CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation

Yue Wang[1], **Weishi Wang**[12], **Shafiq Joty**[12], **and Steven C.H. Hoi**[1]
[1] Salesforce Research Asia
[2] Nanyang Technological University, Singapore
{wang.y,weishi.wang,sjoty,shoi}@salesforce.com

Abstract

Pre-trained models for Natural Languages (NL) like BERT and GPT have been recently shown to transfer well to Programming Languages (PL) and largely benefit a broad set of code-related tasks. Despite their success, most current methods either rely on an encoder-only (or decoder-only) pre-training that is suboptimal for generation (resp. understanding) tasks or process the code snippet in the same way as NL, neglecting the special characteristics of PL such as token types. We present CodeT5, a unified pre-trained encoder-decoder Transformer model that better leverages the code semantics conveyed from the developer-assigned identifiers. Our model employs a unified framework to seamlessly support both code understanding and generation tasks and allows for multi-task learning. Besides, we propose a novel identifier-aware pre-training task that enables the model to distinguish which code tokens are identifiers and to recover them when they are masked. Furthermore, we propose to exploit the user-written code comments with a bimodal dual generation task for better NL-PL alignment. Comprehensive experiments show that CodeT5 significantly outperforms prior methods on understanding tasks such as code defect detection and clone detection, and generation tasks across various directions including PL-NL, NL-PL, and PL-PL. Further analysis reveals that our model can better capture semantic information from code. Our code and pre-trained models are released at https://github.com/salesforce/CodeT5.

1 Introduction

Pre-trained language models such as BERT (Devlin et al., 2019), GPT (Radford et al., 2019), and T5 (Raffel et al., 2020) have greatly boosted performance in a wide spectrum of natural language processing (NLP) tasks. They typically employ a pre-train then fine-tune paradigm that aims to derive generic language representations by self-supervised training on large-scale unlabeled data, which can be transferred to benefit multiple downstream tasks, especially those with limited data annotation. Inspired by their success, there are many recent attempts to adapt these pre-training methods for programming language (PL) (Svyatkovskiy et al., 2020; Kanade et al., 2020; Feng et al., 2020), showing promising results on code-related tasks.

However, despite their success, most of these models rely on either an encoder-only model similar to BERT (Svyatkovskiy et al., 2020; Feng et al., 2020) or a decoder-only model like GPT (Kanade et al., 2020), which is suboptimal for generation and understanding tasks, respectively. For example, CodeBERT (Feng et al., 2020) requires an additional decoder when applied for the code summarization task, where this decoder cannot benefit from the pre-training. Besides, most existing methods simply employ the conventional NLP pre-training techniques on source code by regarding it as a sequence of tokens like NL. This largely ignores the rich structural information in code, which is vital to fully comprehend the code semantics.

In this work, we present CodeT5, a pre-trained encoder-decoder model that considers the token type information in code. Our CodeT5 builds on the T5 architecture (Raffel et al., 2020) that employs denoising sequence-to-sequence (Seq2Seq) pre-training and has been shown to benefit both understanding and generation tasks in natural language. In addition, we propose to leverage the developer-assigned identifiers in code. When writing programs, developers tend to employ informative identifiers to make the code more understandable, so that these identifiers would generally preserve rich code semantics, *e.g.,* the "binarySearch" identifier in Figure 2 directly indicates its functionality. To fuse such code-specific knowledge, we propose a novel identifier-aware objective that trains the model to distinguish which tokens are identifiers and recover them when they are masked.

Furthermore, we propose to leverage the code

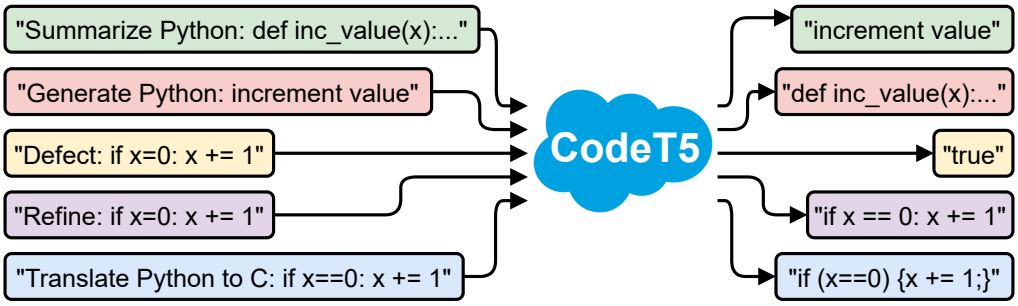

Figure 1: Illustration of our CodeT5 for code-related understanding and generation tasks.

and its accompanying comments to learn a better NL-PL alignment. Developers often provide documentation for programs to facilitate better software maintenance (de Souza et al., 2005), so that such PL-NL pairs are widely available in most source code. Specifically, we regard the NL→PL generation and PL→NL generation as dual tasks and simultaneously optimize the model on them.

We pre-train CodeT5 on the CodeSearchNet data (Husain et al., 2019) following (Feng et al., 2020) that consists of both unimodal (PL-only) and bimodal (PL-NL) data on six PLs. In addition to that, we further collect extra data of C/C# from open-source Github repositories. We fine-tune CodeT5 on most tasks in the CodeXGLUE benchmark (Lu et al., 2021), including two understanding tasks: code defect detection and clone detection, and generation tasks such as code summarization, generation, translation, and refinement. As shown in Figure 1, we also explore multi-task learning to fine-tune CodeT5 on multiple tasks at a time using a task control code as the source prompt. In summary, we make the following contributions:

- We present one of the first unified encoder-decoder models CodeT5 to support both code-related understanding and generation tasks, and also allows for multi-task learning.
- We propose a novel identifier-aware pre-training objective that considers the crucial token type information (identifiers) from code. Besides, we propose to leverage the NL-PL pairs that are naturally available in source code to learn a better cross-modal alignment.
- Extensive experiments show that CodeT5 yields state-of-the-art results on the fourteen sub-tasks in CodeXGLUE. Further analysis shows our CodeT5 can better capture the code semantics with the proposed identifier-aware pre-training and bimodal dual generation primarily benefits NL↔PL tasks.

2 Related Work

Pre-training on Natural Language. Pre-trained models based on Transformer architectures (Vaswani et al., 2017) have led to state-of-the-art performance on a broad set of NLP tasks. They can be generally categorized into three groups: encoder-only models such as BERT (Devlin et al., 2019), RoBERTa (Liu et al., 2019b), and ELECTRA (Clark et al., 2020), decoder-only models like GPT (Radford et al., 2019), and encoder-decoder models such as MASS (Song et al., 2019), BART (Lewis et al., 2020), and T5 (Raffel et al., 2020). Compared to encoder-only and decoder-only models that respectively favor understanding and generation tasks, encoder-decoder models can well support both types of tasks. They often employ denoising sequence-to-sequence pre-training objectives that corrupt the source input and require the decoder to recover them. In this work, we extend T5 to the programming language and propose a novel identifier-aware denoising objective that enables the model to better comprehend the code.

Pre-training on Programming Language. Pre-training on the programming language is a nascent field where much recent work attempts to extend the NLP pre-training methods to source code. CuBERT (Kanade et al., 2020) and CodeBERT (Feng et al., 2020) are the two pioneer models. CuBERT employs BERT's powerful masked language modeling objective to derive generic code-specific representation, and CodeBERT further adds a replaced token detection (Clark et al., 2020) task to learn NL-PL cross-modal representation. Besides the BERT-style models, Svyatkovskiy et al. (2020) and Liu et al. (2020) respectively employ GPT and UniLM (Dong et al., 2019) for the code completion task. Transcoder (Rozière et al., 2020) explores programming language translation in an unsupervised setting. Different from them, we explore

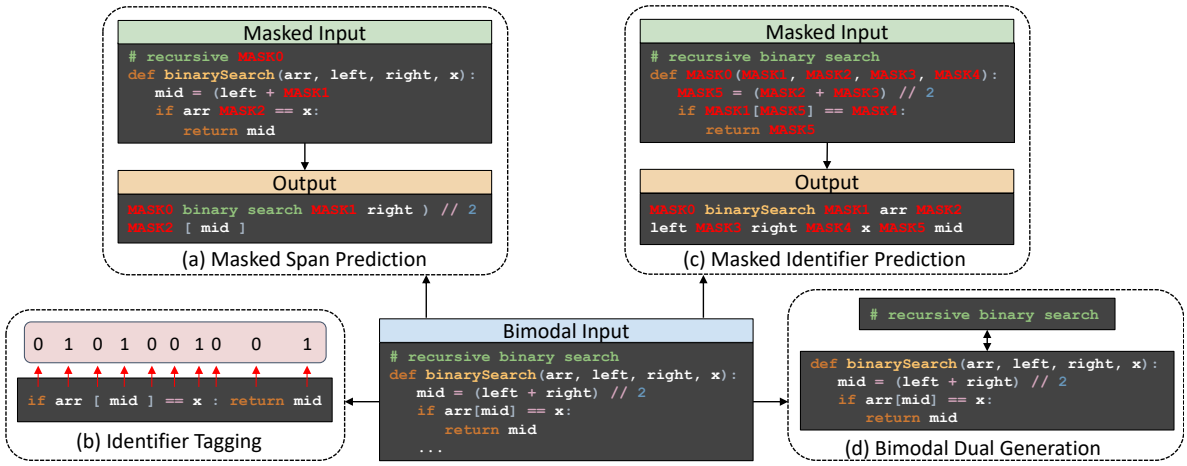

Figure 2: Pre-training tasks of CodeT5. We first alternately train span prediction, identifier prediction, and identifier tagging on both unimodal and bimodal data, and then leverage the bimodal data for dual generation training.

encoder-decoder models based on T5 for programming language pre-training and support a more comprehensive set of tasks.

Some emerging work (Clement et al., 2020; Mastropaolo et al., 2021; Elnaggar et al., 2021) in the recent literature also explore the T5 framework on code, but they only focus on a limited subset of generation tasks and do not support understanding tasks like us. Apart from these, PLBART (Ahmad et al., 2021) based on another encoder-decoder model BART can also support both understanding and generation tasks. However, all the above prior work simply processes code in the same way as natural language and largely ignores the code-specific characteristics. Instead, we propose to leverage the identifier information in code for pre-training.

Recently, GraphCodeBERT (Guo et al., 2021) incorporates the data flow extracted from the code structure into CodeBERT, while Rozière et al. (2021) propose a deobfuscation objective to leverage the structural aspect of PL. These models only focus on training a better code-specific encoder. Zügner et al. (2021) proposes to capture the relative distances between code tokens over the code structure. By contrast, we specifically focus on the identifiers that reserve rich code semantics and fuse such information into a Seq2Seq model via two novel identifier tagging and prediction tasks.

3 CodeT5

Our CodeT5 builds on an encoder-decoder framework with the same architecture as T5 (Raffel et al., 2020). It aims to derive generic representations for programming language (PL) and natural language (NL) via pre-training on unlabeled source code. As illustrated in Figure 2, we extend the denoising Seq2Seq objective in T5 by proposing two identifier tagging and prediction tasks to enable the model to better leverage the token type information from PL, which are the identifiers assigned by developers. To improve the NL-PL alignment, we further propose a bimodal dual learning objective for a bidirectional conversion between NL and PL.

In the following, we introduce how CodeT5 encodes PL and NL inputs (§3.1) and our proposed identifier-aware pre-training tasks (§3.2), followed by the fine-tuning with task-specific transfer learning and multi-task training (§3.3).

3.1 Encoding NL and PL

At the pre-training stage, our model would receive either PL-only or NL-PL as inputs depending on whether the code snippet has accompanying NL descriptions or not. For the NL-PL bimodal inputs, we concatenate them into a sequence with a delimiter token `[SEP]` and represent the whole input sequence into the format as $\mathbf{x} = (\texttt{[CLS]}, w_1, ..., w_n, \texttt{[SEP]}, c_1, ..., c_m, \texttt{[SEP]})$, where n and m denote the number of NL word tokens and PL code tokens, respectively. The NL word sequence will be empty for PL-only unimodal inputs.

In order to capture more code-specific features, we propose to leverage token type information from code. We focus on the type of identifiers (*e.g.,* function names and variables) as they are one of the most PL-agnostic features and reserve rich code semantics. Specifically, we convert the PL segment into an Abstract Syntax Tree (AST) and extract the node types for each code token. Finally, we construct a sequence of binary labels $\mathbf{y} \in \{0, 1\}^m$ for the PL segment, where each $y_i \in \{0, 1\}$ represents whether the code token c_i is an identifier or not.

3.2 Pre-training Tasks

We now introduce our proposed pre-training tasks that enable CodeT5 to learn useful patterns from either PL-only or NL-PL bimodal data.

Identifier-aware Denoising Pre-training. Denoising Sequence-to-Sequence (Seq2Seq) pre-training has been shown to be quite effective in a broad set of NLP tasks (Song et al., 2019; Raffel et al., 2020; Lewis et al., 2020). This denoising objective typically first corrupts the source sequence with some noising functions and then requires the decoder to recover the original texts. In this work, we utilize a span masking objective similar to T5 (Raffel et al., 2020) that randomly masks spans with arbitrary lengths and then predicts these masked spans combined with some sentinel tokens at the decoder. We refer this task to **Masked Span Prediction (MSP)**, as illustrated in Figure 2 (a).

Specifically, we employ the same 15% corruption rate as T5 and ensure the average span length to be 3 by uniformly sampling spans of from 1 to 5 tokens. Moreover, we employ the *whole word masking* by sampling spans before subword tokenization, which aims to avoid masking partial subtokens and is shown to be helpful (Sun et al., 2019). Notably, we pre-train a shared model for various PLs to learn robust cross-lingual representations. We describe the masked span prediction loss as:

$$\mathcal{L}_{MSP}(\theta) = \sum_{t=1}^{k} -\log P_\theta(x_t^{\text{mask}} | \mathbf{x}^{\setminus \text{mask}}, \mathbf{x}_{<t}^{\text{mask}}), \tag{1}$$

where θ are the model parameters, $\mathbf{x}^{\setminus \text{mask}}$ is the masked input, $\mathbf{x}^{\text{mask}}$ is the masked sequence to predict from the decoder with k denoting the number of tokens in $\mathbf{x}^{\text{mask}}$, and $\mathbf{x}_{<t}^{\text{mask}}$ is the span sequence generated so far.

To fuse more code-specific structural information (the identifier node type in AST) into the model, we propose two additional tasks: *Identifier Tagging (IT)* and *Masked Identifier Prediction (MIP)* to complement the denoising pre-training. .

• **Identifier Tagging (IT)** It aims to notify the model with the knowledge of whether this code token is an identifier or not, which shares a similar spirit of syntax highlighting in some developer-aided tools. As shown in Figure 2 (b), we map the final hidden states of the PL segment at the CodeT5 encoder into a sequence of probabilities $\mathbf{p} = (p_1, ..., p_m)$, and compute a binary cross entropy loss for sequence labeling:

$$\mathcal{L}_{IT}(\theta_e) = \sum_{i=1}^{m} -[y_i \log p_i + (1 - y_i) \log(1 - p_i)], \tag{2}$$

where θ_e are the encoder parameters. Note that by casting the task as a sequence labeling problem, the model is expected to capture the code syntax and the data flow structures of the code.

• **Masked Identifier Prediction (MIP)** Different from the random span masking in MSP, we mask all identifiers in the PL segment and employ a unique sentinel token for all occurrences of one specific identifier. In the field of software engineering, this is called *obfuscation* where changing identifier names does not impact the code semantics. Inspired by Rozière et al. (2021), we arrange the unique identifiers with the sentinel tokens into a target sequence $\mathbf{I}$ as shown in Figure 2 (c). We then predict it in an auto-regressive manner:

$$\mathcal{L}_{MIP}(\theta) = \sum_{j=1}^{|I|} -\log P_\theta(I_j | \mathbf{x}^{\setminus \mathbf{I}}, \mathbf{I}_{<j}), \tag{3}$$

where $\mathbf{x}^{\setminus \mathbf{I}}$ is the masked input. Note that *deobfuscation* is a more challenging task that requires the model to comprehend the code semantics based on obfuscated code and link the occurrences of the same identifiers together.

We alternately optimize these three losses with an equal probability, which constitutes our proposed identifier-aware denoising pre-training.

Bimodal Dual Generation. In the pre-training phase, the decoder only sees discrete masked spans and identifiers, which is disparate from the downstream tasks where the decoder needs to generate either fluent NL texts or syntactically correct code snippets. To close the gap between the pre-training and fine-tuning, we propose to leverage the NL-PL bimodal data to train the model for a bidirectional conversion as shown in Figure 2 (d). Specifically, we regard the NL→PL generation and PL→NL generation as dual tasks and simultaneously optimize the model on them. For each NL-PL bimodal datapoint, we construct two training instances with reverse directions and add language ids (*e.g.,* <java> and <en> for Java PL and English NL, respectively). This operation can be also seen as a special case of T5's span masking by either masking the full NL or PL segment from the bimodal inputs. This task aims to improve the alignment between the NL and PL counterparts.

3.3 Fine-tuning CodeT5

After pre-training on large-scale unlabeled data, we adapt CodeT5 to downstream tasks via either task-specific transfer learning or multi-task learning.

Task-specific Transfer Learning: Generation vs. Understanding Tasks. Code-related tasks can be categorized into generation and understanding tasks. For the former one, our CodeT5 can be naturally adapted with its Seq2Seq framework. For understanding tasks, we investigate two ways of either generating the label as a unigram target sequence (Raffel et al., 2020), or predicting it from the vocabulary of class labels based on the last decoder hidden state following Lewis et al. (2020).

Multi-task Learning. We also explore a multi-task learning setting by training a shared model on multiple tasks at a time. Multi-task learning is able to reduce computation cost by reusing the most of model weights for many tasks and has been shown to improve the model generalization capability in NL pre-training (Liu et al., 2019a). We follow Raffel et al. (2020) to employ the same unified model for all tasks without adding any task-specific networks but allow to select different best checkpoints for different tasks. To notify the model with which task it is dealing with, we design a unified format of task control codes and prepend it into the source inputs as shown in Figure 1. For instance, we employ "Translate Java to CSharp:" as the source prompt for the code-to-code translation task from Java to CSharp.

As different tasks have different dataset sizes, we follow Conneau and Lample (2019) to employ a balanced sampling strategy. For N number of datasets (or tasks), with probabilities $\{q_i\}_{i=1}^{N}$, we define the following multinomial distribution to sample from:

$$q_i = \frac{r_i^{\alpha}}{\sum_{j=1}^{N} r_j^{\alpha}}, \text{ where } \quad r_i = \frac{n_i}{\sum_{k=1}^{N} n_k}, \qquad (4)$$

where n_i is number of examples for i-th task and α is set to 0.7. This balanced sampling aims to alleviate the bias towards high-resource tasks.

4 Experimental Setup

4.1 Pre-training Dataset

We follow Feng et al. (2020) to employ CodeSearchNet (Husain et al., 2019) to pre-train CodeT5, which consists of six PLs with both unimodal and bimodal data. Apart from that, we additionally collect two datasets of C/CSharp from BigQuery[1] to ensure that all downstream tasks have overlapped PLs with the pre-training data. In total, we employ around 8.35 million instances for pre-training. Table 1 shows some basic statistics. To obtain the identifier labels from code, we leverage the tree-sitter[2] to convert the PL into an abstract syntax tree and then extract its node type information. We filter out reserved keywords for each PL from its identifier list. We observe that PLs have different identifier rates, where Go has the least rate of 19% and Ruby has the highest rate of 32%.

	PLs	W/ NL	W/o NL	Identifier
CodeSearchNet	Ruby	49,009	110,551	32.08%
	JavaScript	125,166	1,717,933	19.82%
	Go	319,132	379,103	19.32%
	Python	453,772	657,030	30.02%
	Java	457,381	1,070,271	25.76%
	PHP	525,357	398,058	23.44%
Our	C	1M	-	24.94%
	CSharp	228,496	856,375	27.85%
	Total	3,158,313	5,189,321	8,347,634

Table 1: Dataset statistics. "Identifier" denotes the proportion of identifiers over all code tokens for each PL.

4.2 Code-specific Tokenizer

Tokenization is a key ingredient for the success of pre-trained language models like BERT and GPT. They often employ a Byte-Pair Encoding (BPE) tokenizer (Sennrich et al., 2016) to alleviate the Out-of-Vocabulary (OoV) issues. Specifically, we train a Byte-level BPE tokenizer following Radford et al. (2019) and set the vocabulary size to 32,000 as T5. We add additional special tokens (`[PAD]`, `[CLS]`, `[SEP]`, `[MASK0]`, ..., `[MASK99]`). This tokenzier is trained on all of our pre-training data with non-printable characters and low-frequent tokens (occurring <3 times) filtered. We compare it with T5's default tokenizer and find that our tokenizer largely reduces the length of tokenized code sequence by 30% - 45% on downstream tasks. This will accelerate the training and especially benefit generation tasks due to the shorter sequence to predict. We also spot a severe problem for applying the T5's default tokenizer on source code, where it would encode some common code tokens such as brackets ['{', '}'] into unknown tokens.

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

[2] https://tree-sitter.github.io/tree-sitter/

Methods	Ruby	JavaScript	Go	Python	Java	PHP	Overall
RoBERTa	11.17	11.90	17.72	18.14	16.47	24.02	16.57
CodeBERT	12.16	14.90	18.07	19.06	17.65	25.16	17.83
DOBF	-	-	-	18.24	19.05	-	-
PLBART	14.11	15.56	18.91	19.30	18.45	23.58	18.32
CodeT5-small	14.87	15.32	19.25	20.04	19.92	25.46	19.14
+dual-gen	15.30	15.61	19.74	19.94	19.78	26.48	19.48
+multi-task	15.50	15.52	19.62	20.10	19.59	25.69	19.37
CodeT5-base	15.24	16.16	19.56	20.01	20.31	26.03	19.55
+dual-gen	**15.73**	16.00	19.71	20.11	20.41	**26.53**	19.75
+multi-task	15.69	**16.24**	**19.76**	**20.36**	**20.46**	26.09	**19.77**

Table 2: Smoothed BLEU-4 scores on the code summarization task. The "Overall" column shows the average scores over six PLs. Best results are in bold.

Methods	EM	BLEU	CodeBLEU
GPT-2	17.35	25.37	29.69
CodeGPT-2	18.25	28.69	32.71
CodeGPT-adapted	20.10	32.79	35.98
PLBART	18.75	36.69	38.52
CodeT5-small	21.55	38.13	41.39
+dual-gen	19.95	39.02	42.21
+multi-task	20.15	35.89	38.83
CodeT5-base	22.30	40.73	43.20
+dual-gen	**22.70**	**41.48**	**44.10**
+multi-task	21.15	37.54	40.01

Table 3: Results on the code generation task. EM denotes the exact match.

4.3 Downstream Tasks and Metrics

We cover most generation and understanding tasks in the CodeXGLUE benchmark (Lu et al., 2021) and employ the provided public datasets and the same data splits following it for all these tasks.

We first consider two cross-modal generation tasks. **Code summarization** aims to summarize a function-level code snippet into English descriptions. The dataset consists of six PLs including Ruby, JavaScript, Go, Python, Java, and PHP from CodeSearchNet (Husain et al., 2019). We employ the smoothed BLEU-4 (Lin and Och, 2004) to evaluate this task. **Code generation** is the task to generate a code snippet based on NL descriptions. We employ the Concode data (Iyer et al., 2018) in Java where the input contains both NL texts and class environment contexts, and the output is a function. We evaluate it with BLEU-4, exact match (EM) accuracy, and CodeBLEU (Ren et al., 2020) that considers syntactic and semantic matches based on the code structure in addition to the n-gram match.

Besides, we consider two code-to-code generation tasks. **Code translation** aims to migrate legacy software from one PL to another, where we focus on translating functions from Java to CSharp and vice versa. **Code refinement** aims to convert a buggy function into a correct one. We employ two Java datasets provided by Tufano et al. (2019) with various function lengths: small (fewer than 50 tokens) and medium (50-100 tokens). We use BLEU-4 and exact match to evaluate them.

We also investigate how CodeT5 performs on two understanding-based tasks. The first one is **defect detection** that aims to predict whether a code is vulnerable to software systems or not. We use the C dataset provided by Zhou et al. (2019) for experiment. The second task is **clone detection** which aims to measure the similarity between two code snippets and predict whether they have the same functionality. We experiment with the Java data provided by Wang et al. (2020). We employ F1 score and accuracy for evaluating these two tasks respectively. In total, our CodeT5 supports six tasks and fourteen sub-tasks in CodeXGLUE with a unified encoder-decoder model.

4.4 Comparison Models

We compare CodeT5 with state-of-the-art (SOTA) pre-trained models that can be categorized into three types: encoder-only, decoder-only, and encoder-decoder models. As **encoder-only** models, we consider RoBERTa (Liu et al., 2019b), RoBERTa (code) trained with masked language modeling (MLM) on code, CodeBERT (Feng et al., 2020) trained with both MLM and replaced token detection (Clark et al., 2020), GraphCodeBERT (Guo et al., 2021) using data flow from code, and DOBF (Rozière et al., 2021) trained with the identifier deobfuscation objective. Note that although DOBF employs a Seq2Seq model during pre-training, it only aims to train a better encoder for downstream tasks without exploring the potential benefit of the pre-trained decoder.

For **decoder-only** models, we compare GPT-2 (Radford et al., 2019) and its adaptations on code domain including CodeGPT-2, and CodeGPT-adapted. The difference is that the latter one utilizes a GPT-2 checkpoint for model initialization while the former one is trained from scratch. As **encoder-decoder** models, the current SOTA model for the CodeXGLUE benchmark is PLBART (Ahmad et al., 2021) based on BART (Lewis et al., 2020) architecture. For pre-training data, most of these models employ CodeSearchNet (Husain et al., 2019) except DOBF and PLBART. DOBF is pre-trained on 7.9M Java and 3.6M Python files from BigQuery while PLBART employs a much larger data with 470M Python and 210M Java functions, and 47M NL posts from StackOverflow.

Methods	Java to C#		C# to Java		Refine Small		Refine Medium	
	BLEU	EM	BLEU	EM	BLEU	EM	BLEU	EM
Naive Copy	18.54	0	18.69	0	78.06	0	90.91	0
RoBERTa (code)	77.46	56.10	71.99	57.90	77.30	15.90	90.07	4.10
CodeBERT	79.92	59.00	72.14	58.80	77.42	16.40	91.07	5.20
GraphCodeBERT	80.58	59.40	72.64	58.80	**80.02**	17.30	**91.31**	9.10
PLBART	83.02	64.60	78.35	65.00	77.02	19.21	88.50	8.98
CodeT5-small	82.98	64.10	79.10	65.60	76.23	19.06	89.20	10.92
+dual-gen	82.24	63.20	78.10	63.40	77.03	17.50	88.99	10.28
+multi-task	83.49	64.30	78.56	65.40	77.03	20.94	87.51	11.11
CodeT5-base	**84.03**	**65.90**	**79.87**	**66.90**	77.43	21.61	87.64	13.96
+dual-gen	81.84	62.00	77.83	63.20	77.66	19.43	90.43	11.69
+multi-task	82.31	63.40	78.01	64.00	78.06	**22.59**	88.90	**14.18**

Table 4: BLEU-4 scores and exact match (EM) accuracies for code translation (Java to C# and C# to Java) and code refinement (small and medium) tasks.

Methods	Defect Accuracy	Clone F1
RoBERTa	61.05	94.9
CodeBERT	62.08	96.5
DOBF	-	96.5
GraphCodeBERT	-	97.1
PLBART	63.18	**97.2**
CodeT5-small	63.40	97.1
+dual-gen	63.47	97.0
+multi-task	63.58	-
CodeT5-base	**65.78**	**97.2**
+dual-gen	62.88	97.0
+multi-task	65.02	-

Table 5: Results on the code defect detection and clone detection tasks.

4.5 Model Configurations

We build CodeT5 based on Huggingface's T5 (Raffel et al., 2020) PyTorch implementation[3] and employ two sizes of CodeT5-small (60M) and CodeT5-base (220M). We set the maximum source and target sequence lengths to be 512 and 256, respectively. We use the mixed precision of FP16 to accelerate the pre-training. We set the batch size to 1024 and employ the peak learning rate of 2e-4 with linear decay. We pre-train the model with the denoising objective for 100 epochs and bimodal dual training for further 50 epochs on a cluster of 16 NVIDIA A100 GPUs with 40G memory. The total training time for CodeT5-small and CodeT5-base is 5 and 12 days, respectively.

In the fine-tuning phase, we find that the tasks in CodeXGLUE (Lu et al., 2021) are quite sensitive to some hyper parameters such as learning rate, training steps, and batch size. We conduct a grid search and select the best parameters based on the validation set. In multi-task learning, we cover all downstream tasks except clone detection.

5 Results and Analysis

In this section, we compare CodeT5 with SOTA models on a broad set of CodeXGLUE downstream tasks (§5.1), and investigate the effects of our bimodal dual generation and multi-task learning (§5.2), followed by a detailed analysis on the proposed identifier-aware pre-training (§5.3).

5.1 CodeXGLUE Downstream Tasks

We evaluate two sizes of our model: CodeT5-small and CodeT5-base that are pre-trained with identifier-aware denoising. In addition, we consider the model that continues to train with bimodal dual generation (dual-gen) and show the results with multi-task fine-tuning. The results of all comparison models are obtained from their original papers and also the CodeXGLUE paper (Lu et al., 2021).

Code Summarization. We show code summarization results of smoothed BLEU-4 on six PL data in Table 2. We observe all our model variants significantly outperform prior work with either an encode-only (RoBERTa, CodeBERT, DOBF) or encoder-decoder framework (PLBART). Moreover, the salient performance gap between these two groups of models confirms that encode-only frameworks are suboptimal for generation tasks. Compared to the SOTA encoder-decoder model PLBART, we find that even our CodeT5-small yields better overall scores (also on Python and Java) given that our model is much smaller (60M vs. 140M) and PLBART is pre-trained with much larger Python and Java data (> 100 times). We attribute such improvement to our identifier-aware denoising pre-training and better employment of bimodal training data[4]. By increasing the model size, our CodeT5-base boosts the overall performance by over 1.2 absolute points over PLBART.

Code Generation. We compare CodeT5 with GPT-style models and PLBART in Table 3. Our CodeT5-small outperforms all decoder-only models and also the SOTA PLBART, which again confirms the superiority of encoder-decoder models at generating code snippets. Moreover, our CodeT5-base further significantly pushes the SOTA results across three metrics. Particularly, it achieves around 4.7 points improvement on CodeBLEU over PLBART, indicating our CodeT5 can better comprehend the code syntax and semantics with the help of identifier-aware pre-training.

[3] https://huggingface.co/

[4] Apart from bimodal dual generation, we concatenate NL and PL for training while PLBART deals with them separately.

Type	Code
Target	`public long ramBytesUsed(){ return BASE_RAM_BYTES_USED+((index!=null)? index.ramBytesUsed() : 0);}`
CodeT5	`public long ramBytesUsed(){ long sizeInBytes = BASE_RAM_BYTES_USED; if (index != null){ sizeInBytes += index.ramBytesUsed();} return sizeInBytes;}`

Figure 3: One translation (C# to Java) example that is semantically correct but with a 50.23% BLEU-4 score.

Code-to-Code Generation Tasks. We compare two code-to-code generation tasks: code translation and code refinement in Table 4 and further consider one naive copy baseline by copying the source input as the target prediction. In the code translation task, our CodeT5-small outperforms most of baselines and obtains comparable results with PLBART, which shows the advantages of encoder-decoder models in the code-to-code generation setting. Our CodeT5-base further achieves consistent improvements over PLBART across various metrics for translating from Java to C# and vice versa.

Here we show one CodeT5's output of translating C# to Java in Figure 3. In this case, despite the poor BLEU score, CodeT5 is able to generate a function that reserves the same functionality and even has better readability compared to the ground-truth. This reveals that CodeT5 has a good generalization ability instead of memorizing and repeating what it has seen before. On the other hand, it also suggests that BLEU score is not a perfect evaluation metric for code generation tasks, where sometimes a higher score can instead reflect the problematic copy issues of neural models.

Another code-to-code generation task is code refinement, a challenging task that requires to detect which parts of code are buggy and fix them via generating a bug-free code sequence. Due to the large overlap of source and target code, even the naive copy approach yields very high BLEU scores but zero exact matches. Therefore, we focus on the exact match (EM) metric to evaluate on this task. As shown in Table 4, we observe that EM scores for the small data are consistently higher than the medium one, indicating that it is harder to fix bugs for a longer code snippet. Our CodeT5-base significantly outperforms all baselines on EM and especially boosts over 4.8 points for the more challenging medium task (13.96 vs. GraphCodeBERT's 9.10), reflecting its strong code understanding capability.

Understanding Tasks. We compare with two understanding tasks of defect detection and clone detection in Table 5. Specifically, we generate the binary labels as a unigram sequence from the decoder for the defect detection task, while for the clone detection task, we first obtain the sequence embedding of each code snippet using the last decoder state following Lewis et al. (2020) and then predict the labels by measuring their similarity. Both CodeT5-small and CodeT5-base outperform all baselines on the defect detection task while CodeT5-base yields 2.6 accuracy score improvement than PLBART. For the clone detection task, our CodeT5 models achieve comparable results to the SOTA GraphCodeBERT and PLBART models. These results demonstrate that with an encode-decoder framework, our CodeT5 can still be adapted well for understanding tasks.

Methods	Sum-PY (BLEU)	Code-Gen (CodeBLEU)	Refine Small (EM)	Defect (Acc)
CodeT5	20.04	41.39	19.06	63.40
-MSP	18.93	37.44	15.92	64.02
-IT	19.73	39.21	18.65	63.29
-MIP	19.81	38.25	18.32	62.92

Table 6: Ablation study with CodeT5-small on four selected tasks. "Sum-PY" denotes code summarization on Python and "Code-Gen" denotes code generation.

5.2 Effects of Bimodal Dual Generation and Multi-task Learning

We examine the effects of bimodal dual generation at pre-training and multi-task learning at fine-tuning. The bimodal pre-training brings consistent improvements for code summarization and generation tasks on both CodeT5-small and CodeT5-base. However, this pre-training task does not help and even sometimes slightly hurts the performance for PL-PL generation and understanding tasks. We anticipate this is because bimodal dual generation learns a better alignment between PL and NL that naturally benefits the former tasks involving both PL and NL. As a side effect, this objective could bias the model towards the PL-NL tasks and affect its performance on PL-PL tasks.

In multi-task learning, it generally improves most of downstream tasks except the code translation and defect detection. Particularly, it largely boosts the performance on code summarization, which is not surprising as code summarization takes up the largest portion of sub tasks (six out of thirteen) and thereby benefit the most from the multi-task learning. Besides, we observe that multi-task learning consistently improves the per-

Type	Code
Source	**Text:** returns the string value of the specified field. the value is obtained from whichever scan contains the field. **Env:** Scan s1; Scan s2; boolean hasField

CodeT5	String function (String arg0){
	if (s1.hasField(arg0))
	return s1.getString(arg0);
	else return s2.getString(arg0);}

W/o MIP+IT	String function (String arg0){ return s1.getString(arg0);}

Figure 4: One code generation example on Concode test set, where our CodeT5 gives a correct prediction. The important identifiers are highlighted.

formance of code refinement, which might benefit from the joint training of both small and medium refinement data. Another possible reason is that multi-task training with defect detection would enable the model to better comprehend the code semantics for bug detection, which is also a necessary intermediate step for code refinement.

5.3 Analyzing Identifier-aware Pre-training

We provide an ablation study to examine the contribution of each component in our identifier-aware objective. Specifically, we compare the performance of our CodeT5-small on four selected tasks by ablating each of the three objectives: masked span prediction (MSP), identifier tagging (IT), and masked identifier prediction (MIP). As shown in Table 6, we observe that generally removing one of the objectives would reduce the performance for all tasks, indicating that all objectives contribute to the better code understanding of our CodeT5. However, the effect of each objective differs across tasks. Specifically, removing MSP would largely reduce the performance of all generation tasks but instead increase the defect detection performance. This shows that masked span prediction is more crucial for capturing syntactic information for generation tasks. On the contrary, removing MIP would hurt the defect detection task the most, indicating that it might focus more on code semantic understanding. By combining these objectives, our CodeT5 can better capture both syntactic and semantic information from code.

We further provide outputs from CodeT5 and its variant without MIP and IT on code generation in Figure 4. We observe that CodeT5 can correctly generate the exact function, while the model without MIP and IT fails to recover the identifiers of "s2" and "hasField". This shows our identifier-aware denoising pre-training can better distinguish and leverage the identifier information.

Methods	MSP		MIP	
	Acc	#Pred M	Acc	#Pred M
MSP-only	50.13	99.80	2.94	1.60
MIP-only	1.68	82.40	42.75	98.80
MIP+MSP	48.26	99.60	42.72	98.60

Table 7: Compare MSP and MIP on a subset of Java in CodeSearchNet. "#Pred M" denotes the ratio of prediction numbers that matches the sentinel token numbers.

We also investigate the identifier tagging performance and find it achieves over 99% F1 for all PLs, showing that our CodeT5 can confidently distinguish identifiers in code. We then check whether MSP and MIP tasks would have conflicts as they employ the same sentinel tokens for masking. In identifier masking, all occurrences of one unique identifier are replaced with the same sentinel token, resulting in a many-to-one mapping compared to the one-to-one mapping in span prediction. We compare models pre-trained with either MSP or MIP, and both on these two tasks in Table 7. We report the prediction accuracy and also the ratio of how often they can generate the same number of predictions as the sentinel tokens. We observe that pre-training only with either MIP or MSP would bias the model towards that task, achieving poor accuracy and higher mismatch in number of predictions when applied to the other task. Interestingly, we find that MIP-only objective can better recover the correct number of predictions in the MSP task than MSP-only does for the MIP task, meaning that it is easier to adapt from many-to-one mapping to one-to-one mapping and difficult for the opposite. At last, combining them can help our model to make a good trade-off on both tasks.

6 Conclusion

We have presented CodeT5, a pre-trained encoder-decoder model that incorporates the token type information from code. We propose a novel identifier-aware pre-training objective to better leverage the identifiers and propose a bimodal dual generation task to learn a better NL-PL alignment using code and its comments. Our unified model can support both code understanding and generation tasks and allow for multi-task learning. Experiments show that CodeT5 significantly outperforms all prior work in most CodeXGLUE tasks. Further analysis also reveals its better code comprehension capability across various programming languages.

Broader Impact and Ethical Consideration

Our work generally belongs to NLP applications for software intelligence. With the goal of improving the development productivity of software with machine learning methods, software intelligence research has attracted increasing attention in both academia and industries over the last decade. Software code intelligence techniques can help developers to reduce tedious repetitive workloads, enhance the programming quality and improve the overall software development productivity. This would considerably decrease their working time and also could potentially reduce the computation and operational cost, as a bug might degrade the system performance or even crash the entire system. Our work addresses the fundamental challenge of software code pre-training, our study covers a wide range of code intelligence applications in the software development lifecycle, and the proposed CodeT5 method achieves the state-of-the-art performance on many of the benchmark tasks, showing its great potential benefit towards this goal.

We further discuss the ethical consideration of training CodeT5 and the potential risks when applying it into real-world downstream applications:

Dataset bias. The training datasets in our study are source code including user-written comments from open source Github repositories and publicly available, which do not tie to any specific application. However, it is possible that these datasets would encode some stereotypes like race and gender from the text comments or even from the source code such as variables, function and class names. As such, social biases would be intrinsically embedded into the models trained on them. As suggested by Chen et al. (2021), interventions such as filtration or modulation of generated outputs may help to mitigate these biases in code corpus.

Computational cost. Our model pre-training requires non-trivial computational resources though we have tried our best to carefully design our experiments and improve experiments to save unnecessary computation costs. In fact, compared to the recent large-scale language model Codex (Chen et al., 2021), our CodeT5-base has a much smaller model size of 220M than theirs of 12B ($\sim 55\times$). In addition, we experiment on Google Cloud Platform which purchases carbon credits to reduce its carbon footprint, *e.g.,* training CodeT5-base produced around 49.25 kg CO_2 which was totally offset by the provider. Furthermore, we release our pre-trained models publicly to avoid repeated training for the code intelligence research community.

Automation bias. As CodeT5 can be deployed to provide coding assistance such as code generation for aiding developers, automation bias of machine learning systems should be carefully considered, especially for developers who tend to overrely on the model-generated outputs. Sometimes these systems might produce functions that superficially appear correct but do not actually align with the developer's intents. If developers unintentionally adopt these incorrect code suggestions, it might cause them much longer time on debugging and even lead to some significant safety issues. We suggest practitioners using CodeT5 should always bear in mind that its generation outputs should be only taken as references which require domain experts for further correctness and security checking.

Security implications. We train CodeT5 on existing code corpus including CodeSearchNet (Husain et al., 2019) and a small fraction of Google BigQuery, both of which are originally collected from public Github repositories. Pre-trained models might encode some sensitive information (*e.g.,* personal addresses or identification numbers) from the training data. Though we have conducted multi-rounds of data cleaning to mitigate this before training our models, it is still possible that some sensitive information cannot be completely removed. Besides, due to the non-deterministic nature of generation models like CodeT5, it might produce some vulnerable code to harmfully affect the software and even be able to benefit more advanced malware development when deliberately misused.

Acknowledgements

We thank Akhilesh Deepak Gotmare, Amrita Saha, Junnan Li, and Chen Xing for valuable discussions. We thank Kathy Baxter for the ethical review. We also thank our anonymous reviewers for their insightful feedback on our paper.

References

Wasi Uddin 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,*

NAACL-HLT 2021, Online, June 6-11, 2021, pages 2655–2668. Association for Computational Linguistics.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: pre-training text encoders as discriminators rather than generators. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Colin B. 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 2020, Online, November 16-20, 2020*, pages 9052–9065. Association for Computational Linguistics.

Alexis Conneau and Guillaume Lample. 2019. Cross-lingual language model pretraining. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 7057–7067.

Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia Marçal de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: documenting & Designing for Pervasive Information, SIGDOC 2005, Coventry, UK, September 21-23, 2005*, pages 68–75. ACM.

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, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186.

Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. 2019. Unified language model pre-training for natural language understanding and generation. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 13042–13054.

Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. Codetrans: Towards cracking the language of silicone's code through self-supervised deep learning and high performance computing. *CoRR*, abs/2104.02443.

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 *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP 2020, Online Event, 16-20 November 2020*, pages 1536–1547. 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 B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

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.

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, Brussels, Belgium, October 31 - November 4, 2018*, pages 1643–1652. Association for Computational Linguistics.

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, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 5110–5121. PMLR.

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, ACL 2020, Online, July 5-10, 2020*, pages 7871–7880. 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, 20th International Conference on Computational Linguistics, Proceedings of the Conference, 23-27 August 2004, Geneva, Switzerland*.

Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 473–485. IEEE.

Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. 2019a. Multi-task deep neural networks for natural language understanding. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 4487–4496. Association for Computational Linguistics.

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

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. 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. *CoRR*, abs/2102.04664.

Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader-Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 336–347. IEEE.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67.

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. *CoRR*, abs/2009.10297.

Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Baptiste Rozière, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A deobfuscation pre-training objective for programming languages. *CoRR*, abs/2102.07492.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics.

Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2019. MASS: masked sequence to sequence pre-training for language generation. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 5926–5936. PMLR.

Yu Sun, Shuohuan Wang, Yu-Kun Li, Shikun Feng, Xuyi Chen, Han Zhang, Xin Tian, Danxiang Zhu, Hao Tian, and Hua Wu. 2019. ERNIE: enhanced representation through knowledge integration. *CoRR*, abs/1904.09223.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: code generation using transformer. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1433–1443. ACM.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4):19:1–19:29.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, pages 261–271. IEEE.

Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 10197–10207.

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-agnostic representation learning of source code from structure and context. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.