# TransCoder: Towards Unified Transferable Code Representation Learning Inspired by Human Skills

**Qiushi Sun**[♡], **Nuo Chen**[♢], **Jianing Wang**[♢], **Xiang Li**[♢✉], **Ming Gao**[♢]
[♡]National University of Singapore [♢]East China Normal University
qiushisun@u.nus.edu, nuochen@stu.ecnu.edu.cn
lygwjn@gmail.com, {xiangli,mgao}@dase.ecnu.edu.cn

**Abstract**

Code pre-trained models (CodePTMs) have recently demonstrated a solid capacity to process various code intelligence tasks, *e.g.*, code clone detection, code translation, and code summarization. The current mainstream method that deploys these models to downstream tasks is to fine-tune them on individual tasks, which is generally costly and needs sufficient data for large models. To tackle the issue, in this paper, we present *TransCoder*, a unified Transferable fine-tuning strategy for Code representation learning. Inspired by human inherent skills of knowledge generalization, TransCoder drives the model to learn better code-related knowledge like human programmers. Specifically, we employ a tunable prefix encoder to first capture cross-task and cross-language transferable knowledge, subsequently applying the acquired knowledge for optimized downstream adaptation. Besides, our approach confers benefits for tasks with minor training sample sizes and languages with smaller corpora, underscoring versatility and efficacy. Extensive experiments conducted on representative benchmarks clearly demonstrate that our method can lead to superior performance on various code-related tasks and encourage mutual reinforcement, especially in low-resource scenarios. Our codes are available at https://github.com/QiushiSun/TransCoder.

**Keywords:** Neural Code Intelligence, Pre-trained Models, Transfer Learning

## 1. Introduction

With the remarkable success pre-trained language models (Devlin et al., 2019; Radford et al., 2019; Liu et al., 2019; Raffel et al., 2020; Qiu et al., 2020, *inter alia*) have achieved in natural language processing (NLP), leveraging pre-training strategies has gradually become the de-facto paradigm for neural code intelligence. Under the assumption of "Software Naturalness" (Hindle et al., 2016), researchers have proposed several code pre-trained models (CodePTMs; Feng et al., 2020; Xu and Zhu, 2022; Xu et al., 2022; Nijkamp et al., 2023b,a) , which utilize unsupervised objectives in the pre-training stage and are fine-tuned on downstream tasks. These CodePTMs have shown great capacity in code understanding (Mou et al., 2016; Svajlenko et al., 2014) and code generation (Nguyen et al., 2015; Husain et al., 2019). The applications of code intelligence in real-world scenarios can be mainly divided into two dimensions: languages and tasks; when adapting these powerful CodePTMs to downstream tasks of code representation learning, each model requires to be tuned on a specific task in a particular language. This faces challenges that include data insufficiency and imbalance.

To address this kind of problem, the idea of meta-learning has been widely adopted for model adaptation (Finn et al., 2017; Jamal and Qi, 2019; Rajasegaran et al., 2020). Wang et al. (2021a) employs a multitask meta-knowledge acquisition strategy to train a meta-learner that can capture cross-task transferable knowledge from a series of NLP tasks. Nevertheless, the challenge escalates when migrating this approach to the realm of code representation, especially in the "Big Code" era (Allamanis et al., 2018). The complicated scenario, marked by multiple task types and a myriad of programming languages (PLs), renders the direct application of prior NLP methodologies suboptimal (Phan et al., 2021). In light of these complexities, utilizing the idea of a "meta-leaner" for various forms of knowledge transfer appears to be promising in the field of code intelligence. Therefore, a research question naturally arises: *Can we develop a unified learning framework for CodePTMs that captures transferable knowledge across both code-related tasks and programming languages to enhance the code representation learning?*

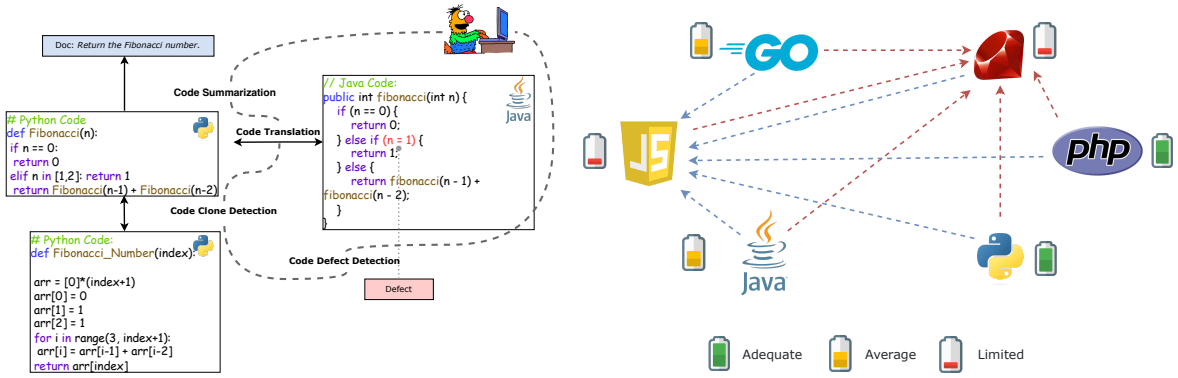
As a programmer, learning trajectories of humans have told us that: the experience of performing multiple code-related tasks (*e.g.*, debugging, writing docs) or learning various PLs (*e.g.*, C, Python) should not cancel each other out but be able to reinforce each other. The insights and skills gleaned from one language create a foundational understanding that eases the learning curve for subsequently encountered languages operating on similar principles. As a programmer masters more PLs, it becomes progressively attainable to pick up the next one. These are the innate human capability of knowledge generalization.

Drawing upon the human capabilities outlined

✉ Corresponding author.



*LREC-COLING 2024*, pages 16713–16726




(a) Cross-task code representation learning (b) Cross-language code representation learning

Figure 1: (a) A CodePTM (*e.g.*, CodeT5, PLBART) will learn through a series of code-related tasks such as code summarization and clone detection in the learning process, in order to acquire cross-task knowledge of code representation. (b) In the currently available code corpora (both bimodal and unimodal data), there is an imbalance between different PLs. Nonetheless, different languages share similar programming principles so that they can "support" each other through the models' learning cross-language knowledge.

previously (Zhu et al., 2022), this work proposes TransCoder, a unified transferable code representation learning framework. In particular, Figure 1(a) showcases the procedure of learning multiple tasks that cover both code understanding and generation. Then, Figure 1(b) illustrates the scenario of using other PLs with an abundant volume of data[1] to enhance the one with insufficient examples, which is similar to multilingual learning in NLP (Üstün et al., 2022). We describe this procedure of learning cross-task and cross-language representation as *universal code-related knowledge acquisition*. To be specific, a transferable prefix that plays the role of "universal-learner" is employed to acquire and bridge the knowledge of different tasks and PLs based on continual learning (Chen and Liu, 2018). This process can be analogous to people learning different languages and using them in various contexts. Once a sufficient amount of knowledge has been "assimilated" for a particular task, the prefix will be concatenated to a fresh CodePTM to commence the next phase of training, and so forth. It is a similar process for cross-language knowledge acquisition by TransCoder.

In addition, the adaptability of transferable representations enables low-resource learning on downstream tasks, which can serve as an alternative under extreme data scarcity. Extensive experiments among several PLs and code-related tasks demonstrate the effectiveness of TransCoder. Our main contributions are summarized as follows:

- In this paper, we propose TransCoder, which is a novel transferable framework for both code understanding and generation.
- A prefix-based universal learner is presented to capture general code-related knowledge, enabling unified transferability and generalization. Besides, TransCoder makes it feasible to drive cross-language code representation learning that empowers PLs with insufficient training data to conquer challenging tasks.
- Extensive experiments demonstrate the capability of TransCoder, which not only boosts reciprocal reinforcement among tasks but also mitigates the unbalanced data problem.

[1]the data includes bimodal data that refer to parallel data of NL-PL pairs and unimodal stands for pure codes without NL comments.

## 2. Related Works

**Pre-trained Language Models for Code** Pre-trained language models have significantly advanced performance in a broad spectrum of NLP tasks. Inspired by the success of Transformer-based pre-trained models, attempts to apply pre-training to source code have emerged in recent years (Xu et al., 2022; Zan et al., 2023). Feng et al. (2020) first propose CodeBERT, which is pre-trained on NL-PL pairs in six languages with masked language modeling (MLM) and replaced token detection (RTD; Yang et al., 2019) objectives. GraphCodeBERT (Guo et al., 2021) inaugurates the utilization of information contained in the Abstract Syntax Tree (AST), extracting structural information from source code to enhance pre-training. PLBART (Ahmad et al., 2021) adapts the BART architecture and is pre-trained with denoising objectives on Python/Java code data and NL posts. Builds on the T5 (Raffel et al., 2020) model architecture, CodeT5 (Wang et al., 2021b, 2023) is designed to support both code understanding and generation. It allows for multi-task learning and considers crucial token types (*e.g.*, identifiers). UniXcoder (Guo et al., 2022) uses a multi-layer

Transformer-based model that follows UniLM (Dong et al., 2019) to utilize mask attention matrices with prefix adapters. Recently, Nijkamp et al. (2023b,a) release a family of models across different scales for program synthesis, filling a gap between larger and smaller CodePTMs.

**Transfer-Learning** Over the past few years, there have been substantial advancements in transfer-learning techniques within the NLP community, leading to significant improvements in a wide range of tasks (Ruder et al., 2019). Transfer-learning leverages the knowledge contained in related source domains to enhance target learners' performance on target domains (Zhuang et al., 2021). Previous research demonstrates effective transfer from data-rich source tasks (Phang et al., 2018), especially for the scenario that reasoning and inference are required (Pruksachatkun et al., 2020). When considering various tasks as related, transfer learning among multitasks has already shown superior performance compared to single-task learning (Aghajanyan et al., 2021). Wang et al. (2021a) employ prompt tuning-based methods to transfer knowledge across similar NLP tasks for the purpose of mutual reinforcement. Recently, Vu et al. (2022) adopt the method of learning a prompt on one or more source tasks and utilizing it to initialize the prompt for target tasks.

**Continual Learning** Continual learning is also referred to as incremental or lifelong learning (Chen and Liu, 2018; Rao et al., 2019; Parisi et al., 2019), which is inspired by the learning process of human beings. The primary criterion is acquiring knowledge gradually while preventing catastrophic forgetting (Kirkpatrick et al., 2017) and then applying the knowledge for future learning. To be specific, the model undergoes training on multiple tasks or datasets in a sequential manner in order that remember the previous tasks when learning the new ones. Existing studies on continual learning also propose combinations of different techniques (Cao et al., 2021) to reduce the risk of forgetting, as well as continual pre-training (Sun et al., 2020) that supports customized training tasks. Recently, such techniques have been applied to code generation models through sequential training (Yadav et al., 2023) and prompting (Razdaibiedina et al., 2023).

# 3. TransCoder

In this section, we formally present the model architecture and the universal knowledge acquisition[2] process of TransCoder.

## 3.1. Overview

Bengio et al. (2021) indicates the next stage of deep learning by comparing current AI approaches with the learning capabilities of human beings. In the domain of code intelligence, a recent work (Zhu et al., 2022) considers human behaviors in reading programs. In this paper, we take a step forward in that TransCoder takes human *inherent skills* into account. Like human programmers, the models have the capacity to learn programming languages incrementally and enhance their overall comprehension of codes when being exposed to related tasks. Specifically, we define the transferable knowledge of code representation learning as *universal code-related knowledge*. As is demonstrated in Figure 2, *knowledge prefixes* concatenated with the CodePTMs play the role of the *universal knowledge provider* that bridges different downstream tasks. The first stage of TransCoder is *Source Tasks Training*, acquiring cross-domain and cross-language knowledge by continual learning (Sun et al., 2020), which is the core idea that enables CodePTM to "learn coding" like humans. The second stage is *Target Task Specification*, which applies the learned knowledge to new unseen code-related tasks through prefix concatenation.

## 3.2. Universal Knowledge Prefix

Inspired by deep prompt-tuning (Li and Liang, 2021; Liu et al., 2022; Xie et al., 2022; Choi and Lee, 2023), which prepends tunable prefixes to hidden states at every layer of the transformer-based models. We employ knowledge prefixes $\mathcal{F}_{uni}$ that could be used to acquire universal code-related knowledge. The knowledge prefixes are then concatenated to the CodePTM $\psi_i$ for both source task training and target task specification. Precisely, the prefixes inject universal knowledge into the attention modules of the CodePTMs, both the prefix and the knowledge injection procedure would not be constrained by the model backbone (*e.g.*, CodeT5, PLBART, CodeGen). Moreover, the prefix further supports cross-task and cross-language learning in a unified way: No matter what kinds of tasks we are tackling or how many languages are involved, the usage and effectiveness remain the same.

[2]For a more precise understanding, here we clarify that TransCoder acquires cross-task and cross-language knowledge by employing continual learning. Our core idea is to learn universal knowledge across tasks, not to learn a better parameter initialization oriented toward few-shot learning or fast adaptation. Even so, TransCoder can indeed support low-resource scenarios, as is shown in Section 3.4 and Section 4.5.

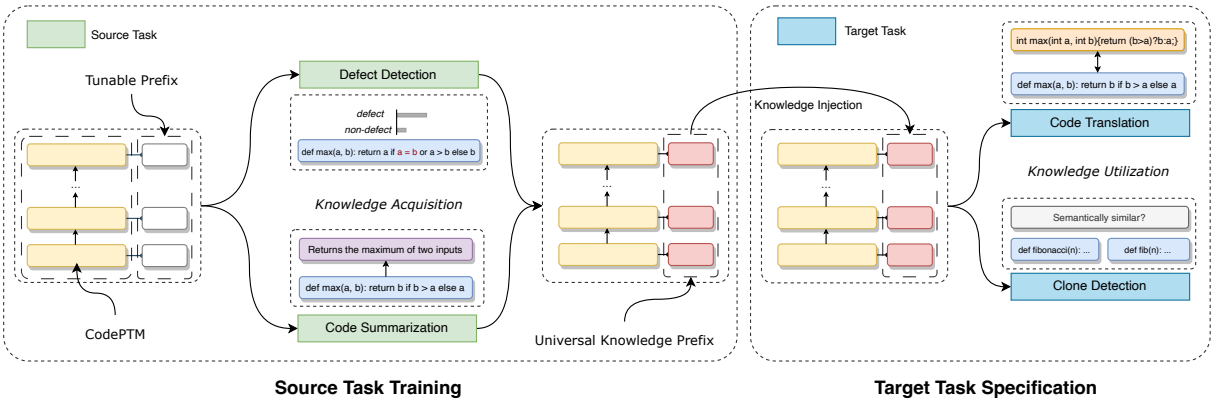



Figure 2: An illustration of the architecture of TransCoder. (1) In the source task training stage, tunable universal knowledge prefixes are first randomly initialized and prepended with a CodePTM (*e.g.*, CodeT5, PLBART, CodeGen). The whole model is tuned by back-propagation. (2) For the target tasks specification stage, we prepend these universal knowledge prefixes to a new CodePTM, effectively infusing universal knowledge into the model. For brevity, we choose a cross-task scenario and use some representative tasks as illustrations in this figure, which means using the knowledge acquired from code summarization/defect detection to enhance the performance of code translation/clone detection. The order of tasks or languages could be rearranged flexibly. (Best viewed in color.)

### 3.3. Training Procedure

**Source Tasks Training.** The first stage of TransCoder is source task training, which aims to acquire and transfer code-related knowledge to the universal knowledge prefix. We define a task that provides universal code-related knowledge as a source task[3] $t$, and the collection of $k$ source tasks as $T$. For code summarization that has six different PLs, we "break" it into six independent source tasks. L ikely, the task of code translation between C# and Java is defined as two source tasks as well. We note their training data as source data $D(t)$. Figure 1(a) gives an intuitive understanding of source task training. Similarly, Figure 1(b) provides a straightforward illustration of using tasks in other programming languages for universal knowledge acquisition and then supporting target tasks.

**Universal Knowledge Acquisition.** Algorithm 1 briefly describes how TransCoder acquires code-related universal knowledge. The inner loop employs the strategy of continual learning (Sun et al., 2020). For each task $t$ sampled from $T$, the trainer uses back-propagation to optimize the entire model with the cross-entropy loss. In this process, both the parameters of the knowledge prefix and the CodePTM $\psi_i$ are updated by the back-propagation. The outer loop ensures the knowledge prefix "absorbs" sufficient universal knowledge through $e$ epochs of source task training. In each epoch, the trainer will iterate through all of the source tasks.

[3]For the code summarization task, the training data for each source task corresponds to a sub-dataset of CodeSearchNet dataset partitioned by different programming languages.

**Algorithm 1** Source Tasks Training

**Input:** Source task $t$, Universal knowledge prefix $\mathcal{F}_{uni}$, Universal knowledge $\theta$, Task-specific model $f_t$ , CodePTM $\psi_i$
**Output:** Updated universal knowledge: $\theta$

**function** $Trainer(\mathcal{F}_{uni},\theta,f_t,\psi_i)$
  Randomly initialize $\theta$
  **for** each epoch $e$ **do**
    **for** each task $i$ in $t$ **do**
      Adaptive sample data batch $D(i)$ from source tasks.
      Get $loss_i \leftarrow L(f_i(\mathcal{F}_{uni}(\theta), \psi_i))$
      Update universal knowledge and CodePTM model parameters:
      $\theta, \psi_i \leftarrow \theta, \psi_i - \alpha \text{grad}_{\theta,\psi_i} loss_i$
    **end for**
  **end for**
**end function**

In practice, we first randomly select a source task $t$ from available tasks and then sample a batch from the task's training data $\mathcal{D}(t)$. The knowledge prefix $\mathcal{F}_{uni}$ first holds random knowledge $\theta_0$ , *i.e.*, initialized with random parameters. During source task training:

$$\overbrace{\theta_0 \longrightarrow \theta_1}^{T_0 \cdots T_k} \rightarrow \cdots \rightarrow \theta_{e-1} \rightarrow \theta_e$$

Each transition refers to the trainer iterating through all the $k$ source tasks in one epoch. During this procedure, the knowledge prefix $\mathcal{F}_{uni}$ will be moved to a new CodePTM whenever a new task comes, as is shown in Figure 2. After $e$ epochs of training, $\mathcal{F}_{uni}$ captures enough universal code-related knowledge among different tasks or PLs.

**Source Data Sampling.** In source task training, we are aware that the training sample sizes of each source task might vary considerably. Learning directly from these datasets would make the universal knowledge prefix biased toward tasks with large datasets. Thus, we employ an "Adaptive Sampling" strategy. When TransCoder samples $M$ training data from $\mathcal{D}(1), \mathcal{D}(2), \cdots, \mathcal{D}(M)$, instead of randomly selecting training samples, it employs stratified sampling where training instances are selected with the probability proportional to the dataset distribution $P(\mathcal{D}(k))$:

$$P\left(\mathcal{D}(k)\right) = \frac{\log |\mathcal{D}(k)| + \delta}{\sum_{\tilde{k}=1}^{M} \log |\mathcal{D}(\tilde{k})| + \delta}, \quad (1)$$

where $\delta > 0$ is the smoothing factor. This strategy results in the over-sampling of small datasets and the under-sampling of large datasets.

**Target Task Specification.** This is the second stage of TransCoder. As is detailed in Algorithm 1, the universal knowledge acquired in source task training grants us knowledge prefixes that could be concatenated to new CodePTMs. It enhances the target task's performance by injecting code-related knowledge $\theta_e$ by prefix $\mathcal{F}_{uni}$. Figure 2 gives an example of utilizing universal knowledge to enhance target tasks. The CodePTM with the universal knowledge prefix is trained together in this stage for downstream adaptation.

### 3.4. Learning under Low-resource Scenarios

Apart from learning from the full dataset, the universal knowledge acquired from source tasks innately enables TransCoder to conduct code-related tasks under low-resource scenarios. This capability is instrumental in enhancing performance across tasks involving less common languages (*e.g.*, Ruby), where available data is a fraction compared to prevailing ones, and aids in processing new, less-resourced languages.

Under this setting, both the source tasks training algorithm and data sampling strategy are identical to Algorithm 1 and Equation 1 respectively. However, in the target task specification stage, we only need to utilize 5% - 20% data for downstream adaption. The settings are detailed in section 4.5.

## 4. Experiments

In this section, we demonstrate the performance of TransCoder on cross-task and cross-language code representation learning, including the scenario of learning from inadequate data. We aim to answer the research questions (RQs) listed below with our results and analysis.

- **RQ1**: Through learning transferable knowledge, can TransCoder boost mutual enhancement on code downstream tasks?
- **RQ2**: To what extent can TransCoder help programming languages with small sample sizes to get better results on complex tasks?
- **RQ3**: Under low-resource scenarios, to what extent can TransCoder surpass fine-tuning? Additionally, is TransCoder capable of matching the model fine-tuned on the whole dataset?

### 4.1. Tasks and Languages

We conduct our experiments on four code representation learning tasks from the CodeXGLUE benchmark (Lu et al., 2021) that can be divided into code understanding and code generation. All dataset statistics are available in Appendix A.

**Code Generation.** For code generation tasks, code summarization (Alon et al., 2019) aims to generate natural language comments for the given code snippet in a different programming language. We use the CodeSearchNet dataset for the experiment, Additionally, we can observe a huge gap between the data volume of Java and Ruby.

The second code generation task is code translation (Nguyen et al., 2015; Rozière et al., 2020). This task involves translating a code snippet from one programming language to another. Compared with the data volume of code summarization, this task has fewer available training samples.

**Code Understanding.** Code understanding tasks evaluate models' capability of understanding code semantics and their relationships. Clone detection (Svajlenko et al., 2014; Mou et al., 2016) measures the similarity between two code snippets. Defect detection (Zhou et al., 2019) seeks to predict whether the source code contains vulnerability to software systems. In our experiment, we use the BigCloneBench dataset in Java language and Devign dataset in C language.

### 4.2. Experimental Setup

**Backbone Models.** In our experiments, we employ the following CodePTMs with different scales and architectures as backbone models to verify the efficacy of our approach: CodeT5-base (Wang et al., 2021b), PLBART-base (Ahmad et al., 2021), CodeGen (Nijkamp et al., 2023b) [4], and CodeT5+ (Wang et al., 2023) [5].

[4] https://huggingface.co/Salesforce/codegen-350M-multi

[5] https://huggingface.co/Salesforce/codet5p-770m

| Methods | CLS2Trans | | Sum2Trans | | CLS2Sum | Trans2Sum | Sum2CLS | | Trans2CLS | |
|---|---|---|---|---|---|---|---|---|---|---|
| | BLEU | EM | BLEU | EM | BLEU | BLEU | Clone F1 | Defect Acc | Clone F1 | Defect Acc |
| **CodeT5** | | | | | | | | | | |
| Fine-Tuning | **81.63** | 65.80 | 81.63 | 65.80 | 19.56 | 19.56 | **94.97** | 64.35 | 94.97 | 64.35 |
| TransCoder | 81.43 | **67.00** | **82.12** | **68.20** | **20.39** | **19.77** | 93.70 | **66.58** | **95.39** | **66.36** |
| **PLBART** | | | | | | | | | | |
| Fine-Tuning | **78.17** | 62.70 | **78.17** | **62.70** | 17.93 | 17.93 | **92.85** | 62.27 | 92.85 | 62.27 |
| TransCoder | 76.00 | **63.40** | 74.50 | 56.00 | **18.62** | **18.25** | 92.28 | **64.58** | **92.91** | **64.98** |
| **CodeGen** | | | | | | | | | | |
| Fine-Tuning | 82.45 | 64.79 | **82.05** | 63.59 | 18.37 | 18.37 | 93.61 | 63.15 | 93.61 | 63.15 |
| TransCoder | **84.05** | **65.65** | 81.78 | **64.87** | **19.47** | **20.14** | **94.91** | **65.79** | **94.45** | **65.79** |
| **CodeT5+** | | | | | | | | | | |
| Fine-Tuning | 83.97 | 63.91 | 83.97 | 67.41 | 20.15 | 20.15 | 95.14 | 66.71 | 95.14 | 66.70 |
| TransCoder | **85.02** | **65.77** | **84.56** | **69.85** | **21.87** | **21.56** | **96.08** | **69.71** | **96.60** | **68.94** |

Table 1: The performance on the code cross-task learning with four backbones.

**Experimental Details.** All the experiments covered in this paper are conducted on a Linux server with 4 interconnected NVIDIA RTX 3090 GPUs. The model is trained with the Adam (Kingma and Ba, 2015) optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.999$. For the source tasks training, we set epoch $e$ as 2 for all cross-language knowledge acquisition. And for the cross-task part, epoch $e$ is set as 2/6/6 for using code summarization, code understanding (clone detection and defect detection), and code translation as source tasks respectively.

Regarding hyperparameter settings, we adhere to the configurations established by Husain et al. (2019) and Nijkamp et al. (2023b) for different tasks to bolster the fairness of our comparative analysis.

**Evaluations.** The code translation task between C# and Java is evaluated on the averaged BLEU-4 scores (Papineni et al., 2002) and Exact Match (EM). For code summarization, we report the average of TransCoder's performance with smoothed BLEU-4 (Lin and Och, 2004) metrics on six programming language tasks. At last, two code understanding tasks: clone detection and defect detection are evaluated on F1 score and accuracy respectively.

### 4.3. Main Results

**Effectiveness of Cross-Tasks Scenario.** Table 1 shows the main experiment result of cross-task code representation learning by TransCoder. Due to the space limit, we abbreviated the name of different cross-task scenarios in the form of `<Source>2<Target>`. For instance, "CLS2Trans" refers to using two classification tasks (clone detection and defect detection) as source tasks and then selecting code translation as the target task. Similarly, "Trans2Sum" means employing Java → C# and C# → Java translation as source tasks, and choosing code summarization as the target task.

From these experimental results regarding different source tasks and target tasks, we make the following observations. 1) Learning with TransCoder improves the overall performance of each code downstream task when using different backbones. 2) Regardless of the source tasks involved, target tasks with smaller training sample sizes are most benefited from TransCoder. *e.g.*, there exists a notable improvement in defect detection (with a modest sample size) in all combinations of backbones and source tasks. 3) Performance gains are evident across different backbones, with experiments on the CodeT5 series showing superior performance compared to other models of similar size. We hold the view that this phenomenon could be attributed to the distinct pre-training corpora: CodeT5 incorporates the CodeSearchNet (CSN) dataset, whereas PLBART and CodeGen do not. The universal knowledge is extracted from the bimodal part of the CSN dataset to a large extent, which innately aligns the inherent knowledge and grants a smoother learning process.

**Effectiveness on Cross-Languages Scenario.** For the community, there is a significant gap between the data volume of Java and Ruby. The latter only has $< 15\%$ of the former's available training sample size. Other languages like JavaScript also have a certain degree of data scarcity compared to Java or Python. This is another scenario where TransCoder proves to be useful.

Table 2 demonstrates the main results of cross-language learning by applying TransCoder. We observe that 1) The overall performance of summarizing code for different programming languages is enhanced. 2) Different languages get various levels of performance increase with the aid of TransCoder in code summarization. It is worth noticing that languages with relatively smaller training sample sizes are most benefited from our method.

Specifically, the performance on summarizing

| Settings | Ruby | JavaScript | Go | Python | Java | PHP | Overall |
|---|---|---|---|---|---|---|---|
| **CodeT5** | | | | | | | |
| Fine-Tuning | 15.24 | 16.21 | 19.53 | 19.90 | 20.34 | 26.12 | 19.56 |
| TransCoder | **16.88** | **18.45** | **20.40** | **20.17** | **21.28** | **27.28** | **20.74** |
| **PLBART** | | | | | | | |
| Fine-Tuning | 13.97 | 14.13 | 18.10 | **19.33** | 18.50 | **23.56** | 17.93 |
| TransCoder | **15.32** | **15.00** | **18.67** | 19.27 | **19.44** | 23.52 | **18.54** |
| **CodeGen** | | | | | | | |
| Fine-Tuning | 13.48 | 16.54 | 18.09 | 18.31 | 19.41 | 24.41 | 18.37 |
| TransCoder | **15.93** | **19.92** | **19.64** | **20.41** | **21.02** | **26.12** | **20.51** |
| **CodeT5+** | | | | | | | |
| Fine-Tuning | 15.63 | 17.93 | 19.64 | 20.47 | 20.83 | 26.39 | 20.15 |
| TransCoder | **18.09** | **20.12** | **20.79** | **21.93** | **22.10** | **27.65** | **21.78** |

Table 2: Comparison between cross-language learning by TransCoder and full fine-tuning, using four backbone models with varying scales.

JavaScript code is significantly higher than the baseline, and Ruby also has a notable improvement. This phenomenon is in line with our expectation that universal code-related knowledge can compensate for the shortcomings of smaller sample sizes. Furthermore, as the model size increases, the capacity to memorize and utilize universal knowledge is enhanced, leading to significant improvements across all languages.

### 4.4. Further Analysis

**Ablation Study.** Using the knowledge prefix to absorb and transfer knowledge from different kinds of code-related tasks is one of the key ideas of TransCoder. Here we conduct an ablation study to understand the effect of using knowledge prefixes. We employed a prefix with random knowledge $\theta_0$, *i.e.*, with randomly initialized parameters to verify the effectiveness of the universal knowledge acquisition process on source tasks.

As is illustrated in Table 3. We observe that using the prefix with random knowledge brings worse performance on both two kinds of knowledge transfer, particularly in defect detection. This further proves the capability of knowledge prefixes to leverage crucial information of code.

| **Methods** | Sum2CLS | | Trans2CLS | |
|---|---|---|---|---|
| | Clone F1 | Defect Acc | Clone F1 | Defect Acc |
| **CodeT5** | | | | |
| Random Knowl. | 92.38 | 60.76 | 93.68 | 60.29 |
| Universal Knowl. | **93.70** | **66.58** | **95.39** | **66.36** |
| **PLBART** | | | | |
| Random Knowl. | 90.96 | 61.02 | 91.15 | 61.64 |
| Universal Knowl. | **92.28** | **64.58** | **92.91** | **64.98** |

Table 3: Ablation study of the effectiveness of universal code-related knowledge.

We also evaluate the effectiveness of universal code-related knowledge by observing the source task training process. Figure 3 indicates that with the help of TransCoder, the model can converge faster and reach a new optimum on an unseen code understanding task.

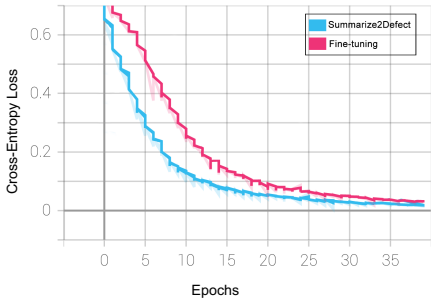



Figure 3: Comparison of defect detection task between fine-tuning and TransCoder with the knowledge of code summarization (PLBART backbone).

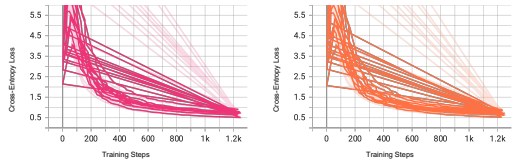
**Advantages over Sequential Fine-Tuning.** TransCoder leverages transferable knowledge across different languages and tasks to aid the learning of unseen new tasks. We compare it with straightforward sequential fine-tuning to further demonstrate the effectiveness of our approach.

| **Methods** | Sum2CLS | | CLS2Trans | | |
|---|---|---|---|---|---|
| | Clone F1 | Defect Acc | BLEU | EM | CB |
| **CodeGen** | | | | | |
| Sequential Tuning | 93.04 | 64.89 | 83.39 | 64.12 | 84.49 |
| TransCoder | **94.91** | **65.79** | **84.05** | **65.65** | **86.02** |
| **CodeT5+** | | | | | |
| Sequential Tuning | 94.21 | 68.02 | 84.41 | 62.74 | 85.74 |
| TransCoder | **96.08** | **69.70** | **85.02** | **65.77** | **87.97** |

Table 4: Ablation study of the effectiveness of universal code-related knowledge. CB: CodeBLEU.

| Settings | Ruby | JavaScript | Go | Python | Java | PHP | Overall |
|---|---|---|---|---|---|---|---|
| **5% Data** | | | | | | | |
| Fine-Tuning | 13.96 | 14.68 | 18.04 | 18.30 | 18.74 | 23.42 | 17.86 |
| TransCoder | 14.21 | 15.14 | 19.16 | 19.36 | 18.23 | 23.68 | 18.30 |
| **10% Data** | | | | | | | |
| Fine-Tuning | 15.22 | 15.12 | 19.06 | 19.20 | 19.32 | 24.95 | 18.81 |
| TransCoder | 16.05 | 16.63 | 20.21 | 20.07 | 20.48 | 26.20 | 19.94 |
| **20% Data** | | | | | | | |
| Fine-Tuning | 15.23 | 16.01 | 19.44 | 19.91 | 20.38 | 25.51 | 19.41 |
| TransCoder | 16.11 | 17.25 | 20.28 | 20.11 | 20.73 | 26.96 | 20.24 |

Table 5: Varying degrees of low-resource cross-language learning by TransCoder, using code summarization and CodeT5 backbone for evaluation.




(a) I (b) II

(c) III (d) IV

Figure 4: Employing different source tasks training order when utilizing CodeT5 backbone. Four Roman numerals denote four distinct, randomly determined sequences of source tasks.

As shown in Table 4, TransCoder significantly outperforms the direct sequential learning of different tasks. This highlights the advantage of infusing knowledge through a universal knowledge prefix.

**Stability of TransCoder.** To assess the potential oscillation in performance due to the training sequence of source tasks, we perform experiments on the training sequence of source tasks. We train our universal knowledge prefix for cross-language learning in various orders. The results, visualized in Figure 4, clearly indicate minimal variation in the final performance metrics, and problems like "catastrophic forgetting" do not occur. This also verifies the effectiveness of our adaptive sampling strategy that prevents the knowledge prefix from being over-reliant on certain tasks.

Additional comparative experiments on other backbones and more case studies presented in Appendix B.

### 4.5. Low-resource Scenarios

As stated in section 3.4, TransCoder is also capable of conducting code-related downstream tasks under low-resource settings. To verify our claim, we conduct experiments on code summarization, which is the most challenging task for code representation learning. In the experiment, whenever we specify a language as the target task of code summarization, we provide the knowledge of five other languages to TransCoder as source tasks.

Table 5 demonstrates the main results of applying TransCoder to three different scenarios, which only use 5%, 10%, and 20% of the CodeSearchNet (Husain et al., 2019) bimodal data[6] for summarizing code across six programming languages.

It is clear that TransCoder significantly surpasses fine-tuning under the same availability of data in most cases. Moreover, we only require 10% of the data to approximate the performance of model tuning on the whole dataset for each language. Besides, by utilizing 20% of the data, the performance of code summarization can significantly surpass fine-tuning the model on the **whole dataset**. Additionally, it is worth noticing that the major improvement in code summarization still comes from Ruby and JavaScript, which further confirms that acquiring knowledge from languages with rich corpus can aid subsequent learning from the ones with modest sizes. Consequently, TransCoder emerges as a practical tool for bridging the gap in language models' proficiency across programming languages of varying corpus richness and availability.

## 5. Conclusion

In this paper, we proposed **TransCoder**, which is a unified transfer learning framework for both code understanding and generation. We draw inspiration from 1) how human beings learn to write codes that the difficulty of learning the next programming language decreases with the aid of previous efforts and experience, and 2) learning multiple code-related tasks will deepen the understand-

[6]For a fair comparison, we sample the training data from CodeSearchNet at the same specified rate for all languages.

ing of the programming domain. TransCoder employs a two-stage learning strategy that 1) in the source task training stage, our proposed knowledge prefix will absorb universal code-related knowledge through continual learning among various tasks. 2) in the target task specification stage, the model will be benefited from the prepared knowledge and utilize it to learn new target tasks. Extensive evaluations of prevailing backbones and benchmarks demonstrate that TransCoder significantly enhances the performance of a series of code-related tasks among several programming languages. Moreover, further analysis shows that our method can drive PTMs to conduct downstream tasks under low-resource scenarios, while concurrently mitigating computational overhead. We hold the view that TransCoder serves as not only a significant stride towards optimizing performance on established tasks but also a catalyst for advancing the application of neural code intelligence within less-resourced programming languages.

## Limitations and Ethical Consideration

**Limitations.** The limitations of our work are shown below:

- We primarily rely on the CodeXGLUE Benchmark Dataset (Lu et al., 2021) to evaluate the effectiveness of our method. We hold the view that incorporating more code corpora with diversity, *e.g.*, BigQuery[7], can further exploit the benefits of TransCoder and extend it to more tasks and languages.
- Due to the limited computational resources, our experiments mainly rely on models with encoder-decoder and decoder-only architecture for both code understanding and generation tasks. We leave experiments based on other CodePTMs with encoder-only (Kanade et al., 2020; Feng et al., 2020) architecture as future works.
- Owing to constraints in data availability, we assess TransCoder's efficacy under low-resource scenarios by employing a subsampling technique on the existing dataset. A more comprehensive evaluation involving less commonly used programming languages in the real world remains a prospect for future endeavors.

**Ethical Considerations.** The contribution in this paper, TransCoder, is fully methodological. Our approach focuses on a new learning framework for code-related downstream applications, which encourages mutual reinforcement between tasks and languages. It can be applied to a series of scenarios, *e.g.*, low-resource and sample imbalance. Therefore, this contribution would be beneficial to the NLP community and has no direct negative social / ethical impacts.

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

## Acknowledgement

We thank our anonymous reviewers for their insightful comments and suggestions. This work is supported by Shanghai "Science and Technology Innovation Action Plan" Project (No.23511100700).

## Bibliographical References

Armen Aghajanyan, Anchit Gupta, Akshat Shrivastava, Xilun Chen, Luke Zettlemoyer, and Sonal Gupta. 2021. Muppet: Massive multi-task representations with pre-finetuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5799–5811, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81.

Uri Alon, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*.

Yoshua Bengio, Yann LeCun, and Geoffrey E. Hinton. 2021. Deep learning for AI. *Commun. ACM*, 64(7):58–65.

Yue Cao, Hao-Ran Wei, Boxing Chen, and Xiaojun Wan. 2021. Continual learning for neural machine translation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 3964–3974, Online. Association for Computational Linguistics.

Zhiyuan Chen and Bing Liu. 2018. Lifelong machine learning, second edition. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 12(3):1–207.

YunSeok Choi and Jee-Hyong Lee. 2023. CodePrompt: Task-agnostic prefix tuning for program and language generation. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 5282–5297, Toronto, Canada. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

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.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1126–1135. PMLR.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland. Association for Computational Linguistics.

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

Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar T. Devanbu. 2016. On the naturalness of software. *Commun. ACM*, 59(5):122–131.

Muhammad Jamal and Guo-Jun Qi. 2019. Task agnostic meta-learning for few-shot learning. *CVPR*, pages 11711–11719.

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*, volume 119 of *Proceedings of Machine Learning Research*, pages 5110–5121. PMLR.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526.

Xiang Lisa Li and Percy Liang. 2021. Prefix-Tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 4582–4597. 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*, pages 501–507.

Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. 2022. P-tuning: Prompt tuning can be comparable to fine-tuning across scales and tasks. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2:*

*Short Papers)*, pages 61–68, Dublin, Ireland. 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. 2019. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692*.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 1287–1293.

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2015. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *ICASE*, pages 585–596. IEEE.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023a. Codegen2: Lessons for training llms on programming and natural languages.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023b. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. 2019. Continual lifelong learning with neural networks: A review. *Proceedings of the National Academy of Sciences*, 113(C):54–71.

Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Annibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: Multi-task learning with code-text transformer. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, pages 40–47, Online. Association for Computational Linguistics.

Jason Phang, Thibault Févry, and Samuel R. Bowman. 2018. Sentence encoders on stilts: Supplementary training on intermediate labeled-data tasks. *ArXiv*, abs/1811.01088.

Yada Pruksachatkun, Jason Phang, Haokun Liu, Phu Mon Htut, Xiaoyi Zhang, Richard Yuanzhe Pang, Clara Vania, Katharina Kann, and Samuel R. Bowman. 2020. Intermediate-task transfer learning with pretrained language models: When and why does it work? In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5231–5247, Online. Association for Computational Linguistics.

Xipeng Qiu, TianXiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. 2020. Pre-trained models for natural language processing: A survey. *SCIENCE CHINA Technological Sciences*, 63(10):1872–1897.

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

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. *Journal of Machine Learning Research*, 21(140):1–67.

Jathushan Rajasegaran, Salman Khan, Munawar Hayat, Fahad Shahbaz Khan, and Mubarak Shah. 2020. itaml : An incremental task-agnostic meta-learning approach. *CVPR*.

Dushyant Rao, Francesco Visin, Andrei Rusu, Razvan Pascanu, Yee Whye Teh, and Raia Hadsell. 2019. Continual unsupervised representation learning. In *NeurIPS*, pages 7645–7655.

Anastasia Razdaibiedina, Yuning Mao, Rui Hou, Madian Khabsa, Mike Lewis, and Amjad Almahairi. 2023. Progressive prompts: Continual learning for language models. In *The Eleventh International Conference on Learning Representations*.

Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *NeurIPS*.

Sebastian Ruder, Matthew E. Peters, Swabha Swayamdipta, and Thomas Wolf. 2019. Transfer learning in natural language processing. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*, pages 15–18, Minneapolis, Minnesota. Association for Computational Linguistics.

Yu Sun, Shuohuan Wang, Yukun Li, Shikun Feng, Hao Tian, Hua Wu, and Haifeng Wang. 2020. Ernie 2.0: A continual pre-training framework for language understanding. *Proceedings of*

*the AAAI Conference on Artificial Intelligence*, 34:8968–8975.

Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480. IEEE.

Ahmet Üstün, Arianna Bisazza, Gosse Bouma, Gertjan van Noord, and Sebastian Ruder. 2022. Hyper-X: A unified hypernetwork for multi-task multilingual transfer. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 7934–7949, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Tu Vu, Brian Lester, Noah Constant, Rami Al-Rfou', and Daniel Cer. 2022. SPoT: Better frozen model adaptation through soft prompt transfer. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5039–5059, Dublin, Ireland. Association for Computational Linguistics.

Chengyu Wang, Jianing Wang, Minghui Qiu, Jun Huang, and Ming Gao. 2021a. TransPrompt: Towards an automatic transferable prompting framework for few-shot text classification. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 2792–2802, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021b. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Tianbao Xie, Chen Henry Wu, Peng Shi, Ruiqi Zhong, Torsten Scholak, Michihiro Yasunaga, Chien-Sheng Wu, Ming Zhong, Pengcheng Yin, Sida I. Wang, Victor Zhong, Bailin Wang, Chengzu Li, Connor Boyle, Ansong Ni, Ziyu Yao, Dragomir Radev, Caiming Xiong, Lingpeng Kong, Rui Zhang, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. 2022. UnifiedSKG: Unifying and multi-tasking structured knowledge grounding with text-to-text language models. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 602–631, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, page 1–10, New York, NY, USA. Association for Computing Machinery.

Yichen Xu and Yanqiao Zhu. 2022. A survey on pretrained language models for neural code intelligence.

Prateek Yadav, Qing Sun, Hantian Ding, Xiaopeng Li, Dejiao Zhang, Ming Tan, Parminder Bhatia, Xiaofei Ma, Ramesh Nallapati, Murali Krishna Ramanathan, Mohit Bansal, and Bing Xiang. 2023. Exploring continual learning for code generation models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 782–792, Toronto, Canada. Association for Computational Linguistics.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. XLNet: Generalized autoregressive pretraining for language understanding. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2023. Large language models meet nl2code: A survey.

Zhou, Yaqin and Liu, Shangqing and Siow, Jingkai and Du, Xiaoning and Liu, Yang. 2019. *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks*. Curran Associates, Inc.

Renyu Zhu, Lei Yuan, Xiang Li, Ming Gao, and Wenyuan Cai. 2022. A neural network architecture for program understanding inspired by human behaviors. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5142–5153, Dublin, Ireland. Association for Computational Linguistics.

Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2021. A comprehensive survey on transfer learning. *Proceedings of the IEEE*, 109(1):43–76.

## Language Resource References

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

Shuai Lu and Daya Guo and Shuo Ren and Junjie Huang and Alexey Svyatkovskiy and Ambrosio Blanco and Colin Clement and Dawn Drain and Daxin Jiang and Duyu Tang and Ge Li and Lidong Zhou and Linjun Shou and Long Zhou and Michele Tufano and MING GONG and Ming Zhou and Nan Duan and Neel Sundaresan and Shao Kun Deng and Shengyu Fu and Shujie LIU. 2021. *CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation*.

## A. Dataset Statistics

| Language | Training | Dev | Testing |
|---|---|---|---|
| Go | 167,288 | 7,325 | 8,122 |
| Java | 164,923 | 5,183 | 10,955 |
| JavaScript | 58,025 | 3,885 | 3,291 |
| PHP | 241,241 | 12,982 | 14,014 |
| Python | 251,820 | 13,914 | 14,918 |
| Ruby | 24,927 | 1,400 | 1,261 |

Table 7: CodeSearchNet (Husain et al., 2019) data statistics for the code summarization task.

| Dataset | Language | Training | Dev | Testing |
|---|---|---|---|---|
| CodeTrans | Java - C# | 10,300 | 500 | 1,000 |

Table 8: CodeTrans (Nguyen et al., 2015) datasets statistics for code translation task.

| Dataset | Language | Training | Dev | Testing |
|---|---|---|---|---|
| BigCloneBench | Java | 900K | 416K | 416K |
| Devign | C | 21K | 2.7K | 2.7K |

Table 9: BigCloneBench (Svajlenko et al., 2014) and Devign (Zhou et al., 2019) datasets statistics for Clone detection and Defect Detection tasks.

## B. Case Analysis

Figure 5 demonstrates the effectiveness of universal code-related knowledge when using CodeT5 as the backbone model.

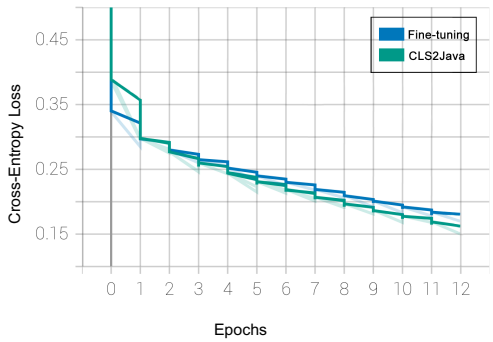



Figure 5: Comparison between TransCoder (with code understanding knowledge) and fine-tuning on summarizing Java code based on CodeT5 backbone.

It is clear that, through involving TransCoder, the loss function can be further optimized on the training set.

## C. Details of Code Summarization

Due to the space limit, we report the overall performance for code summarization in section 4.3. The details of summarizing code snippets of six different programming languages based on two backbone CodePTMs are given in Table 6.

## D. Hyperparameters Settings

The hyperparameters for model fine-tuning (baselines) and TransCoder are listed in 10 and Table 11.

| Methods | Ruby | JavaScript | Go | Python | Java | PHP | Overall |
|---|---|---|---|---|---|---|---|
| **CodeT5** | | | | | | | |
| Fine-Tuning | 15.24 | 16.21 | 19.53 | 19.90 | 20.34 | 26.12 | 19.56 |
| CLS2Sum | 16.62 | 16.54 | 20.67 | 20.07 | 22.41 | 26.03 | 20.39 |
| Trans2Sum | 15.88 | 16.30 | 19.99 | 19.81 | 20.63 | 26.03 | 19.77 |
| **PLBART** | | | | | | | |
| Fine-Tuning | 13.97 | 14.13 | 18.10 | 19.33 | 18.50 | 23.56 | 17.93 |
| CLS2Sum | 16.12 | 14.40 | 18.86 | 19.30 | 19.31 | 23.70 | 18.62 |
| Trans2Sum | 14.72 | 14.11 | 18.51 | 19.35 | 19.31 | 23.51 | 18.25 |

Table 6: Trans2Sum and CLS2Sum

| **Hyperparameter** | **value** |
|---|---|
| Batch Size | 8,16 |
| Learning Rate | {8e-6, 2e-5} |
| Max Source Length | {256, 320, 512} |
| Max Target Length | {3, 128, 256, 512} |
| Epoch | {2, 30, 50, 100} |

Table 10: Hyperparameters for fine-tuning

| **Hyperparameter** | **value** |
|---|---|
| Batch Size | 8,16,32 |
| Learning Rate | {2e-5, 1e-4, 5e-4} |
| Prefix Length | 32 |
| Source Train Epoch | {2, 4, 6} |
| Target Train Epoch | {2, 30, 50, 100} |
| Smoothing Factor $\delta$ | {0.5, 1} |

Table 11: Hyperparameters for TransCoder