sequence is complete). However, this does not always yield the optimal answer. On the other hand, if we do a beam search, where we continually keep track of and recurse from the K highest probability prefixes, we can get an approximation that works quite well in practice[1].

To actually model the probability distribution, the standard Seq2Seq model uses a recurrent neural network, such as an LSTM (Hochreiter & Schmidhuber, 1997; Sutskever et al., 2014). One LSTM, the encoder, iterates over the input sequence and updates its state accordingly to create a vector embedding of the text. This vector is then fed as the initial state to a decoder LSTM that then iterates over the existing prefix of program tokens. The final decoder LSTM cell's output is then put through a learned linear transformation and fed through a softmax gate to arrive at a probability distribution for the next token. At training time, the model is fed the actual prefixes from an example program. At prediction time, the token sequence is determined from the model's own output.

[1] when implementing this, it is more convenient to use the log probabilities

3 OUR METHOD

We make three main modifications to the standard Seq2Seq model. The first of the modifications is a new technique we introduce, *token pairing*, that involves preprocessing the data to find common program substructures that can be condensed into a new token for the network to use. The other two, the addition of an attention mechanism and the addition of a learned syntax layer, are additions onto the model architecture and have been studied in previous works (Bunel et al., 2018; Bahdanau et al., 2014).

3.1 TOKEN PAIRING

Consider again a program that sums over a list as a subprocedure:

```
...(reduce (code for calculating the list) 0 +) ...
```

The issue lies in the fact that the `reduce`, `0`, and + tokens appear far apart despite being part of one logical unit, a summation. Instead, if they were combined into one token, `sum`, there would be no need for the model to remember to place the `0` and + in the correct spot, and as a result, the code becomes much easier for a neural network to produce:

```
...(sum (code for calculating the list)) ...
```

To generalize, we would like to expand AlgoLisp's vocabulary to include tokens that represent compositions of existing tokens. To do this, we iteratively find the most frequent 2-token substructure in the abstract syntax trees (ASTs) over all training programs and collapse them into a new token. This will always take the form of a parent token and a child token. Note that the child token can appear in multiple positions so the algorithm needs to distinguish between these different placements. The children of the child token and the remaining children of the parent token will become children of the new token in the order they appear originally. We repeat this until we arrive at the desired vocabulary size. With these new tokens, the dataset is remade to have all programs be written in the expanded vocabulary, condensing them as much as possible.

In addition to helping deal with the tree structure of programs in a linear settings, token pairing also makes the programs shorter, which has its own benefits. First, it makes training more stable, especially in the early stages. In fact, although we train all of our models with curriculum training (detailed in Table 2), we found that it was not actually required when we used more token pairing. Additionally, this reduces the depth of the search space, which makes it more likely the model will be able to converge on the correct sequence. However, we must also consider that the width of the search space increases, so there is a tradeoff going on that we explore more closely in Section 4.1.

3.2 MODEL ARCHITECTURE

Our model makes use of three LSTM's, all with a hidden size of H. One LSTM is the encoder that begins with a learned initial state and is run on the sequence of input tokens. Input tokens are given as learned vector embeddings of dimension d_{emb}. The outputs of the encoders are stacked into an l by H matrix, $U_{\texttt{keys}}$, to be used by the attention mechanism, where l is the length of the input sequence.

The final state of the encoder LSTM is then fed as the initial state of the decoder LSTM. The decoder LSTM is given a `<BEGIN CODE>` token and the known prefix of program tokens, also embedded as vectors of dimension d_{emb}. The final decoder cell's output, $\tilde{h}_d^{(t)}$, is then used to attend to the encoder's outputs:

$$h_d^{(t)} = \tilde{h}_d^{(t)} W_i + \left(\text{softmax}\left(\tilde{h}_d^{(t)} U_{\texttt{keys}}^T\right) U_{\texttt{keys}}\right) W_a + B_a,$$

where $W_i, W_a \in \mathbb{R}^{H \times d_{\text{emb}}}$ and $B_a \in \mathbb{R}^{d_{\text{emb}}}$ are learned weights and biases. This mechanism is similar to dot-product attention except that the keys are not separate from the values. This technique has

```
program ::= symbol
symbol ::= constant | argument | function_call | function | lambda
constant ::= number | string | True | False
function_call ::= (function_name arguments)
arguments ::= symbol | arguments, symbol
function_name ::= reduce | filter | map | head | + | - ...
lambda ::= lambda function_call
```

Figure 2: Partial specification of AlgoLisp's DSL (Polosukhin & Skidanov, 2018)

been shown to greatly improve performance in recurrent neural networks. Finally, $h_d^{(t)}$ is compared against the program token embeddings:

$$\tilde{o} = h_d^{(t)} E_{\text{dec}}^T$$

where E_{dec} is a n_{dec} by d_{emb} matrix representing the n_{dec} program tokens' vector embeddings.

In a simpler model, $\tilde{o}$ would be fed into a softmax to generate a probability distribution but instead, we have a third LSTM designed to penalize syntactically invalid tokens. The syntax LSTM has a learned starting state and is run on the same token embeddings as the decoder LSTM. The final output of the syntax LSTM, $\tilde{h}_s^{(t)}$, is used with $\tilde{o}$ to generate the final probability distribution:

$$o^{(s)} = \lambda \tanh\left(\tilde{h}_s^{(t)} W_s + B_s\right)$$
$$o = \text{softmax}\left(\tilde{o} - \exp\left(-o^{(s)}\right)\right)$$

where $W_s \in \mathbb{R}^{H \times d_{\text{emb}}}$ and $B_s \in \mathbb{R}^{d_{\text{emb}}}$ are learned parameters. We choose the hyperparameter λ to cap how large the penalization can be. We use $\lambda = 20$ in our implementation which is enough to avoid overflow errors.

We use the negative log probability as our loss function and add an additional term to prevent the syntax layer from penalizing valid programs:

$$\mathcal{L}(\Theta) = -\frac{1}{NL} \sum_{i=1}^{N} \sum_{j=1}^{L} \left(\log\left(\epsilon + o_{i,y_j}\right) - o_{i,y_j}^{(s)}\right).$$

Here, L is the length of the longest program (for training, we pad shorter programs with <STOP> tokens so they are all equal), and N is the number of training problems. o_{i,y_j} is the probability the model assigns to the correct j-th token of the i-th training problem. $o_{i,y_j}^{(s)}$ is defined similarly for the syntax penalization. We use $\epsilon = 10^{-10}$ to help stabilize the loss function.

4 Experiments

We evaluate our model on the AlgoLisp dataset, created by Polosukhin & Skidanov (2018). The dataset consists of 100,000 program synthesis tasks specified in English. Each problem comes with 10 input-output pairs against which programs can be tested. Additionally, the dataset comes with example solutions, written in its Lisp-inspired domain-specific language (DSL). The grammar for the DSL is shown in Figure 2 and a few examples of programs written in the DSL are shown in Table 1. The large variety of programs, relative shortness of NL descriptions, and number of previous works against which to compare makes the AlgoLisp data base very appealing.

The problems are split into three sets: train (79214 examples), "dev" (10819 examples), and test (9967 examples). Text descriptions are up to 164 tokens long and programs are up to 217 tokens long. On average, the NL descriptions are 38 tokens and the programs are 24 tokens.

Statement	Example Solution
Consider an array of numbers a, what is the length of the longest subsequence of a that starts with 1 and increases by 1?	`(- (reduce a 1 (lambda2 (if (== arg1 arg2) (+ arg 1) arg1))) 1)`
Consider an array of number, your task is to find if element at position 0 (0 based) in the given array is non-positive.	`(<= (head a) 0)`
Given an array of strings a, what is the element of a that has the biggest value (lexicographically)?	`(reduce a "" str_max)`

Table 1: Sample of problems and their example solutions written in AlgoLisp's DSL.

Cleaning and Custom Interpreter 10% of the problems in the original dataset have example solutions that do not actually pass their associated test cases. Following previous work (Bednarek et al., 2018; Nye et al., 2019; Chen et al., 2019), we evaluate on both the full dataset and a cleaned dataset with those 10% of problems removed. After closer examination, we also found that the public AlgoLisp executor created by Polosukhin & Skidanov (2018) is faulty. Specifically, their executor uses Python range objects instead of the corresponding list whenever `range` is called. Operations like `slice` also maintain these objects as ranges as opposed to converting them to lists. As a result, their executor may output a range object while the correct answer is the associated list, and this is reported as a wrong answer. To avoid this issue, we create and use our own interpreter. With our own interpreter, only 3% of the problems in the original dataset are faulty. However, for the sake of comparability with other works, we decided to use the builtin executor for the purpose of cleaning the dataset. Note that our choice to use a custom interpreter will contribute to the gap in accuracy between our model and previous work on the full dataset but will not affect our accuracy on the cleaned dataset.

For all of our models, we chose $H = 512$ and $d_{\text{emb}} = 128$. Parameters were updated using Adam Optimization with a learning rate of 0.001, $\beta_1 = 0.9, \beta_2 = 0.999$, and $\epsilon = 10^{-8}$ (Kingma & Ba, 2014). Additionally, gradients were clipped to have a global norm of at most 1. Training was done with a batch size of 32 and we followed the curriculum shown in Table 2. Every 1000 iterations, we save the model and calculate its loss on the "dev" set. Then, in the end, we use the one that had the best performance. When evaluating our final models, we perform a search with a beam size of 10. We then measure the percent of problems for which our model is able to produce at least one correct solution (passes all the I/O pairs).

4.1 Varying Amount of Token Pairing

Our first goal was to finding an optimal amount of token pairing to use. Although token pairing has the benefits described in Section 3.1, as we increase the number of tokens in the expanded vocabulary, we are increasing the number of tokens the model must consider and hence also increasing the width of the search space. Thus, we expect that increasing the number of tokens beyond a certain point would be less helpful or hurt the performance of our model. After training and evaluating four different models using different amounts of token pairing (shown in Table 3), we found that this was indeed the case. Interestingly, it seems to depend on which version of the dataset is being used. On the cleaned dataset, using 150 tokens was clearly the best option of the four we tried. On the

Curriculum Number	Description	Duration
1	Shortest 25% of training problems (combined text and example program)	1 epoch
2	Shortest 50%	1 epoch
3	Shortest 75%	1 epoch
4	All training problems	10 epochs

Table 2: Curriculum used for training the models in this work.

Model	Full		Cleaned		Parameters
	Dev	Test	Dev	Test	
TokPair-81 (no token pairing)	95.09%	94.15%	97.71%	97.11%	4.16M
TokPair-150	97.07%	**96.22%**	**99.53%**	**99.24%**	4.20M
TokPair-200	**97.15%**	96.09%	99.02%	98.89%	4.24M
TokPair-300	97.01%	96.13%	98.98%	98.68%	4.30M

Table 3: Performance of our model using different amounts of token pairing on both full and cleaned datasets. The number after "TokPair-" signifies the size of the expanded vocabulary. The original vocabulary consists of 81 tokens.

Model	Full		Cleaned		Parameters
	Dev	Test	Dev	Test	
TokPair-150	**96.89%**	**96.03%**	**99.53%**	**99.24%**	4.20M
SketchAdapt	88.80%	90.00%	95.00%	95.80%	$\sim$7M
Seq2Tree	86.10%	85.80%	-	-	-
TP-N2F	-	84.02%	-	93.48%	-
SAPSpre VH Att	86.67%	83.80%	95.02%	92.98%	5.73M

Table 4: Our best model compared to previous results. Values not reported in corresponding work are left empty.

full dataset, the results are a little less clear as to what value is optimal. Nonetheless, it is clear that simply adding more tokens will not continue to improve the model significantly.

4.2 Comparison to Previous Work

Taking our best model (TokPair-150), we now compare it against previous results in Table 4. We significantly improve upon the previous state of the art, SketchAdapt, introduced in Nye et al. (2019), reducing the error by a factor of 5.5, achieving a less-than-one-percent failure rate. Some other noteworthy works against which we compare are Seq2Tree (Polosukhin & Skidanov, 2018), TP-N2F (Chen et al., 2019), and SAPS (Bednarek et al., 2018). Although some of the works ommit the size of their model, we still use less parameters than the previous state-of-the-art, demonstrating that our results are not simply from having a bigger model.

4.3 I/O Pair Versus Golden Program Evaluation

For all the other experiments, we relied on I/O pair evaluation, where each program our model produced is run on test cases to determine whether or not it is correct. The other common metric is to see if the produced program is exactly equal to the example, "golden", program. Of course, if our model produced a unique program different from the example program, then golden program evaluation would report a failure while I/O pair evaluation would report a success. Since the style of programming is relatively consistent throughout the examples in the dataset, a difference in accuracy between I/O pair evaluation and golden program evaluation may indicate the model's capability of producing its own style of code. To see if our model indeed does this, we evaluate our TokPair-150 model using both I/O pairs and against a golden program. In addition to using a beam size of 10 like in the other experiments, where multiple candidate programs are produced, we also decided to also

Evaluation Method	beam_size=10		beam_size=1	
	Dev	Test	Dev	Test
IO Pair	99.53%	99.24%	94.99%	94.08%
Golden	99.34%	98.89%	94.29%	92.67%
Difference	0.19%	0.35%	0.70%	1.41%

Table 5: TokPair-150 model evaluated using both I/O pairs and against a golden program. Testing was performed using both a beam size of 10 and a beam size of 1 (equivalent to greedily choosing next token).

look at using a beam size of 1, or simply choosing the next token greedily and arriving at a single final program. The data for this experiment is shown in Table 5. Surprisingly, there were a decent amount of unique working programs produced by our model. This is more clear in the case where the beam size is 1, where almost 1.5% of the programs that passed the I/O pairs on the test set were not the golden program. After all, when 10 different programs are produced, it is more likely that at least one is the golden program.

4.4 Novel Programs and Token Utilization

Naturally, we were inclined to inspect whether or not the non-golden programs produced actually worked or if they simply were able to slip by the test cases. We found that a few of the programs were in fact incorrect. However, there were still many programs that were correct but were just implemented differently from the golden program. For example, in one case the golden program was

```
(map (slice (reverse a) 0 (/ (len (reverse a)) 2)) (partial0 2 *))
```

and the model produced

```
(slice (map (reverse a) (partial0 2 *)) 0 (/ (len (map (reverse a)
                        (partial0 2 *))) 2)).
```

Though perhaps even more surprisingly, there were cases where the program produced by the model was identical to the golden solution but it was synthesized using a different sequence of tokens. This means that, although all the training data was condensed as much as possible using the expanded vocabulary, the model was still able to understand and use lower-level tokens and circumvent the aid of higher-level ones in certain cases.

5 Conclusion and Future Work

Our work was focused on the AlgoLisp dataset of English description-to-code programming problems. We took the bare Seq2Seq LSTM model for neural program synthesis and augmented it with an attention mechanism and a jointly trained syntax layer to improve syntactical validity. Additionally, we applied a novel technique, *token pairing*, to ease the difficulties of forcing a tree structure upon a linear model. We experimented with changing the amount of token pairing and found that using 150 tokens was about optimal on the cleaned dataset. We then took our best model and demonstrated that it outperformed previous works significantly while using fewer parameters, reducing the error by a factor of 5.5 and entering the $< 1\%$ failure range. In addition to this, we found that there are instances where our model is able to produce working code different from the example programs. Furthermore, our model demonstrates the ability to still use lower-level tokens in cases where it would have been trained to use higher-level tokens. Finally, in addition to our new model and token pairing technique, we also created an AlgoLisp interpreter that avoids the issues with the builtin executor.

For areas of possible continued work in the future, we would be interested in exploring ways that token pairing could be done implicitly as a part of the model architecture. We might also consider training our model in a different paradigm such as reinforcement learning (Bunel et al., 2018) or priority-queue training (Abolafia et al., 2018). It may also be interesting to see what would happen if the training data was not fully condensed using the new tokens, but rather had a variety of lower-level and higher-level tokens, and if that would improve the model's performance or aid in producing unique working programs.

Acknowledgments

The authors would like to thank Ping Yan and Jun Wang for their constant love and support. The authors would also like to thank Charles Leiserson, Srini Devadas, and Slava Gerovitch for their invaluable feedback and advice.

This research was supported in part by the MIT PRIMES program, and in part by LANL grant 531711. William S. Moses was supported in part by a DOE Computational Sciences Graduate Fel-

lowship DE-SC0019323. Research was sponsored by the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

Daniel A Abolafia, Mohammad Norouzi, Jonathan Shen, Rui Zhao, and Quoc V Le. Neural program synthesis with priority queue training. *arXiv preprint arXiv:1801.03526*, 2018.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Jakub Bednarek, Karol Piaskowski, and Krzysztof Krawiec. Ain't nobody got time for coding: Structure-aware program synthesis from natural language. *arXiv preprint arXiv:1810.09717*, 2018.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

Kezhen Chen, Qiuyuan Huang, Hamid Palangi, Paul Smolensky, Kenneth D Forbus, and Jianfeng Gao. Natural-to formal-language generation using tensor product representations. *arXiv preprint arXiv:1910.02339*, 2019.

Jacob Devlin, Rudy Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural program meta-induction, 2017a.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. *arXiv preprint arXiv:1703.07469*, 2017b.

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

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538 (7626):471–476, 2016.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

Guillaume Lample and François Charton. Deep learning for symbolic mathematics. *arXiv preprint arXiv:1912.01412*, 2019.

Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. *arXiv preprint arXiv:1902.06349*, 2019.

Illia Polosukhin and Alexander Skidanov. Neural program search: Solving programming tasks from description and examples. *arXiv preprint arXiv:1802.04335*, 2018.

Raul Puri, Robert Kirby, Nikolai Yakovenko, and Bryan Catanzaro. Large scale language modeling: Converging on 40gb of text in four hours. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 290–297. IEEE, 2018.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pp. 3104–3112, 2014.