Exploring Data Augmentation for Code Generation Tasks

Pinzhen Chen[*]
School of Informatics
University of Edinburgh
`pinzhen.chen@ed.ac.uk`

Gerasimos Lampouras
Noah's Ark Lab
Huawei
`gerasimos.lampouras@huawei.com`

Abstract

Advances in natural language processing, such as transfer learning from pre-trained language models, have impacted how models are trained for programming language tasks too. Previous research primarily explored code pre-training and expanded it through multi-modality and multi-tasking, yet the data for downstream tasks remain modest in size. Focusing on data utilization for downstream tasks, we propose and adapt augmentation methods that yield consistent improvements in code translation and summarization by up to 6.9% and 7.5% respectively. Further analysis suggests that our methods work orthogonally and show benefits in output code style and numeric consistency. We also discuss test data imperfections.

1 Introduction

Recent years have seen the rapid development of pre-trained models (PLMs) to enable knowledge transfer from generic texts to specific downstream tasks (Devlin et al., 2019; Liu et al., 2019). PLMs have been applied to the programming language domain as well, following the same paradigm of (continuing) training PLMs on code and text data, and then fine-tuning them for specific tasks (Kanade et al., 2020; Feng et al., 2020). PLMs are often adapted to programming languages by including code-specific modalities as part of the input like serialized syntax trees and data flows (Guo et al., 2021, 2022; Tipirneni et al., 2022). Such works have outperformed rule-based tools in various tasks, e.g. the CodeXGLUE benchmark (Lu et al., 2021).

Despite the abundance of raw code available for pre-training, code data that meet downstream needs stay modest in size. This is due to the fact that, unlike texts, code datasets cannot be easily curated by people without programming knowledge. For example, code translation data in CodeXGLUE is sized at 10K, which is orders of magnitude smaller than their natural language counterparts that often include millions of instances (Kocmi et al., 2022).

We are therefore motivated to enrich data in the fine-tuning phase of code PLMs, using automatic data augmentation (DA) methods like back-translation, monolingual, multilingual, and numeric augmentation. We extensively experiment on code translation, where a programming language is converted to another, and summarization, where a textual description is produced from a code block. Even with limited resources, we can lift performance by 6.9% for translation and 7.5% for summarization compared to baselines. Through manual inspection and extra evaluation measures, we demonstrate that our methods lead to desirable enhancements special to code, namely better output code style and number correctness.

2 Methodology

2.1 Data synthesis

Back-translation (BT, Sennrich et al., 2016) is a data augmentation technique originated from machine translation, where an auxiliary model is used to construct pseudo-parallel data from monolingual resources. It can be straightforwardly applied to code translation. Formally, to train a model $f()$ that converts a programming language PL_x into PL_y, we first train an inverse model $g(PL_y) \to PL_x$ with the same parallel data. Having the inverse model $g()$, extra monolingual data in PL_y is translated into PL'_x to form pseudo-parallel pairs PL'_x-PL_y that can be used to train $f()$.

For code summarization, back-translation is not applicable as "monolingual" natural language (NL) summaries unaligned to code hardly exist. Hence we propose to use the summaries originally associated with a single programming language as a pivot for other programming languages. After

[*]Work done during an internship at Huawei Noah's Ark Lab. Our code will be available at https://github.com/huawei-noah/noah-research/tree/master/NLP/DA4CodeGeneration

inversing code-to-text data which has source side code available in multiple programming languages ($PL_1 \rightarrow NL, \dots, PL_n \rightarrow NL$), we train a multilingual text-to-code generator, which outputs a designated programming language given a natural language summary and a target language tag ($NL + tag_{\{1,\dots n\}} \rightarrow \{PL_1, \dots, PL_n\}$). This generator can iteratively produce code in different PLs by inputting summaries regardless of the original $PL \rightarrow NL$ alignment. These synthesized data, despite having a lower quality, can augment the training data for summarization.

2.2 Utilization of multilinguality

Currey et al. (2017) suggested that including monolingual data in the target language as an additional autoencoding (AE) objective benefits translation models trained on limited data. We migrate this objective to code translation by mixing $PL_x \rightarrow PL_y$ and $PL_y \rightarrow PL_y$ data. This effectively builds a multilingual encoder that enables knowledge transfer, given the high similarity between programming languages, namely the overlap of numerals, syntax tokens, reserved keywords, etc. This process constrains the decoder side to a single programming language PL_y to not add complexity.

In code summarization, as the target NL is fundamentally divergent from the input PL, the autoencoding objective might not be useful. In contrast, we train a "multilingual" code summarization model $\{PL_1, \dots, PL_n\} \rightarrow NL$ where the system takes an arbitrary programming language to produce a natural language summary. Such a many-to-one model allows encoder knowledge sharing too and exposes the decoder to more NL summaries.

2.3 Numeric awareness

Referenced variables and their values are unique components of programming languages; to enhance understanding of these values, previous works on pre-training suggested attending to appropriate modalities, e.g. data flow (Guo et al., 2021). Such sophisticated handling of values might not be necessary for code translation, as copying them over to the target suffices. However, given a small training size, any translation model will still only be exposed to sparse numerical input. To increase model robustness, we augment the data by creating new instances where, in all code tokens containing a number, each digit is randomly replaced with another digit, consistently on both the source and target sides. We do not distinguish between purely numerical tokens and tokens including a number. For instance, a variable "num1" could become "num4" in the augmented code pair. The method guarantees that the number-swapped synthetic code is grammatical and compilable.

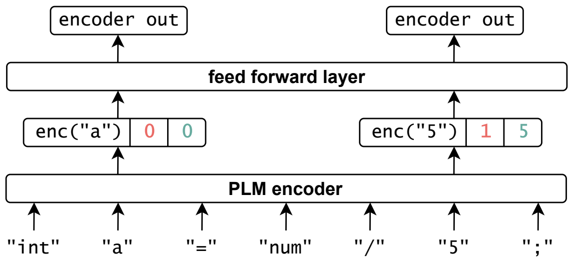

Figure 1: Numeric encoding with a PLM encoder, exemplifying how "a" and "5" are encoded differently.

Apart from numerical augmentation, we propose to include input numbers directly in the encoder output as mathematical values, complementary to their string embedding representations. As illustrated in Figure 1, we append two dimensions to the original encoder output. Particularly, one dimension (red, left) is a binary value (0/1) indicating whether the respective input is a number, while the other dimension (green, right) inherits the input's value, or 0 if the input is not numeric. The expanded embedding can be reduced to its original size via a feed-forward layer; such a change requires no modification to the pre-trained encoder.

3 Experiments

3.1 Tasks, datasets and evaluation

We benchmark our methods on the code task suite CodeXGLUE (Lu et al., 2021). Its translation task uses code originally developed in Java and then migrated to C#, so the corresponding C#-Java snippets are considered parallel. Training, validation, and test sizes are 10K, 0.5K, and 1K. For back-translation, we translated 377K lines of monolingual Java, albeit out-of-domain, from other CodeXGLUE tasks, into C#. To ensure that the target side consists of genuine data, we only experimented with the C#→Java direction as there is no other C# code in the benchmark for BT.

The summarization task employs CodeSearchNet (Husain et al., 2019) and covers six languages: Ruby, JavaScript, Go, Python, Java, and PHP. Training sizes range from 25K to 250K, totalling 908K; validation and test sets are between 1K and 15K. We performed multilingual back-translation by re-

	BLEU	EM	CodeBLEU[†]
CodeBERT			
paper	72.14	58.0	-
replicate	72.92	57.4	78.93 (72.92 / 73.61 / 87.08 / 82.10)
BT	77.34	61.4	83.36 (77.34 / 78.11 / **90.34** / 87.64)
+ AE	**77.60**	**61.8**	**83.47** (**77.60 / 78.30** / 90.02 / **87.96**)
GraphCodeBERT			
paper	72.64	58.8	-
replicate	72.66	58.9	78.55 (72.66 / 73.35 / 87.44 / 80.74)
BT	75.15	60.7	82.13 (75.15 / 75.86 / 90.06 / 87.46)
+ AE	**76.15**	**62.5**	**82.88** (**76.15 / 76.87 / 90.54 / 87.95**)

[†]average (n-gram / weighted n-gram / syntax / data flow)

Table 1: Test results for C#→Java translation.

	Ruby	JS	Go	Py	Java	PHP	Avg.
CodeBERT							
paper	12.16	14.90	18.07	19.06	17.65	25.16	17.83
monolingual	12.39	14.13	17.89	18.22	18.66	25.14	17.73
+ rule-trans	-	15.35	-	-	-	-	-
+ BT	13.76	15.00	18.30	18.60	19.64	25.69	18.50
multilingual	**14.93**	15.53	18.68	18.71	19.70	25.96	18.92
+ rule-trans	14.58	15.65	18.77	18.95	**19.86**	25.98	18.97
+ BT	14.91	**15.81**	**18.88**	**18.97**	19.69	**26.10**	**19.06**
UniXcoder							
paper	14.87	15.85	19.07	19.13	20.31	26.54	19.30
monolingual	14.81	15.28	18.93	19.05	20.22	26.66	19.16
multilingual	**15.15**	15.64	19.03	19.22	**20.45**	26.59	19.35
+ BT	14.94	**15.85**	**19.29**	**19.36**	20.43	**26.69**	**19.43**

Table 2: Test results for code summarization in BLEU.

versing the dataset so no external data is introduced; this leads to a five-fold BT data of 4.5M (908K×5). All programming languages share an equal amount of original and synthetic data combined. Moreover, to compare the quality of neural back-translation against hand-written rule-based conversion, we created 80K JavaScript-summary pairs from Python-summary data using `jsbuilder`.

We report code translation results in BLEU-4 (Papineni et al., 2002), exact line matches (EM, in %), and CodeBLEU, a weighted sum of four accuracies: n-grams, weighted n-grams, syntax, and data flow (Ren et al., 2020). Code summarization performance is measured by the de facto choice of BLEU-4 on natural language texts.

3.2 Systems

For all tasks, we use the CodeXGLUE baseline, i.e. CodeBERT with a Transformer decoder, for our base and inverse models (for data synthesis). We continue training PLMs on the augmented data, then fine-tune on the original data, except for numeric augmentation where we mix the synthetic data with the training set. Monolingual and multilingual summarization experiments share the same configurations. For numeric encoding with CodeBERT, we add a feed-forward layer to make the baseline as deep as our proposed network.

To provide results with stronger baselines, we also test with GraphCodeBERT (Guo et al., 2021) for translation and UniXcoder (Guo et al., 2022) for summarization. This helps to verify the stability of data augmentation performance across distinct PLM architectures. We stick to the relevant PLMs' hyperparameters except for batch size. Model and training details, with links to the preprocessing and evaluation scripts, can be found in Appendix A.

3.3 Results and Discussions

We first show translation results in Table 1, where back-translation surpasses baselines by a large margin for both PLMs; on top of it, autoencoding brings a small gain. Table 2 indicates that back-translation also steadily helps code summarization overall. An interesting pattern from both PLMs is that BT helps Ruby and Java less than other languages. Furthermore, learning a single multilingual model is better than learning separate monolingual models, potentially due to transfer learning between programming languages and the increase in natural language data size on the output side.

Table 3 reports the results for numeric augmentation and numeric encoding in translation. Adding number-swapped data to training surpasses the baseline, while our numeric encoding proposal under-performs the baseline. To accommodate the neural network weights which are orders of magnitude smaller than the variable values encountered in code, we investigate linear and logarithmic value scaling. As the scaling gets smaller, result numbers gradually catch up; the optimal is a logarithmic transformation, whereby the model attains the highest performance.

To directly assess our value-aware augmentation, we compute and append output token accuracies to Table 3, with a distinction between numeric and non-numeric tokens. We can observe that the numerical approaches aid number generation without compromising non-numbers, and the improvement in number correctness is generally consistent with the improvement in BLEU and EM. Additional visualization in Appendix B.2 implies that DA models can maintain numeric consistency even when the output is extremely long and complicated.

	BLEU	**EM**	**CodeBLEU**	**Token Accuracy**	
				numeric	**non-numeric**
CodeBERT + FFN	72.88	58.0	78.07 (72.88 / 73.66 / 86.15 / 79.59)	74.50	86.72
+ numeric augmentation	**74.00**	**59.5**	**79.43** (**74.00** / **74.72** / **87.01** / **82.00**)	**76.14**	**87.30**
numeric encoding	72.95	58.1	78.77 (72.95 / 73.74 / 86.96 / 81.45)	73.74	86.84
+ numeric augmentation with value scaling					
$\times 10^{2}$	71.32	51.6	77.71 (71.32 / 72.25 / 86.22 / 81.05)	72.48	85.98
$\times 1$ (no scaling)	72.51	57.4	78.45 (72.51 / 73.38 / 86.47 / 81.46)	72.92	86.49
$\times 10^{-2}$	73.48	**59.2**	79.41 (73.48 / 74.28 / 87.31 / 82.56)	74.11	87.11
$\times 10^{-4}$	74.01	58.9	79.73 (74.07 / 74.75 / 87.29 / 82.87)	74.93	**87.48**
$\log_{10}()$	**74.16**	59.1	**79.84** (**74.16** / **74.91** / **87.39** / **82.90**)	**75.22**	87.32

Table 3: Test results for C#→Java translation with numeric augmentation and encoding.

	BLEU	**EM**	**CodeBLEU**	**Token Accuracy**	
				numeric	**non-numeric**
CodeBERT replicate	72.92	57.4	78.93 (72.92 / 73.61 / 87.08 / 82.10)	74.64	87.54
BT	77.34	**61.4**	83.36 (77.34 / 78.11 / 90.34 / 87.64)	78.09	88.62
+ num. aug. original only	**77.69**	61.0	**83.44** (**77.69** / **78.33** / 90.19 / 87.56)	**78.54**	**88.69**
+ num. aug. BT and original	77.37	60.9	83.43 (77.37 / 78.07 / **90.36** / **87.94**)	77.16	88.55
BT + AE	77.60	61.8	83.47 (77.60 / 78.30 / 90.02 / **87.96**)	77.16	88.64
+ num. aug. original only	**77.96**	**62.0**	**83.63** (**77.96** / **78.62** / **90.15** / 87.82)	**78.01**	**88.79**

Table 4: Test results for C#→Java translation with multiple augmentation techniques.

Finally, Table 4 examines if the above methods, namely back-translation and numeric augmentation, work orthogonally. It is observed that better results are achieved when numeric augmentation is applied to the original data, but not to the back-translated data. This is probably because BT is already of inferior quality, so numerical augmentation introduces extra noise. Nevertheless, combining BT and AE with numeric augmentation over the original data leads to the best outcome.

4 Analysis

Upon inspecting the translation test outputs, we find that our data-augmented model is better exposed to the target Java language: it has learned the Java programming conventions instead of following the input code style. We present test instances focused on element retrieval methods, by listing sources, references, and outputs from the CodeBERT baseline and our BT-augmented model in Table 5. Whilst direct retrieval of an element through reference to its position is possible in Java, we observe that the baseline tends to imitate the code style in source C#, but the DA model closely follows the Java coding convention where the inbuilt method `get()` is favoured over directly accessing the attributes by indices.

We should note that in the translation test set a small proportion of code pairs seem to be divergent, which can lead to an inaccurate estimate of translation performance. We record a few examples of these imperfections in Appendix B.1, but leave in-depth investigation and refinement for future work.

5 Related Works

Recent research at the intersection of natural language processing and programming languages concentrated on pre-training. Kanade et al. (2020) trained CuBERT to obtain embeddings for code understanding tasks. Feng et al. (2020) developed CodeBERT by training RoBERTa on bimodal text-code data with replaced token detection (Clark et al., 2020). In GraphCodeBERT, Guo et al. (2021) incorporated data flow edge prediction and data-variable alignment. Researchers expanded decoder-only models to the code domain too, e.g. CodeGPT, Codex, and Pangu-Coder (Lu et al., 2021; Chen et al., 2021; Christopoulou et al., 2022). Universal encoder-decoder code PLMs have also been presented: PyMT5, CodeT5, PLBART, UniXcoder, and StructCoder (Clement et al., 2020; Wang et al., 2021; Ahmad et al., 2021; Guo et al., 2022; Tipirneni et al., 2022). UniXcoder, which we used, adopts attention masks to control encoder-decoder behaviours in a shared encoder-decoder network.

Datasets for specific tasks concerning code are usually small, so data augmentation can help to boost performance. Roziere et al. (2020) combined

```
// test #85
C# source      ... GetEscherRecord(int index){return escherRecords[index];}
Java reference ... getEscherRecord(int index){return escherRecords.get(index);}
baseline       ... getEscherRecord(int index) {return escherRecords[index];}
DA model       ... getEscherRecord(int index) {return escherRecords.get(index);}

// test #90
C# source      public virtual IQueryNode GetChild(){return GetChildren()[0];}
Java reference public QueryNode getChild() {return getChildren().get(0);}
baseline       public QueryNode getChild() {return getChildren() == 0);}
DA model       public QueryNode getChild() {return getChildren().get(0);}

// test #978
C# source      public virtual SrndQuery GetSubQuery(int qn) { return m_queries[qn]; }
Java reference public SrndQuery getSubQuery(int qn) {return queries.get(qn);}
baseline       public SrndQuery getSubQuery(int qn) {return queries[qn];}
DA model       public SrndQuery getSubQuery(int qn) {return queries.get(qn); }
```

Table 5: C#-Java output translations of element retrieval methods, before and after data augmentation.

cross-lingual masked modelling and iterative back-translation to build an unsupervised code transcompiler. Ahmad et al. (2022) ran code-to-text summarization then text-to-code generation, to obtain translation data. In contrast, we train a text-to-code generation model by reversing the summarization data; our methods differ in both the procedure and the intended task. Also, Yu et al. (2022) crafted rules for source code transformation, whilst our investigation is on automatic neural methods. Finally, techniques like dead code insertion and variable renaming in malware obfuscation (You and Yim, 2010), as well as string manipulation (e.g. token noising, swapping, deletion) can be useful. Nonetheless, these methods are not task-specific, meaning they could be more appropriate for the generic code pre-training stage.

6 Conclusion

We adapt several data augmentation techniques to programming language translation and summarization. Our investigation includes data synthesis, knowledge sharing via multilinguality, and numeric-aware techniques. Enhanced performance is observed in experiments conducted on a variety of pre-trained code language models, and our analysis demonstrates that these methods can benefit output code style and numeric correctness.

7 Limitations

We identify the main limitation to lie in evaluation since we relied on automatic text metrics for both code and text generation. Ideally, code should be treated with software testing practices such as code review, compilation, unit testing, etc. Evaluation is further undermined given the test data issues revealed in Section 4 and Appendix B.1, so more human analysis should be of interest.

We also do not cover all potential code generation tasks, e.g. code synthesis, where a code snippet is created given a textual description. In this task, the source side carries much less information than the target. We apply a back-translation-style augmentation, but it does not significantly surpass the state-of-the-art PLM. Due to space constraints, we offer some preliminary views in Appendix C.

Acknowledgements

We are grateful to Ignacio Iacobacci for his comments on numeric input scaling, and to the reviewers for their suggestions on qualitative analysis. We also thank the MindSpore team for providing technical support.[1,2]

Pinzhen Chen is supported by UK Research and Innovation under the UK government's Horizon Europe funding guarantee [grant number 10052546 – High Performance Language Technologies].

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2022. Summarize and generate to back-translate: Unsupervised translation of programming languages. *arXiv preprint*, abs/2205.11116v1.

[1]https://www.mindspore.cn/en
[2]https://github.com/mindspore-ai

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, ..., and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *arXiv preprint*, abs/2107.03374v2.

Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhong-Yi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, Hao Yu, Li Yan, Pingyi Zhou, Xin Wang, Yu Ma, Ignacio Iacobacci, Yasheng Wang, Guangtai Liang, Jia Wei, ..., and Qun Liu. 2022. Pangu-coder: Program synthesis with function-level language modeling. *arXiv preprint*, abs/2207.11280v1.

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. Electra: Pre-training text encoders as discriminators rather than generators. In *International Conference on Learning Representations*.

Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: multi-mode translation of natural language and python code with transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Anna Currey, Antonio Valerio Miceli Barone, and Kenneth Heafield. 2017. Copied monolingual data improves low-resource neural machine translation. In *Proceedings of the Second Conference on Machine Translation*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. 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)*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Hamel Husain, Hongqi Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint*, abs/1909.09436v3.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning*.

Tom Kocmi, Rachel Bawden, Ondřej Bojar, Anton Dvorkovich, Christian Federmann, Mark Fishel, Thamme Gowda, Yvette Graham, Roman Grundkiewicz, Barry Haddow, Rebecca Knowles, Philipp Koehn, Christof Monz, Makoto Morishita, Masaaki Nagata, Toshiaki Nakazawa, Michal Novák, Martin Popel, and Maja Popović. 2022. Findings of the 2022 conference on machine translation (WMT22). In *Proceedings of the Seventh Conference on Machine Translation (WMT)*.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized BERT pretraining approach. *arXiv preprint*, abs/1907.11692v1.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, ..., and Shujie Liu. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

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*.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, M. Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a method for automatic evaluation of code synthesis. *arXiv preprint*, abs/2009.10297v2.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems*.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Improving neural machine translation models with monolingual data. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. 2022. Structcoder: Structure-aware transformer for code generation. *arXiv preprint*, abs/2206.05239v1.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*.

Ilsun You and Kangbin Yim. 2010. Malware obfuscation techniques: A brief survey. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*.

Shiwen Yu, Ting Wang, and Ji Wang. 2022. Data augmentation by program transformation. *Journal of Systems and Software*, 190:111304.

A Model Configurations

Our training and model configurations are summarized here and in Table 6. We retain the relevant PLMs' default configurations as much as possible, except for a grid search on the learning rate for code summarization with UniXcoder. We also changed the batch size to utilize our GPUs.

The randomly initialized Transformer decoder attached to CodeBERT and GraphCodeBERT has 6 layers, 12 heads, 768 hidden dimensions, and other hyperparameters as default in PyTorch. For the numeric encoding experiments with CodeBERT, we append 2 dimensions to CodeBERT's 768d encoder output, then transform it back to 768d using a linear layer. To ensure a fair comparison, a 768d-to-768d layer is added to the baseline to make it as deep.

All experiments are given a fixed budget to run. We save the best checkpoint according to validation BLEU. Results in the paper are based on a single run, but the experiments were benchmarked on PLMs of different architectures to reflect stability.

B More Inspections on Translation Test

B.1 Test imperfections

We show a few translation test instances that are not perfectly parallel in Table 7. In these cases, the code in two languages will not function exactly the same when being executed.

B.2 Numeric consistency

Complementing the number accuracy figures reported in Section 3.3, we list translation outputs containing numbers in Table 9 for visualization. It conveys the idea that our DA models can ensure number consistency even in very long and complicated outputs. In the baseline outputs, for example in test #436, number incorrectness further leads to undesirable hallucinations, which can be prevented in the DA model's output.

Hyperparameter	**Value**
PLM checkpoints	CodeBERT: https://huggingface.co/microsoft/codebert-base GraphCodeBERT: https://huggingface.co/microsoft/graphcodebert-base UniXcoder: https://huggingface.co/microsoft/unixcoder-base CodeGPT: https://huggingface.co/microsoft/CodeGPT-small-java-adaptedGPT2 StructCoder: https://github.com/reddy-lab-code-research/structcoder
trainable parameters	CodeBERT: 172.5M + numeric encoding: + 591k GraphCodeBERT: 172.5M UniXcoder: 126.5M CodeGPT: 124.4M StructCoder: 223.4M
learning rate	translation: 5e^{-5} summarization: 1e^{-5}, 5e^{-5}, **1e^{-6}** , 5e^{-6} synthesis: 5e^{-5}
optimizer	Adam (epsilon=1e^{-8})
training loss	cross-entropy (perplexity)
validation metric	best BLEU
beam size	10
CodeXGLUE	https://github.com/microsoft/CodeXGLUE
`jsbuilder`	https://github.com/tvst/jsbuilder
`javalang`	https://github.com/c2nes/javalang

Table 6: Model and training configurations.

// test #307	
C# source	public override string ToString(){return "IndexSearcher(" + reader + "; executor=" + executor + ")";}
Java reference	public String toString() {return "IndexSearcher(" + reader + "; executor=" + executor + "; **sliceExecutionControlPlane** " + **sliceExecutor** + ")";}
// test #518	
C# source	public override PushConnection OpenPush() throw {**new NGit.Errors.NotSupportedException(JGitText.Get().pushIsNotSupportedForBundleTransport**);}
Java reference	public PushConnection openPush() throws {**TransportException return new TcpPushConnection();**}
// test #892	
C# source	public Builder(): **base(){lastDocID = -1;wordNum = -1;word = 0;}**
Java reference	public Builder() **{this(true);}**
// test #902	
C# source	public override string ToString(){return "term="+ term+", field=" +field+", value="+value;}
Java reference	public String toString() {return "term="+term+", field=" +field+", value="+valueToString()**+",docIDUpto="+docIDUpto;**}

Table 7: C#-Java test instances that are not perfectly parallel, with divergence shown in bold.

C Code Synthesis with Augmentation

For code synthesis, while reversing the summarization data is a natural solution, the difficulty lies in forming the class environment (visible and usable variables and methods) because. We parse the code in a summarization instance to obtain positive tokens, as well as randomly sample tokens from other genuine code as negative signals. In other words, from $PL \rightarrow NL$ pairs, we construct code synthesis data $NL + parse(PL) + random(PL') \rightarrow PL$.

We experiment on CodeXGLUE's code synthesis task, which samples data from CONCODE (Iyer et al., 2018) at 100K/2K/2K for training/validation/test. The source contains a text description as well as the available class variable and function names, and the target is the corresponding Java code. We reverse the Java summary data to create 181K synthetic data; to get available variable and method names, the code is parsed by `javalang` into tokens. Following CodeXGLUE, we use CodeGPT-adapted as a base model; we further experiment with StructCoder (Tipirneni et al., 2022) which is a more up-to-date code PLM.

The outputs are evaluated by BLEU, EM, and CodeBLEU, similar to translation. Note that the test references are not publicly available, and test predictions need to be sent to the CodeXGLUE authors for evaluation, so we report results on both the validation and test set for reproducibility.

We notice that for CodeGPT, our augmentation work yields a small gain on validation and test sets. However, it does not improve upon the latest PLM for a few possible reasons: 1) StructCoder is remarkably stronger than CodeGPT, thus the room for improvement is small; 2) the summarization data we used to augment the synthesis task could be different in terms of topic, length, style, etc, resulting in a domain drift.

	BLEU	EM	CodeBLEU
CodeGPT on validation			
replicate	28.13	16.1	31.65
+ augmentation	**29.04**	**16.6**	**32.35**
StructCoder on validation			
replicate	37.30	18.2	40.42
+ augmentation	**37.48**	**18.7**	**40.47**
CodeGPT on test			
paper	32.79	20.1	35.98
replicate	32.66	**20.1**	35.89
+ augmentation	**33.45**	19.2	**36.47**
StructCoder on test			
paper	40.91	22.4	44.77
replicate	**41.57**	**22.6**	**44.61**
+ augmentation	41.32	21.4	44.04

Table 8: Results for code synthesis.

```
// test #131
C# source      public ScaleClusterRequest(): base("CS", "2015-12-15", "ScaleCluster"
                   , "cs", "openAPI"){UriPattern = "/clusters/[ClusterId]";
                   Method = MethodType.PUT;}
Java reference public ScaleClusterRequest() {super("CS", "2015-12-15", "ScaleCluster"
                   , "csk");setUriPattern("/clusters/[ClusterId]");
                   setMethod(MethodType.PUT);}
baseline       publicscaleClusterRequest() {super("CS", "2018-12-15", "ScaleCluster"
                   , "cs");setUriPattern("/clusters/[ClusterId]");
                   setMethod(MethodType.PUT);}
DA model       public ClusterRequest() {super("CS", "2015-12-15", "ScaleCluster"
                   , "cs");setUriPattern("/clusters/[ClusterId]");
                   setMethod(MethodType.PUT);}

// test #436
C# source      public void CopyTo(byte[] b, int o){FormatHexByte(b, o + 0, w1);
                   FormatHexByte(b, o + 8, w2);FormatHexByte(b, o + 16, w3);
                   FormatHexByte(b, o + 24, w4);FormatHexByte(b, o + 32, w5);}
Java reference public void copyTo(byte[] b, int o) {formatHexByte(b, o + 0, w1);
                   formatHexByte(b, o + 8, w2);formatHexByte(b, o + 16, w3);
                   formatHexByte(b, o + 24, w4);formatHexByte(b, o + 32, w5);}
baseline       public void copyTo(byte[] b, int o) {formatHexByte(b, o1);
                   formatHexByte(b, o2);formatHexByte(b, o2);
                   formatHexByte(b, o3);formatHexByte(b,o + 24, w4);
                   formatHexByte(b, o + 32, w5);}
DA model       public void copyTo(int[] b, int o) {formatHexByte(b, o + 0, w1);
                   formatHexByte(b, o + 8, w2);formatHexByte(b, o + 16, w3);
                   formatHexByte(b, o + 24, w4);formatHexByte(b, o + 32, w5);}

// test #716
C# source      public override void Decode(byte[] blocks, int blocksOffset, int[]
                   values, int valuesOffset, int iterations){for (int j = 0;
                   j < iterations; ++j){var block = blocks[blocksOffset++];
                   values[valuesOffset++] = ((int)((uint)block >> 7)) & 1;
                   values[valuesOffset++] = ((int)((uint)block >> 6)) & 1;
                   values[valuesOffset++] = ((int)((uint)block >> 5)) & 1;
                   values[valuesOffset++] = ((int)((uint)block >> 4)) & 1;
                   values[valuesOffset++] = ((int)((uint)block >> 3)) & 1;
                   values[valuesOffset++] = ((int)((uint)block >> 2)) & 1;
                   values[valuesOffset++] = ((int)((uint)block >> 1)) & 1;
                   values[valuesOffset++] = block & 1;}}
Java reference public void decode(byte[] blocks, int blocksOffset, int[]
                   values, int valuesOffset, int iterations) {for (int j = 0;
                   j < iterations; ++j) {final byte block = blocks[blocksOffset++];
                   values[valuesOffset++] = (block >>> 7) & 1;
                   values[valuesOffset++] = (block >>> 6) & 1;
                   values[valuesOffset++] = (block >>> 5) & 1;
                   values[valuesOffset++] = (block >>> 4) & 1;
                   values[valuesOffset++] = (block >>> 3) & 1;
                   values[valuesOffset++] = (block >>> 2) & 1;
                   values[valuesOffset++] = (block >>> 1) & 1;
                   values[valuesOffset++] = block & 1;}}
baseline       public void decode(byte[] blocks, int blocksOffset, int[]
                   values, int valuesOffset, int iterations) {for (int j = 0;
                   j < iterations; ++j) {final byte block = blocks[blocksOffset++];
                   values[valuesOffset++] = (block >>> 7) & 1;
                   values[valuesOffset++] = (block >>> 6) & 1;
                   values[valuesOffset++] = (block >>> 5) & 1;
                   values[valuesOffset++] = (block >>> 4) & 1;
                   values[valuesOffset++] = (block >>> 4) & 1;
                   values[valuesOffset++] = (block >>> 2) & 1;
                   values[valuesOffset++] = (block >>> 1) & 1;
                   values[valuesOffset++] = block & 1;}}
DA model       public void decode(byte[] blocks, int blocksOffset, int[]
                   values, int valuesOffset, int iterations) {for (int j = 0;
                   j < iterations; ++j) {final byte block = blocks[blocksOffset++];
                   values[valuesOffset++] = (block >>> 7) & 1;
                   values[valuesOffset++] = (block >>> 6) & 1;
                   values[valuesOffset++] = (block >>> 5) & 1;
                   values[valuesOffset++] = (block >>> 4) & 1;
                   values[valuesOffset++] = (block >>> 3) & 1;
                   values[valuesOffset++] = (block >>> 2) & 1;
                   values[valuesOffset++] = (block >>> 1) & 1;
                   values[valuesOffset++] = block & 1;}}
```

Table 9: C#-Java output translations containing numbers, before and after data augmentation.