# CoTexT: Multi-task Learning with Code-Text Transformer

Long Phan[1], Hieu Tran[3], Daniel Le[1], Hieu Nguyen[1], James Anibal[1], Alec Peltekian[1], and Yanfang Ye[1]

[1]Case Western Reverse University, Ohio, USA
[3]University of Science, VNU-HCM, Vietnam
{*lnp26, yxy1032*}*@case.edu*



## Abstract

We present CoTexT, a transformer-based architecture encoder-decoder pre-trained model that learns the representative context between natural language (NL) and programming language (PL) through multi-task learning. CoTexT is pre-trained, in self-supervised fashion, based on large programming language corpus to learn general-purpose understanding and code-text generation supporting downstream NL-PL task such as code summarizing/documentation, code generation, defect detection, code debugging, etc. We train CoTexT on different combination of available PL corpus including both "bimodal" and "unimodal" data where the former is the combinations of both natural texts and their corresponding code snippets in an input sequence and the latter is merely code snippets. We evaluate multi-task learning CoTexT on different generation and classification tasks on CodeXGLUE and it achieves state-of-the-art on all downstream tasks.

## 1 Introduction

In recent years, pre-trained language model (LM) has played a crucial role and proved its effectiveness in the development of many natural language processing (NLP) systems. Before the emerging of large LMs, traditional word embedding gives each word / token a global representation. Large pre-trained models such as ELMo (Peters et al., 2018), GPT (Brown et al., 2020), BERT (Devlin et al., 2019), XLNet (Yang et al., 2020) derive effective contextualized word vector representations from large pre-trained corpus to enable better understanding of representation on the language and significantly improve a broad range of downstream NLP tasks. These LMs also make use of pre-training learning objectives such as Masked Language Modeling (MLM) (Devlin et al., 2019) where a random tokens in a sequence is masked out and the model predicts the original token to further improve the model ability to understand the context. Consequently, the success of these pre-trained models in NLP has open a path for different domain-specific pre-train LMs, such as BioBERT (Lee et al., 2019a) on biomedical text, or TaBERT (Yin et al., 2020) on NL text and tabular data.

We introduce CoTexT (Code and Text Transfer Transformer), a pre-trained model for both natural language (NL) and programming language (PL) such as Java, Python, Javascript, PHP, etc. CoTexT follows the encoder-decoder architecture propose by (Vaswani et al., 2017) with attention mechanism. We then adapts the model on the T5 framework proposed and implemented as a Python library by (Raffel et al., 2019) to perform exhausted experiments on multi-task learning of multiple programming language during self-learning and downstream fine-tuning tasks.

We train CoTexT from available large programming language corpus ( including Java, Python, JavaScript, Ruby, etc) in which we test different settings with different combinations of both unimodal and bimodal data to produce best result for each corresponding task. We then fine-tune CoTexT on five CodeXGLLUE's tasks (Lu et al., 2021) including CodeSummarization, CodeGeneration, Defect Detection and Code Refinement (small and medium dataset). Results show that we achieve state-of-the-art on all metrics of the four tasks and achieve competitive result with outperformed accuracy especially on Code Refinement Medium Dataset. We find that CoTexT outperforms rivals state-of-the-art related model like CodeBERT (Feng et al., 2020) and PLBART (Ahmad et al., 2021a).

In this paper we offer the following contributions:

- Three different versions of CoTexT that achieve state-of-the-art on the CodeXGLUE's

CodeSummarization, CodeGeneration, Defect Detection and Code Refinement (small and medium dataset) tasks.

- We publicize our CoTexT pre-trained checkpoints and related source code available for future studies and improvements

## 2 Related Work

Most recent work on domain adaptation of BERT performs on its domain better than general BERT. BioBERT (Lee et al., 2019b) is further trained from $\text{BERT}_{\text{BASE}}$ on biomedical articles such as PubMed abstracts and PMC articles. SciBERT (Beltagy et al., 2019) is also a family of BERT, which is trained on the full text of biomedical and computer science papers. The experimental results of these models on domain-specific datasets show the improvement in performance compared to $\text{BERT}_{\text{BASE}}$.

More closely to our work, CodeBERT (Feng et al., 2020) is made to take NL-PL pairs as the input and is trained on bimodal data of NL-PL pairs. That helps this model learns general-purpose representations of both natural language and programming language. Instead of considering only syntactic-level structure, GraphCodeBERT is released by (Guo et al., 2021) and uses data flow in the pre-training stage to capture the semantic-level structure of code. More recently, PLBART (Ahmad et al., 2021b) is a pre-trained sequence-to-sequence model for NL and PL. By using denoising autoencoding, the model can performs well on NL-PL understanding and generation tasks.

## 3 CoTexT

### 3.1 Vocabulary

Following the implementations of T5's framework by (Raffel et al., 2019), we use Sentence Piece Unsupervised Text Tokenizer proposed by (Kudo and Richardson, 2018). Sentence Piece model is a re-implementation of sub-word units, in which it extracts the sub-words that contain the semantic context of a sequence. This approach will overcome the drawbacks of word level tokenization and the need for enormous vocabulary sets to cover all words in a dataset. We employ Sentence Piece thoroughly as a vocabulary model for all of our contribute CoTexT models. However, during experiment we found that the provided SentencePiece

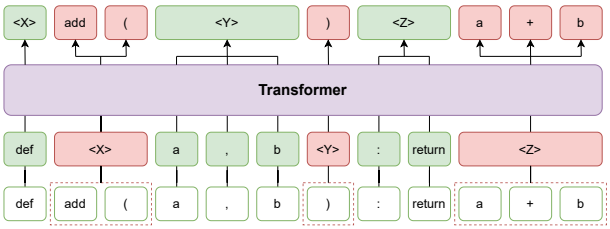


Figure 1: An illustration about Fill-in-the-blank objective

model[1] has some out-of-vocab for special tokens in code generation (such as "[", "{", "$", etc). These tokens turn out to carry a crucial representative context in programming language. Therefore, for the effective of training, we encode all of these missing tokens into a natural language representations during both self-supervised and supervised training.

### 3.2 Pre-training CoTexT

Inspired by the attempt of CodeBERT (Feng et al., 2020), we train CoTexT on both bimodal and unimodal data. Bimodal data refers to the parallel data of code snippets and its corresponding natural text in each sequence pl-nl pair, while unimodal data refers to a sequence of code only. We use two main datasets during self-supervised training: **CodeSearchNet Corpus Collection** (Husain et al., 2020) and **GitHub Repositories**[2] data. The combinations of corpus used to train CoTexT are listed in Table 1. For the efficiency of computational, for every C4 corpora used for pretraining, we init the checkpoints from the original T5 work (Raffel et al., 2019).

#### 3.2.1 CodeSearchNet Corpus Collection

CodeSearchNet Corpus (Husain et al., 2020) contains datapoints from open-source non-fork Github repositories in which all are in the form of code function and it documents natural languages across six programming language (Python, Java, Javascript, PHP, Ruby, and Go). For bimodal data, we simply concatenate natural language document of the code snippet and the correspond function into one input sequence. These self-supervised learning data are then processed as described in 3.1

[1] https://github.com/google/sentencepiece

[2] https://console.cloud.google.com/marketplace/details/github/github-repos

Table 1: Pre-training CoTexT on different combinations of natural language and programming language copora

| Model | N-modal | Corpus combination |
|---|---|---|
| T5 | NL | C4 |
| CoTexT (1-CC) | PL | **C**4 + **C**odeSearchNet |
| CoTexT (2-CC) | NL-PL | **C**4 + **C**odeSearchNet |
| CoTexT (1-CCG) | PL | **C**4 + **C**odeSearchNet + **G**ithub Repos |

#### 3.2.2 GitHub repositories

We download a large-collection of Java and Python functions from GitHub repositories dataset available on Google BigQuery. These Java and Python functions are then extracted and natural language descriptions are filtered out following the preprocessing pipeline from (Lachaux et al., 2020). These datapoints also run through a replacing special tokens pipeline as described in 3.1

### 3.3 Input/Output Representations

Under the T5's framework, CoTexT converts all NLP problems into a text-to-text format. This mean that during both self-supervised pre-training and supervised training, we always need an input text sequence and a target text sequence. For bimodal model, we concatenate natural language text and its corresponding programming language text as an input. For unimodal model, we simply feed in the model each code function as an input sequence. During self-supervised training, a span-based masking (Raffel et al., 2019) is in the input sequence, in which spans are randomly masked and the target sequence is formed as the concatenation of the same sentinel tokens and the real masked spans / tokens.

### 3.4 Model Architecture

CoTexT follows the sequence-to-sequence encoder-decoder architecture proposed by (Vaswani et al., 2017). We initialize the Base T5 model released by (Raffel et al., 2019) which has 220 million parameters. We train the model with a 0.001 learning rate and both input and target length of 1024. With the provided TPU v2-8 on Google Colab, we train with the recommended setting of model parallelism 2 and batch size 128.

### 3.5 Multi-task Learning

The model is trained with maximum likelihood objective (that is using "teacher forcing" (Williams and Zipser, 1989)) regardless of the text-code or code-text tasks. Therefore, for CoTexT, we leverage the ability of using Multi-Task learning (Raffel et al., 2019) to perform experiment on both text-code and code-text generation on CodeSummarization, CodeGeneration, and Code Refinement tasks. To specify each task the model should perform during supervised training and evaluating, we just simply add a task-specific prefix to the input sequence before fed to the model. For example, during fine-tuning downstream task CodeSummarization task for each programming language Java, Python, Javascript, PHP, Go, Ruby, we simply prepend a prefix for each PL name in the input sequence.

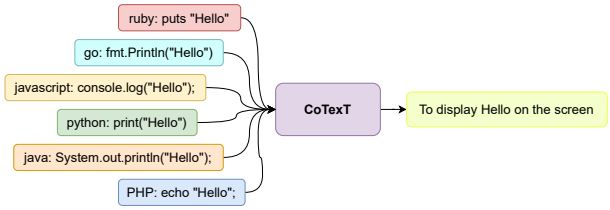


Figure 2: An illustration about Multi-task learning

## 4 Experiments

In this section, we will first have an overview of the benchmark dataset for code intelligence CodeXGLUE, then we explain the experimental setup on the tasks we perform and the results. The evaluation datasets are summarized in Table 3

### 4.1 CodeXGLUE

General Language Understanding Evaluation benchmark for CODE (CodeXGLUE) (Lu et al., 2021) is a benchmark dataset to foster machine learning studies on code understanding and generation problems. It includes a collection of code intelligence tasks (both classification and generation), a platform for model evaluation, and a leaderboard for comparison. It includes 14 datasets and 10 code intelligence tasks including code-text, text-code, code-code, and text-text scenarios. For CoTexT, we will focus on perform exhaustive experiments and

benchmark on Code Summarization, Code Generation, Code Refinement, and Defect Detection tasks.

### 4.2 Evaluation Tasks

We evaluate our programming language and natural language generation tasks on TPU v2-8 with the setting provided from the original T5 (Raffel et al., 2019). The input length and target length for each task are described in Table 2.

#### 4.2.1 Code Summarization

For Code Summarization, the objective is to generate a natural language documentation for a given code snippet. The task includes a CodeSearchNet dataset (Husain et al., 2019) with 6 different programming languages: Python, Java, Javascript, PHP, Ruby, Go. The data comes from public open-source non-fork GitHub repositories and each documentation output corresponds to the first paragraph.

#### 4.2.2 Code Generation

Text-to-Code Generation target aims to generate a code function given a natural language description. This task uses CONCODE dataset (Iyer et al., 2018), a well-known dataset on Java language generation. Each tuple is a set of natural language description, code environments, and code snippets. The goal is to generate the correction Java function, given the natural language description in the form of Javadoc-style method comments.

#### 4.2.3 Code Refinement

Code Refinement, or Code Repair, aims to fix bugs in the Java code automatically. The task use the Bug2Fix corpus released by CodeXGLUE itself (Lu et al., 2021), dividing the task into 2 subsets SMALL and MEDIUM based on the code length. The former subset includes Java code functions with tokens $\leq 50$ and the later includes code functions with tokens $\geq 50$ and $\leq 100$.

#### 4.2.4 Defect Detection

Defect Detection task is to classify whether a code snippet contains a source code that is vulnerabilities to software development and production, such as attacking software system, resource leaks, or DoS attack. The task uses Devign dataset (Zhou et al., 2019) that contains C programming language from open-source projects and labels based on security-related commits.

### 4.3 Experimental Setup

#### 4.3.1 Baselines

We compare our model with some well-known pre-trained models:

- CodeGPT, CodeGPT-adapted are based on the architecture and training objective of GPT-2 (Budzianowski and Vulic, 2019). CodeGPT is pre-trained from scratch on CodeSearchNet dataset (Lu et al., 2021) while CodeGPT-adapted further learns this dataset from GPT-2 checkpoint.

- CodeBERT (Feng et al., 2020) employs the same architecture as RoBERTa (Liu et al., 2020) but its objective is to minimize the combination of masked language modeling and replaced token detection.

- PLBART (Ahmad et al., 2021b) is a Transformer-based model. It trained BART (Lewis et al., 2020) on PL corpora with three strategies, including token masking, token deletion, and token infilling.

#### 4.3.2 Performance Metrics

- BLEU (Papineni et al., 2002) is a metric for automatically evaluating machine-translated text. It calculates the n-gram similarity of a candidate to a set of reference texts by counting the number of times an n-gram occurs in the reference and then clips the count of n-grams in the candidate translation to the maximum count in the reference. Similar to (Feng et al., 2020) and (Ahmad et al., 2021b), we employ smoothed BLEU-4 score (Lin and Och, 2004) for Code Summarization and corpus-level BLEU score for all remain tasks.

- CodeBLEU (Ren et al., 2020) is designed to consider syntactic and semantic features of codes based on the abstract syntax tree and the data flow structure.

- Accuracy is the ratio of the number of generated sequences that harmonise the reference to the total number of observations.

### 4.4 Results

#### 4.4.1 Code Summarization

The results of Code Summarization task are shown in Table 5. First, we observe that the base T5,

Table 2: The input and target sequence length settings for each self-supervised learning, code summarization, code generation, code refinement, and defect detection task

| Task | Dataset | Task Type | Input Length | Target Length |
|---|---|---|---|---|
| Self-supervised Learning | CodSearchNet Corpus<br>GitHub Repositories | | 1024<br>1024 | 1024<br>1024 |
| Code Summarization | CodeSearchNet | Multi-Task | 512 | 512 |
| Code Generation | CONCODE | Single-Task | 256 | 256 |
| Code Refinement | $\text{Bugs2Fix}_{\text{small}}$<br>$\text{Bugs2Fix}_{\text{medium}}$ | Multi-Task | 512 | 512 |
| Defect Detection | Devign | Single-Task | 1024 | 5 |

Table 3: Data statistics about Code Intelligence datasets

| **Category** | **Task** | **Dataset** | **Size** | | | **Language** |
|---|---|---|---|---|---|---|
| | | | Train | Val | Test | |
| Code-Text | Code Summarization (Lu et al., 2021) | CodeSearchNet | 164K | 5.1K | 10.9K | Java |
| | | | 58K | 3.8K | 3.2K | Javascript |
| | | | 251K | 13.9K | 14.9K | Python |
| | | | 241K | 12.9K | 14K | PHP |
| | | | 167K | 7.3K | 8.1K | Go |
| | | | 24K | 1.4K | 1.2K | Ruby |
| Code-Code | Defect Detection (Zhou et al., 2019) | Devign | 21K | 2.7K | 2.7K | C |
| | Code Refinement (Lu et al., 2021) | $\text{Bugs2Fix}_{\text{small}}$ | 46K | 5.8K | 5.8K | Java |
| | | $\text{Bugs2Fix}_{\text{medium}}$ | 52K | 6.5K | 6.5K | |
| Text-Code | Code Generation (Iyer et al., 2018) | CONCODE | 100K | 2K | 2K | Java |

Table 4: Test result on Code Generation task

| **Model** | **Text2Code Generation** | | |
|---|---|---|---|
| | EM | BLEU | CodeBLEU |
| PLBART | 18.75 | <u>36.69</u> | 38.52 |
| CodeGPT-adapted | 20.10 | 32.79 | 35.98 |
| CodeGPT | 18.25 | 28.69 | 32.71 |
| T5 | 18.65 | 32.74 | 35.95 |
| CoText (1-CCG) | 19.45 | 35.40 | 38.47 |
| CoText (2-CC) | <u>20.10</u> | 36.51 | <u>39.49</u> |
| CoText (1-CC) | **20.10** | **37.40** | **40.14** |

*Notes:* The best scores are in bold and second best scores are underlined. The baseline scores were obtained from the CodeXGLUE's Leaderboard (https://microsoft.github.io/CodeXGLUE/)

Table 5: Test result on Code Summarization task

| **Model** | **All** | **Ruby** | **Javascript** | **Go** | **Python** | **Java** | **PHP** |
|---|---|---|---|---|---|---|---|
| RoBERTa | 16.57 | 11.17 | 11.90 | 17.72 | 18.14 | 16.47 | 24.02 |
| CodeBERT | 17.83 | 12.16 | 14.90 | 18.07 | 19.06 | 17.65 | **25.16** |
| PLBART | 18.32 | **14.11** | **15.56** | 18.91 | 19.3 | 18.45 | 23.58 |
| T5 | 18.35 | 14.18 | 14.57 | 19.17 | 19.26 | 18.35 | 24.59 |
| CoTexT (1-CCG) | 18.00 | 13.23 | 14.75 | 18.95 | 19.35 | 18.75 | 22.97 |
| CoTexT (2-CC) | 18.38 | 13.07 | 14.77 | **19.37** | 19.52 | **19.1** | 24.47 |
| CoTexT (1-CC) | **18.55** | 14.02 | 14.96 | 18.86 | **19.73** | 19.06 | 24.58 |

*Notes:* The best scores are in bold and second best scores are underlined. The baseline scores were obtained from the CodeXGLUE's Leaderboard (https://microsoft.github.io/CodeXGLUE/)

Table 6: Test result on Code Refinement task

| | **Small test set** | | | **Medium test set** | | |
|---|---|---|---|---|---|---|
| **Model** | BLEU | Acc(%) | CodeBLEU | BLEU | Acc(%) | CodeBLEU |
| Transformer | 77.21 | 14.70 | 73.31 | 89.25 | 3.70 | 81.72 |
| CodeBERT | 77.42 | 16.40 | 75.58 | **91.07** | 5.16 | **87.52** |
| PLBART | 77.02 | 19.21 | / | 88.5 | 8.98 | / |
| T5 | 74.94 | 15.3 | 75.85 | 88.28 | 4.11 | 85.61 |
| CoTexT (1-CCG) | 76.87 | 20.39 | 77.34 | 88.58 | 12.88 | 86.05 |
| CoTexT (2-CC) | 77.28 | **21.58** | **77.38** | 88.68 | 13.03 | 84.41 |
| CoTexT (1-CC) | **77.79** | 21.03 | 76.15 | 88.4 | **13.11** | 85.83 |

*Notes:* The best scores are in bold and second best scores are underlined. The baseline scores were obtained from the CodeXGLUE's Leaderboard (https://microsoft.github.io/CodeXGLUE/)

Table 7: Test result on Defect Detection task

| Model | Accuracy |
|---|---|
| RoBERTa | 61.05 |
| CodeBERT | 62.08 |
| PLBART | 63.18 |
| T5 | 61.93 |
| CoTexT (1-CCG) | **66.62** |
| CoTexT (2-CC) | 64.49 |
| CoTexT (1-CC) | <u>65.99</u> |

*Notes:* The best scores are in bold and second best scores are underlined. The baseline scores were obtained from the CodeXGLUE's Leaderboard (https://microsoft.github.io/CodeXGLUE/)

which is pre-trained only on the general domain corpus C4 is effective on this task. In fact, it achieves higher overall results on BLEU-4 metric compared to all other related model on the CodeXGLUE's leaderboard. On the other hand, CoTexT achieves state-of-the-art on the overall scores and 2 popular programming language Java and Python while still achieving competitive results on all other languages, proving that our approach in training CoTexT on a large programming language corpus is effective on code-text generation.

#### 4.4.2 Code Generation

In Table 4, we reported our results for Code Generation task in Java. The result shows that our proposed model achieves state-of-the-art on 3 metrics Exact Match (EM), BLEU, and CodeBLEU. Moreover, CoTexT outperforms other model on CodeBLEU metrics, indicating that CoTexT is effective on code generation.

#### 4.4.3 Code Refinement

The Code Refinement results of each model are shown in Table 6. For this task, the base T5, which is pre-trained only on natural language text, does not perform well compared to other transformer-based models. Yet, after the training on a large programming language corpus, the result improves significantly on all metrics for both small and medium test sets. CoTexT achieves state-of-the-art on the small test set and on the accuracy metric for the medium test set. The relatively low BLEU BLEU scores and CodeBLEU in the medium test set can be attribute to the substitution of the function name in the medium-length task into other general method name during training and evaluating. Yet, CoTexT is still able to outperform all other state-of-the-art models on the accuracy metric in code-code generating.

#### 4.4.4 Defect Detection

The Defect Detection results are shown in Table 7. For this task, extra training on large programming corpus allows CoTexT to outperform all of other baseline models and achieve state-of-the-art on the task. One of the notifying contribution here is that when training on a larger and broader corpus like GitHub Repos, there is a promising improvement in the result as the CodeSearchNet Collection doesn't contain the C programming language using in the task.

## 5 Conclusion

In this manuscript, we introduced CoTexT, which is a pre-trained language representation for both programming language and natural language. CoTexT focused on text-code and code-text understanding and generating. Leveraging multi-task under the T5's framework (Raffel et al., 2019), we showed that pre-training on a large programming language corpus is effective for natural language and programming language domain. CoTexT achieves state-of-the-art on 4 test CodeXGLUE's code intelligence tasks: Code Summarization, Code Generation, Code Refinement, and Code Detection. For future works, we want to test CoTexT on a broader range of programming language and natural language generation tasks, such as autocompletion or code translating.

## References


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

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021b. 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*.

Iz Beltagy, Kyle Lo, and Arman Cohan. 2019. SciBERT: A pretrained language model for scientific text. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3615–3620, Hong Kong, China. Association for Computational Linguistics.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

Pawel Budzianowski and Ivan Vulic. 2019. Hello, it's GPT-2 - how can I help you? towards the use of pretrained language models for task-oriented dialogue systems. *CoRR*, abs/1907.05774.

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)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

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*, pages 1536–1547, Online. Association for Computational Linguistics.

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. Graphcode{bert}: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Codesearchnet challenge: Evaluating the state of semantic code search.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *CoRR*, abs/1808.09588.

Taku Kudo and John Richardson. 2018. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *CoRR*, abs/1808.06226.

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages.

Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. 2019a. Biobert: a pre-trained biomedical language representation model for biomedical text mining. *CoRR*, abs/1901.08746.

Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. 2019b. Biobert: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics*.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.

Chin-Yew Lin and Franz Josef Och. 2004. ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 501–507, Geneva, Switzerland. COLING.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2020. Ro{bert}a: A robustly optimized {bert} pretraining approach.

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, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation.

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 on Association for Computational Linguistics*, ACL '02, page 311–318, USA. Association for Computational Linguistics.

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. *CoRR*, abs/1802.05365.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *CoRR*, abs/1706.03762.

Ronald J. Williams and David Zipser. 1989. A learning algorithm for continually running fully recurrent neural networks. *Neural Comput.*, 1(2):270–280.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2020. Xlnet: Generalized autoregressive pretraining for language understanding.

Pengcheng Yin, Graham Neubig, Wen tau Yih, and Sebastian Riedel. 2020. Tabert: Pretraining for joint understanding of textual and tabular data.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks.