Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22)
Survey Track

Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code

Changan Niu[1] , **Chuanyi Li**[1] , **Bin Luo**[1] **and Vincent Ng**[2]

[1]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[2]Human Language Technology Research Institute, University of Texas at Dallas, Richardson, Texas, USA
niu.ca@outlook.com, {lcy,luobin}@nju.edu.cn, vince@hlt.utdallas.edu

Abstract

Recent years have seen the successful application of deep learning to software engineering (SE). In particular, the development and use of pre-trained models of source code has enabled state-of-the-art results to be achieved on a wide variety of SE tasks. This paper provides an overview of this rapidly advancing field of research and reflects on future research directions.

1 Introduction

Once upon a time the state of software intelligence in software engineering (SE) was very rudimentary, with many of the decisions supported by gut feeling and at best through consultation with senior developers [Hassan and Xie, 2010]. As a wealth of data has been generated in the software development and evolution lifecycle over the years, the software development and evolution paradigm has also shifted from human experience-based to data-driven decision making. While AI researchers are fully aware of the impact deep learning has on AI application domains such as computer vision and natural language processing (NLP), many are not aware of the extensive and successful applications of deep learning technologies to SE tasks in recent years.

Though successful, the application of deep learning is not without challenges. One such challenge concerns the need for a large, typically costly-to-obtain, annotated training set to train the millions or even billions of parameters in deep neural networks. To address this data annotation bottleneck, NLP researchers have come up with an idea that can arguably be considered a breakthrough in recent deep learning research, namely *pre-training* [Dai and Le, 2015; Howard and Ruder, 2018; Peters *et al.*, 2018]. Rather than training a model from scratch (i.e., with randomly initialized network weights), which typically requires a lot of task-specific annotated data, one can first *pre-train* it on one or more so-called *self-supervised* tasks (i.e., tasks for which annotated data can be automatically generated and therefore large amounts of training data are readily available) so that its weights encode general linguistic and commonsense knowledge about language, and then the resulting *pre-trained* model can be *fine-tuned* to learn the target task using (a potentially small amount of) task-specific annotated training data in the usual supervised manner. A large number of pre-trained language models have been developed and widely used in NLP, such as BERT [Devlin *et al.*, 2018], XLNet [Yang *et al.*, 2019], RoBERTa [Liu *et al.*, 2019], ELECTRA [Clark *et al.*, 2019], GPT-2 [Radford *et al.*, 2019], T5 [Raffel *et al.*, 2020], and BART [Lewis *et al.*, 2020].

Can these pre-trained models be applied to SE tasks? Since source code can be viewed as a sequence of code tokens in the same way that natural language (NL) can be viewed as a sequence of word tokens, we can in principle retrain these models on source code and apply them to SE tasks. In practice, this is not ideal, as there are code-specific characteristics that may not be properly taken into account by these models. For instance, source code is not as homogeneous as NL: it is composed of both the code in a function body, which is written in programming language (PL), as well as optional comments written in NL. Treating both code and comments in a uniform manner (i.e., as a sequence of tokens) may not be the best way to exploit the two sources of information. In addition, code has syntactic structures (as defined in Abstract Syntax Trees (ASTs)) and semantic structures (as defined in Control Flow Graphs (CFGs)). While a few syntax-aware pre-trained models are recently developed in the NLP community (e.g., Xu et al. [2021]), the majority of existing pre-trained models fail to exploit structured information. Consequently, SE researchers have developed a number of pre-trained models of source code (CodePTMs) that take into account the characteristics specific to source code in the past few years.

Our goal in this paper is to raise the awareness of the AI audience on the impact that AI technologies — in this case the development and use of pre-trained models — have on SE, an important AI application domain, specifically by providing them with a survey of the recent development of CodePTMs and their successful application to SE tasks. We believe this survey will be of particular interest to (1) NLP researchers, especially those focusing on text summarization and generation, since many SE tasks (e.g., code summarization) involve NL generation; and (2) applied machine learning researchers, since the development of these models could have a big impact on SE. Though our target audience is AI researchers, we believe this paper could also be of high interest for the SE technology providers, raising their awareness on the added value AI technology could have in augmenting SE tooling to leverage the increasing complexity of software systems.

Type	I-O	Task	Definition	ID - Dataset	Metrics
Und.	C-V	WB	**Wrong Binary Operator**: Check if a given piece of code contains any incorrect binary operators.	K1 - Kanade et al. [2020]	Acc
		ET	**Exception Type**: Predict the precise exception type.	K1 - Kanade et al. [2020]	Acc
		BD	**Bug Detection / Defect Detection**: Check if a given function contains a defect.	D1 - Devign [2019]	Acc
				P1 - Pradel et al. [2018]	Acc
		CD	**Clone Detection**: Determine whether two code snippets are semantically equivalent.	B1 - BigCloneBench [2014]	F1
				C1 - CLCDSA [2019]	P/R/F1
		CC	**Code Classification**: Classify the category of a given function.	P2 - POJ-104 [2016]	Acc/MAP@R
		FD	**Function-Docstring Mismatch**: Determine whether a given function and the docstring correspond to each other.	K1 - Kanade et al. [2020]	Acc
	C-C	CR	**Code-to-Code Retrieval**: Retrieve semantically similar code for a given piece of query code.	C1 - CLCDSA [2019]	Acc/MRR/NDCG
				P2 - POJ-104 [2016]	MAP@R
		VM	**Variable-Misuse Localization and Repair**: Identify the location of a misused variable and return the correct one.	V1 - Vasic et al. [2019]	Acc
		CT	**Cloze Test**: Predict the masked token from code.	D2 - De Sousa et al. [2021]	Acc
	NL-C	CS	**Code Search / Text-to-Code Retrieval**: Find the most relevant piece of code from a set of candidates for a given natural language description.	C2 - CodeSearchNet [2019]	MRR
				C3 - AdvText [2021]	MRR/F1/Acc
Gen.	C-C	CP	**Code Completion**: Predict the missing/following token(s) of a given code context.	S1 - Svyatkovskiy et al. [2020]	RL/EditSim.
				L1 - Liu et al. [2020]	Acc
				A1 - Alon et al. [2020]	Acc@k
		TL	**Code Translation**: Translate the code in one programming language to the code in another programming language.	C4 - Chen et al. [2018]	BLEU/Acc/CBLEU
				T1 - TransCorder [2020]	Acc
				C1 - CLCDSA [2019]	BLEU/RL/CIDER
		BF	**Bug Fixing**: Repair buggy code by generating the correct version.	T2 - Tufano et al. [2019b]	BLEU/Acc/CBLEU
		MG	**Mutant Generation**: Inject in working code a mutant for a real bug.	T3 - Tufano et al. [2019a]	Acc
		AG	**Assert Generation**: Generate a correct unit test assert statement.	W1 - Watson et al. [2020]	Acc@k
	C-NL	SU	**Code Summarization / Code Documentation**: Generate a textual description that describes the functionality of a function.	C2 - CodeSearchNet [2019]	BLEU
				H1 - Haque et al. [2020]	BLEU/RL
				H2 - Hu et al. [2018a]	BLEU
				H3 - Hu et al. [2018b]	BLEU/METEOR
				M1 - Miceli et al. [2017]	BLEU
		MN	**Method Naming / Extreme Code Summarization**: Predict the function name of a given function body.	A2 - Allamanis et al. [2016]	P/R/F1
				E1 - ETH Py150 [2016]	P/R/F1
	NL-C	CG	**Code Generation**: Generate code given a natural language description.	C5 - CONCODE [2018]	BLEU/Acc/CBLEU

Table 1: Categorization of the 18 SE tasks to which CodePTMs have been applied.

2 SE Tasks, Datasets, and Evaluation Metrics

SE studies problems concerning the design, development, maintenance, testing, and evolution of software systems. Table 1 enumerates the key SE tasks to which pre-trained models have been applied. As can be seen in the first two columns, we classify each task along two dimensions: (1) whether the task concerns *understanding* (**Und.**) or *generation* (**Gen.**); and (2) the type of input assumed by the task and the type of output produced (**I-O**), where **C**, **NL**, and **V** denote code, natural language, and extracted/predicted value, respectively.

In addition, Table 1 shows for each task the benchmark dataset(s) and the corresponding evaluation metric(s). These metrics are fairly standard. For retrieval and classification tasks, metrics such as Acc (Accuracy [Kanade *et al.*, 2020]), Acc@k (Accuracy computed over the top k predicted answers [Watson *et al.*, 2020]), Precision(P)/Recall(R)/F1 [Nafi *et al.*, 2019], MRR (Mean Reciprocal Rank [Husain *et al.*, 2019]), MAP@R (Mean Average Precision [Mou *et al.*, 2016]), and NDCG (Normalized Discounted Cumulative Gain [Nafi *et al.*, 2019]) are typically used. For generation tasks, metrics developed in the NLP community for summarization and translation tasks, such as BLEU [Papineni *et al.*, 2002], ROUGE-L (RL) [Haque *et al.*, 2020], METEOR [Hu *et al.*, 2018b], CIDER [Zhang *et al.*, 2021], and EditSim [Svyatkovskiy *et al.*, 2020] (an edit distance-based metric), as well as variants developed in the SE community, such as CodeBLEU (CBLEU) [Ren *et al.*, 2020], are used.

3 CodePTMs

In this section, we provide an overview of 20 CodePTMs recently developed in the SE community. To enable the reader to better understand their similarities and differences, as well as their relative strengths and weaknesses, we classify them along four dimensions, as described below.

3.1 Architecture

First, existing CodePTMs differ in terms of the underlying network architecture. To understand network architectures, we need to briefly introduce the concepts of encoding and decoding. An encoder encodes an input sequence as a fixed-length vector representation, whereas a decoder generates an output sequence based on the representation of an input.

Rather than designing new network architectures, SE researchers base the design of CodePTMs on existing architectures. Broadly, these architectures can be divided into four categories: (1) **Long Short-Term Memory** (LSTM [Hochreiter and Schmidhuber, 1997]), which is a classical recurrent neural network architecture, (2) **Transformer** (TF) [Vaswani *et al.*, 2017], which is a comparatively newer encoder-decoder architecture[1] that is faster to train and can

[1]Recall that an encoder-decoder architecture is commonly used for sequence-to-sequence tasks, where the encoder encodes an input sequence as a fixed-length, typically task-specific, representation, and the decoder then generates an output sequence token by token

	Type		Task	Full Name and Description
NLP	LM		FLM [2020]	Forward LM: maximizes the conditional probabilities of all the words by taking their previous words as contexts.
			FNP [2021]	Future N-gram Prediction: a variant of FLM that involves predicting the next n ($n > 1$) tokens simultaneously instead of one token.
			BiLM [2020]	Bidirectional LM: combines a forward LM and a backward LM, and jointly maximizes the likelihood of the tokens both directions.
	MLM		BMLM [2021]	Basic version of MLM: randomly masks a certain percentage of tokens in the input, then predicts the masked tokens.
			WWM [2020]	Whole Word Masking: if a word is masked, mask all subwords/tokens in it; then predict these masked tokens.
			MASS [2022]	MAsked Seq2Seq: reconstructs the sentence fragment given the remaining part of the sentence in the encoder-decoder framework.
			SMLM [2021]	Seq2Seq MLM: randomly masks a set of token spans in the input and sequentially predicts them in the encoder-decoder framework.
	DAE		DAE [2021]	Denoising Auto-Encoding: corrupts the input (by masking, deleting tokens, etc.) and uses the model to recover the original input.
	CTL		NSP [2020]	Next Sentence Prediction: determines whether two given sentences (i.e., logical lines of code) are coherent.
			RTD [2020]	Replaced Token Detection: identifies the replaced tokens in the input (i.e., tokens produced by a small generator network).
SE	CMA	CA	IMLM [2020]	Identifier MLM: an adaptation of MLM to source code that masks only the identifiers in the code text.
			SIMLM [2021b]	Seq2Seq IMLM: an adaptation of Seq2Seq MLM to source code that masks only the identifiers in the code text.
			IT [2021b]	Identifier Tagging: determines if the input token at each position is an identifier or not via binary classification.
			CCL [2021]	Code Contrastive Learning: minimizes/maximizes the distances between the representations of similar/dissimilar code snippets.
		SA	EP [2021]	Edge Prediction: masks the edges connecting randomly selected nodes in a DFG, then predicts the masked edges.
			NOP [2021]	Node Order Prediction: randomly changes the order of some nodes in an AST, then determines if a change occurs.
		CN	BDG [2021b]	Bimodal Dual Generation: generates a NL summary if code is given, and generates code if NL is given.
			MNG [2022]	Method Name Generation: generates the sub-token sequence of the method name based on a given method body.
		CS	NA [2021]	Node Alignment: samples nodes in a DFG, masks the edge connecting each node to its code token, then predicts the masked edges.
			TMLM [2021]	Tree MLM: masks some terminal nodes/identifiers in ASTs/code on encoder/decoder side, then generates complete code sequence.
			VGVAE [2021]	vMF-Gaussian Variational Autoencoder: disentangles code semantics from code syntax under the supervision of a masked AST.
			CAP [2022]	Code-AST Prediction: determines whether the given code and AST correspond to each other.
			CLR [2021]	Cross-Language Reconstruction: reconstructs the code snippet in one PL from functionally equivalent code snippets in other PLs.
			PD [2021]	Posterior Distribution: reduces difference in distributions of functionally equiv. code snippets in different PLs over code semantics.
			ACP [2021]	Attentive Code Position: predicts the node type of a code token in an AST through an attention mechanism.
		CNS	MCL [2021a]	Multi-modal Contrastive Learning: maximizes/minimizes the representation similarity between positive/negative samples.

Table 2: Categorization and description of the pre-training tasks used by existing CodePTMs.

better capture long-distance dependencies than LSTM; (3) **Transformer-Encoder** (TE), which corresponds to the architecture of the encoder part of TF; and (4) **Transformer-Decoder** (TD), which corresponds to the architecture of the decoder part of TF. While it is possible to use encoder-only models (such as TE) and decoder-only models (such as TD) for sequence-to-sequence (seq2seq) tasks, it has been shown to be disadvantageous and impractical to do so [Niu *et al.*, 2022]. In particular, encoder-only models and decoder-only models are disadvantaged when applied to generation/decoding and classification tasks, respectively.

3.2 Modality

When using a neural model to process source code, being able to integrate the NL embedded in the code (e.g., documentations, variable names) and the code structure (e.g., ASTs) can improve the model's ability to understand the code [Ernst, 2017; Hu *et al.*, 2018b; LeClair *et al.*, 2019; Zügner *et al.*, 2021]. Therefore, the use of NL and code structure as inputs in addition to the code itself has become a common practice in CodePTMs. As Code, NL, and Structure differ in representation and processing, they can be viewed as features of different input modalities. Hence, along the second dimension, we divide CodePTMs into three categories — unimodal (**Uni**), bimodal (**Bi**), and multimodal (**Multi**) — based on the number of input modalities they employ.

When a model employs more than one input modality, we can either (1) concatenate the features extracted from different modalities to form a single training instance or (2) use the features extracted from different modalities to create different training instances. We refer to these two strategies as *Together* and *Standalone*, respectively. As can be imagined, an advantage of *Together* over *Standalone* is that the former allows cross-modal representations to be learned by a model.

based on the input and the tokens that have been generated so far.

3.3 Pre-Training Tasks

Along the third dimension, we differentiate CodePTMs based on the tasks used to pre-train them. At a high level, we can divide these tasks into two categories depending on whether the task originates in NLP (**NLP**) or is specifically designed for source code (**SE**), as shown in Table 2.

As can be seen from the table, the NLP pre-training tasks can be subdivided into four categories: (1) Language modeling (**LM**) [Qiu *et al.*, 2020], which refers to the collection of tasks that aim to predict a given word given the surrounding context; (2) Masked Language Modeling (**MLM**) [Devlin *et al.*, 2018], which refers to the collection of tasks that aim to predict the masked tokens; (3) Denoising Auto-Encoding (**DAE**) [Lewis *et al.*, 2020], which aim to recover the original (i.e., uncorrupted) text from corrupted text; and (4) Contrastive Learning (**CTL**) [Jain *et al.*, 2021], which allows a model to learn which data points are similar or different. The SE pre-training tasks, on the other hand, can be subdivided into three categories according to their input modalities: (1) Code-Aware (**CA**) tasks, which aim to mine latent information from code text; (2) Structure-Aware (**SA**) tasks, which aim to learn representations of the code structure; and (3) Cross-Modal-Aware (**CMA**) tasks, which seek to acquire knowledge from multiple input modalities. The CMA tasks can be further subdivided into three categories based on which input modalities are involved, namely Code-NL (**CN**), Code-Structure (**CS**) and Code-NL-Structure (**CNS**).

When more than one task is used to pre-train a CodePTM, the tasks involved can be learned *simultaneously* (i.e., each data instance supports all of the tasks involved[2] and the task

[2]A data instance *supports* a task if the task's loss can be computed based on the instance. For example, a code-only data instance (i.e., a code snippet without the paired docstring) supports both MLM and NSP because the losses of both tasks can be calculated based on the code snippet. However, it does not support BDG

Arch.	Mod.	Pre-Training Tasks				PL	CodePTM	SE Understanding Tasks										SE Generation Tasks							
		NLP	CA	SA	CMA			WB	ET	BD	CD	CC	FD	CR	VM	CT	CS	CP	TL	BF	MG	AG	SU	MN	CG
LSTM	Uni	✓				Mono	SCELMo			**P1**															
	Bi				✓	Multi	CodeDisen				**C1**			**C1**					**C1**						
TE	Uni	✓				Mono	CuBERT	**K1**	**K1**				**P2**		**V1**										
							C-BERT			D1															
							JavaBERT									**D2**									
		✓	✓			Multi	CugLM											**L1**							
	Bi	✓				Multi	CodeBERT										C2						C2		
		✓	✓			Mono	OSCAR					**P2**		P2											
	Multi	✓			✓	Multi	GraphCodeBERT				B1						C2		C4	T2					
		✓	✓	✓	✓	Multi	SynCoBERT			D1	**B1**			**P2**			C2,C3		C4						
TD	Uni	✓				Multi	GPT-C											**S1**							
TF	Uni		✓			Multi	DOBF				B1						C3		**T1**				C2		
		✓				Mono	DeepDebug													T2					
	Bi	✓				Mono	T5-learning													T2	**T3**	**W1**	**H1**		
						Multi	PLBART			D1	B1								C4				C2		C5
							CoTexT			D1										T2			C2		C5
							ProphetNet-Code																**C2**		
		✓	✓		✓	Multi	CodeT5			**D1**	B1								**C4**	**T2**			**C2**		**C5**
				✓	✓	Multi	TreeBERT																**H2**	**A2**,**E1**	
	Multi	✓			✓	Multi	SPT-Code										C2	**A1**	**C4**	T2			C2,**H3**,**M1**		

Table 3: Categorization of existing CodePTMs along four dimensions and their performances on downstream SE tasks. If a CodePTM is applied to a task, we list the ID of the benchmark dataset on which the CodePTM was evaluated (see Table 1 for the ID associated with each dataset), boldfacing the ID if the CodePTM achieved SOTA results on the corresponding dataset.

losses can be jointly minimized), *sequentially* (i.e., the model is first trained on the first task for a specified number of steps and then trained on the remaining tasks one by one), or *alternately* (i.e., the tasks are randomly optimized as batches of the data instances corresponding to a particular task are selected at random during training). Hence, simultaneous pre-training holds the strictest requirements on the data and the tasks because it requires that for each data instance, all the pre-training tasks can be completed in one forward propagation such that their losses can be added to form the final optimization objective and jointly minimized during backward propagation. In other words, if it can perform simultaneous pre-training, it will also be possible to perform sequential/alternate pre-training but not vice versa. Nevertheless, the selection of a pre-training strategy in existing CodePTMs seems random when multiple options are available[3].

3.4 Programming Languages

Along the last dimension, we categorize CodePTMs depending on whether they are pre-trained on one PL (**Monolingual (Mono)**) or multiple PLs (**Multilingual (Multi)**).

3.5 Categorization and Pre-Training Details

The first five columns of Table 3 categorize 20 CodePTMs along the four dimensions discussed in the previous subsections, namely Architecture (**Arch.**), Modality (**Mod.**, Pre-Training Tasks, and Programming Languages (**PL**). We believe this categorization can help the reader better understand the similarities and differences between different CodePTMs.

Note, however, that Table 3 only provides a *high-level* categorization of the CodePTMs. For instance, we still do not know which two input modalities are used by a bimodal CodePTM, and neither do we know which PLs are used to pre-train a multilingual CodePTM. Table 4 fills this gap by providing the details of how each CodePTM is pre-trained. Specifically, **CodePTM** cites the paper that proposed each CodePTM, whereas **Input**, **Objective**, and **Dataset** show the input modalities, the pre-training tasks, and the PLs involved in pre-training each CodePTM. The datasets can be divided into four types, namely, *GitHub Repos* (a dataset obtained from GitHub, e.g., JS GitHub Repos is a dataset built by GitHub JavaScript repositories), *BigQuery* (a platform that includes activity from over 3M open source GitHub repositories, e.g., "Python from BigQuery" is the dataset collected by querying Python functions on BigQuery), *CodeSearchNet* [Husain *et al.*, 2019] (a dataset that is obtained by scraping open-source repositories and pairing individual functions with their docstrings and which includes more than 6.4M codes of 6 PLs including Java, Python, JavaScript, PHP, Go and Ruby), and *CLCDSA* [Nafi *et al.*, 2019] (a dataset collected from Online Judge (OJ) sites across four PLs (i.e., Java, Python, C# and C++) where functionally similar solutions written in different PLs are available for a given problem).

4 Discussion

Next, we explore the relationship between CodePTMs (Section 3) and SE tasks (Section 2). The right half of Table 3 depicts this relationship by showing whether a CodePTM has been applied to a particular SE task, and if so, which benchmark dataset(s) it has been evaluated on and whether state-of-the-art (SOTA) results have been achieved. Below we discuss our key observations, which are based in part on Table 3 and in part on conclusions drawn from the literature.

Architecture. As can be seen in Table 3, TE-based CodePTMs are applied mostly to Understanding tasks, whereas TD- and TF-based CodePTMs are applied mostly to Generation tasks. This is understandable. As mentioned in Section 3.1, encoder-only models are disadvantaged when applied to Generation tasks. The reason is that they can only

because the code-docstring alignment is needed by BDG.

[3]For example, IT in CodeT5 can be pre-trained simultaneously with any of the other tasks, but it is still pre-trained alternatively.

CodePTM	Input	Objective	Dataset	Dataset Size
SCELMo [2020]	Code	BiLM	JS GitHub Repos	150K Files
CodeDisen [2021]	Code + AST Seq	VGVAE + CLR + PD + ACP	CLCDSA	26K Functions
CuBERT [2020]	Code	BMLM + NSP	Python from BigQuery	7.4M Files
C-BERT [2020]	Code	WWM	C GiHub Repos	5.8GB
JavaBERT [2021]	Code	BMLM	Java GitHub Repos	3M Files
CugLM [2020]	Code	IMLM + NSP + FLM	Java, TS GitHub Repos	617K Files
CodeBERT [2020]	Code + Doc	BMLM & RTD	CodeSearchNet	6.5M Functions
OSCAR [2021]	IR + AEI	BMLM + CCL	C/C++ GitHub Repos	500K Functions
GraphCodeBERT [2021]	Code + Doc + DFG Nodes	BMLM + EP + NA	CodeSearchNet (Bimodal)	2.3M Functions
SynCoBERT [2021a]	Code + Doc + AST Seq	BMLM + IT + TEP + MCL	CodeSearchNet	6.5M Functions
GPT-C [2020]	Code	FLM	Python, C#, JS/TS GitHub Repos	4.7M Files
DOBF [2021]	Code	SIMLM(Seq2Seq IMLM)	Java, Python from BigQuery	11.5M Files
DeepDebug [2021]	Code	SMLM(Seq2Seq MLM)	Java GitHub Repos	8M Files
T5-learning [2021]	Code	SMLM(Seq2Seq MLM)	CodeSearchNet (Java)	1.5M Functions
PLBART [2021]	Code & Posts	DAE (masking / deletion / infilling)	Java, Python GitHub Repos	680M Functions
			StackOverflow Posts	47M Posts
CoTexT [2021]	Code + Doc	SMLM(Seq2Seq MLM)	CodeSearchNet	6.5M Functions
			Java, Python from BigQuery	6.4M Functions
ProphetNet-Code [2021]	Code & Doc	FNP	CodeSearchNet (Bimodal)	2.3M Functions
CodeT5 [2021b]	Code + Doc	SMLM(Seq2Seq MLM) / IT / SIMLM(Seq2seq IMLM) / BDG	CodeSearchNet	6.5M Functions
			C, C# from BigQuery	1.85M Functions
TreeBERT [2021]	Code + AST Paths	TMLM + NOP	Java, Python from BigQuery	21.3M Files
SPT-Code [2022]	Code + Names + AST Seq	CAP & MASS & MNG	CodeSearchNet	6.5M Functions

Table 4: Details of how the CodePTMs are pre-trained. The pre-training scheme employed for each CodePTM is characterized by (1) the input modalities (if multiple modalities are involved, they can be handled via a Together (+) or Standalone (&) strategy); (2) the pre-training objectives (if multiple pre-training objectives are involved, they can be learned jointly (+), sequentially (&), or alternately (/)); (3) the dataset on which the CodePTM is pre-trained; and (4) the size of the dataset.

map an input sequence to an output sequence with a priori known length, but for Generation tasks the output length is typically not known a priori. In contrast, the presence of decoders in TD- and TF-based CodePTMs naturally makes them more suited to Generation tasks.

Modality. We make two modality-related observations. First, for CodePTMs that use structured information as input (e.g., features extracted from DFGs, ASTs, and AEI[4]), removing such information from the input always reduces their performances on downstream SE tasks [Guo *et al.*, 2021; Zhang *et al.*, 2021; Wang *et al.*, 2021a; Jiang *et al.*, 2021].

Second, the use of NL as an input modality appears to contribute positively to model performance on a downstream task only if NL is present in the input or output of the task [Feng *et al.*, 2020; Niu *et al.*, 2022]. Otherwise, the use of NL could lead to a performance deterioration [Phan *et al.*, 2021; Niu *et al.*, 2022]. For example, CodeT5, which is pre-trained using NL and Code, achieves SOTA results on all the NL-related SE tasks to which it is applied (e.g., TL, SU, and MN), but it is surpassed by SynCoBERT, which is pre-trained only on Code, in performance on CD, a Code-related-only task.

Pre-training tasks. We make two pre-training tasks-related observations. First, after fine-tuning on task-specific training data, a pre-trained model generally yields better results on SE downstream tasks than its "no pre-training" counterpart that is trained only on task-specific training data, and the discrepancy in their performances is especially obvious when the amount of task-specific training data is small [Zhou *et al.*, 2021; Kanade *et al.*, 2020; Buratti *et al.*, 2020; Roziere *et al.*, 2021]. This is true even when the underlying pre-trained model is taken from the NLP domain, such as RoBERTa, without pre-training it again on source code [Feng *et al.*, 2020; Ahmad *et al.*, 2021].

Second, keeping the pre-training task's type as similar as possible to that of the downstream task tends to yield the best results. Theoretically, pre-training will be beneficial for a downstream task precisely when the knowledge learned during pre-training can be successfully exploited when the model learns the downstream task. Such knowledge transfer tends to be more effective if the pre-training task is closer to the downstream task. For instance, for Understanding tasks, it is better to use a pre-training task that is also an Understanding task, such as MLM. Note that MLM is used by all the models that achieve SOTA results on Understanding tasks, such as CuBERT. In contrast, (I)MASS, which focuses on Generation, tends to work much better as a pre-training task than (I) MLM, which focuses on Understanding, on seq2seq downstream tasks such as code summarization [Jiang *et al.*, 2021].

Programming languages. We make two PL-related observations. First, knowledge transfer tends to be a lot more effective if a CodePTM is trained on a PL that is syntactically similar to the one used in the downstream task. In contrast, knowledge learned by a CodePTM from PLs that are syntactically different from the one used in the downstream task may even lead to the performance degradation. For instance, PLBART, which is pre-trained on Java and Python, performs better on C# code translation but worse on PHP code summarization than RoBERTa, a PTM that is trained on NL text only. The reason is that C# is syntactically similar to Java, while PHP has a syntax mismatch with Java and Python.

Second, multilingual pre-training and fine-tuning generally yield better results than their monolingual counterparts. For example, CodeT5, which is pre-trained on 6 PLs, out-

[4]AEI (Abstract Environment Information) describes a program's semantics with a mathematical characterization of its behaviors.

performs T5-learning and DeepDebug, both of which are only pre-trained on Java, on code translation. In addition, when performing multilingual pre-training, *language-aware* pre-training, where the training instances that belong to different PLs are being differentiated (by adding language-specific symbols to the input or appending a language-type embedding to each token, for instance), tend to yield a pre-trained model that can better discriminate between PLs than *language-agnostic* pre-training, as demonstrated via GPT-C on code completion.

5 How Effective are CodePTMs?

CodePTMs have been successfully applied to a variety of SE tasks, but *how* effective are they? To enable the reader to gain insights into this question, we present some quantitative results in this section. More specifically, we show in Table 5 the best result achieved by a CodePTM on each commonly used evaluation dataset for each SE task (see the "Best CodePTM" column). To help the reader gauge the effectiveness of CodePTMs, we show in the "Best non-CodePTM" column the best result achieved by an approach that does not involve pre-training on each dataset. As can be seen, many of the best non-CodePTM-based approaches are neural models that involve Tree-LSTM and Transformer, for instance. The last column of the table shows for each dataset the *relative error reduction rate*, which is computed as the error reduced by the best-performing CodePTM relative to the error made by the best non-CodePTM-based system on the dataset. A positive value indicates that the SOTA result is achieved using a CodePTM. As can be seen, the SOTA results on all of the datasets are achieved using CodePTMs, with the relative error reduction rates ranging from 0.9–78.7 when expressed in percentages. These results provide suggestive evidence that CodePTMs are a promising approach to a wide variety of SE tasks. Nevertheless, it is clear that CodePTMs are more effective at relative error reduction on certain SE tasks/datasets than other tasks/datasets. Additional analysis is needed to determine the reason.

Task	DS	Best CodePTM	Best non-CodePTM	ER
WB	K1	82.3 (CuBERT [2020])	73.8 (GREAT [2020])	32.4
ET	K1	79.1 (CuBERT [2020])	49.5 (Transformer [2020])	58.6
BD	D1	65.7 (CodeT5 [2021b])	62.4 (code2vec [2021])	8.8
CD	B1	97.4 (SynCoBERT [2021a])	95.0 (FA-AST [2020])	48.0
	C1	90.0 (CodeDisen [2021])	81.0 (Tree-LSTM [2019])	47.3
CC	P2	98.0 (OSCAR [2021])	96.6 (ProGraML [2021])	43.2
FD	K1	98.0 (CuBERT [2020])	91.0 (Transformer [2020])	78.7
CR	C1	31.6 (CodeDisen [2021])	16.6 (Pontes et al. [2018])	17.9
	P2	88.2 (SynCoBERT [2021a])	82.4 (MISIM [2020])	32.9
VM	V1	95.2 (CuBERT [2020])	80.5 (BiLSTM [2020])	75.4
CT	D2	94.4 (JavaBERT [2021])	–	
CS	C2	74.0 (SynCoBERT [2021a])	41.9 (Transformer [2021])	55.2
	C3	38.1 (SynCoBERT [2021a])	–	
CP	S1	82.8 (GPT-C [2020])	–	
	L1	81.9 (CugLM [2020])	71.7 (Transformer-XL [2019])	36.0
	A1	26.5 (SPT-Code [2022])	24.7 (SLM [2020])	2.4
TL	C4	66.4 (CodeT5 [2021b])	35.4 (Transformer [2021])	47.9
	T1	41.8 (DOBF [2021])	34.7 (Transformer [2021])	10.9
	C1	29.7 (CodeDisen [2021])	25.8 (Tree-LSTM [2019])	5.3
BF	T2	18.3 (CodeT5 [2021b])	12.7 (S2S+COPY [2021])	6.5
MG	T3	28.0 (T5-learning [2021])	17.0 (Tufano [2019a])	13.3
AG	W1	66.0 (T5-learning [2021])	65.0 (Watson et al. [2020])	2.9
SU	C2	19.7 (CodeT5 [2021b])	15.5 (Transformer [2020])	4.9
	H1	21.0 (T5-learning [2021])	19.0 (Haque et al. [2020])	2.5
	H2	20.4 (TreeBERT [2021])	19.7 (GNN+GRU [2020])	0.9
	H3	49.1 (SPT-Code [2022])	48.2 (AST-Trans [2022])	1.6
	M1	36.1 (SPT-Code [2022])	34.7 (AST-Trans [2022])	2.1
MN	A2	60.1 (TreeBERT [2021])	57.5 (GNN+GRU [2020])	6.1
	E1	39.0 (TreeBERT [2021])	34.4 (GNN+GRU [2020])	7.0
CG	C5	22.3 (CodeT5 [2021b])	12.2 (Iyer et al. [2019])	11.5

Table 5: Relative error reduction rates achieved by CodePTMs on SE tasks/datasets. "DS" shows the commonly used evaluation datasets for each SE task (see Table 1 for details on these datasets). "Best CodePTM" shows the best result achieved to date by a CodePTM on the corresponding dataset and the name of the CodePTM. "Best non-CodePTM" shows the best result achieved to date by an approach that does not involve pre-training on the corresponding dataset and the name of the approach (note that "–" indicates that non-CodePTM-based approaches have not been applied to the corresponding dataset). "ER" shows the relative error reduction rate for each dataset. Information on the evaluation metric used for each dataset can be found in Table 1.

6 Concluding Remarks

Though CodePTMs have proven their success in SE, we believe that they have not reached their full potential. In this section, we outline some promising future directions.

6.1 Thinking beyond NLP

Tokenization and embedding. Currently, CodePTMs use the tokenization and embedding methods developed in NLP. For example, they use SentencePiece as the tokenizer as well as token and position embeddings. However, code is not exactly the same as NL: code contains different types of lexical tokens such as variables, control symbols, and keywords. We speculate that NLP tokenization and embedding methods would not yield optimal performances for CodePTMs, and recommend that researchers look into the possibility of developing code-specific versions of these methods.

Pre-training methods. Pre-training tasks that can better exploit code-specific characteristics (e.g., code structure, the presence of branches, and the use of different identifiers taken from a largely unrestricted vocabulary to express the same meaning) may be needed in order to train more powerful CodePTMs. Most of the existing SE-specific pre-training tasks (see Section 3.3) still do not completely step outside the NLP mindset. IMLM, for example, is just a version of MLM that masks identifiers, and in fact, pre-training on IMLM has even yielded worse results than pre-training on MLM for DOBF [Roziere *et al.*, 2021]. We believe that the design of code-specific pre-training methods is currently limited in part by the NLP tokenization and embedding methods that are currently in use, and that a fundamental overhaul in the design of code-specific pre-training methods that involves designing code-specific tokenization and embedding methods will likely be needed.

6.2 Learning Code Form and Functionality

Code has both *form*, which is defined by combinations of particular code identifiers, and *function*, which is independent of any particular code identifiers [Jain *et al.*, 2021]. Note that the CodePTMs listed in Table 4 all learn representations of source code from the "form" instead of the "function" perspective. Learning code functionality, however, will undoubtedly help CodePTMs understand the code better and achieve

higher performances on SE tasks. So we believe that designing CodePTMs that can learn both code form and code functionality would be a valuable research direction.

6.3 Adaptation to Downstream Tasks

Currently, fine-tuning is the primary method for transferring the knowledge acquired during pre-training to downstream tasks. However, fine-tuning can be inefficient because all model parameters need to be updated. To mitigate this problem, the NLP community has proposed several solutions, such as (1) model reprogramming (i.e., freezing the original parameters of the PTMs and adding small fine-tunable adaption modules for specific tasks [Chen, 2022]), (2) using prompt tuning [Brown *et al.*, 2020], and (3) using model compression (e.g., pruning and knowledge distillation). How to adapt or extend these methods for fine-tuning CodePTMs is a promising research direction.

6.4 CodePTMs for Niche Applications

Rather than attempting to design a single CodePTM that works well on all SE tasks, we recommend that specialized CodePTMs be designed for different classes of SE tasks (e.g., Understanding vs. Generation). Our recommendation is based on our earlier observations that different model design choices may be better suited for different kinds of tasks. For instance, theoretically speaking, TE-based models tend to work better than TD- and TF-based models on Understanding tasks, whereas the reverse is generally true for Generation tasks. One may even go as far as designing task-specific CodePTMs. The reason is that having pre-training tasks that are more similar to the downstream task at hand could enable a more effective transfer of the knowledge acquired during pre-training, as discussed previously. We believe that specialized CodePTMs have an additional advantage: they tend to be smaller and hence may be more efficient, potentially allowing us to address the efficiency issues associated with model architecture (Section 6.1) and fine-tuning (Section 6.3).

6.5 Unified Evaluation and Analysis

Our understanding of the strengths and weaknesses of existing CodePTMs is currently limited by the tasks on which they are evaluated. To better understand CodePTMs, it is important to conduct a systematic evaluation of all CodePTMs on all the benchmark datasets associated with the 18 SE tasks we discussed. In addition to a comprehensive quantitative evaluation, a qualitative analysis that involves analyzing the common errors made by each model would be important.

Acknowledgments

We thank the three anonymous reviewers for their helpful comments on an earlier draft of this paper. This work was supported in part by the National Natural Science Foundation of China (No. 61802167) and the US National Science Foundation (Grant IIS-1528037). Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views or official policies, either expressed or implied, of the funding agencies. Chuanyi Li is the corresponding author.

References

[Ahmad *et al.*, 2021] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *NAACL-HLT*, 2021.

[Allamanis *et al.*, 2016] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *ICML*, 2016.

[Alon *et al.*, 2020] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. In *ICML*, 2020.

[Brown *et al.*, 2020] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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 Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *NeurIPS*, 2020.

[Buratti *et al.*, 2020] Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, et al. Exploring software naturalness through neural language models. *arXiv:2006.12641*, 2020.

[Chen *et al.*, 2018] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *NeurIPS*, 2018.

[Chen, 2022] Pin-Yu Chen. Model reprogramming: Resource-efficient cross-domain machine learning. *arXiv preprint arXiv:2202.10629*, 2022.

[Clark *et al.*, 2019] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. In *ICLR*, 2019.

[Coimbra *et al.*, 2021] David Coimbra, Sofia Reis, Rui Abreu, Corina Păsăreanu, and Hakan Erdogmus. On using distributed representations of source code for the detection of c security vulnerabilities. *CoRR*, 2021.

[Dai and Le, 2015] Andrew M Dai and Quoc V Le. Semi-supervised sequence learning. *Advances in neural information processing systems*, 28, 2015.

[Dai *et al.*, 2019] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *ACL*, 2019.

[de Sousa and Hasselbring, 2021] Nelson Tavares de Sousa and Wilhelm Hasselbring. Javabert: Training a transformer-based model for the java programming language. *arXiv:2110.10404*, 2021.

[Devlin *et al.*, 2018] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805*, 2018.

[Drain *et al.*, 2021] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. Generating bug-fixes using pretrained transformers. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, 2021.

[Ernst, 2017] Michael D. Ernst. Natural language is a programming language: Applying natural language processing to software development. In *2nd Summit on Advances in Programming Languages*, 2017.

[Feng *et al.*, 2020] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *EMNLP: Findings*, 2020.

[Guo *et al.*, 2021] 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. Graphcodebert: Pre-training code representations with data flow. In *ICLR*, 2021.

[Haque *et al.*, 2020] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. Improved automatic summarization of subroutines via attention to file context. In *MSR*, 2020.

[Hassan and Xie, 2010] Ahmed E Hassan and Tao Xie. Software intelligence: the future of mining software engineering data. In *FSE/SDP workshop*, 2010.

[Hellendoorn *et al.*, 2020] Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *ICLR*, 2020.

[Hochreiter and Schmidhuber, 1997] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 1997.

[Howard and Ruder, 2018] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In *ACL*, 2018.

[Hu *et al.*, 2018a] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *ICPC*, 2018.

[Hu *et al.*, 2018b] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. Summarizing source code with transferred api knowledge. In *IJCAI*, 2018.

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

[Iyer *et al.*, 2018] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. In *EMNLP*, 2018.

[Iyer *et al.*, 2019] Srinivasan Iyer, Alvin Cheung, and Luke Zettlemoyer. Learning programmatic idioms for scalable semantic parsing. In *EMNLP*, 2019.

[Jain *et al.*, 2021] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. Contrastive code representation learning. In *EMNLP*, 2021.

[Jiang *et al.*, 2021] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. Treebert: A tree-based pre-trained model for programming language. In *UAI*, 2021.

[Kanade *et al.*, 2020] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *ICML*, 2020.

[Karampatsis and Sutton, 2020] Rafael-Michael Karampatsis and Charles Sutton. Scelmo: Source code embeddings from language models. *arXiv:2004.13214*, 2020.

[LeClair *et al.*, 2019] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. In *ICSE*, 2019.

[LeClair *et al.*, 2020] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. Improved code summarization via a graph neural network. In *ICPC*, 2020.

[Lewis *et al.*, 2020] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *ACL*, 2020.

[Liu *et al.*, 2019] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv:1907.11692*, 2019.

[Liu *et al.*, 2020] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. Multi-task learning based pre-trained language model for code completion. In *ASE*, 2020.

[Lu *et al.*, 2021] 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. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *NeurIPS Datasets and Benchmarks Track (Round 1)*, 2021.

[Mastropaolo *et al.*, 2021] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *ICSE*, 2021.

[Miceli-Barone and Sennrich, 2017] Antonio Valerio Miceli-Barone and Rico Sennrich. A parallel corpus of python functions and documentation strings for automated

code documentation and code generation. In *IJCNLP*, 2017.

[Mou *et al.*, 2016] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, 2016.

[Nafi *et al.*, 2019] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K Roy, and Kevin A Schneider. Clcdsa: cross language code clone detection using syntactical features and api documentation. In *ASE*, 2019.

[Niu *et al.*, 2022] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. Spt-code: Sequence-to-sequence pre-training for learning source code representations. *arXiv:2201.01549*, 2022.

[Panthaplackel *et al.*, 2021] Sheena Panthaplackel, Miltiadis Allamanis, and Marc Brockschmidt. Copy that! editing sequences by copying spans. In *AAAI*, 2021.

[Papineni *et al.*, 2002] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *ACL*, 2002.

[Peng *et al.*, 2021] Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. How could neural networks understand programs? In *ICML*, 2021.

[Peters *et al.*, 2018] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *NAACL-HLT*, 2018.

[Phan *et al.*, 2021] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye. Cotext: Multi-task learning with code-text transformer. *arXiv:2105.08645*, 2021.

[Pontes *et al.*, 2018] Elvys Linhares Pontes, Stéphane Huet, Andréa Carneiro Linhares, and Juan-Manuel Torres-Moreno. Predicting the semantic textual similarity with siamese cnn and lstm. In *TALN*, 2018.

[Pradel and Sen, 2018] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2018.

[Qi *et al.*, 2021] Weizhen Qi, Yeyun Gong, Yu Yan, Can Xu, Bolun Yao, Bartuer Zhou, Biao Cheng, Daxin Jiang, Jiusheng Chen, Ruofei Zhang, et al. Prophetnet-x: Large-scale pre-training models for english, chinese, multi-lingual, dialog, and code generation. *arXiv:2104.08006*, 2021.

[Qiu *et al.*, 2020] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences*, 2020.

[Radford *et al.*, 2019] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. In *OpenAI Blog*, 2019.

[Raffel *et al.*, 2020] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR*, 2020.

[Raychev *et al.*, 2016] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. In *OOPSLA*, 2016.

[Ren *et al.*, 2020] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv:2009.10297*, 2020.

[Roziere *et al.*, 2020] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *NeurIPS*, 2020.

[Roziere *et al.*, 2021] Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. Dobf: A deobfuscation pre-training objective for programming languages. *arXiv:2102.07492*, 2021.

[Shido *et al.*, 2019] Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. Automatic source code summarization with extended tree-lstm. In *IJCNN*, 2019.

[Svajlenko *et al.*, 2014] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *ICSME*, 2014.

[Svyatkovskiy *et al.*, 2020] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *ESEC/FSE*, 2020.

[Tang *et al.*, 2022] Ze Tang, Xiaoyu Shen, Chuanyi Li, Jidong Ge, Liguo Huang, Zhelin Zhu, and Bin Luo. Asttrans: Code summarization with efficient tree-structured attention. In *ICSE*, 2022.

[Tufano *et al.*, 2019a] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *ICSE*, 2019.

[Tufano *et al.*, 2019b] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *TOSEM*, 2019.

[Vasic *et al.*, 2019] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh singh. Neural program repair by jointly learning to localize and repair. In *ICLR*, 2019.

[Vaswani *et al.*, 2017] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, 2017.

[Wang *et al.*, 2020] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *SANER*, 2020.

[Wang *et al.*, 2021a] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv:2108.04556*, 2021.

[Wang *et al.*, 2021b] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*, 2021.

[Watson *et al.*, 2020] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. In *ICSE*, 2020.

[Xu *et al.*, 2021] Zenan Xu, Daya Guo, Duyu Tang, Qinliang Su, Linjun Shou, Ming Gong, Wanjun Zhong, Xiaojun Quan, Daxin Jiang, and Nan Duan. Syntax-enhanced pre-trained model. In *ACL/IJCNLP (1)*, 2021.

[Yang *et al.*, 2019] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *NeurIPS*, 32, 2019.

[Ye *et al.*, 2020] Fangke Ye, Shengtian Zhou, Anand Venkat, Ryan Marcus, Nesime Tatbul, Jesmin Jahan Tithi, Niranjan Hasabnis, Paul Petersen, Timothy Mattson, Tim Kraska, et al. Misim: A neural code semantics similarity system using the context-aware semantics structure. *CoRR*, 2020.

[Zhang *et al.*, 2021] Jingfeng Zhang, Haiwen Hong, Yin Zhang, Yao Wan, Ye Liu, and Yulei Sui. Disentangled code representation learning for multiple programming languages. In *ACL: Findings*, 2021.

[Zhou *et al.*, 2019] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *NeurIPS*, 2019.

[Zhou *et al.*, 2021] Xin Zhou, DongGyun Han, and David Lo. Assessing generalizability of CodeBERT. In *ICSME*, 2021.

[Zügner *et al.*, 2021] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. In *ICLR*, 2021.