Replicating Parser Behavior using Neural Machine Translation

Carol V. Alexandru, Sebastiano Panichella and Harald C. Gall
Software Evolution and Architecture Lab
University of Zurich, Switzerland
{alexandru,panichella,gall}@ifi.uzh.ch

***Abstract*—More than other machine learning techniques, neural networks have been shown to excel at tasks where humans traditionally outperform computers: recognizing objects in images, distinguishing spoken words from background noise or playing "Go". These are hard problems, where hand-crafting solutions is rarely feasible due to their inherent complexity. Higher level program comprehension is not dissimilar in nature: while a compiler or program analysis tool can extract certain facts from (correctly written) code, it has no intrinsic 'understanding' of the data and for the majority of real-world problems, a human developer is needed - for example to find and fix a bug or to summarize the bahavior of a method. We perform a pilot study to determine the suitability of neural machine translation (NMT) for processing plain-text source code. We find that, on one hand, NMT is too fragile to accurately tokenize code, while on the other hand, it can precisely recognize different types of tokens and make accurate guesses regarding their relative position in the local syntax tree. Our results suggest that NMT may be exploited for annotating and enriching out-of-context code snippets to support automated tooling for code comprehension problems. We also identify several challenges in applying neural networks to learning from source code and determine key differences between the application of existing neural network models to source code instead of natural language.**

I. Introduction

Deep learning has become an ever more prominent tool for solving *hard* problems in computer science [1] and other disciplines; particularly those problems, where humans typically outperform their artificial counterparts. For example, people can easily spot and identify multiple objects in an image, even if they are occluded, badly lit or otherwise hard to discern from the surrounding imagery. Manually writing a program with the same purpose is extremely hard and has kept researchers busy for decades, whereas machine learning, and especially deep learning, has provided several breakthroughs in image recognition technology within the past decade alone [2].

In a way, program comprehension (PC) falls into the same problem category: while an interpreter can read a program line by line, link different sources and execute arbitrarily complex pieces of (correctly written) code, it has no inheret *understanding* of the program - it is just a 'dumb' automaton, handcrafted by a human designer. To actually comprehend what is going on in a given piece of code, to fix a bug or to change some specific program behavior, a human developer is most often required, unless the bug or change falls into a narrow category of problems that have been solved programmatically. From this point of view, PC is another hard problem where humans outperform machines and where research is struggling to yield effective automated means of answering relatively simple questions: 'Why does this method crash given some specific input?', 'Is this code thread safe?', 'Why does this assertion fail?'. Questions like these might be answered by an experienced developer just by reading the source code, but a program that can answer all three questions necessitates an immensely complex, yet narrowly targeted toolchain.

As such, we argue that for many PC tasks, a traditional model of the source code (as a compiler builds it) is not enough. We believe that in the long run, deep learning could be a key towards automating PC. The goal, of course, is to create automated tools that can aid developers more effectively in understanding and modifying source code. However, Rome was not built in a day, and as a first step in the right direction, we ask: *can neural networks learn the first lesson of mastering a new programming language, namely recognizing its syntactic components?* After all, even for more complex comprehension tasks (i.e., how two separate pieces of code interact), any deep learning model would first need to learn an internal model of the source code itself in order to know "what is what".

Hence, we explore the ability of neural machine translation (NMT) to interpret plain-text source code directly. Our contributions are: (i) a fast and simple tool for gathering NMT training data from GitHub[1], (ii) an examination of the suitability of NMT to *tokenize* source code and *annotate* tokens with typing and location metadata, (iii) an outline for future avenues of research applying NMT towards PC.

II. Related Work

Recurrent neural networks (RNN) have found widespread use in other fields, with natural language processing being most closely related to source code processing. Sequence-to-sequence translation, as outlined by Sutskever et al., has been shown to match or even outperform existing systems when translating sentences from one language to another [3]. Vinyals et al. discovered that neural machine translation can also parse natural language sentences into parse trees with similar performance as the Berkley Parser [4]. Both these examples show that a relatively simple RNN can mimmick more complex, human-engineered and state-of-the-art tooling behavior. These results in natural language processing inspire confidence that deep learning may also be effective for solving similar problems in source code processing. In fact, White et al. show that RNNs significantly outperform n-gram based models for predicting the next token in a sequence [1].

[1]The tool, ParseNN, is open source: https://bitbucket.org/sealuzh/parsenn

III. Approach

Parsers traditionally use a rule-based lexer to group source code characters. The resulting tokens are fed into a parser to construct a parse tree. We follow the same two-step process, however both translations are performed by an NMT model. Since parse trees can be much more deeply nested and since they are generally more complex than natural language parse trees, the result of the second step in this preliminary study is not a parse tree, but rather an annotation sequence identifying the type for each token, as well as its depth in the parse tree, indicating the relative location of different tokens.

A. Neural Machine Translation

Luong et al. describe NMT as a neural network modelling the conditional probability $p(y|x)$ of translating a source sequence $x_1, ..., x_n$ to a target sequence $y_1, ..., y_m$ [5]. The model consists of an *encoder*, that creates some numerical representation s for each source sequence, while a *decoder* generates one output word at a time. Thus, the conditional probability of translating any particular sentence is defined as:

$$\ln p(y|x) = \sum_{j=1}^{m} \ln p(y_j|y_1, ..., y_{j-1}, s)$$

In other words, the probability of any single word y_j being appended to the output depends both on the input and on any previously generated output words. Recurrent neural networks (RNN), consisting of Long Short-Term Memory (LSTM) [6] or gated recurrent units (GRU) [7], lend themselves naturally to model this problem because they make predictions based not only on input data, but also on *previous internal states*. While the basic sequential model works with input and output sequences consisting of single words, it is also possible to use sequences where each word has more than one feature [8].

To map words to their numerical representation, *vocabularies* V_x and V_y are created for the input and the output sequences. They contain a mapping from words to integers for the top-k most common words in each dataset. They also contain four control words, for the *start* and *end* of sentence, *padding* and *unknown* words (assigned to all uncommon words not present in the vocabulary).

To further improve predictions by an RNN, *attention-based models* compute *context vectors* to determine the relevant information in the source sequence for the next output word. The context vector may be computed over the entire input sequence (*global attention*), or only parts of it by attempting to align which words in the input sequence are most predictive for the next output word (*local attention*). Through *input feeding*, the previous context vector may be fed back into the model at each step, such that the 'attention' moves over the source sequence continuously. Finally, instead of only going forward through the sequences, a *bi-directional RNN* can average predictions going through the sequences both forward and in reverse, which can also improve predictions.

As with all neural networks, the training goal is to minimize the prediction error of the model by tweaking its internal weights using one of several optimization techniques. The prediction error, typically called *loss*, is defined as:

$$loss = -\frac{1}{N} \sum_{j=1}^{N} \ln p_{y_j}$$

Usually, *per-word perplexity* is used as the performance metric for sequence prediction models. Perplexity is defined as:

$$perplexity = e^{-\frac{1}{N} \sum_{j=1}^{N} \ln p_{y_j}} = e^{loss}$$

Metaphorically speaking, the perplexity x of a model indicates that it predicts the correct word as often as an 'x-sided' die. Thus, better models exhibit lower perplexity and with larger target vocabularies, increased preplexity is expected.

B. Training data

As we are training two separate models for the two translation steps, we need four kinds of data, as shown in table I. For the first translation step to tokenize source code, we avoid using the tokens themselves in the output sequence because many of them would be 'unknown' (e.g., class and method names are often unique in the dataset). Instead, we use a sequence of just three simple lexing instructions: (a) '0' $\rightarrow$ continue (or begin) the current token, (b) '1' $\rightarrow$ end the current token, (c) ' ' $\rightarrow$ skip the current character. For the second translation step, we *do* use tokens as words in the input sequence (where some may be "unknown"), but these words can still be annotated correctly thanks to the surrounding context - they are not necessarily informative.

The example provided in Table I shows the original input sequence (A1), the lexing instructions (A2) required to produce the given tokens (B1), as well as the corresponding annotations (B2). It shows that the tokens `public` and `;` are on the same level (a statement level element, most likely), while the other elements are located further down the tree. Some levels are noticably missing, which is because the original AST created by the parser creates many nodes not represented by a literal token. We address this problem in section IV.

Neural translation models in natural language normally operate at the sentence-level. As there are no 'sentences' in source code, we decided to simply split the input data on newlines. This has one particular disadvantage, namely that the resulting model cannot recognize multi-line tokens (such as multi-line strings). We discuss this problem, as well as possible alternatives in section IV. For now, when the parser we use to generate the training data encounters a multi-line token, the corresponding input and output sequences are thrown away and excluded from the training data.

C. Training Data Acquisition

To gather the training data, we first used the GitHub API to obtain the Git URLs of the top 1000 Java projects hosted on GitHub, ordered by the number of stars they received. We then wrote a tool to automatically apply the following process for each of the projects:

1) First, the Git repository is cloned.

TABLE I
EXEMPLARY TRAINING DATA

ID	Sequence	Seq. type	Example
A1	Plain text code	Characters	`public String s = "Hello, World";`
A2	Lexing instructions	Characters	`000001 000001 1 1 000000000000011`
B1	Tokens	Words	`[public] [String] [s] [=] ["Hello, World"] [;]`
B2	Annotations	Words|Integers	`[ClassOrInterfaceModifier|11] [ClassOrInterfaceType|13] [VariableDeclaratorId|14] [VariableDeclarator|13] [Literal|17] [FieldDeclaration|11]`

TABLE II
ONE SUCCESSFUL AND ONE FAILED TRANSLATION (ERRORS HIGHLIGHTED)

Successful translation	`List<Throwable> errors = TestHelper.trackPluginErrors();`
	`000110000000011 000001 1 00000000011000000000000000001111`
	`[ClassOrInterfaceType|14] [TypeArguments|15] [ClassOrInterfaceType|18] [TypeArguments|15] [VariableDeclaratorId|15] [VariableDeclarator|14] [Primary|19] [Expression|17] [Expression|17] [Expression|16] [Expression|16] [LocalVariableDeclarationStatement|11]`

Failed translation	`@Test(expected = NullPointerException.class)` `10001100000001 1 00000000000000000000000000000011`

2) The bare Git tree is traversed to obtain the Git blob IDs of all Java files present in the latest revision.
3) An ANTLR-generated lexer and parser are then used to parse each file and simultaneously extract all four data sequences, meanwhile ensuring that all the sequences are aligned correctly. Accidentally skipping any line in any of the sequences would result in a fatal misalignment of the training data. As we need each word (be it individual characters, tokens or annotations) to be separated by a blank space for further training, we replace any existing spaces within a word with an unassigned unicode character (0xFF00) and then concatenate the words using spaces when writing them to file.

By the end of this process, we had obtained four data files, all containing space-separated words which can directly be used by the translation model. The automated extraction process for 1000 Java projects took 4.3 hours. From this data, we eventually used 25 million sentences for training and a separate 2 million for validation.

D. Model Training and evaluation

We experimented with different frameworks (based on Tensorflow and Torch) as well as different hyperparameters for the models. We found OpenNMT [8] to be most suitable (in terms of speed, resource requirements and ease of use), as it is a mature, full-featured framework rather than a research prototype. All models were trained until no improvement in preplexity was made for 2 full training epochs, and using the following OpenNMT default parameters unless otherwise noted: 2 layers, 500 hidden LSTM units, input feeding enabled, batchsize: 64, dropout probability: 0.3 and a learning rate decay rate of 0.5 applied at the end of each epoch where perplexity did not improve. Combining both models, we built an annotation engine that translates plain-text source code to annotated token sequences. Table II shows examples for both successful and failed translations. The source and target vocabulary sizes are denoted as $V_{x^{size}}$ and $V_{y^{size}}$.

Tokenization. For translating plain-text source code ($V_{x^{size}}$: 2189) to lexing instructions ($V_{y^{size}}$: 7) we trained a bi-directional RNN operating on input and output sequences up to 100 words (characters) long. After training for 7 epochs, requiring 24 hours for each epoch, the resulting model exhibited a perplexity of **1.11**. Although this is very low, given that the target vocabulary is tiny, and the number of tokens per sequence quite large (a single line of code can contain dozens of characters), any single mistake leads to a faulty tokenization (see the second row in table II). A similar training session using a uni-directional RNN diverged in the second epoch.

Token annotation. For annotating tokens ($V_{x^{size}}$: 50004, $V_{y^{size}}$: 91), we trained both a uni-directional (5½ hours per epoch) and a bi-directional (7 hours per epoch) model for 11 epochs on sequences up to 50 words in length. Both models reached the same perplexity of **1.28**, although the uni-directional one learned more quickly in the beginning. This is a good result given the non-trivial target vocabulary. Given how we constructed our training data, we know that some predictions cannot accurately be made: some input sequences consist only of a single curly bracket ({}), and without additional context it is impossible to predict the scope and depth of such a token. Using a different code splitting strategy, as discussed in section IV, would certainly alleviate this issue and may improve the perplexity.

Conclusion. NMT is not very well suited for tokenizing source code, but it is highly capable of recognizing token types and their relative locations in an implied parse tree.

IV. FUTURE WORK

Our results suggest that NMT can effectively annotate tokenized source code given enough training data, but our strategy for tokenizing source code using lexing instructions is ineffective. It may be better to first use a naive tokenization to reduce the sequence length (e.g., splitting on non-alphanumerics) and to train a model for extracting proper tokens from that sequence. A direct translation from characters to tokens may also be attempted, given that *attention* may be enough to replace rare target-tokens (names of variables, method, etc.) with the correct characters from the input sequence.

A. Alternative input data separation.

We split our input data on newlines. This means that our training sequences are variable in length and that many sequences likely exhibited similar features. On one hand, this makes for a more regular data set, where the neural network can more easily learn certain patterns (e.g., `public` at the

beginnig of a line can easily be recognized as a modifier). On the other hand, this makes it more difficult for the neural network to correctly translate irregular code. There exist alternatives that may be worth exploring:

- Splitting the data such that the resulting token sequences have a fixed length. This implies that the input sequence is variable-length, because each token may entail a different number of characters. This would likely make the approach more robust to strangely formatted code, as there is no longer any assumption of where on a line a token occurs more often. On the other hand, more training data would be needed and the model would also need longer to train in order to perform well.
- Simlilarily, a fixed input length could be used, i.e., splitting the input every n characters. This has the drawback that characters at the beginning and end may not describe an entire token. For these characters, a special 'partial' token could be used in the target sequence.
- Splitting based on semantics: for example for Java, we may split the token sequences into class and method definitions, control statements (`for`, `while` etc.) and block statements. We suspect that this approach could slightly outperform our current approach, since there are fewer 'spurious' sequences, such as single closing brackets, but only experimentation will tell.

B. Translating text or tokens to parse trees

While Vinyals et al. have already shown that NMT can parse natural language sentences into parse trees, doing the same for source code is more difficult: Typical natural language tree banks work with under 50 different token types [9], a typical Java grammar may have up to 100. Natural language trees are also much less deeply nested compared to source code. To make a direct translation from naively tokenized source code to a linearized tree representation, it is likely necessary to use multiple features for each target token, either (i) the token type (same as in this study) and the parent token or scope, or (ii) the token type and whether the current token starts or ends a child scope. It may also make sense to use a meta-model representation of the code (e.g., FAMIX [10]), instead of the source language grammar when creating the training data.

C. Parsing noisy, out-of-context sources

Code snippets found on online forums or StackOverflow are often taken out of their original program context and may be missing parts necessary for compilation or further processing (such as imports or variable definitions). Furthermore, they may contain noise, such as ellipses or generic placeholders (like `foo` or `X`). Even island parsers have difficulty parsing noisy code and we propose that neural machine translation could significantly improve on existing techniques. Concretely, one can (i) train a simple sequence-learning model (with the goal of predicting the next token or character) on correctly formatted code from GitHub, (ii) use the model to detect and catalogue noise in sources from StackOverflow, (iii) apply the same noise to the original sources used to train the model generating synthetic 'noisy' input sequences, and (iv) train a new model that de-noises the code using placeholders or best-match tokens from the original training data. Contrary to existing methods of working with code from StackOverflow (e.g., displaying full snippets deemed relevant [11]), a model able to parse these snippets could form the basis for more sophisticated recommender systems.

V. Conclusion

This pilot study yields a negative and a positive result: NMT is not the ideal model for simply tokenizing source code, but it is certainly a strong contender for annotating source code with contextual information. Our training data creation tool can easily be adapted to create large-scale datasets containing other kinds of sequences (and annotations) to perform further experiments. Creating full-fledged parse-trees is more difficult when working with source code than when working on natural language, but simpler representations (e.g., using meta models) may be feasible.

We conclude that NMT can learn the first lesson in mastering a programming language: recognizing syntax and identifying types of tokens. Perhaps, NMT and related neural network models can learn to understand more complex concepts as well - from scratch, rather than by the hand of a developer hard-coding solutions for specific problems.

Acknowledgements

This research is partially supported by the Swiss National Science Foundation (Project №149450 – "Whiteboard"). We thank the Nvidia Corporation for providing the Titan X GPU used for this research.

References

[1] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, May 2015.

[2] A. Karpathy and F. Li, "Deep visual-semantic alignments for generating image descriptions," *CoRR*, vol. abs/1412.2306, 2014. [Online]. Available: http://arxiv.org/abs/1412.2306

[3] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *CoRR*, vol. abs/1409.3215, 2014. [Online]. Available: http://arxiv.org/abs/1409.3215

[4] O. Vinyals, L. Kaiser, T. Koo, S. Petrov, I. Sutskever, and G. E. Hinton, "Grammar as a foreign language," *CoRR*, vol. abs/1412.7449, 2014. [Online]. Available: http://arxiv.org/abs/1412.7449

[5] M. thang Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation."

[6] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[7] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, *On the properties of neural machine translation: Encoder-decoder approaches*, 2014.

[8] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "OpenNMT: Open-Source Toolkit for Neural Machine Translation," *ArXiv e-prints*.

[9] A. Taylor, M. Marcus, and B. Santorini, "The penn treebank: An overview," 2003.

[10] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz, "A meta-model for language-independent refactoring," in *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, 2000.

[11] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining stackoverflow to turn the ide into a self-confident programming prompter," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. ACM, 2014.