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

Anonymous Authors[1]

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 desirable for a task without regard to whether the chosen language itself 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 or task 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 process of development.

Programming languages are similar to natural languages in many ways. Natural language translation involves transforming a sequence of words in one language to a sequence in another. Methods such as sequence-to-sequence translation, which map input sequences to output sequences, have achieved great performance for this task (Bahdanau et al., 2014; Cho et al., 2014; Eriguchi et al., 2016; He et al., 2016; Vaswani et al., 2017). While similar to natural languages, programming languages are remarkably different in how they are structured, making it harder to use the same tools for translation. For instance, the RNN-based sequence generator, which can generate words in a natural language easily, finds it difficult to generate syntactically correct programs when the lengths grow large (Karpathy et al., 2015).

Techniques for program language translation have generally been variants of statistical machine translation (SMT) (Lopez, 2008), the process of learning to model the probability distribution of phrases of text in one language given phrases in the other language and then using that distribution to generate a probable translation. Nguyen applied traditional SMT methods from natural language processing to the task but found that these methods produced many syntactically incorrect programs (Nguyen et al., 2013). Following up on this work, Nguyen et al. found that SMT methods could be improved by incorporating knowledge of program syntax (2016a). They also saw success matching program elements in different programming languages by learning Word-to-Vec encodings of different tokens based on their usage in their surrounding context and pairing tokens with similar representations (2016b).

Recently, similar to the use of neural networks in natural language processing 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 the challenge of programming language translation (Chen et al., 2018). The novel tree-to-tree encoder/decoder model proposed in Chen et al.'s work took advantage of the inherent tree structure of programs and performed better than traditional sequence-to-sequence models for the task of program translation. The tree-to-tree model improved on previous state-of the art program translation approaches by margins of 20 points for real-world translation projects.

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

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

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. Section 3 presents the framework for our work. Section 4 describes our experiments and presents our results. Lastly, Section 5 concludes and mentions possible directions for future work.

2. Background

Our model is heavily inspired by the tree-to-tree encoder/decoder model introduced by Chen et al. Their tree-to-tree model employs a tree LSTM (Tai et al., 2015) to encode the source tree into an embedding (Chen et al., 2018). First, the input program tree is binarized using the Left-Child Right-Sibling representation. The input tree is then encoded by an LSTM beginning at the leaves of the tree.

Each node N in the tree has a token t_s and up to two children; a left child N_L and a right child N_R. If the children maintain the LSTM states (h_L, c_L) and (h_R, c_R), then the LSTM state (h, c) for N is computed as:

$$(h, c) = LSTM(([h_L : h_R], [c_L, c_R]), t_s). \tag{1}$$

The hidden state and cell state of any missing children are represented as vectors of zeros.

The decoder then generates the target tree by first copying the LSTM state into the root node of the target tree and inserting it into a queue of nodes to be expanded. As long as there are nodes to be expanded, one is popped out and an attention mechanism is applied to determine what nodes in the input tree are most relevant (Chen et al., 2018). The attention mechanism is based on computing a dot product of a representation of the hidden state with all the encoded representations from the input tree (Luong et al., 2015) and produces a context vector e_s. From there, the hidden state and the context vector that arise are then used to determine the probabilities of the next token as shown in Equations (2) and (3).

$$e_t = \mathbf{tanh}(W_1[e_s; h]) \tag{2}$$

$$t_t = \mathbf{argmax\ softmax}(We_t) \tag{3}$$

W and W_1 are trainable weigh matrices. Each token t_t is a non-terminal, terminal, or <EOS> token. If a node's token is not <EOS>, the decoder will generate two children for the expanding node.

Chen et al. then generate the hidden state and cell state for each of the node's children with another set of LSTMs. To allow for different states to be produced for each of the node's children, they have a set of LSTMs, $LSTM_1 \dots LSTM_m$, where m is the maximum number of children a node can have (in their case two, since output trees were binarized as well). Hidden and cell states for the i^{th} child of N are generated from its hidden and cell states (h, c) as follows:

$$(h_i, c_i) = LSTM_i((h, c), [Bt_t; e_t]) \tag{4}$$

where B is an embedding matrix. To help the LSTM incorporate information from a node's attention when generating its children, the model uses parent attention feeding — concatenating the embedding representation of the parent's value with its attention vector before feeding the two into the LSTM. The new child nodes are then pushed into the queue of nodes to be expanded. The target tree generation process stops when the queue is empty.

Recent work by Yin and Neubig (2017) built on previous language translation and semantic parsing techniques by developing a grammar-based neural architecture to take into account the target language syntax. Their work was able to generate complex Python programs from natural language descriptions, outperforming previous work in this field. To do so, they created a neural network that converts a natural language statement into a syntactically correct AST for the target language. This network, however, generates nodes in the AST as a series of sequential instructions to extend or terminate a tree branch.

Work by Chen et al. comparing the performance of a tree-to-tree model with a tree-to-sequence model suggests that generating the tree directly could yield higher accuracy than generating the tree as a sequence (Chen et al., 2018).

In this paper, we apply the concept of a grammar model to the task of program language translation to leverage the existing grammar rules of a language to enhance the translation accuracy. The benefits of using grammar rules are that they generate only syntactically valid programs and they eliminate the redundancy of the end of tree token, thus increasing training speed.

3. Grammar-Based Tree-to-Tree model

We implement a tree-to-tree encoder/decoder model patterned off (Chen et al., 2018). The tree encoder is almost identical to the one described in the paper except that our model does not use dropout (Srivastava et al., 2014) as

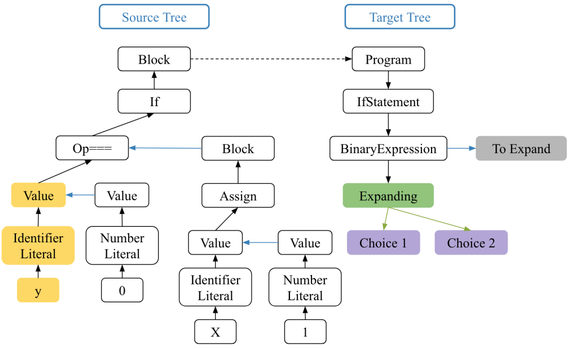

Figure 1. This diagram, adapted from Chen et al., shows a target tree being generated by a binarized input tree. When a node is expanded (green block), it can only generate children from grammatically correct choices (purple blocks) and add them to the queue of unexpanded nodes (gray). Our attention mechanism lets the decoder focus on relevant nodes in the input (yellow blocks).

we did not find using dropout improved training accuracy. However, we modify their tree decoder to make use of the grammar of the target language when generating nodes. Unlike in the paper 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 nodes that can be generated by that node at each child index. A category is a unique set of valid children of a parent. For instance, in the language FOR (described further in the Experiments section), the node <PLUS> can only generate tokens within the $Expression$ category: (<PLUS>, <MINUS>, <VAR>, or <CONST>). Each category k has an associated learnable weight matrix $\boldsymbol{W}_k$.

To generate each child, we create a node embedding e_t computed the same way as the tree decoder made 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:

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

Note that as the number of tokens in class k is typically 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 generates nodes from the root of the tree down to the leaves. Since the program's grammar does not allow any tokens to be generated from terminal tokens, branches of the tree end automatically when a terminal character 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 one of the tasks described by Chen et al. The task 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 10,000 validation and 10,000 test 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 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. Each model was trained 5 times over half a million examples. Program accuracy was measured on the test set by counting the percentage of perfectly syntactically correct translated programs.

Results are summarized in Table 2. The grammar-based model achieved an average XXX% accuracy, outperforming our reimplementation of the tree-to-tree model. The reimplemented tree-to-tree model performed worse than the results reported by (Chen et al., 2018), whereas our reimplementation of their tree-to-sequence model achieved comparable accuracy. 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

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

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).

Model	μ Accuracy	σ Accuracy
Grammar Tree2Tree	**999**	
Reimplemented Tree2Tree	888	
Chen et al. Tree2Tree (Easy)	99.76%	Unknown
Chen et al. Tree2Tree (Hard)	97.50%	Unknown
Reimplemented Tree2Seq	666	
Chen et al. Tree2Seq (Easy)	98.36%	Unknown
Chen et al. Tree2Tree (Hard)	87.84%	Unknown

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. While our results still point to the grammar decoder performing better than a typical tree decoder, in the future we will attempt to more closely replicate their dataset and see if we can obtain more reliable baselines. To make it easier for future research to evaluate on the same task, we release our dataset 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-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 necessary training data 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 between them) or by finding real-world programs implemented in both programs (which may be difficult to collect and can introduce noise to the dataset due to differences in program implementations).

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

Finally, our model currently caps the number of variable names and literals at a fixed number determined before training. Consequently, if a dataset contains even one training program with an exceptionally high number of variables or literals, we would need to support generation of all of those tokens, which slows down training time. Conversely, if a program in the test set has more variables or literals than our model supports, there is no way to translate it correctly. Future work could explore alternative ways to generate 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 different parts of the program. We could also integrate a language model into our decoder to help with translating unusual expressions never 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. 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. 2018. URL http://arxiv.org/abs/1802.03691.

Cho, Kyunghyun, van Merrienboer, Bart, Gulcehre, Caglar, Bahdanau, Dzmitry, Bougares, Fethi, Schwenk, Holger, and Bengio, Yoshua. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. 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. pp. 3–7, 2016. doi: 10.18653/v1/P16-1078. URL http://arxiv.org/abs/1603.06075.

He, Di, Xia, Yingce, Qin, Tao, Wang, Liwei, Yu, Nenghai, Liu, Tieyan, and Ma, Wei-Ying. Dual learning for machine translation. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 29*, pp. 820–828. Curran Associates, Inc., 2016. URL http://papers.nips.cc/paper/6469-dual-learning-for-machine-\translation.pdf.

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. 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. Tung Thanh. 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. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 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. 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.