IJARSCT

ISSN (Online) 2581-9429

International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)

Impact Factor: 6.252

Volume 2, Issue 2, June 2022

Unsupervised Translation for Programming Languages

Mr. Arunkumar Joshi[1] and Ms. Priya I Doddamani[2]

Assistant Professor, Department of Computer Science and Engineering[1]

Final Year Student, Department of Computer Science and Engineering[2]

Smt Kamala and Sri Venkappa M. Agadi College of Engineering & Technology, Laxmeshwar, Karnataka, India

Abstract: *A transcompiler, also known as source-to-source translator, is a system that converts source code from a high-level programming language (such as C++ or Python) to another. Transcompilers are primarily used for interoperability, and to port codebases written in an obsolete or deprecated language (e.g. COBOL, Python 2) to a modern one. They typically rely on handcrafted rewrite rules, applied to the source code abstract syntax tree. Unfortunately, the resulting translations often lack readability, fail to respect the target language conventions, and require manual modifications in order to work properly. The overall translation process is time consuming and requires expertise in both the source and target languages, making code-translation projects expensive. Although neural models significantly outperform their rule-based counterparts in the context of natural language translation, their applications to transcompilation have been limited due to the scarcity of parallel data in this domain. In this paper, we propose to leverage recent approaches in unsupervised machine translation to train a fully unsupervised neural transcompiler. We train our model on source code from open source GitHub projects, and show that it can translate functions between C++, Java, and Python with high accuracy. Our method relies exclusively on monolingual source code, requires no expertise in the source or target languages, and can easily be generalized to other programming languages. We also build and release a test set composed of 852 parallel functions, along with unit tests to check the correctness of translations. We show that our model outperforms rule-based commercial baselines by a significant margin.*

Keywords: Transcompiler

I. INTRODUCTION

Transcompiler, transpiler, or source-to-source compiler, is a translator which converts between programming languages that operate at a similar level of abstraction. Transcompilers differ from traditional compilers that translate source code from a high-level to a lower-level programming language (e.g. assembly language) to create an executable. Initially, transcompilers were developed to port source code between different platforms (e.g. convert source code designed for the Intel 8080 processor to make it compatible with the Intel 8086). More recently, new languages have been developed (e.g. CoffeeScript, TypeScript, Dart, Haxe) along with dedicated transcompilers that convert them into a popular or omnipresent language (e.g. JavaScript). These new languages address some shortcomings of the target language by providing new features such as list comprehension (CoffeeScript), object-oriented programming and type checking (TypeScript), while detecting errors and providing optimizations. Unlike traditional programming languages, these new languages are designed to be translated with a perfect accuracy (i.e. the compiled language does not require manual adjustments to work properly). In this paper, we are more interested in the traditional type of transcompilers, where typical use cases are to translate an existing codebase written in an obsolete or deprecated language (e.g. COBOL, Python 2) to a recent one, or to integrate code written in a different language to an existing codebase.

Translating source code from one Turing-complete language to another is always possible in theory. Unfortunately, building a translator is difficult in practice: different languages can have a different syntax and rely on different platform APIs and standard-library functions. Currently, the majority of transcompilation tools are rule-based; they essentially tokenize the input source code and convert it into an Abstract Syntax Tree (AST) on which they apply handcrafted rewrite rules. Creating them requires a lot of time, and advanced knowledge in both the source and target languages. Moreover, translating from a dynamically-typed language (e.g. Python) to a statically-typed language (e.g. Java) requires to infer the variable types which is difficult (and not always possible) in itself.

The applications of neural machine translation (NMT) to programming languages have been limitedso far, mainly because of the lack of parallel resources available in this domain. In this paper, we propose to apply recent approaches in unsupervised machine translation, by leveraging large amount of monolingual source code from GitHub to train a model, TransCoder, to translate between three popular languages: C++, Java and Python. To evaluate our model, we create a test set of 852 parallel functions, along with associated unit tests. Although never provided with parallel data, the model manages to translate functions with a high accuracy, and to properly align functions from the standard library across the three languages, outperforming rule-based and commercial baselines by a significant margin. Our approach is simple, does not require any expertise in the source or target languages, and can easily be extended to most programming languages. Although not perfect, the model could help to reduce the amount of work and the level of expertise required to successfully translate a codebase. The main contributions of the paper are the following:

- We introduce a new approach to translate functions from a programming language to another, that is purely based on monolingual source code.
- We show that TransCoder successfully manages to grasp complex patterns specific to each language, and to translate them to other languages.
- We show that a fully unsupervised method can outperform commercial systems that leverage rule-based methods and advanced programming knowledge.
- We build and release a validation and a test set composed of 852 parallel functions in 3 languages, along with unit tests to evaluate the correctness of generated translations.
- We will make our code and pretrained models publicly available.

II. RELATED WORK

Source-to-source translation. Several studies have investigated the possibility to translate programming languages with machine translation. For instance, Nguyen et al. [36] trained a Phrase-Based Statistical Machine Translation (PBSMT) model, Moses [27], on a Java-C# parallel corpus. They created their dataset using the implementations of two open source projects, Lucene and db4o, developed in Java and ported to C#. Similarly, Karaivanov et al. [22] developed a tool to mine parallel datasets from ported open source projects. Aggarwal et al. [1] trained Moses on a Python 2 to Python 3 parallel corpus created with 2to3, a Python library 2 developed to port Python 2 code to Python 3. Chen et al. [12] used the Java-C# dataset of Nguyen et al. [36] to translate code with tree-to-tree neural networks. They also use a transcompiler to create a parallel dataset CoffeeScript-Javascript. Unfortunately, all these approaches are supervised, and rely either on the existence of open source projects available in multiple languages, or on existing transcompilers, to create parallel data. Moreover, they essentially rely on BLEU score [38] to evaluate their translations [1, 10, 22, 36], which is not a reliable metric, as a generation can be a valid translation while being very different from the reference.

Translating from source code. Other studies have investigated the use of machine translation from source code. For instance, Oda et al. [37] trained a PBSMT model to generate pseudo-code. To create a training set, they hired programmers to write the pseudo-code of existing Python functions. Barone and Sennrich [10] built a corpus of Python functions with their docstrings from open source GitHub repositories. They showed that a neural machine translation model could be used to map functions to their associated docstrings, and vice versa. Similarly, Hu et al. [21] proposed a neural approach, DeepCom, to automatically generate code comments for Java methods.

Other applications. Another line of work studied the applications of neural networks to code suggestion [2, 11, 34], or error detection [13, 18, 47]. Recent approaches have also investigated the use of neural approaches for code decompilation [16, 24]. For instance, Katz et al. [23] propose a sequence-to-sequence model to predict the C code of binary programs. A common issue with standard seq2seq models, is that the generated functions are not guaranteed to compile, and even to be syntactically correct. To address this issue, several approaches proposed to use additional constraints on the decoder, to ensure that the generated functions respect the syntax of the target language [3, 4, 5, 40, 48]. Recently, Feng et al. [15] introduced Codebert, a transformer pretrained with a BERT-like objective [14] on open source GitHub repositories. They showed that pretraining improves the performance on several downstream tasks such as code documentation generation and code completion.

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
Volume 2, Issue 2, June 2022
Impact Factor: 6.252

Unsupervised Machine Translation. The quality of NMT systems highly depends on the quality of the available parallel data. However, for the majority of languages, parallel resources are rare or nonexistent. Since creating parallel corpora for training is not realistic (creating a small parallel corpus for evaluation is already challenging [19]), some approaches have investigated the use of monolingual data to improve existing machine translation systems [17, 20, 41, 49]. More recently, several methods were proposed to train a machine translation system exclusively from monolingual corpora, using either neural models [30, 8] and statistical models [32, 7]. We describe now some of these methods and how they can be instantiated in the setting of unsupervised transcompilation.

III. MODEL

For TransCoder, we consider a sequence-to-sequence (seq2seq) model with attention [44, 9], composed of an encoder and a decoder with a transformer architecture [45]. We use a single shared model for all programming languages. We train it using the three principles of unsupervised machine translation identified in Lample et al. [32], namely initialization, language modeling, and back-translation. In this section, we summarize these principles and detail how we instantiate them to translate programming languages. An illustration of our approach is given in Figure 1.

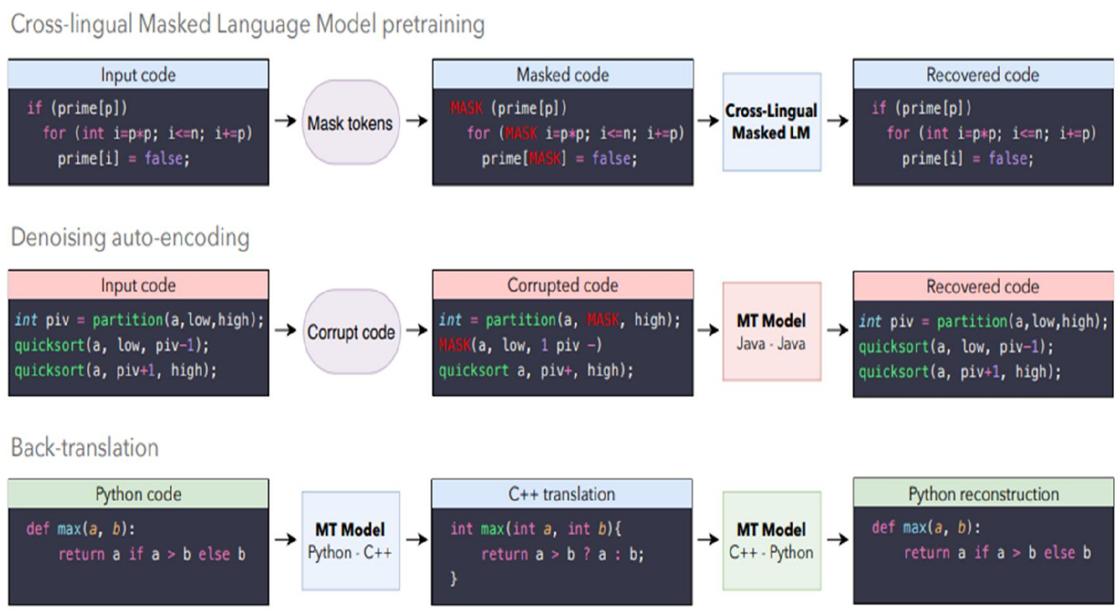

Figure 1: Illustration of the three principles of unsupervised machine translation used approach.

3.1 Cross Programming Language Model pretraining

Pretraining is a key ingredient of unsupervised machine translation Lample et al. [32]. It ensures that sequences with a similar meaning are mapped to the same latent representation, regardless of their languages. Originally, pretraining was done by initializing the model with cross-lingual word representations [30, 8]. In the context of unsupervised English-French translation, the embedding of the word "cat" will be close to the embedding of its French translation "chat". Cross-lingual word embeddings can be obtained by training monolingual word embeddings and aligning them in an unsupervised manner [31, 6]. Subsequent work showed that pretraining the entire model (and not only word representations) in a cross-lingual way could lead to significant improvements in unsupervised machine translation [29, 33, 43]. In particular, we follow the pretraining strategy of Lample and Conneau [29], where a Cross-lingual Language Model (XLM) is pretrained with a masked language modeling objective [14] on monolingual source code datasets.

3.2 Denoising auto-encoding

We initialize the encoder and decoder of the seq2seq model with the XLM model pretrained in Section 3.1. The initialization is straightforward for the encoder, as it has the same architecture as the XLM model. The transformer decoder, however, has extra parameters related to the source attention mechanism [45]. Following Lample and Conneau [29], we initialize these parameters randomly. XLM pretraining allows the seq2seq model to generate high quality representations of input sequences. However, the decoder lacks the capacity to translate, as it has never been trained to decode a sequence based on a source representation. To address this issue, we train the model to encode and decode sequences with a Denoising Auto-Encoding (DAE) objective [46]. The DAE objective operates like a supervised machine translation algorithm, where the model is trained to predict a sequence of tokens given a corrupted version of that sequence. To corrupt a sequence, we use the same noise model as the one described in Lample et al. [30]. Namely, we randomly mask, remove and shuffle input tokens.

3.3 Back-translation

In practice, XLM pretraining and denoising auto-encoding alone are enough to generate translations. However, the quality of these translations tends to be low, as the model is never trained to do what it is expected to do at test time, i.e. to translate functions from one language to another. To address this issue, we use back-translation, which is one of the most effective methods to leverage monolingual data in a weakly-supervised scenario. Initially introduced to improve the performance of machine translation in the supervised setting [41], back-translation turned out to be an important componentof unsupervised machine translation [30, 32, 8]. In the unsupervised setting, a source-to-target model is coupled with a backward target-to-source model trained in parallel. The target-to-source model is used to translate target sequences into the source language, producing noisy source sequences corresponding to the ground truth target sequences. The source-to-target model is then trained in a weakly supervised manner to reconstruct the target sequences from the noisy source sequences generated by the target-to-source model, and vice versa. The two models are trained in parallel until convergence. An example of back-translation is illustrated in Figure 1.

IV. EXPERIMENTS

4.1 Training Details

We use a transformer with 6 layers, 8 attention heads, and set the dimensionality of the model to 1024. We use a single encoder and a single decoder for all programming languages. During XLM pretraining, we alternate between batches of C++, Java, and Python, composed of 32 sequences of source code of 512 tokens. At training time, we alternate between the denoising auto-encoding and back-translation objectives, and use batches of around 6000 tokens. We optimize TransCoder with the Adam optimizer [25], a learning rate of 10□4, and use the same learning rate scheduler as Vaswani et al. [45]. We implement our models in PyTorch [39] and train them on 32 V100 GPUs. We use float16 operations to speed up training and to reduce the memory usage of our models.

4.2 Training Data

We download the GitHub public dataset available on Google BigQuery4. It contains more than 2.8 million open source GitHub repositories. We filter projects whose license explicitly permits the re-distribution of parts of the project, and select the C++, Java, and Python files within those projects. Ideally, a transcompiler should be able to translate whole projects. In this work, we decide to translate at function level. Unlike files or classes, functions are short enough to fit into a single batch, and working at function level allows for a simpler evaluation of the model with unit tests (c.f. Section 4.4). We pretrain TransCoder on all source code available, and train the denoising auto-encoding and back-translation objectives on functions only. Please refer to Section A.3 and Table 3 in the appendix for more details on how the functions are extracted, and for statistics about our training set. We carry out an ablation study to determine whether it is better to keep or remove comments from source code. Keeping comments in the source code increases the number of anchor points across languages, which results in a better overall performance (c.f. Table 6 in the appendix). Therefore, we keep them in our final datasets and experiments.

4.3 Preprocessing

Recent approaches in multilingual natural language processing tend to use a common tokenizer [28], and a shared vocabulary for all languages. This reduces the overall vocabulary size, and maximizes the token overlap between languages, improving the cross-linguality of the model [14, 29]. In our case, a universal tokenizer would be suboptimal,

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
Impact Factor: 6.252
Volume 2, Issue 2, June 2022

as different languages use different patterns and keywords. The logical operators && and || exist in C++ where they should be tokenized as a single token, but not in Python. The indentations are critical in Python as they define the code structure, but have no meaning in languages like C++ or Java. We use the javalang5 tokenizer for Java, the tokenizer of the standard library for Python6, and the clang7 tokenizer for C++. These tokenizers ensure that meaningless modifications in the code (e.g. adding extra new lines or spaces) do not have any impact on the tokenized sequence. An example of tokenized code is given in Figure 3 in the appendix. We learn BPE codes [42] on extracted tokens, and split tokens into subword units. The BPE codes are learned with fastBPE8 on the concatenation of tokenized C++, Java, and Python files.

4.4 Evaluation

GeeksforGeeks is an online platform9 with computer science and programming articles. It gathers many coding problems and presents solutions in several programming languages. From these solutions, we extract a set of parallel functions in C++, Java, and Python, to create our validation and test sets. These functions not only return the same output, but also compute the result with similar algorithm. In Figure 4 in the appendix, we show an example of C++-Java-Python parallel function that determines whether an integer represented by a string is divisible by 13.

The majority of studies in source code translation use the BLEU score to evaluate the quality of generated functions [1, 10, 22, 36], or other metrics based on the relative overlap between the tokens in the translation and in the reference. A simple metric is to compute the reference match, i.e. the percentage of translations that perfectly match the ground truth reference [12]. A limitation of these metrics is that they do not take into account the syntactic correctness of the generations. Two programs with small syntactic discrepancies will have a high BLEU score while they could lead to very different compilation and computation outputs. Conversely, semantically equivalent programs with different implementations will have low BLEU scores. Instead, we introduce a new metric, the computational accuracy, that evaluates whether the hypothesis function generates the same outputs as the reference when given the same inputs. We consider that the hypothesis is correct if it gives the same output as the reference for every input. Section B and Table 4 in the appendix present more details on how we create these unit tests, and give statistics about our validation and test sets. At inference, TransCoder can generate multiple translations using beam search decoding [26]. In machine translation, the considered hypotheses are typically the ones with the highest log-probabilities in the beam. In our case, we have access to unit tests to verify the correctness of the generated hypotheses, so we report two sets of results for our computational accuracy metric: Beam N, the percentage of functions with at least one correct translation in the beam, and Beam N - Top 1 the percentage of functions where the hypothesis in the beam with the highest log-probability is a correct translation. We select our best model using greedy decoding (Beam 1) for speed efficiency.

4.5 Results

We report the results on our test set in Table 1, using greedy decoding (beam size 1), for the three metrics presented in Section 4.4. In Table 2, we report our results with beam search decoding, and compare TransCoder to existing baselines. We give an example of unsupervised translation from Python to C++ in Figure 2.

Evaluation metric differences. In Table 1, we observe that a very large fraction of translations differ from the reference, and are considered as invalid by the reference match metric although they successfully pass the unit tests. For instance, when translating from C++ to Java, only 3.1% of the generations are strictly identical to the ground truth reference, although 60.9% of them return the expected outputs. Moreover, the performance in terms of BLEU is relatively flat and does not correlate well with the computational accuracy. These results highlight the issues with the traditional reference match and BLEU metrics commonly used in the field.

	C++ → Java	C++ → Python	Java → C++	Java → Python	Python → C++	Python → Java
Reference Match	3.1	6.7	24.7	3.7	4.9	0.8
BLEU	85.4	70.1	97.0	68.1	65.4	64.6
Computational Accuracy	60.9	44.5	80.9	35.0	32.2	24.7

Table 1: Results of TransCoder on our test set with greedy decoding.

DOI: 10.48175/568

Beam search decoding. In Table 2, we study the impact of beam search, either by considering all hypotheses in the beam that pass the unit tests (Beam N) or by only considering the ones with the highest log-probabilities (Beam N - Top 1). Compared to greedy decoding (Beam 1), beam search significantly improves the computational accuracy, by up to 33.7% in Java ! Python with Beam. When the model only returns the hypothesis with the highest log-probability, the performance drops, indicating that TransCoder often finds a valid translation, although it sometimes gives a higher log-probability to incorrect hypotheses. More generally, beam search allows minor variations of the translations which can make the unit tests succeed, such as changing the return or variable types in Java and C++, or fixing small errors such as the use of / instead of the // operator in Python. More examples of errors corrected by beam search are presented in Figure 9 in the appendix. In a real use-case, checking whether the generated functions are syntactically correct and compile, or creating unit tests from the input function would be better approaches than comparing log-probabilities in order to select an hypothesis from the beam. Table 5 in the appendix shows that many failures.

	C++ → Java	C++ → Python	Java → C++	Java → Python	Python → C++	Python → Java
Baselines	61.0	-	-	38.3	-	-
TransCoder Beam 1	60.9	44.5	80.9	35.0	32.2	24.7
TransCoder Beam 5	70.7	58.3	86.9	60.0	44.4	44.3
TransCoder Beam 10	73.4	62.0	89.3	64.4	49.6	51.1
TransCoder Beam 10 - Top 1	65.1	46.9	79.8	49.0	32.4	36.6
TransCoder Beam 25	74.8	67.2	91.6	68.7	57.3	56.1

Table 2: Computational accuracy with beam search decoding and comparison to baselines.

Comparison to existing baselines. We compare TransCoder with two existing approaches: j2py10, a framework that translates from Java to Python, and a commercial solution from Tangible Software Solutions11, that translates from C++ to Java. Both systems rely on rewrite rules manually built using expert knowledge. The latter handles the conversion of many elements, including core types, arrays, some collections (Vectors and Maps), and lambdas. In Table 2, we observe that TransCoder significantly outperforms both baselines in terms of computational accuracy, with 74.8% and 68.7% in the C++ ! Java and Java ! Python directions, compared to 61% and 38.3% for the baselines. TransCoder particularly shines when translating functions from the standard library. In rule-based transcompilers, rewrite rules need to be manually encoded for each standard library function, while TransCoder learns them in an unsupervised way. In Figure 10 of the appendix, we present several examples where TransCoder succeeds, while the baselines fail to generate correct translations.

V. CONCLUSION

In this paper, we show that approaches of unsupervised machine translation can be applied to source code to create a transcompiler in a fully unsupervised way. TransCoder can easily be generalized to any programming language, does not require any expert knowledge, and outperforms commercial solutions by a large margin. Our results suggest that a lot of mistakes made by the model could easily be fixed by adding simple constraints to the decoder to ensure that the generated functions are syntactically correct, or by using dedicated architectures [12]. Leveraging the compiler output or other approaches such as iterative error correction [16] could also boost the performance.

REFERENCES

[1] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. Using machine translation for converting python 2 to python 3 code. Technical report, PeerJ PrePrints, 2015.

[2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 281–293, 2014.

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. ICLR, 2019.

[4] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models for any-code generation. arXiv preprint arXiv:1910.00577, 2019.
[5] Matthew Amodio, Swarat Chaudhuri, and Thomas Reps. Neural attribute machines for program generation. arXiv preprint arXiv:1705.09231, 2017.
[6] Mikel Artetxe, Gorka Labaka, and Eneko Agirre. Learning bilingual word embeddings with (almost) no bilingual data. In Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), volume 1, pages 451–462, 2017.
[7] Mikel Artetxe, Gorka Labaka, and Eneko Agirre. Unsupervised statistical machine translation. arXiv preprint arXiv:1809.01272, 2018.
[8] Mikel Artetxe, Gorka Labaka, Eneko Agirre, and Kyunghyun Cho. Unsupervised neural machine translation. In International Conference on Learning Representations (ICLR), 2018.
[9] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473, 2014.
[10] Antonio Valerio Miceli Barone and Rico Sennrich. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. arXiv preprint arXiv:1707.02275, 2017.
[11] Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. Learning python code suggestion with a sparse pointer network. arXiv preprint arXiv:1611.08307, 2016.
[12] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In Advances in neural information processing systems, pages 2547–2557, 2018.
[13] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. IEEE Transactions on Software Engineering, 2019.
[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. CoRR, abs/1810.04805, 2018.
[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155, 2020.
[16] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. In Advances in Neural
Information Processing Systems, pages 3703–3714, 2019.
[17] Caglar Gulcehre, Orhan Firat, Kelvin Xu, Kyunghyun Cho, Loic Barrault, Huei-Chi Lin, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. On using monolingual corpora in neural machine translation. arXiv preprint arXiv:1503.03535, 2015.
[18] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In Thirty-First AAAI Conference on Artificial Intelligence, 2017.
[19] Francisco Guzmán, Peng-Jen Chen, Myle Ott, Juan Pino, Guillaume Lample, Philipp Koehn, Vishrav Chaudhary, and Marc'Aurelio Ranzato. Two new evaluation datasets for low-resource machine translation: Nepali-english and sinhala-english. arXiv preprint arXiv:1902.01382, 2019.
[20] Di He, Yingce Xia, Tao Qin, Liwei Wang, Nenghai Yu, Tie-Yan Liu, and Wei-Ying Ma. Dual learning for machine translation. In Advances in neural information processing systems, pages 820–828, 2016.
[21] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In Proceedings of the 26th Conference on Program Comprehension, pages 200–210, 2018.
[22] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical translation of programming languages. In Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, pages 173–184, 2014.
[23] Deborah S Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 346–356. IEEE, 2018.

[24] Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. Towards neural decompilation. arXiv preprint arXiv:1905.08325, 2019.
[25] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.
[26] Philipp Koehn. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In Conference of the Association for Machine Translation in the Americas, pages 115–124. Springer, 2004.
[27] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Ondrej Bojar Chris Dyer, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In Annual Meeting of the Association for Computational Linguistics (ACL), demo session, 2007.
[28] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. arXiv preprint arXiv:1808.06226, 2018.
[29] Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. arXiv preprint arXiv:1901.07291, 2019.
[30] Guillaume Lample, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. Unsupervised machine translation using monolingual corpora only. ICLR, 2018.
[31] Guillaume Lample, Alexis Conneau, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Word translation without parallel data. In ICLR, 2018.
[32] Guillaume Lample, Myle Ott, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. Phrase-based & neural unsupervised machine translation. In EMNLP, 2018.
[33] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. arXiv preprint arXiv:1910.13461, 2019.
[34] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. Code completion with neural attention and pointer networks. IJCAI, 2018.
[35] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. Journal of machine learning research, 9(Nov):2579–2605, 2008.
[36] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Lexical statistical machine translation for language migration. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pages 651–654, 2013.
[37] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (t). In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 574–584. IEEE, 2015.
[38] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting on association for computational linguistics, pages 311–318. Association for Computational Linguistics, 2002.
[39] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. NIPS 2017 Autodiff Workshop, 2017.
[40] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. arXiv preprint arXiv:1704.07535, 2017.
[41] Rico Sennrich, Barry Haddow, and Alexandra Birch. Improving neural machine translation models with monolingual data. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, pages 86–96, 2015.
[42] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, pages 1715–1725, 2015.
[43] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mass: Masked sequence to sequence pre-training for language generation. In International Conference on Machine Learning, pages 5926–5936, 2019.
[44] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Advances in neural information processing systems, pages 3104–3112, 2014.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Advances in neural information processing systems, pages 5998–6008, 2017.

[46] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In Proceedings of the 25th international conference on Machine learning, pages 1096–1103, 2008.

[47] Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embedding for program repair. arXiv preprint arXiv:1711.07163, 2017.

[48] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. arXiv preprint arXiv:1704.01696, 2017.

[49] Hao Zheng, Yong Cheng, and Yang Liu. Maximum expected likelihood estimation for zeroresource neural machine translation. In IJCAI, 2017.