Automatic Code Documentation Generation Using Transformers

Sangam Man Buddhacharya Ambareesh Ramakrishnan Prashant Tandan
Suhas V. Sumukh
Oregon State University
{buddhacs, ramakria, tandanp, sumukhs}@oregonstate.edu

Abstract

In this work, we assessed the performance of a hybrid architecture in automatic code summarization task by combining an encoder trained on programming languages with a decoder trained on natural language. This hybrid architecture leverages multi-layer perceptrons (MLPs) to bridge the gap between code and English embeddings, enhancing the model's flexibility and performance. Our experiments show that training all layers of the model end-to-end on a code summarization dataset is crucial for achieving high performance. However, our model's summaries, though slightly different from reference labels and resulting in a lower BLEU score compared to our replicated CodeT5+, accurately summarized the function's task. Although our model does not surpass the CodeT5+ baseline, likely due to the limited size of our training dataset, it demonstrates the potential of incorporating larger datasets and further fine-tuning. This study underscores the importance of comprehensive training and paves the way for future enhancements in automated code documentation.

1. Introduction

Code documentation plays a pivotal role in modern software development, facilitating code comprehension, maintainability, and collaboration. However, the manual process of documenting code is often neglected due to time constraints and the prioritization of code implementation over documentation. This leads to incomplete, inconsistent, and poor-quality documentation, hindering code understanding and knowledge transfer within development teams (20).

This research focuses on leveraging transformer-based models to automate the generation of natural language documentation for source code. Specifically, we utilize CodeT5+ (23) as the encoder model for its robust understanding of programming languages, paired with a GPT-2 based decoder for generating high-quality English documentation. The encoder-decoder architecture is designed to combine the deep programming knowledge of the encoder with the linguistic proficiency of the decoder, much like pairing a skilled programmer with a proficient English speaker to produce comprehensive and accurate documentation.

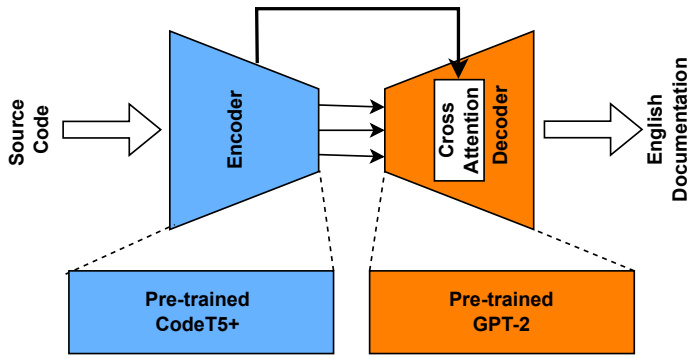
Existing approaches, such as Codex (5) and Code-T5 (24), have demonstrated the potential of transformer models in code summarization tasks. However, these models face limitations. Codex relies on GPT-3, which is costly and not fine-tunable for new programming languages, while Code-T5 may lack comprehensive language understanding due to its end-to-end training on code-summary datasets alone.

By automating code documentation, this research aims to save developers' time, improve code maintainability, facilitate better collaboration, and potentially reduce bugs and errors through enhanced documentation quality. This approach represents a significant advancement in the application of AI to software development, offering a scalable and efficient method for maintaining comprehensive code documentation across diverse codebases and programming languages.

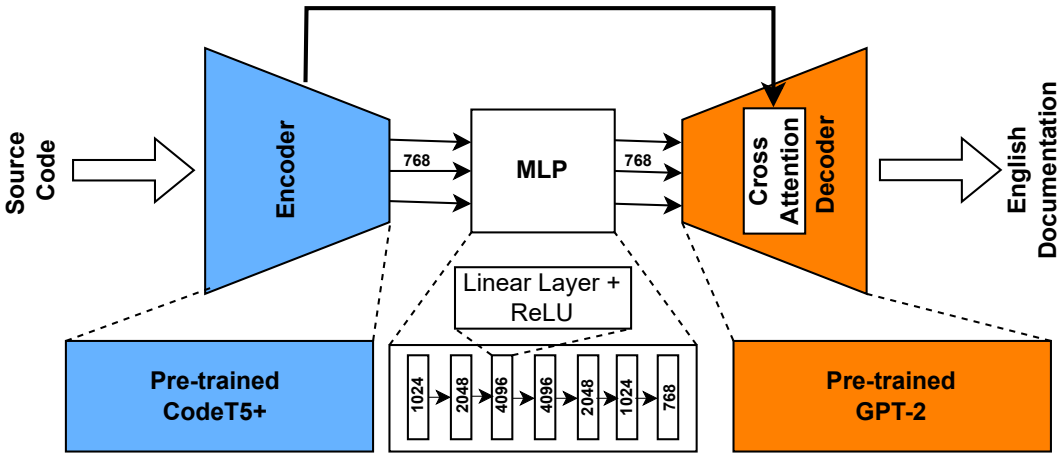
Our approach addresses these issues by adopting a modular training strategy where the CodeT5+ encoder is pre-trained on diverse code tasks using bimodal code-text corpora, and the decoder, a pre-trained GPT-2, is solely responsible for English translation.

The modular training of the encoder and decoder separately ensures that each component excels in its specialized task before integrating them. The pre-trained encoder captures intricate code representations, while the GPT-2 decoder, trained extensively on natural language corpora, translates these representations into coherent and meaningful documentation. This method promises to enhance both the understanding and generation capabilities of the model, ultimately leading to higher-quality code documentation (1).

Previous works, such as CodeT5+ (23) and PLBART (2), have explored similar encoder-decoder architectures for code documentation generation. However, our approach differs in the modular training strategy and the utilization of a bimodal data pre-trained encoder, aiming to enhance



(a) Model without linear layers between encoder and decoder



(b) Model with linear layers between encoder and decoder

Figure 1. Our architecture with and without multiple linear layer.

the model's understanding of both programming languages and natural language.

Initial experiments using the CodeXGLUE benchmark (15), a widely adopted dataset for code summarization and documentation generation, and BLEU score evaluations suggest that our approach is effective. The results indicate that the embeddings generated by the encoder when processed by the GPT-2 decoder, produce documentation that is both accurate and contextually relevant. This dual-focus training method leverages the strengths of both models, addressing the shortcomings of existing end-to-end training approaches and offering a promising solution to the challenges of automated code documentation (14).

2. Related Work

Automating code documentation generation has been an active area of research, with various approaches explored in prior work. In the early efforts from 2010 to 2014, the majority of code commenting and summarization systems relied on information retrieval methods. However, in the past nine years, most of these systems have predominantly utilized deep neural networks. Given a piece of source code without comments and a dataset containing source code with comments, information retrieval algorithms initially assess the similarity between the target code and the dataset. Code is first transformed into an abstract syntax tree representation. The algorithm then identifies one or more pieces of code from the dataset that closely match the target code, and uses their comments to generate comments for the target program (20). Other rule-based or template-

based systems used heuristics and patterns to extract information from source code and generate documentation (11; 21). However, these approaches were limited in their ability to produce natural language descriptions and often resulted in rigid, unnatural documentation.

The attention mechanism is almost always found in the encoder-decoder frameworks today. Before transformer architectures became popular, reinforcement learning (RL) was seen as a promising paradigm for code summarization due to its ability to learn without explicit labels and its flexibility in handling complex tasks. However, as research progressed, it became evident that RL's application in code summarization faced challenges, such as the difficulty in defining suitable rewards and the complexity of incorporating RL into existing neural network frameworks. This realization has led to a reevaluation of RL's suitability for code summarization tasks, favoring simpler and more direct approaches that leverage the power of neural networks (22)(5).

Effective source code modeling enhances the extraction of both syntactic and semantic information. Common approaches include token-based, tree-based, and graph-based methods (25). However, we argue that a robust model should independently determine semantic relationships like data dependencies and control flow among tokens. Therefore, we have opted for token-based modeling.

We've utilized pre-trained models built on Transformer architectures that have achieved leading performance across various NLP tasks. They can be broadly categorized into three main architectures: encoder-only, decoder-only, and encoder-decoder models.

2.1. Encoder-only Models

These models, exemplified by CodeBERT (7) and Unixcoder (10), leverage pre-trained language models to encode source code into rich contextual representations. While effective for code understanding tasks like code retrieval (13) and code summarization, these models lack the capability to generate natural language documentation directly.

2.2. Decoder-only Models

On the other hand, decoder-only models, such as GPT-3 (4), Codegen2 (16), Code Llama (19), Codex (5), and InCoder (8), have demonstrated impressive performance in code generation tasks, including code synthesis and code completion. These models are trained on large corpora of code and can generate code or documentation in an autoregressive manner. However, they may lack the bidirectional context understanding provided by encoder-decoder architectures. For example, Code Llama, which fine-tunes the general-purpose Llama 2 model, relies only on its substantial size and raw computational power to understand programming concepts broadly.

2.3. Encoder-Decoder Models

Encoder-decoder models, such as CodeT5 (24), CodeTransformer (1), and UniXcoder (10), combine the strengths of both encoder and decoder components. They leverage the bidirectional context understanding of the encoder and the generation capabilities of the decoder, making them suitable for both code understanding and generation tasks. However, as noted by Wang et al. 2021b(24) and Ahmad et al. [2021](1), these models do not always outperform specialized encoder-only or decoder-only models. UniXcoder(10) adopts a UniLM-style design (6) to support various tasks by manipulating input attention masks, but suffers from inter-task interference, leading to performance degradation on sequence-to-sequence tasks like code generation. While UniXcoder and CodeT5 use code-specific features, our approach based on CodeT5+ does not.

Much work on PLBART (1), based on BART, also uses an encoder and an autoregressive decoder. While it is pre-trained in programming languages, unlike our approach, it isn't pre-trained in natural languages. Our approach shares similarities with encoder-decoder models by employing an encoder-decoder architecture but introduces a novel modular training strategy and leverages Code-T5+ (23) for encoder pre-training. By pre-training the encoder on diverse code tasks using Code-T5+, we aim to imbue the encoder with a comprehensive understanding of programming languages. This contrasts with end-to-end training approaches like CodeT5 (24), which may lack a thorough understanding of both programming languages and natural language. Additionally, our approach leverages the pre-trained GPT-2 model (18) as the decoder, harnessing its robust language understanding capabilities. This modular approach, where the encoder and decoder are trained separately on their respective specialized tasks before integration, distinguishes our work from previous efforts.

While recent works like CodeT5+ (23) have explored encoder-decoder architectures for code documentation generation, our approach differs in its modular training strategy and the utilization of Code-T5+ for encoder pre-training. CodeT5+ employs a shallow encoder and freezes the deep decoder language model during training to improve computational efficiency. However, our method keeps the pre-trained GPT-2 decoder trainable to allow for fine-tuning on the code documentation task. Another related work is the research on parameter-efficient training of large language models (12; 17), which aims to scale up models using limited computational resources. We adopt a common strategy to freeze a large part of the pre-trained model and only train a small number of additional parameters. In our approach, we adopt a similar "shallow encoder and deep decoder" architecture (9), freezing the deep pre-trained GPT-2 decoder while keeping the smaller encoder and cross-attention layers trainable.

In summary, our work builds upon recent progress in encoder-decoder models for code processing while exploring a novel modular training strategy. Through extensive pre-training of specialized encoders for code understanding (CodeT5+) and decoders for natural language generation (GPT-2), we aim to leverage the best capabilities of each component to generate high-quality code documentation that accurately captures the semantics and functionality expressed in the source code.

3. Methodology

3.1. Our Architecture

As shown in the figure 1, we employ a sequence-to-sequence transformer-based encoder-decoder architecture, utilizing CodeT5+ (23) as the encoder and GPT-2 (18) as the decoder. The pretrained encoder model, CodeT5+ (23), is leveraged for its training on diverse code generation and code detection tasks, while the decoder, GPT-2 (18), is pre-trained on a large corpus of English text. As illustrated in figure 1, we tested two distinct architectures: one that includes multiple linear layers (MLP) between the encoder and decoder, and another without the MLP. The MLP consists of seven sequentially stacked linear layers. It initially increases the dimensionality from 768 to 1024, 2048, and 4096 before reducing it back to the original dimension of 768, which is compatible with the GPT-2 (18) input. We applied ReLU activation after each linear layer. The MLP is added after the encoder's final dropout layer to learn a new embedding space suitable for the GPT-2 (18) decoder input. Additionally, because GPT-2 (18) lacks a cross-attention mechanism, we explicitly integrated cross-attention into the decoder.

3.2. Datasets

In our research, we utilized the CodeXGlue (15) which is a standard benchmark dataset created by Microsoft Research to evaluate the code intelligence for multiple programming languages and diverse tasks. For our task, we used 251,820 training, 13,914 validation, and 14,918 test samples of Python code with docstrings. Specifically, we concentrated on the Python code summarization dataset, sourced directly from the Hugging Face dataset hub. This dataset contains both code and docstring tokens, which we utilized as-is. For tokenization, we employed the CodeT5+ tokenizer for the code tokens and the GPT-2 (18) tokenizer for decoding the target labels. We set the pad_token_id, eos_token_id, and decoder_start_token_id to 50256.

3.3. Experiments

We have performed different experiments on the base model and also tried to replicate the CodeT5+ (23) paper. Furthermore, We conducted experiments by freezing different modules of the model to reduce the number of trainable parameters and prevent the disruption of useful representations learned in previous tasks, known as catastrophic forgetting.

3.3.1 Base CodeT5+ model

To evaluate the code intelligence capacity of CodeT5+ (23), we calculated the BLEU score on the code summarization testing dataset using the base CodeT5+ bimodal (220M) model, which was pre-trained on code generation and code detection tasks, excluding code summarization.

3.3.2 Replicated CodeT5+ (23)

We attempted to replicate the CodeT5+ model as described in the paper (23). To construct the architecture, we used the base model (220M bimodal), pre-trained on various code generation tasks as a starting model. Utilizing the Sequence-to-Sequence encoder-decoder model from HuggingFace, we trained it for the code summarization task, maintaining the same settings as our architecture to ensure a fair comparison.

3.3.3 CodeT5+ encoder (frozen) and GPT-2 decoder with MLP

In this setting, we froze only the encoder section and trained both the MLP and the decoder layer end-to-end on the code summarization task.

3.3.4 CodeT5+ encoder (frozen) and GPT-2 decoder (frozen) with MLP

In this setting, we froze both the encoder and decoder layers and trained only the MLP and cross-attention layers.

3.3.5 CodeT5+ encoder and GPT-2 decoder with MLP

In this setting, we trained the entire model with MLP end-to-end without freezing any layers of the model.

3.3.6 CodeT5+ encoder and GPT-2 decoder without MLP

In this setting, we trained the entire model end-to-end without freezing any layers of the model. This model does not contain an MLP layer between the encoder and decoder.

3.4. Evaluation Metrics

As Smoothed BLEU has become a standard metric for code documentation evaluation and it enables consistent benchmarking and comparison across different systems, we

Replicated base models	BLEU score
Base CodeT5+ model	13.47
CodeT5+ fine-tuned on code summarization (Replicated)	19.32
CodeT5+ fine-tuned on code summarization(Wang et al.) (23)	**20.16**

Table 1. BLEU-4 score for base models. The score increased when the model was fine-tuned on code summarization task specifically. The best result is bolded.

Experimental models	BLEU score
CodeT5+ encoder (frozen) and GPT-2 decoder (frozen) with MLP	10.23
CodeT5+ encoder (frozen) and GPT-2 decoder with MLP	14.38
CodeT5+ encoder and GPT-2 decoder without MLP	15.65
CodeT5+ encoder and GPT-2 decoder with MLP	**17.89**

Table 2. BLEU-4 score for each of the experiments. CodeT5+ encoder with GPT-2 decoder performs better when there is a block of multi-layer perceptron in between. However, it doesn't perform better than the CodeT5+ model fine-tuned on code summarization task. The best result is bolded.

use BLEU-4 as the performance metric, which assesses the token-based similarity between the predicted and ground-truth summaries from the dataset. We calculated this score for all of the experiments conducted.

$$matches_n = \sum_{ngram \in T} \min(\text{Count}_T(\text{ngram}), \text{Count}_R(\text{ngram})) \tag{1}$$

$$possible_n = \sum_{ngram \in T} \text{Count}_T(\text{ngram}) \tag{2}$$

$$p_n = \frac{matches_n + 1}{possible_n + 1} \tag{3}$$

where T is the model-generated memorization and R is the ground truth memorization.

$$\text{GeoMean} = \exp\left(\frac{1}{N}\sum_{n=1}^{N} \log p_n\right) \tag{4}$$

$$\text{BP} = \begin{cases} 1 & \text{if } \frac{c}{r} > 1 \\ \exp\left(1 - \frac{r}{c}\right) & \text{if } \frac{c}{r} \leq 1 \end{cases} \tag{5}$$

$$\text{BLEU} = \text{GeoMean} \times \text{BP} \tag{6}$$

For our setting, we use ngram equal to 4.

3.5. Implementation

We implemented our proposed model architecture using the PyTorch framework and the HuggingFace library. We utilized the Weights & Biases (3) logging platform throughout our experiments to track and monitor progress every 10 iterations. The batch size was fixed at 8, employing a decaying learning rate initialized at $1e^{-6}$. We set maximum source and target sequence lengths to 320 and 128, respectively. Each experiment was trained for at least 100 epochs, leveraging 4 x V100-SMX3-32GB GPUs for computational acceleration for 10 days. For calculating a BLEU score, we used the model with the best validation loss, with n-gram equal to 4.

4. Results

We trained our models in the experimental settings explained above. Then, we calculated the BLEU-4 score for all of the experiments conducted.

4.1. Replicated CodeT5+ Model

The results for the replicated base models are presented in Table 1. The base CodeT5+ bimodal model (220 million parameters), achieved a BLEU score of 13.47, demonstrating its code intelligence capacity. When fine-tuned on the code summarization task using the CodeXGLUE dataset, the BLEU score significantly improved to 19.32. The original paper reported a BLEU score of 20.16. The discrepancy in scores may be due to differences in our hyperparameter settings in the paper. Fine-tuning the hyperparameters more closely with those in the original paper could potentially achieve the reported score.

4.2. Our Experimental Models

The experimental results of our architecture are detailed in Table 2. Initially, we achieved a BLEU score of 15.65 using a model that combined a CodeT5+ encoder with a GPT-2 decoder, trained without freezing any layers. This

Example 1

Input (Code)

```
def get_url_args(url):
    url_data = urllib.parse.urlparse(url)
    arg_dict = urllib.parse.parse_qs(url_data.
        query)
    return arg_dict
```

Output(Summaries)

Ground Truth	*Returns a dictionary from a URL params*
Finetuned CodeT5+	*Parse url and return a dictionary of arguments*
Our Best Experimental Model	*Generates a dictionary using URL query parameters.*

Example 2

Input (Code)

```
def _warn(self,msg ):
    self._warnings.append(msg)
    sys.stderr.write("Coverage.py warning: %s\n" %
        msg)
```

Output(Summaries)

Ground Truth	*Use msg as a warning.*
Finetuned CodeT5+	*Record a warning.*
Our Best Experimental Model	*Add msg to warning.*

Table 3. Example input-output pairs for replicated fine-tuned model and the best model from our experiments i.e. CodeT5+ encoder and GPT-2 decoder with MLP.

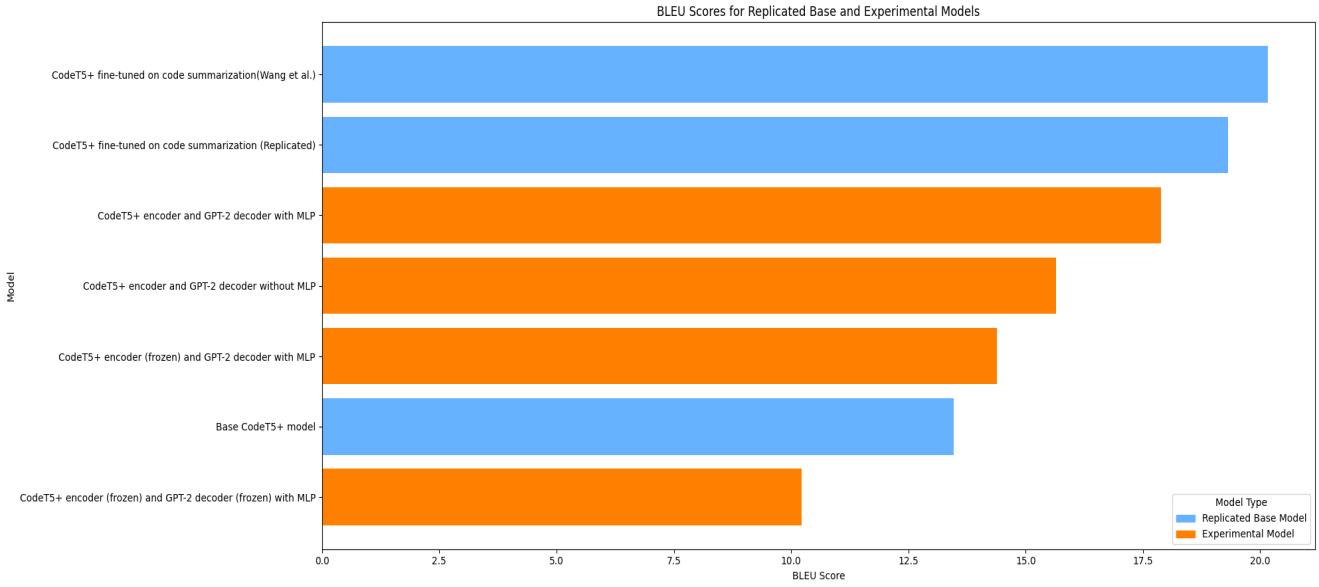


Figure 2. Comparison of all models.

score improved to 17.89 after incorporating an MLP layer between the encoder and decoder. Thus, adding a multi-layer perceptron after the encoder proves to be more effective than directly combining encoders and decoders from two different architectures.

Moreover, when both the encoder and decoder were frozen, and only the MLP layer and the cross-attention head of the decoder were trained, the BLEU score dropped significantly to 10.23. This indicates that simply converting the embeddings from the encoder suitable for the decoder input using multiple linear layers is insufficient. Similarly, freezing the encoder while training only the MLP layer and decoder resulted in a BLEU score of 15.56. These findings suggest that training the entire model end-to-end is more effective for downstream tasks such as code summarization.

4.3. Replicated CodeT5+ vs. Our Experimental models

From Tables 1 and 2, we can clearly see that the models that were not using any intermediate multi-layer perceptrons or had either encoder or decoder frozen during the training didn't compare to the CodeT5+ replicated results. However, the model that combines the encoder and decoder architectures with a block of multi-layer perceptrons and trains all layers end-to-end on the code summarization dataset has achieved a BLEU score comparable to the baseline CodeT5+ model. The performance of CodeT5+ is better than our architecture. This might be the case as both the encoder and decoder of CodeT5+ are being trained on a large code dataset for different code generation and code detection tasks, but our GPT2 decoder is only being fine-tuned on a small Python dataset, so the available dataset from CodeXGlue might be insufficient to fully train the GPT2 decoder in a combination of encoders of CodeT5+. Incorporating other larger datasets might improve the performance of our architecture.

5. Conclusion

We conducted various experiments to evaluate our proposed architecture, which combines an encoder (code expert) model with a decoder (English language expert) model for the code summarization task under different settings. Our findings indicate that incorporating MLP layers between the encoder and decoder significantly improves performance. The MLP layer enhances the model's flexibility by converting embeddings from the encoder into a format more compatible with the decoder. Furthermore, experimental results demonstrated that training all layers (encoder, MLP, and decoder) of the network end-to-end on the code summarization dataset is crucial for optimal performance. Training only the MLP layer to transform embeddings from the encoder (CodeT5+ trained on code summarization) into inputs for the decoder (trained on English language) is insufficient for effective code summarization. We also observed that the decoder (GPT-2) (only trained with Language Corpus) is required to have some code understanding, although its task is to generate English summarization.

However, while our model generated a summary that was slightly different from the reference label, resulting in a lower BLEU score compared to the original CodeT5+ paper, it still accurately summarized the code in relation to the function's task.

We found a few limitations in our approach during this project. Firstly, it requires training all the layers of the model to get better performance, increasing the resources to train the model efficiently. Secondly, since the GPT2 model is only trained on a large corpus of English text, it is required to train the GPT2 model along with the CodeT5+ encoder model on a large source code-documentation pair to make the GPT2 model familiar with the source code. We could not train the model in these settings due to a lack of available resources and time constraints. However, incorporating the above-mentioned training could improve the performance of our architecture, comparable to or beyond the BLEU score of the current CodeT5+ model.

This project explores the future possibility of adding multiple self-attention heads in place of multiple linear layers in order to improve performance, as our results show improved performance after incorporating the MLP layer.

References

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*, 2020. 1, 3

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation, 2021. 1

[3] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com. 5

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020. 3

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. 1, 3

[6] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. Unified language model pre-training for natural language understanding and generation. *Advances in neural information processing systems*, 32, 2019. 3

[7] 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. 3

[8] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022. 3

[9] Linyuan Gong, Di He, Zhuohan Li, Tao Qin, Liwei Wang, and Tieyan Liu. Efficient training of bert by progressively stacking. In *International conference on machine learning*, pages 2337–2346. PMLR, 2019. 3

[10] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022. 3

[11] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*, pages 35–44, 2010. 3

[12] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021. 3

[13] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019. 3

[14] Junaed Younus Khan and Gias Uddin. Automatic code documentation generation using gpt-3. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–6, 2022. 2

[15] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021. 2, 4

[16] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*, 2023. 3

[17] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022. 3

[18] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019. 3, 4

[19] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023. 3

[20] Xiaotao Song, Hailong Sun, Xu Wang, and Jiafei Yan. A survey of automatic generation of source code comments: Algorithms and techniques. *IEEE Access*, 7:111411–111428, 2019. 1, 2

[21] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the 25th IEEE/ACM international conference on Automated software engineering*, pages 43–52, 2010. 3

[22] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 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, 2018. 3

[23] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. CodeT5+: Open code large language models for code understanding and generation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088, Singapore, Dec. 2023. Association for Computational Linguistics. 1, 3, 4, 5

[24] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021. 1, 3

[25] Chunyan Zhang, Junchao Wang, Qinglei Zhou, Ting Xu, Ke Tang, Hairen Gui, and Fudong Liu. A survey of automatic source code summarization. *Symmetry*, 14(3):471, 2022. 3