
Program Language Translation Using a Grammar-Driven Tree-to-Tree Model

Anonymous Authors¹

Abstract

The task of translating between programming languages differs from the challenge of translating natural languages in that programming languages are designed with a far more rigid set of structural and grammatical rules. Previous work has used a tree-to-tree encoder/decoder model to take advantage of the inherent tree structure of programs during translation. Neural decoders, however, by default do not exploit known grammar rules of the target language. In this paper, we describe a tree decoder that leverages knowledge of a language’s grammar rules to exclusively generate syntactically correct programs. We find that this grammar-based tree-to-tree model outperforms the state of the art tree-to-tree model in translating between two programming languages on a previously used synthetic task.

1. Introduction

Program translation is the process of converting code in one programming language to code in another, ideally with minimal human effort. It has the possibility to significantly alter the ways in which programs are developed. With a perfect translator, a programmer could freely choose a programming language to use without regard to whether the chosen language is the most efficient for the task at hand. Effective program translation would thus enable programmers to focus on the content and development of a specific program as opposed to the details of a particular language. With such a translation method, a developer could easily import code to different platforms, streamlining the development process.

Programming languages are similar to natural languages in many ways, and natural language translation has been studied extensively. Sequence-to-sequence translation models, which map input sequences to output sequences, have achieved great performance (Bahdanau et al., 2014; Cho

et al., 2014; Eriguchi et al., 2016; Vaswani et al., 2017). While similar to natural languages, programming languages have a distinct structure which makes it harder to use the same tools for translation. For instance, the RNN-based sequence generator, which easily generates phrases in a natural language, finds it difficult to generate long syntactically correct programs (Karpathy et al., 2015).

Techniques for program language translation are often variants of statistical machine translation (SMT) (Lopez, 2008), which involves modeling the probability distribution of phrases in the target language given phrases in the source language. Nguyen found SMT methods from NLP applied to programming language translation produced many syntactically incorrect programs (Nguyen et al., 2013). They later found that SMT methods could be improved by incorporating knowledge of program syntax (2016a). They also saw success matching tokens in different languages through similarities in their usage in context (Nguyen et al., 2016b).

Recently, there has been a rise in the use of neural networks for programming language tasks. Neural networks have been applied to code-generation tasks converting images to code (Beltramelli, 2017) and converting text to code (Yin & Neubig, 2017). They have also been applied to tasks like program induction (Bunel et al., 2016) and program classification (Peng et al., 2015).

Recent work applied tree-based neural networks to programming language translation (Chen et al., 2018). Their tree-to-tree encoder/decoder model performed better than sequence-to-sequence models and improved on state-of-the-art program translation approaches by margins of 20 points for real-world translation projects.

However, this tree-to-tree program translation model faces various issues, including the generation of syntactically invalid programs and inefficiencies stemming from the need to generate end of tree tokens at each branch of the underlying abstract syntax tree (AST). Our work makes this tree-to-tree model more efficient by leveraging the grammar of the language to generate only syntactically correct programs and removing redundant notation such as end of tree node to decrease the number of required operations in the model.

The remainder of this paper is organized as follows. Section 2 presents the tree-to-tree model and discusses prior work.

¹Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

by Chen et al., we do not binarize the output tree, because the target language’s grammar rules cannot easily be applied to trees in which a node’s siblings can appear as its children.

To generate a node’s children, we first call a function that returns a list of the categories of tokens that can be generated by that node at each child index. A category is a unique set of valid children. For instance, in the language FOR (described further in the Experiments section), the node `<PLUS>` can generate tokens in the *Expression* category: (`<PLUS>`, `<MINUS>`, `<VAR>`, or `<CONST>`). Each category k has an associated learnable weight matrix W_k .

To generate each child, we create a node embedding e_t computed the same way as the tree decoder by (Chen et al., 2018). We then generate the node’s token by finding the most probable token out of the set of possibilities as follows:

$$t_t = \text{argmax softmax}(W_k e_t). \tag{5}$$

Note that as the number of tokens in class k is substantially fewer than the total number of tokens in the language, this is an easier prediction problem than for the prior tree decoder.

After this, as in (Chen et al., 2018), we feed an embedded representation of t_t into an LSTM to compute the hidden and cell state for each of its children. We always train using teacher forcing (passing the true value of t_t into the LSTM rather than the generated value) because a single incorrect token that generates different categories of children than the correct token could make the probability of generating a correct token for any of its children zero. This could zero out most of our gradients, slowing down training.

The decoder iteratively creates nodes, starting at the root and generating each child node from the hidden and cell states generated for that child from its parent. Since the program’s grammar does not allow tokens to be generated from terminal tokens, branches of the tree end automatically when a terminal is produced. This means our grammar decoder does not have to learn to generate `<EOS>` tokens, simplifying the translation task and decreasing the number of operations the model needs to perform.

4. Experiments

We tested our grammar-based tree-to-tree model on a task described by Chen et al. which examines the ability of a model to translate between simple programming languages of different paradigms. For the task, we randomly generated a synthetic dataset of 100,000 training programs in FOR, a simple imperative programming language created by (Chen et al., 2018). We also generated test and validation sets with 10,000 programs. The programs were generated by an almost context-free probabilistic grammar. It is not fully context-free since to avoid generating programs that used

Table 1. FOR/LAMBDA training dataset description.

METRIC	FOR	LAMBDA
TOTAL PROGRAM COUNT	100K	100K
AVERAGE PROGRAM LENGTH	22	56
MINIMUM PROGRAM LENGTH	5	13
MAXIMUM PROGRAM LENGTH	104	299
NUMBER OF TOKENS IN LANGUAGE	32	33

variables before they were defined, we kept track of previously defined variables and only used those in expressions. These programs were then fed into a translator function that converts them into a simple functional language called LAMBDA also created by (Chen et al., 2018). More dataset details are available in Table 1.

In Table 2, we compare our model’s performance to the baselines described by (Chen et al., 2018) and to our own reimplemented tree-to-tree and tree-to-sequence models with the architecture and hyperparameters described by Chen et. al. For the hyperparameters of the grammar model we simply used the tree-to-tree model’s hyperparameters and did not try to optimize them. Each model was trained 5 times over half a million examples. We also ran one of each of our three models to convergence (30 epochs). Program accuracy was measured on the test set by counting the percentage of perfectly correct translated programs.

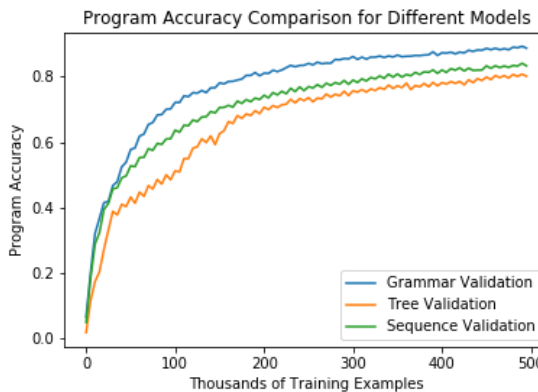


Figure 2. Average validation accuracy for grammar, tree2tree, and tree2seq model for 5 epoch runs.

Results are summarized in Figure 2 and Table 2. The grammar-based model achieved an average 88.82% accuracy, outperforming our reimplementation of the tree-to-tree model and tree-to-seq model. When run to convergence, the difference in accuracy among the models decreased but the grammar model remains more accurate. In addition, the grammar model converges more stably than the other models as the standard deviation of its accuracy is much

Table 2. Program Accuracy on the FOR/LAMBDA translation task. Since our datasets are not identical, performance of the reimplemented Tree2Tree and Tree2Seq models does not perfectly match the results reported by (Chen et al., 2018). The Chen et al. have some entries as N/A as they don't mention multiple runs. The last three rows display the results of running our models to complete convergence.

MODEL	MEAN ACCURACY	σ ACCURACY
GRAMMAR TREE2TREE	88.82%	0.64%
REIMPLEMENTED TREE2TREE	80.69%	7.02%
REIMPLEMENTED TREE2SEQ	83.59%	3.95%
CHEN ET AL. TREE2TREE (EASY)	99.76%	N/A
CHEN ET AL. TREE2SEQ (EASY)	98.36%	N/A
CHEN ET AL. TREE2TREE (HARD)	97.50%	N/A
CHEN ET AL. TREE2SEQ (HARD)	87.84%	N/A
(LONG) GRAMMAR TREE2TREE	93.70%	N/A
(LONG) REIMPLEMENTED TREE2TREE	91.89%	N/A
(LONG) REIMPLEMENTED TREE2SEQ	90.60%	N/A

smaller (while the tree2tree model's convergence depends a lot more on random seed). The grammar model's average run beat the best run of both baselines.

The reimplemented tree-to-tree and tree-to-seq models performed worse than the results reported by (Chen et al., 2018). This discrepancy could be caused by differences between the complexity of our datasets. Since their dataset was unavailable, we implemented a dataset with similar average program lengths. However, our dataset may have used more variables or constants, and it had much greater variation in program lengths. Our average program length was about the length of the programs in the easy version of their task, but our programs with longest length were about twice as long as the programs in the hard version of their task. It is also possible that despite our attempt to faithfully re-implement (Chen et al., 2018) that there are slight differences between the two tree-to-tree models as we lacked their code. Any of those differences would be shared with the grammar model as except for the code corresponding to the decoder, they shared all of their code. To make it easier for future research to evaluate on the same task, we release our dataset and experiment code here, <https://www.dropbox.com/sh/1q4aejr57jk40fs/AADKTVgKqLHuIIzNdLANjGRea?dl=0>.

5. Conclusion

This paper proposes a grammar-based program language translation approach using a grammar decoder that outperforms the state of the art tree-to-tree models for program language translation in both number of operations needed and accuracy. Future work will explore ways to improve convergence and broaden the practical applicability of our approach to various real programming translation problems.

One limitation to this approach is the practical difficulty

of obtaining the training data needed to apply this model to a new pair of languages. Training requires a parallel corpus of programs in two languages. Previous researchers such as (Chen et al., 2018) have obtained such datasets by using languages with an explicit translator between them (which largely obviates the need for a neural translation program) or by finding real-world programs implemented in both programs (which may be difficult to obtain).

Our model also requires a formal grammar for the target language. In the absence of a grammar, we could approximate a grammar from the training set by recording all child tokens of each unique token, but this could make our model incapable of generating rare but valid syntactic patterns.

Finally, our model currently caps the number of variable names and literals at a fixed number determined before training. Future work could explore alternative ways to generate arbitrarily many variables and literals by copying them from the input program using a method similar to that implemented by (Yin & Neubig, 2017).

In future work, we will integrate other ideas from natural language translation to our tasks. One possibility includes self-attention (Vaswani et al., 2017), a mechanism that provides the model at each time step with its state at previous time steps and may help the model learn more complex relationships among parts of the program. We could also integrate a language model into our decoder to help with translating unusual expressions not seen in the training data.

Our current approach makes parallelization of the training process difficult. Since every tree has a different structure, we cannot batch training examples together for faster processing on GPUs. Sequence-to-sequence models can circumvent this by batching programs of the same size or padding shorter sequences. We will need to explore methods of batching tree generation and apply them to our model.

References

Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. Neural machine translation by jointly learning to align and translate. *International Conference on Learning Representations*, abs/1409.0473, 2014. URL <http://arxiv.org/abs/1409.0473>.

Beltramelli, Tony. pix2code: Generating Code from a Graphical User Interface Screenshot. *ArXiv e-prints*, pp. 1–9, 2017. URL <http://arxiv.org/abs/1705.07962>.

Bunel, Rudy, Desmaison, Alban, Kohli, Pushmeet, Torr, Philip H. S., and Kumar, M. Pawan. Adaptive neural compilation. *CoRR*, abs/1605.07969, 2016. URL <http://arxiv.org/abs/1605.07969>.

Chen, Xinyun, Liu, Chang, and Song, Dawn. Tree-to-tree Neural Networks for Program Translation. *International Conference on Learning Representations*, 2018. URL <http://arxiv.org/abs/1802.03691>.

Cho, Kyunghyun, van Merriënboer, Bart, Gulcehre, Caglar, Bahdanau, Dzmitry, Bougares, Fethi, Schwenk, Holger, and Bengio, Yoshua. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *EMNLP*, 2014. ISSN 09205691. doi: 10.3115/v1/D14-1179. URL <http://arxiv.org/abs/1406.1078>.

Eriguchi, Akiko, Hashimoto, Kazuma, and Tsuruoka, Yoshimasa. Tree-to-Sequence Attentional Neural Machine Translation. *ACL*, pp. 3–7, 2016. doi: 10.18653/v1/P16-1078. URL <http://arxiv.org/abs/1603.06075>.

Karpathy, Andrej, Johnson, Justin, and Fei-Fei, Li. Visualizing and Understanding Recurrent Networks. *International Conference on Learning Representations*, pp. 1–12, 2015. ISSN 978-3-319-10589-5. doi: 10.1007/978-3-319-10590-1_53. URL <http://arxiv.org/abs/1506.02078>.

Lopez, Adam. Statistical machine translation. *ACM Comput. Surv.*, 40(3):8:1–8:49, August 2008. ISSN 0360-0300. doi: 10.1145/1380584.1380586. URL <http://doi.acm.org/10.1145/1380584.1380586>.

Luong, Minh-Thang, Pham, Hieu, and Manning, Christopher D. Effective Approaches to Attention-based Neural Machine Translation. *EMNLP*, 2015. ISSN 10495258. doi: 10.18653/v1/D15-1166. URL <http://arxiv.org/abs/1508.04025>.

Nguyen, Anh Tuan, Nguyen, Tung Thanh, and Nguyen, Tien N. Lexical statistical machine translation for language migration. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, pp. 651, 2013. doi: 10.1145/2491411.2494584. URL <http://dl.acm.org/citation.cfm?doid=2491411.2494584>.

Nguyen, Anh Tuan, Nguyen, Tien N. Tung Thanh, and Nguyen, Tien N. Divide-and-conquer approach for multi-phase statistical migration for source code. *Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pp. 585–596, 2016a. doi: 10.1109/ASE.2015.74.

Nguyen, Trong Duc, Nguyen, Anh Tuan, and Nguyen, Tien N. Mapping API elements for code migration with vector representations. *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*, pp. 756–758, 2016b. ISSN 02705257. doi: 10.1145/2889160.2892661. URL <http://dl.acm.org/citation.cfm?doid=2889160.2892661>.

Peng, Hao, Mou, Lili, Li, Ge, Liu, Yuxuan, Zhang, Lu, and Jin, Zhi. Building program vector representations for deep learning. *KSEM*, 9403:547–553, 2015. ISSN 16113349. doi: 10.1007/978-3-319-25159-2_49.

Srivastava, Nitish, Hinton, Geoffrey E., Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. ISSN 15337928. doi: 10.1214/12-AOS1000.

Tai, Kai Sheng, Socher, Richard, and Manning, Christopher D. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *ACL*, 2015. ISSN 9781941643723. doi: 10.1515/popets-2015-0023. URL <http://arxiv.org/abs/1503.00075>.

Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N., Kaiser, Łukasz, and Polosukhin, Illia. Attention Is All You Need. *NIPS*, 2017. URL <http://arxiv.org/abs/1706.03762>.

Yin, Pengcheng and Neubig, Graham. A syntactic neural model for general-purpose code generation. *CoRR*, abs/1704.01696, 2017. URL <http://arxiv.org/abs/1704.01696>.