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

Changan Niu\textsuperscript{1}, Chuanyi Li\textsuperscript{1}, Bin Luo\textsuperscript{1} and Vincent Ng\textsuperscript{2}

\textsuperscript{1}State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
\textsuperscript{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 \textit{pre-training} [Dai and Le, 2015; Howard and Ruder, 2018; Peters \textit{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 \textit{pre-train} it on one or more so-called \textit{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 \textit{pre-trained} model can be \textit{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 \textit{et al.}, 2018], XLNet [Yang \textit{et al.}, 2019], RoBERTa [Liu \textit{et al.}, 2019], ELECTRA [Clark \textit{et al.}, 2019], GPT-2 [Radford \textit{et al.}, 2019], T5 [Raffel \textit{et al.}, 2020], and BART [Lewis \textit{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 \textit{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 of AI technologies — in this case the development and use of pre-trained models — 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.
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], MAP@R (Mean Average Precision) [Mou et al., 2019], and MRR (Mean Reciprocal Rank) [Husain et al., 2002] 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., 2019], METEOR [Hu et al., 2018b], CIDER [Zhong 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 that is faster to train and can

Table 1: Categorization of the 18 SE tasks to which CodePTMs have been applied.
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 [Ni et al., 2022]. In particular, encoder-only models and decoder-only models are disadvantageous when applied to generation 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) and the task

![Table 2: Categorization and description of the pre-training tasks used by existing CodePTMs.](image-url)
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.

<table>
<thead>
<tr>
<th>Arch.</th>
<th>Mod.</th>
<th>Pre-Training Tasks</th>
<th>PL</th>
<th>CodePTM</th>
<th>SE Understanding Tasks</th>
<th>SE Generation Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>Uni</td>
<td>✓</td>
<td>Mono</td>
<td>SELM0</td>
<td>P1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bi</td>
<td>✓ ✓</td>
<td>Multi</td>
<td>CodeDsent</td>
<td>K1, K1, C1, P2, V1, D1, D2</td>
<td></td>
</tr>
<tr>
<td>TE</td>
<td>Uni</td>
<td>✓ ✓</td>
<td>Mono</td>
<td>CubERT</td>
<td>C2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bi</td>
<td>✓ ✓</td>
<td>Multi</td>
<td>C-BERT</td>
<td>D1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uni</td>
<td>✓ ✓</td>
<td>Mono</td>
<td>CodeDsent</td>
<td>P2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bi</td>
<td>✓ ✓</td>
<td>Multi</td>
<td>SynCoBERT</td>
<td>D1, B1, P2</td>
<td></td>
</tr>
<tr>
<td>TD</td>
<td>Uni</td>
<td>✓ ✓</td>
<td>Multi</td>
<td>GPT-C</td>
<td>S1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bi</td>
<td>✓ ✓</td>
<td>Mono</td>
<td>DeepDeBug</td>
<td>C3, C5</td>
<td></td>
</tr>
<tr>
<td>TF</td>
<td>Uni</td>
<td>✓ ✓</td>
<td>Multi</td>
<td>PLBART</td>
<td>D1, B1, C2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bi</td>
<td>✓ ✓</td>
<td>Mono</td>
<td>ProphetNet-Code</td>
<td>C4, C5</td>
<td></td>
</tr>
</tbody>
</table>

Table 4 fills this gap by pre-training 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), BigQuery (a platform that includes activity from over 3M open source GitHub 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 CLDCSA [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).

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

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
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 AEI4), 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-

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.

<table>
<thead>
<tr>
<th>CodePTM</th>
<th>Input</th>
<th>Objective</th>
<th>Dataset</th>
<th>Dataset Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCELMo [2020]</td>
<td>Code</td>
<td>BiLM</td>
<td>JS GitHub Repos</td>
<td>150K Files</td>
</tr>
<tr>
<td>CodeDivein [2021]</td>
<td>Code + AST Seq</td>
<td>VQVAE + CLR + PD + ALP</td>
<td>CLCDSN</td>
<td>26K Functions</td>
</tr>
<tr>
<td>CuBERT [2020]</td>
<td>Code</td>
<td>BMLM + NSP</td>
<td>Python from BigQuery</td>
<td>7.4M Files</td>
</tr>
<tr>
<td>C-BERT [2020]</td>
<td>Code</td>
<td>WWM</td>
<td>C GitHub Repos</td>
<td>5.8GB</td>
</tr>
<tr>
<td>JavaBERT [2021]</td>
<td>Code</td>
<td>BMLM</td>
<td>Java GitHub Repos</td>
<td>3M Files</td>
</tr>
<tr>
<td>CugLM [2020]</td>
<td>Code</td>
<td>IMLM + NSP + FLM</td>
<td>Java, TS GitHub Repos</td>
<td>617K Files</td>
</tr>
<tr>
<td>OSCAR [2021]</td>
<td>IR + AEI</td>
<td>BMLM + CCL</td>
<td>C/C++ GitHub Repos</td>
<td>500K Functions</td>
</tr>
<tr>
<td>GraphR.codeBERT [2021]</td>
<td>Code + Doc + DFG Nodes</td>
<td>BMLM + EP + NA</td>
<td>CodeSearchNet (Biomedical)</td>
<td>2.3M Functions</td>
</tr>
<tr>
<td>SynCoBERT [2021a]</td>
<td>Code + Doc + AST Seq</td>
<td>BMLM + IT + TEP + MCL</td>
<td>CodeSearchNet</td>
<td>6.5M Functions</td>
</tr>
<tr>
<td>OPT-C [2021]</td>
<td>Code</td>
<td>FLM</td>
<td>Python, C#, JS/TS GitHub Repos</td>
<td>4.7M Files</td>
</tr>
<tr>
<td>DOCF [2021]</td>
<td>Code</td>
<td>SMLM(Seq2Seq MLM)</td>
<td>Java, Python from BigQuery</td>
<td>11.5M Files</td>
</tr>
<tr>
<td>DeepDebug [2021]</td>
<td>Code</td>
<td>SMLM(Seq2Seq MLM)</td>
<td>Java GitHub Repos</td>
<td>8M Files</td>
</tr>
<tr>
<td>TS-learning [2021]</td>
<td>Code</td>
<td>SMLM(Seq2Seq MLM)</td>
<td>CodeSearchNet (Java)</td>
<td>1.5M Functions</td>
</tr>
<tr>
<td>CoTeX [2021]</td>
<td>Code + Doc</td>
<td>SMLM(Seq2Seq MLM)</td>
<td>CodeSearchNet</td>
<td>6.5M Functions</td>
</tr>
<tr>
<td>ProphetNet-Code [2021]</td>
<td>Code &amp; Doc</td>
<td>FNP</td>
<td>CodeSearchNet (Biomedical)</td>
<td>2.3M Functions</td>
</tr>
<tr>
<td>CodeTS [2021b]</td>
<td>Code + Doc</td>
<td>SMLM(Seq2Seq MLM) / IT / SMLM(Seq2seq IMLM) / BDG</td>
<td>CodeSearchNet</td>
<td>6.5M Functions</td>
</tr>
<tr>
<td>TreeBERT [2021]</td>
<td>Code + AST Paths</td>
<td>TMLM + NOP</td>
<td>C, C# from BigQuery</td>
<td>1.85M Functions</td>
</tr>
</tbody>
</table>
performs T5-learning and DeepDebug, both of which are only pre-trained on Java, on code translation. In addition, when performing multilingual 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), tends 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.

## 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 PLM 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


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


