Language Modelling for Source Code with Transformer-XL

Thomas Dowdell
University of Newcastle
Callaghan, NSW, Australia
tomjamesdowdell@gmail.com

Hongyu Zhang
University of Newcastle
Callaghan, NSW, Australia
hongyu.zhang@newcastle.edu.au

ABSTRACT

It has been found that software, like natural language texts, exhibits "naturalness", which can be captured by statistical language models. In recent years, neural language models have been proposed to represent the naturalness of software through deep learning. In this paper, we conduct an experimental evaluation of state-of-the-art neural language models for source code, including RNN-based models and Transformer-XL based models. Through experiments on a large-scale Python code corpus, we find that the Transformer-XL model outperforms RNN-based models (including LSTM and GRU models) in capturing the naturalness of software, with far less computational cost.

KEYWORDS

Language Modeling, software naturalness, Transformer-XL

1 INTRODUCTION

Over years of software development, a large amount of source code has been accumulated online. Through large-scale mining, it has been found that source code contains many repetitive, statistical regularities, which are also termed as "naturalness" of source code [17][4][5][6][32]. The naturalness hypothesis has been confirmed by empirical evidence [8][10][15][19][27]. Researchers have also applied natural language processing techniques, such as Language Modeling, to model the naturalness of source code [17][19] [30] [31]. The constructed language models can be used in many practical programming tasks such as code completion [23][17], syntax error fixing [1][7][24], and API recommendation [14].

Language Models (LMs) are probability distributions over strings. Given a sequence of tokens x_T, language modelling is defined as predicting the token x_t given the previous tokens from x_1 to x_{t-1}, i.e. $max_\theta P(x_t|x_{t-1}, x_{t-2}, ..., x_{t-1}, \theta)$, where θ are the trainable model variables. Multiple language models for source code have already been introduced [15] [22] [28]. A classical language modeling method is the N-gram model. Recent research uses RNN-based models (including LSTM and GRU models) [14][15][23][24] and find that RNN-based models superseded N-gram models.

Transformer models [29] have been recently introduced, and have outperformed RNNs for natural language processing tasks, because of their ability to track long-range dependencies and their parallelizable nature. Many Transformer variants have been introduced, notably the Transformer-XL[11], which is capable of tracking remarkably long-range dependencies.

In this paper, we investigate the effectiveness of the Transformer-XL based language model for source code, where the model is trained to predict the next token, given a sequence of source code tokens. Our experimental results on a Python dataset show that the Transformer-XL models largely outperform the state-of-the-art RNN-based models. This strongly suggests that the long-range analysis of the Transformer-XL is helpful for modelling software naturalness. The Transformer-XL model also contained significantly less time to train in comparison to both the LSTM and GRU models.

2 TRANSFORMER-BASED LANGUAGE MODELS FOR SOURCE CODE

2.1 The Transformer Model

Although LSTM [18] and GRU [9] models are considered the pinnacle of RNN-based models, both have noted problems with long-term dependencies. The Transformer model [29] has been introduced to overcome the limitations of RNN-based models. Transformer models do not calculate the results sequentially, but instead calculate the results in a parallel manner using a self-attention mechanism, also allowing a Transformer-based model to be trained in a shorter amount of time. Transformers can achieve better results with less computational cost than the RNN-based models on a variety of natural language tasks [13].

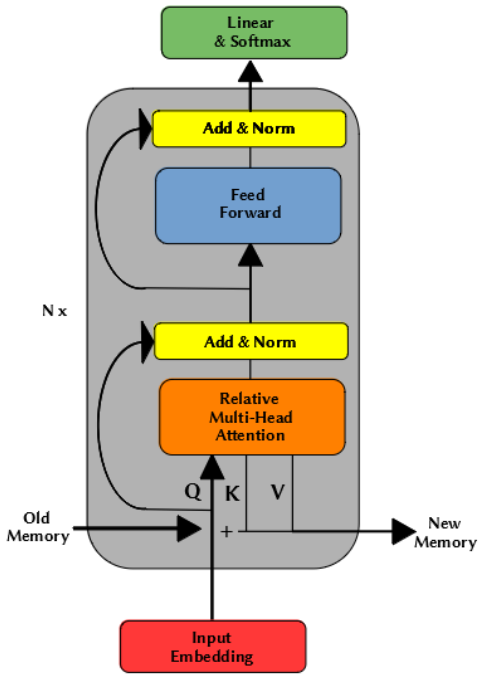

Figure 1: The Transformer-XL model, as specified in [31].

The key to the TransformerâĂŹs success is an attention block, defined as the self-attention mechanism, and a feedforward (FFD) block, both of which are calculated in parallel. The attention block

allows a model to view all data across an entire sequence, unimpeded, in a computationally efficient manner, while the feedforward block analyzes the data in a sequence-independent manner. Transformer-based neural networks take less time to train than the RNNs, require less memory and achieve a lower per-token loss [29]. A single layer of the Transformer can be defined, mathematically, for the l^{th} layer, as:

$$x_0 = inputs$$
$$att_l = Self - Attention\_Mechanism(x_{l-1})$$
$$x_l = AddNorm(att_l, x_{l-1})$$
$$ffd_l = FFD(x_l)$$
$$x_l = AddNorm(ffd_l, x_l)$$

The input sequence of tokens is defined as *inputs*. The functions *AddNorm*, *FFD* and *Self-Attention_Mechanism* are defined as:

$$AddNorm(x, y) = layernorm(x) + y$$
$$FFD(x_l) = W_{l,2} * act(W_{l,1} * x_l + b_{l,1}) + b_{l,2}$$
$$Self-Attention\_Mechanism(x_l) = W_O * softmax(Q * K^T / \sqrt{d_k}) * V$$
$$Q, K, V = W_Q * x_l, W_K * x_l, W_V * x_l$$

The variables $W_{l,1}$, $W_{l,2}$, W_Q, W_K, W_V are all trainable weight matrices, the variables $b_{l,1}$ *and* $b_{l,2}$ are trainable bias vectors and function *act* is an user-defined non-linear activation function. In all models used in this paper, the GELU activation function [16] is used.

The softmax function is applied over the output to yield a categorical probability distribution over each token. At the time-step *t*, the Transformer calculates the input token for time-step *t + 1*.

2.2 The Transformer-XL Model

The Transformer-XL [11] [31] analyzes data in an identical manner to a vanilla transformer, except that, for each sequence, the results calculated for each âĂŸlayerâĂŹ of the transformer are saved and re-inputted, in a gradient-free manner, for the next sequence. This allows the Transformer-XL to âĂŸseeâĂŹ across previous sequences, allowing for a greater amount of input information and therefore greater expressiveness and accuracy. This, conceptually, makes sense, given that a model with greater memory would be capable of analysing this larger memory to generate a better understanding of the underlying source code.

Formally, Transformer-XL redefines the equation that generates the values *Q, K, V* of the attention mechanism as follows:

$$Q = W_Q x$$
$$K, V = W_{K,V}[SG(x_{t-1}) * x]$$

where *[*]* refers to concatenation, *SG* refers to the stop-gradient function and x_{t-1} refers to the input to the attention mechanism from the previous sequence.

Figure 1 shows the structure of Transformer-XL model, as specified in [31]. With the exception of the memory-specific self-attention mechanism, the Transformer-XL layer is identical to the Transformer layer. The self-attention is normalized using layer normalization and a residual connection, followed by an FFD network, as specified in Section 2.1 above.

3 EXPERIMENTS

3.1 Data Collection and Preprocessing

To perform the evaluation, we create a dataset composed of Python source code collected from Github. The dataset contains over 17,000 Python files, which total over 3 million individual lines of code. We use two different forms of tokenization; character-level tokenization and subword-level tokenization. Character-level tokenization contains 141M tokens, while subword tokenization contains 88M tokens.

We did not use word-level tokenization for this task. This is because, unlike natural language, source code does not have an explicit vocabulary. Therefore, there is no way to accurately and effectively model all possible words inside a source code file. However, the source code can be effectively modelled using characters and subwords[25]. For example, the source code *print(x + 3 if x == 0)* can be tokenized into a sequence of subwords, [*print, (, x, +, 3, if, x, ==, 0,)*].

We explicitly set, for subword tokenization, the vocabulary size hyperparameter to 1000. We chose this vocabulary size because, as previous authors have noted [21], a large vocabulary is responsible for one of the most computationally expensive aspects of the model. By setting the vocabulary size to 1000, the model can tokenize the most common words as a single token, but break down less common words into a series of subwords. This allows the model to express any input information without unacceptable computational complexity and information loss. Therefore, subword-level tokenization does not suffer from the out-of-vocabulary (OOV) problem.

Given this dataset, the model is trained to functionally predict the next token in the line, without being able to âĂŸpeek' forward through the line. We specify that the model is trained on a sequence that includes 256 tokens, and the memory from the previous sequence.

We split the collected data in the training, validation and testing data using an 80%/10%/10% split. We were careful to avoid as much code duplication as possible, which has been previously shown to be a common problem for many source code-based machine learning models [3]. In order to do this, we tested that each file in the training dataset did not reappear in either the validation or testing dataset. Further, for each validation and test file, we tested to make sure that the number of lines in each file was not repeated in the training dataset above a certain threshold. If the file was over the threshold, then it was removed from the dataset. We chose 25% as the threshold.

3.2 Model Training

Each model is trained for 50 epochs, where each epoch contains 512 iterations. The learning rate is set to a linear-warmup from 1e-6 to 5e-4 for the first 5120 iterations, and a cosine-decay rate back to 1e-6 for the following iterations. The optimizer used is the Adam Optimizer [20]. We clip the gradient norm to 0.1, which we found was essential for training. We noted that if the gradients were not clipped, then all models would produce inferior results.

Both RNN models and the Transformer-XL model have a hidden size of 512 units, and have a dropout rate set to 0.1 for training. Initially, we trained each model with a depth of 4 layers. RNN-based models do not typically include many more layers due to the RNN's massive computational cost. However, Transformer-based models typically have many more layers, anywhere from 12 [29] to 72 [26] layers.

The depth of a model is considered an essential aspect of a model's ability to learn, but we noted that increasing the number of RNN layers significantly slows down training, but this was not the case for the Transformer-XL model. Considering previous papers[2] have shown that Transformer-based models achieve noticeably superior results by increasing the depth for language modelling, we decided to further experiment by increasing the number of layers to 8.

The experiments were primarily conducted on a single server and a single K40 NVIDIA Tesla GPU, alongside 128GB of RAM.

3.3 Evaluation Metrics

The evaluation metric used for character-level experiments in this paper is the BPC (bytes-per-character) metric [2], where the BPC is calculated per-token. This metric is chosen because BPC is a function of test-entropy, which effectively displays how âĂŸconfident' a model is about guessing the correct answer. The lower the BPC the more confident the model is when guessing the correct answer and the more effective the overall model. The cross-entropy equation can be defined as, given, at time-step t, the predicted output y_t and the target output z_t:

$$loss_t = -1 * z_t * log(y_t)$$

The BPC is defined as:

$$bpc = \frac{loss}{log(2)}$$

For subword-level experiments, the metric is the perplexity rather than the BPC. Perplexity is defined as:

$$perplexity = e^{\text{loss}}$$

Low perplexity is desirable since the perplexity is the exponential of the entropy. Multiple previous papers have stated that, for character-level analysis, BPC is the metric of choice but perplexity is the metric to be used for both word-level and subword-level analysis [2][13][31].

3.4 Experimental Results

The results from all experiments are presented in Table 1. We tested each model 3 times and presented the average results of each model. Each model was initialized randomly, and the random-seed is different for each experiment. We noted that each model, when tested with the same data, performed similarly across different runs and seeds. The validation data loss, for both the character-level and BPC-level analysis, can be seen in Figure 2 and Figure 3.

The LSTM model, which has been used as the previous neural source code modelling work [12], achieved the highest per-character BPC (1.8706) and per-subword Perplexity (6.4552). The GRU model achieved reliably better results than the LSTM (BPC 1.2758 and Perplexity 6.1135), but only by a small margin. The Transformer-XL model, on the other hand, could reliably outperform both the LSTM and GRU models by a substantial margin, for both the 4-layer and 8-layer models. The 8-layer Transformer-XL model, which performed better then all others, achieved a per-character BPC of 1.1297 and per-subword perplexity of 2.7185, which is lower than all other models. It is worth noting that the 4-layer Transformer-XL outperformed both RNN models as well.

The superior results of the 4-layer Transformer-XL model, in comparison to the RNN models, is likely a sign that the Transformer-XL model can extract more rich and useful features than both RNN models.

We also found that an 8-layer Transformer-XL model outperformed a 4-layer Transformer-XL model, but not by the same margin that a Transformer-XL outperforms the RNN. This 8-layer Transformer-XL, despite having twice the depth of the RNN models, was trained in less than half of the time required by the RNN models, even when trained on identical hardware.

Model	Character-Level Test BPC	Subword-Level Test Perplexity
LSTM (4 Layer)	1.8706 *(0.0053)*	6.4552 *(0.0774)*
GRU (4 Layer)	1.2758 *(0.002)*	6.1135 *(0.0146)*
Transformer-XL (4 Layer)	1.1506 *(0.0018)*	2.7278 *(0.0005)*
Transformer-XL (8 Layer)	**1.1297** *(0.0063)*	**2.7185** *(0.0208)*

Table 1: The test BPC and Perplexity of each of the three models, where better models output lower values. The numbers in brackets are standard deviation across the multiple runs. Across both tokenization schemes, the Transformer-XL model outperforms both RNN models.

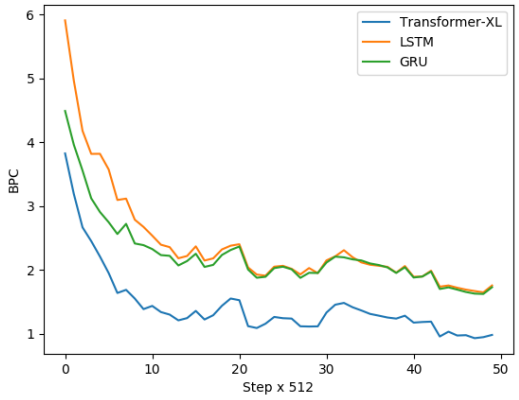

Figure 2: The BPC for the LSTM, GRU and Transformer-XL model when analysing character-level source code. The Transformer-XL, across the entire run, outperforms the RNN models. We chose to display the best-performing model for the LSTM, GRU and Transformer-XL.

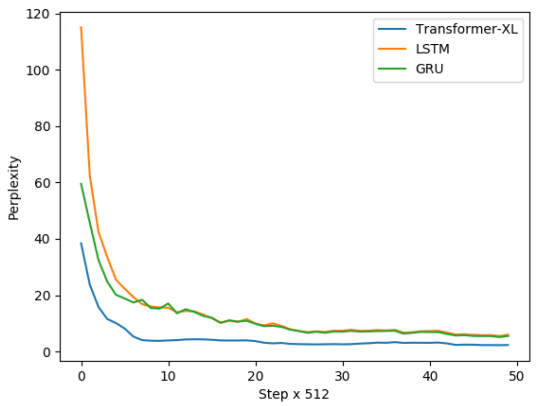

Figure 3: The Perplexity for the LSTM, GRU and Transformer-XL model when analysing subword-level source code. The Transformer-XL outperforms the RNN models.

Model	Normalized Training Time
LSTM (4 Layer)	4.2
GRU (4 Layer)	3.58
Transformer-XL (4 Layer)	1
Transformer-XL (8 Layer)	1.39

Table 2: The normalized training time for each model, where each unit is the normalized length of time for training a model over a single iteration. The 4-layer Transformer-XL model requires the shortest training time, followed by the 8-layer Transformer-XL model. It is worth noting that the 8-layer Transformer-XL, despite having twice the depth of RNNs, takes less than half of the time to train.

We note that the training time for each model depends on the hardware that the model is trained on. For example, an RNN model that is trained on a quicker GPU will be trained faster than a more efficient model on a more slow GPU. In order to reflect this, we recorded the normalized training time, where each model was trained on the same GPU and the time to train was normalized, to better reflect the length of training for each model in comparison to the other models. The normalized training time for each model can be seen in Table 2, from which we can see that Transformer-XL models require much less time to train than the two RNN models (LSTM and GRU).

In summary, our experiments show that the Transformer-XL model outperformed both RNN models when the layers are all set to 4, and the Transformer-XL further improves results when there are 8 layers instead of 4. Furthermore, even though the 8-layer Transformer-XL model has twice the depth and more trainable parameters, it takes less than half of the time to train and achieves notably superior results.

4 DISCUSSION

Our experimental results suggest that the Transformer-XL model is a substantially superior model to either the LSTM or GRU model for modelling source code. This is because the RNN models cannot easily be trained for more layers without incurring massive computational cost, and therefore the RNN models produce inferior result [29]. The Transformer-XL model, on the other hand, can produce noticeably superior results by increasing the layers without an unacceptable computational cost. Therefore, the Transformer-XL can achieve substantially superior using results in practice. We suggest using Transformer-XL in future research work on language modelling for source code.

Previous work on natural language processing has shown that models that are trained on language modelling based tasks can achieve state-of-the-art results on downstream NLP tasks, such as sentiment analysis and question-answering [13]. We would hypothesize that downstream source code related tasks can be performed in a similar way. Pre-training the models to perform language modelling allows a model to “learn" the underlying statistical behaviour (naturalness) of source code, and this understanding can be leveraged to effectively perform downstream tasks.

Future work could focus on applying language models to the source code related downstream tasks such as automatic code completion. These models could also be pushed further. For example, these models could be easily retrained to perform more complicated tasks such as automatic error detection or automatic bug fixing. These tasks cannot be expressed in an identical manner to language modelling, in a manner similar to automatic code completion. However, the software naturalness that the model learns can be leveraged to better perform these downstream tasks. Future work should focus on testing the effectiveness of these downstream tasks, and whether a language model can be trained to improve these tasks effectively.

5 CONCLUSION

This paper has shown that Transformer-based language models largely outperform RNN-based models for source code modelling. The Transformer-XL model achieves substantially better results with less than half the complexity and training time.

Based on this work, we recommend that Transformer-XL can be adopted for language modelling tasks relating to source code. Following the examples set in natural language processing [13], this model can be further applied to other downstream source code related tasks, such as automatic code completion and automatic bug fixing.

Our tool and experimental data are publicly available at https://github.com/Anon-111/Source-Code-Modelling.

REFERENCES

[1] Umair Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation of Error Repair: For the Student Programs, from the Student Programs. In *40th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEETâĂŹ18)*. ACM, 78–87. https://dl.acm.org/citation.cfm?doid=3183377.3183383

[2] Rami Al-Rfou, Dokook Choe, Noah Constant, Mandy Guo, and Llion Jones. 2018. Character-Level Language Modeling with Deeper Self-Attention. *arXiv preprint arXiv:1808.04444v2* (2018).

[3] Miltiadis Allamanis. 2018. The Adverse Effects of Code Duplication in Machine Learning Models of Code. *arXiv preprint arXiv:1812.06468* (2018).

[4] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. *arXiv preprint arXiv:1402.4182* (2014).

[5] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.

[6] Miltiadis Allamanis and Charles Sutton. 2013. Mining Source Code Repositories at Massive Scale using Language Modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 207–216.

[7] Sahil Bhatia and Rishabh Singh. 2016. Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks. *arXiv preprint arXiv:1603.06129* (2016).

[8] Avishkar Bhoopchand, Tim Rockt aschel, Earl Barr, and Sebastian Riedel. 2016. Learning Python Code Suggestion with a Sparse Pointer Network. *arXiv preprint arXiv:1611.08307* (2016).

[9] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv preprint arXiv:1406.1078* (2014).

[10] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing benchmarks for predictive modeling. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. ACM, 86–99. 3049843

[11] Zihing Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. 2019. Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context. *arXiv preprint arXiv:1901.02860* (2019).

[12] Hoa Khanh Dam, Truyen Tran, and Trang Pham. 2016. A Deep Language Model Model for Software Code. *arXiv preprint arXiv:1608.02715* (2016).

[13] Jacob Delvin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[14] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 631âĂŞ642. https://doi.org/10.1145/2950290.2950334

[15] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI'17)*. AAAI Press, 1345–1351. http://dl.acm.org/citation.cfm?id=3298239.3298436

[16] Dan Hendrycks and Kevin Gimpe. 2018. Gaussian Error Linear Units (GELUs). *arXiv preprint arXiv:arXiv:1606.08415v3* (2018).

[17] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 837–847. http://dl.acm.org/citation.cfm?id=2337223.2337322

[18] Sepp Hochreiter and JÃijrgen Schmidhuber. 1997. Long Short-Term Memory. In *Neural Computation*. 1735–1780. https://www.mitpressjournals.org/doi/10.1162/neco.1997.9.8.1735

[19] Rafael-Michael Karampatsis and Charles A. Sutton. 2019. Maybe Deep Neural Networks are the Best Choice for Modeling Source Code. *CoRR* abs/1903.05734 (2019). arXiv:1903.05734 http://arxiv.org/abs/1903.05734

[20] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[21] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. *arXiv preprint arXiv:1909.11942* (2019).

[22] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, 925–936. https://dl.acm.org/citation.cfm?id=3340455

[23] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. *SIGPLAN Not.* 49, 6 (June 2014), 419âĂŞ428. https://doi.org/10.1145/2666356.2594321

[24] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral. 2018. Syntax and sensibility: Using language models to detect and correct syntax errors. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 311–322. https://doi.org/10.1109/SANER.2018.8330219

[25] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural Machine Translation of Rare Words with Subword Units. *arXiv preprint arXiv:1509.07909* (2015).

[26] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jard Casper, and Bryan Catanzaro. 2019. Megatrom-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:arXiv:1909.08053v3* (2019).

[27] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 43–52.

[28] Zhaopeng Tu, Zheopeng Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 269–280.

[29] Ashish Vaswani, Noam Sheezer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

[30] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward Deep Learning Software Repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, Piscataway, NJ, USA, 334–345. http://dl.acm.org/citation.cfm?id=2820518.2820559

[31] Zhilin Yang, Zihing Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. *arXiv preprint arXiv:1906.08237* (2019).

[32] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 783–794. https://doi.org/10.1109/ICSE.2019.00086