# A Transformer-based Approach for Source Code Summarization

**Wasi Uddin Ahmad**
University of California, Los Angeles
wasiahmad@cs.ucla.edu

**Saikat Chakraborty**
Columbia University
saikatc@cs.columbia.edu

**Baishakhi Ray**
Columbia University
rayb@cs.columbia.edu

**Kai-Wei Chang**
University of California, Los Angeles
kwchang@cs.ucla.edu

## Abstract

Generating a readable summary that describes the functionality of a program is known as source code summarization. In this task, learning code representation by modeling the pairwise relationship between code tokens to capture their long-range dependencies is crucial. To learn code representation for summarization, we explore the Transformer model that uses a self-attention mechanism and has shown to be effective in capturing long-range dependencies. In this work, we show that despite the approach is simple, it outperforms the state-of-the-art techniques by a significant margin. We perform extensive analysis and ablation studies that reveal several important findings, e.g., the absolute encoding of source code tokens' position hinders, while relative encoding significantly improves the summarization performance. We have made our code publicly available[1] to facilitate future research.

## 1 Introduction

Program comprehension is an indispensable ingredient of software development and maintenance (Xia et al., 2017). A natural language summary of source code facilitates program comprehension by reducing developers' efforts significantly (Sridhara et al., 2010). Source code summarization refers to the task of creating readable summaries that describe the functionality of a program.

With the advancement of deep learning and the availability of large-scale data through a vast number of open-source repositories, automatic source code summarizing has drawn attention from researchers. Most of the neural approaches generate source code summaries in a sequence-to-sequence fashion. One of the initial works Iyer et al. (2016) trained an embedding matrix to represent the individual code tokens and combine them with a Recurrent Neural Network (RNN) via an attention mechanism to generate a natural language summary. Subsequent works (Liang and Zhu, 2018; Hu et al., 2018a,b) adopted the traditional RNN-based sequence-to-sequence network (Sutskever et al., 2014) with attention mechanism (Luong et al., 2015) on different abstractions of code.

The RNN-based sequence models have two limitations in learning source code representations. First, they do not model the non-sequential structure of source code as they process the code tokens sequentially. Second, source code can be very long, and thus RNN-based models may fail to capture the long-range dependencies between code tokens. In contrast to the RNN-based models, *Transformer* (Vaswani et al., 2017), which leverages self-attention mechanism, can capture long-range dependencies. Transformers have been shown to perform well on many natural language generation tasks such as machine translation (Wang et al., 2019), text summarization (You et al., 2019), story generation (Fan et al., 2018), etc.

To learn the order of tokens in a sequence or to model the relationship between tokens, Transformer requires to be injected with positional encodings (Vaswani et al., 2017; Shaw et al., 2018; Shiv and Quirk, 2019). In this work, we show that, by modeling the pairwise relationship between source code tokens using relative position representation (Shaw et al., 2018), we can achieve significant improvements over learning sequence information of code tokens using absolute position representation (Vaswani et al., 2017).

We want to emphasize that our proposed approach is simple but effective as it outperforms the fancy and sophisticated state-of-the-art source code summarization techniques by a significant margin. We perform experiments on two well-studied datasets collected from GitHub, and the results endorse the effectiveness of our approach

[1] https://github.com/wasiahmad/NeuralCodeSum

over the state-of-the-art solutions. In addition, we provide a detailed ablation study to quantify the effect of several design choices in the Transformer to deliver a strong baseline for future research.

## 2 Proposed Approach

We propose to use *Transformer* (Vaswani et al., 2017) to generate a natural language summary given a piece of source code. Both the code and summary is a sequence of tokens that are represented by a sequence of vectors, $\mathbf{x} = (x_1, \ldots, x_n)$ where $x_i \in R^{d_{model}}$. In this section, we briefly describe the Transformer architecture (§ 2.1) and how to model the order of source code tokens or their pairwise relationship (§ 2.2) in Transformer.

### 2.1 Architecture

The Transformer consists of stacked multi-head attention and parameterized linear transformation layers for both the encoder and decoder. At each layer, the multi-head attention employs $h$ attention heads and performs the self-attention mechanism.

**Self-Attention.** We describe the self-attention mechanism based on Shaw et al. (2018). In each attention head, the sequence of input vectors, $\mathbf{x} = (x_1, \ldots, x_n)$ where $x_i \in R^{d_{model}}$ are transformed into the sequence of output vectors, $\mathbf{o} = (o_1, \ldots, o_n)$ where $o_i \in R^{d_k}$ as:

$$o_i = \sum_{j=1}^{n} \alpha_{ij}(x_j W^V),$$

$$e_{ij} = \frac{x_i W^Q (x_j W^K)^T}{\sqrt{d_k}},$$

where $\alpha_{ij} = \frac{\exp e_{ij}}{\sum_{k=1}^{n} \exp e_{ik}}$ and $W^Q, W^K \in R^{d_{model} \times d_k}, W^V \in R^{d_{model} \times d_v}$ are the parameters that are unique per layer and attention head.

**Copy Attention.** We incorporate the copying mechanism (See et al., 2017) in the Transformer to allow both generating words from vocabulary and copying from the input source code. We use an additional attention layer to learn the copy distribution on top of the decoder stack (Nishida et al., 2019). The copy attention enables the Transformer to copy rare tokens (e.g., function names, variable names) from source code and thus improves the summarization performance significantly (§ 3.2).

### 2.2 Position Representations

Now, we discuss how to learn the order of source code tokens or model their pairwise relationship.

| Dataset | Java | Python |
|---|---|---|
| Train | 69,708 | 55,538 |
| Validation | 8,714 | 18,505 |
| Test | 8,714 | 18,502 |
| Unique tokens in code | 66,650 | 3,07,596 |
| Unique tokens in summary | 46,895 | 56,189 |
| Avg. tokens in code | 120.16 | 47.98 |
| Avg. tokens in summary | 17.73 | 9.48 |

Table 1: Statistics of the experiment datasets. We thank the authors of Wei et al. (2019) for kindly sharing the Python dataset splits. The Java dataset splits are publicly available.

**Encoding absolute position.** To allow the Transformer to utilize the order information of source code tokens, we train an embedding matrix $W^{P_e}$ that learns to encode tokens' absolute positions into vectors of dimension $d_{model}$. However, we show that capturing the order of code tokens is not helpful to learn source code representations and leads to poor summarization performance (§ 3.2).

It is important to note that we train another embedding matrix $W^{P_d}$ that learns to encode the absolute positions of summary tokens.[2]

**Encoding pairwise relationship.** The semantic representation of a code does not rely on the absolute positions of its tokens. Instead, their mutual interactions influence the meaning of the source code. For instance, semantic meaning of the expressions `a+b` and `b+a` are the same.

To encode the pairwise relationships between input elements, Shaw et al. (2018) extended the self-attention mechanism as follows.

$$o_i = \sum_{j=1}^{n} \alpha_{ij}(x_j W^V + a_{ij}^V),$$

$$e_{ij} = \frac{x_i W^Q (x_j W^K + a_{ij}^K)^T}{\sqrt{d_k}},$$

where, $a_{ij}^V$ and $a_{ij}^K$ are relative positional representations for the two position $i$ and $j$. Shaw et al. (2018) suggested clipping the maximum relative position to a maximum absolute value of $k$ as they hypothesize that precise relative position information is not useful beyond a certain distance.

$$a_{ij}^K = w_{clip(j-i,k)}^K, a_{ij}^V = w_{clip(j-i,k)}^V,$$

$$clip(x, k) = \max(-k, \min(k, x)).$$

Hence, we learn $2k + 1$ relative position representations: $(w_{-k}^K, \ldots, w_k^K)$, and $(w_{-k}^V, \ldots, w_k^V)$.

[2] In this work, we do not study alternative ways of learning position representation for the summary tokens.

| Methods | Java | | | Python | | |
|---|---|---|---|---|---|---|
| | BLEU | METEOR | ROUGE-L | BLEU | METEOR | ROUGE-L |
| CODE-NN (Iyer et al., 2016) | 27.60 | 12.61 | 41.10 | 17.36 | 09.29 | 37.81 |
| Tree2Seq (Eriguchi et al., 2016) | 37.88 | 22.55 | 51.50 | 20.07 | 08.96 | 35.64 |
| RL+Hybrid2Seq (Wan et al., 2018) | 38.22 | 22.75 | 51.91 | 19.28 | 09.75 | 39.34 |
| DeepCom (Hu et al., 2018a) | 39.75 | 23.06 | 52.67 | 20.78 | 09.98 | 37.35 |
| API+CODE (Hu et al., 2018b) | 41.31 | 23.73 | 52.25 | 15.36 | 08.57 | 33.65 |
| Dual Model (Wei et al., 2019) | 42.39 | 25.77 | 53.61 | 21.80 | 11.14 | 39.45 |
| Our models and ablation study | | | | | | |
| Base Model | 43.41 | 25.91 | 52.71 | 31.08 | 18.57 | 44.31 |
| Full Model | **44.58** | **26.43** | **54.76** | **32.52** | **19.77** | **46.73** |
| Full Model w/o Relative Position | 44.26 | 26.23 | 53.58 | 31.38 | 18.69 | 44.68 |
| Full Model w/o Copy Attention | 44.14 | 26.34 | 53.95 | 31.64 | 19.17 | 45.42 |

Table 2: Comparison of our proposed approach with the baseline methods. The results of the baseline methods are directly reported from (Wei et al., 2019). The "Base Model" refers to the vanilla Transformer (uses absolute position representations) and the "Full Model" uses relative position representations and includes copy attention.

In this work, we study an alternative of the relative position representations that ignores the *directional* information (Ahmad et al., 2019). In other words, the information whether the $j$'th token is on the left or right of the $i$'th token is ignored.

$$a_{ij}^K = w^K_{clip(|j-i|,k)}, a_{ij}^V = w^V_{clip(|j-i|,k)},$$
$$clip(x,k) = \min(|x|, k).$$

## 3 Experiment

### 3.1 Setup

**Datasets and Pre-processing.** We conduct our experiments on a Java dataset (Hu et al., 2018b) and a Python dataset (Wan et al., 2018). The statistics of the two datasets are shown in Table 1. In addition to the pre-processing steps followed by Wei et al. (2019), we split source code tokens of the form *CamelCase* and *snake_case* to respective sub-tokens[3]. We show that such a split of code tokens improves the summarization performance.

**Metrics.** We evaluate the source code summarization performance using three metrics, BLEU (Papineni et al., 2002), METEOR (Banerjee and Lavie, 2005), and ROUGE-L (Lin, 2004).

**Baselines.** We compare our Transformer-based source code summarization approach with five baseline methods reported in Wei et al. (2019) and their proposed Dual model. We refer the readers to (Wei et al., 2019) for the details about the hyperparameter of all the baseline methods.

**Hyper-parameters.** We follow Wei et al. (2019) to set the maximum lengths and vocabulary sizes for code and summaries in both the datasets. We train the Transformer models using Adam optimizer (Kingma and Ba, 2015) with an initial learning rate of $10^{-4}$. We set the mini-batch size and dropout rate to 32 and 0.2, respectively. We train the Transformer models for a maximum of 200 epochs and perform early stop if the validation performance does not improve for 20 consecutive iterations. We use a beam search during inference and set the beam size to 4. Detailed hyperparameter settings can be found in Appendix A.

### 3.2 Results and Analysis

**Overall results.** The overall results of our proposed model and baselines are presented in Table 2. The result shows that the *Base* model outperforms the baselines (except for ROUGE-L in java), while the *Full* model improves the performance further.[4] We ran the Base model on the original datasets (without splitting the *CamelCase* and *snake_case* code tokens) and observed that the performance drops by 0.60, 0.72 BLEU and 1.66, 2.09 ROUGE-L points for the Java and Python datasets respectively. We provide a few qualitative examples in Appendix C showing the usefulness of the Full model over the Base model.

Unlike the baseline approaches, our proposed model employs the copy attention mechanism. As shown in Table 2, the copy attention improves the performance 0.44 and 0.88 BLEU points for the Java and Python datasets respectively.

**Impact of position representation.** We perform an ablation study to investigate the benefits

[3] The *CamelCase* and *snake_case* tokenization reduces the vocabulary significantly. For example, the number of unique tokens in Java source code reduced from 292,626 to 66,650.

[4] We observe a more significant gain on the Python dataset and a detailed discussion on it is provided in Appendix B.

| Source | Target | BLEU | METEOR | ROUGE-L |
|---|---|---|---|---|
| ✓ | ✓ | 43.41 | 25.91 | 52.71 |
| ✓ | ✗ | 42.34 | 24.74 | 50.96 |
| ✗ | ✓ | **43.59** | **26.00** | **52.88** |
| ✗ | ✗ | 41.85 | 24.32 | 50.87 |

Table 3: Ablation study on absolute positional representations using the "Base Model" on the Java dataset.

| $k$ | Directional | BLEU | METEOR | ROUGE-L |
|---|---|---|---|---|
| 8 | ✓ | 44.22 | 26.35 | 53.86 |
|  | ✗ | 42.61 | 24.67 | 51.10 |
| 16 | ✓ | 44.14 | 26.34 | 53.95 |
|  | ✗ | 44.06 | 26.31 | 53.51 |
| 32 | ✓ | **44.55** | **26.66** | **54.30** |
|  | ✗ | 43.95 | 26.28 | 53.24 |
| $2^i$ | ✓ | 44.37 | 26.58 | 53.96 |
|  | ✗ | 43.58 | 25.95 | 52.73 |

Table 4: Ablation study on relative positional representations (in encoding) for Transformer. While 8, 16, and 32 represents a fixed relative distance for all the layers, $2^i$ (where $i = 1, \ldots, L$; $L = 6$) represents a layer-wise relative distance for Transformer.

|  | #Param. | BLEU | METEOR | ROUGE-L |
|---|---|---|---|---|
| Varying the model size ($d_{model}$) |  |  |  |  |
| 256 | 15.8 | 38.21 | 21.54 | 48.63 |
| 384 | 28.4 | 41.71 | 24.51 | 51.42 |
| 512 | 44.1 | 43.41 | 25.91 | 52.71 |
| 768 | 85.1 | **45.29** | **27.56** | **54.39** |
| Varying the number of layers ($l$) |  |  |  |  |
| 3 | 22.1 | 41.26 | 23.54 | 51.37 |
| 6 | 44.1 | 43.41 | 25.91 | 52.71 |
| 9 | 66.2 | 45.03 | 27.21 | 54.02 |
| 12 | 88.3 | **45.56** | **27.64** | **54.89** |

Table 5: Ablation study on the hidden size and number of layers for the "Base Model" on the Java dataset. We use $d_{model} = H$, $d_{ff} = 4H$, $h = 8$, and $d_k = d_v = 64$ in all settings. We set $l = 6$ and $d_{model} = 512$ while varying $d_{model}$ and $l$ respectively. #Param. represents the number of trainable parameters in millions (only includes Transformer parameters).

of encoding the absolute position of code tokens or modeling their pairwise relationship for the source code summarization task, and the results are presented in Table 3 and 4. Table 3 demonstrates that learning the absolute position of code tokens are not effective as we can see it slightly hurts the performance compared to when it is excluded. This empirical finding corroborates the design choice of Iyer et al. (2016), where they did not use the sequence information of the source code tokens.

On the other hand, we observe that learning the pairwise relationship between source code tokens via relative position representations helps as Table 4 demonstrates higher performance. We vary the clipping distance, $k$, and consider ignoring the directional information while modeling the pairwise relationship. The empirical results suggest that the directional information is indeed important while 16, 32, and $2^i$ relative distances result in similar performance (in both experimental datasets).

**Varying model size and number of layers.** We perform ablation study by varying $d_{model}$ and $l$ and the results are presented in Table 5.[5] In our experiments, we observe that a deeper model (more layers) performs better than a wider model (larger $d_{model}$). Intuitively, the source code summarization task depends on more semantic information than syntactic, and thus deeper model helps.

**Use of Abstract Syntax Tree (AST).** We perform additional experiments to employ the abstract syntax tree (AST) structure of source code in the Transformer. We follow Hu et al. (2018a) and use the Structure-based Traversal (SBT) technique to transform the AST structure into a linear sequence. We keep our proposed Transformer architecture intact, except in the copy attention mechanism, we use a mask to block copying the non-terminal tokens from the input sequence. It is important to note that, with and without AST, the average length of the input code sequences is 172 and 120, respectively. Since the complexity of the Transformer is $O(n^2 \times d)$ where $n$ is the input sequence length, hence, the use of AST comes with an additional cost. Our experimental findings suggest that the incorporation of AST information in the Transformer does not result in an improvement in source code summarization. We hypothesize that the exploitation of the code structure information in summarization has limited advantage, and it diminishes as the Transformer learns it implicitly with relative position representation.

**Qualitative analysis.** We provide a couple of examples in Table 6 to demonstrate the usefulness of our proposed approach qualitatively (more examples are provided in Table 9 and 10 in the Appendix). The qualitative analysis reveals that, in comparison to the Vanilla Transformer model, the copy enabled model generates shorter summaries

[5] Considering the model complexity, we do not increase the model size or number of layers further.

```
public static String selectText(XPathExpression expr, Node context) {
    try {
        return (String)expr.evaluate(context, XPathConstants.STRING );
    } catch (XPathExpressionException e) {
        throw new XmlException(e);
    }
}
```

Base Model: evaluates the xpath expression to a xpath expression .
Full Model w/o Relative Position: evaluates the xpath expression .
Full Model w/o Copy Attention Attention: evaluates the xpath expression as a single element .
Full Model: evaluates the xpath expression as a text string .
Human Written: evaluates the xpath expression as text .

```
def get_hosting_service(name):
    try:
        return hosting_service_registry.get(u'hosting service id', name)
    except ItemLookupError:
        return None
```

Base Model: returns the color limits from the current service name .
Full Model w/o Relative Position: return the hosting service .
Full Model w/o Copy Attention: return the name of the service .
Full Model : return the hosting service name .
Human Written: return the hosting service with the given name .

Table 6: Qualitative example of different models' performance on Java and Python datasets.

with more accurate keywords. Besides, we observe that in a copy enabled model, frequent tokens in the code snippet get a higher copy probability when relative position representations are used, in comparison to absolute position representations. We suspect this is due to the flexibility of learning the relation between code tokens without relying on their absolute position.

## 4 Related Work

Most of the neural source code summarization approaches frame the problem as a sequence generation task and use recurrent encoder-decoder networks with attention mechanisms as the fundamental building blocks (Iyer et al., 2016; Liang and Zhu, 2018; Hu et al., 2018a,b). Different from these works, Allamanis et al. (2016) proposed a convolutional attention model to summarize the source codes into short, name-like summaries.

Recent works in code summarization utilize structural information of a program in the form of *Abstract Syntax Tree* (AST) that can be encoded using tree structure encoders such as Tree-LSTM (Shido et al., 2019), Tree-Transformer (Harer et al., 2019), and Graph Neural Network (LeClair et al., 2020). In contrast, Hu et al. (2018a) proposed a structure based traversal (SBT) method to flatten the AST into a sequence and showed improvement over the AST based methods. Later, LeClair et al. (2019) used the SBT method and decoupled the code structure from the code tokens to learn better structure representation.

Among other noteworthy works, API usage information (Hu et al., 2018b), reinforcement learning (Wan et al., 2018), dual learning (Wei et al., 2019), retrieval-based techniques (Zhang et al., 2020) are leveraged to further enhance the code summarization models. We can enhance a Transformer with previously proposed techniques; however, in this work, we limit ourselves to study different design choices for a Transformer without breaking its' core architectural design philosophy.

## 5 Conclusion

This paper empirically investigates the advantage of using the Transformer model for the source code summarization task. We demonstrate that the Transformer with relative position representations and copy attention outperforms state-of-the-art approaches by a large margin. In our future work, we want to study the effective incorporation of code structure into the Transformer and apply the techniques in other software engineering sequence generation tasks (e.g., commit message generation for source code changes).

## Acknowledgments

This work was supported in part by National Science Foundation Grant OAC 1920462, CCF 1845893, CCF 1822965, CNS 1842456.

## References

Wasi Ahmad, Zhisong Zhang, Xuezhe Ma, Eduard Hovy, Kai-Wei Chang, and Nanyun Peng. 2019. On difficulties of cross-lingual transfer with order differences: A case study on dependency parsing. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2440–2452, Minneapolis, Minnesota. Association for Computational Linguistics.

Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2091–2100. JMLR.org.

Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan. Association for Computational Linguistics.

Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. 2016. Tree-to-sequence attentional neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 823–833, Berlin, Germany. Association for Computational Linguistics.

Angela Fan, Mike Lewis, and Yann Dauphin. 2018. Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 889–898, Melbourne, Australia. Association for Computational Linguistics.

Jacob Harer, Chris Reale, and Peter Chin. 2019. Tree-transformer: A transformer-based method for correction of tree-structured data. *arXiv preprint arXiv:1908.00449*.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018a. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, page 200–210, New York, NY, USA. Association for Computing Machinery.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018b. Summarizing source code with transferred api knowledge. pages 2269–2275.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany. Association for Computational Linguistics.

Diederik P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*.

Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander Rush. 2017. OpenNMT: Open-source toolkit for neural machine translation. In *Proceedings of ACL 2017, System Demonstrations*, pages 67–72, Vancouver, Canada. Association for Computational Linguistics.

Alexander LeClair, Sakib Haque, Linfgei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering*, page 795–806. IEEE Press.

Yuding Liang and Kenny Qili Zhu. 2018. Automatic generation of text descriptive comments for code blocks. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal. Association for Computational Linguistics.

Kyosuke Nishida, Itsumi Saito, Kosuke Nishida, Kazutoshi Shinoda, Atsushi Otsuka, Hisako Asano, and Junji Tomita. 2019. Multi-style generative reading comprehension. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2273–2284, Florence, Italy. Association for Computational Linguistics.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1073–1083, Vancouver, Canada. Association for Computational Linguistics.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, New Orleans, Louisiana. Association for Computational Linguistics.

Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic source code summarization with extended tree-lstm. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

Vighnesh Shiv and Chris Quirk. 2019. Novel positional encodings to enable tree-based transformers. In *Advances in Neural Information Processing Systems 32*, pages 12081–12091. Curran Associates, Inc.

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*, page 43–52, New York, NY, USA. Association for Computing Machinery.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 397–407. ACM.

Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F. Wong, and Lidia S. Chao. 2019. Learning deep transformer models for machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1810–1822, Florence, Italy. Association for Computational Linguistics.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 6563–6573. Curran Associates, Inc.

Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976.

Yongjian You, Weijia Jia, Tianyi Liu, and Wenmian Yang. 2019. Improving abstractive document summarization with salient information modeling. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2132–2141, Florence, Italy. Association for Computational Linguistics.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *Proceedings of the 42nd International Conference on Software Engineering*. IEEE.

# Supplementary Material: Appendices

## A Hyper-Parameters

Table 7 summarizes the hyper-parameters that we used in our experiments.

| | Hyper-parameter | Value |
|---|---|---|
| Embedding | $k$ | 16 |
| Model | $l$ | 6 |
| | $h$ | 8 |
| | $d_{model}$ | 512 |
| | $d_k, d_v$ | 64 |
| | $d_{ff}$ | 2048 |
| Training | dropout | 0.2 |
| | optimizer | Adam |
| | learning rate | 0.0001 |
| | batch size | 32 |
| Testing | beam size | 4 |

Table 7: Hyper-parameters in our experiments. $l$ and $h$ indicates the number of layers and heads in Transformer respectively. $k$ refers to the clipping distance in relative position representations in Transformer.

## B Recurrent Encoder-Decoder vs. Transformer on Python Dataset

While conducting our study using Transformer on the Python dataset, we observed a large gain over the state-of-the-art methods as reported in Wei et al. (2019). However, our initial experiments on this dataset using recurrent sequence-to-sequence models also demonstrated higher performance compared to the results report in Wei et al. (2019). We suspect that such lower performance is due to not tuning the hyper-parameters correctly. So for the sake of fairness and to investigate the true advantages of Transformer, we present a comparison on recurrent Seq2seq model and Transformer in Table 8 using our implementation.[6]

| Methods | BLEU | METEOR | ROUGE-L |
|---|---|---|---|
| Seq2seq | 30.57 | 17.86 | 43.64 |
| Seq2seq w/ copy attention | 29.08 | 17.12 | 42.97 |
| Transformer | 31.08 | 18.57 | 44.31 |
| Transformer w/ copy attention | **31.38** | **18.69** | **44.68** |

Table 8: Comparison between recurrent sequence-to-sequence (Seq2seq) models and Transformer with copy attention mechanism on the Python dataset.

We can see from Table 8, the performance of the recurrent Seq2seq model is much better than the results reported in prior works. However, to our surprise, the copy attention mechanism does not result in improvement for the recurrent Seq2seq model. When we looked into the training perplexity and the validation performance, we also observed lower performance in comparison to the base recurrent Seq2seq model. In comparison, our proposed Transformer-based approach outperforms the recurrent Seq2seq models by a large margin showing its effectiveness for source code summarization.

[6] Our implementation is based on Open-NMT (Klein et al., 2017) and PyTorch 1.3.

## C Qualitative Examples

```
public static terminal find(String with_name) {
    if(with_name == null)
        return null;
    else
        return (terminal)all.get(with_name);
}
```

Base Model: lookup a non terminal by name string

Full Model w/o Relative Position: lookup a terminal terminal by name string

Full Model w/o Copy Attention: lookup a non terminal by name string

Full Model: lookup a terminal by name

Human Written: lookup a terminal by name string .

```
public static String selectText(XPathExpression expr, Node context) {
    try {
        return (String)expr.evaluate(context, XPathConstants.STRING );
    } catch (XPathExpressionException e) {
        throw new XmlException(e);
    }
}
```

Base Model: evaluates the xpath expression to a xpath expression .

Full Model w/o Relative Position: evaluates the xpath expression .

Full Model w/o Copy Attention Attention: evaluates the xpath expression as a single element .

Full Model: evaluates the xpath expression as a text string .

Human Written: evaluates the xpath expression as text .

```
public CTaggingPanel(
    final JFrame parent, final ZyGraph graph, final ITagManager manager) {
    super(new BorderLayout());
    mtagsTree = new CTagsTree(parent, graph, manager);
    final JScrollPane pane = new JScrollPane(mtagsTree);
    pane.setVerticalScrollBarPolicy(
        ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED);
    pane.setHorizontalScrollBarPolicy(
        ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
    add(pane);
    setBorder(new TitledBorder(new LineBorder(Color.LIGHT_GRAY, NUM, BOOL), STRING));
    setDoubleBuffered(BOOL);
}
```

Base Model: creates a new dnetscapesslservername dialog .

Full Model w/o Relative Position: creates a new settings dialog .

Full Model w/o Copy Attention: creates a new toolbar panel .

Full Model: creates a new api panel object .

Human Written: creates a new panel object .

```
public DSignCsr(JFrameparent, PKCS10CertificationRequest pkcs10Csr, File csrFile,
                PrivateKey signPrivateKey, KeyPairType signKeyPairType,
                X509Certificate verificationCertificate, Provider provider)
                throws CryptoException{
    super(parent, Dialog.ModalityType.DOCUMENT_MODAL);
    this.pkcs10Csr = pkcs10Csr;
    this.csrFile = csrFile;
    this.signPrivateKey = signPrivateKey;
    this.signKeyPairType = signKeyPairType;
    this.verificationCertificate = verificationCertificate;
    this.provider = provider;
    setTitle(res.getString(STRING));
    initComponents();
}
```

Base Model: creates a new dsigncsr dialog for a spkac formatted csr .

Full Model w/o Relative Position: creates a new signer dialog for a pkcs # 10 formatted .

Full Model w/o Copy Attention: creates a new dsigncsr dialog for a spkac formatted csr .

Full Model: creates a new dsigncsr dialog for a pkcs # 10 formatted csr .

Human Written: creates a new dsigncsr dialog for a pkcs # 10 formatted csr .

Table 9: Qualitative example of different models' performance in Java dataset.

```python
def get_hosting_service(name):
    try:
        return hosting_service_registry.get(u'hosting service id', name)
    except ItemLookupError:
        return None
```

Base Model: returns the color limits from the current service name .

Full Model w/o Relative Position: return the hosting service .

Full Model w/o Copy Attention: return the name of the service .

Full Model : return the hosting service name .

Human Written: return the hosting service with the given name .

```python
def save_pickle(obj, fname):
    with get_file_obj(fname, 'wb') as fout:
        cPickle.dump(obj, fout, protocol=-1)
```

Base Model: pickle object obj to file fname .

Full Model w/o Relative Position: save object to file .

Full Model w/o Copy Attention: raw_data: object obj to file fname .

Full Model : save object to file fname .

Human Written: save the object to file via pickling .

```python
def get_temp_dir:
    temp = get_environ_variable('TMP')
    if temp is None:
        temp = get_environ_variable('TEMP')
    if temp is None or '' in temp and os.name == 'nt':
        temp = 'C \\temp'
    if temp None or '' in temp and os.name == 'posix':
        temp = '/tmp'
    return temp
```

Base Model: returns the name of the sample environment variable .

Full Model w/o Relative Position: returns the next temporary directory of a file .

Full Model w/o Copy Attention: get the directory related to store the stubbed .

Full Model : return a temporary filename .

Human Written: returns a temporary directory .

```python
def get_exploration_memcache_key(exploration_id, version=None):
    if version:
        return 'exploration-version %s %s' % exploration_id, version
    else:
        return 'exploration %s' % exploration_id
```

Base Model: returns the key for an instance for the project .

Full Model w/o Relative Position: returns a memcache key for the given version .

Full Model w/o Copy Attention: returns a memcache for the exploration id .

Full Model : returns a memcache key for the specified exploration .

Human Written: returns a memcache key for an exploration .

```python
def get_svc_avail_path():
    return AVAIL_SVR_DIRS
```

Base Model: get the actual path .

Full Model w/o Relative Position: returns a list of services .

Full Model w/o Copy Attention: return a list of services that are available .

Full Model : returns a list of available services .

Human Written: return list of paths that may contain available services .

```python
def volume_attach(provider, names, **kwargs):
    client.get_client_info()
    client.extra_action(provider=provider, names=names, action='volume attach',
                        **kwargs)
    return info
```

Base Model: attempt to attach volume .

Full Model w/o Relative Position: attach volume cli example: .

Full Model w/o Copy Attention: attach volume cli example: .

Full Model : attach volume information cli example: .

Human Written: attach volume to a server cli example: .

Table 10: Qualitative example of different models' performance in Python dataset.