# Two Birds with One Stone: Boosting Code Generation and Code Search via a Generative Adversarial Network

SHANGWEN WANG, Key Laboratory of Software Engineering for Complex Systems, College of Computer, National University of Defense Technology, Changsha, China
BO LIN, Key Laboratory of Software Engineering for Complex Systems, College of Computer, National University of Defense Technology, Changsha, China
ZHENSU SUN, School of Computing and Information Systems, Singapore Management University, Singapore
MING WEN, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China
YEPANG LIU, Research Institute of Trustworthy Autonoumous Systems and Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China
YAN LEI, School of Big Data & Software Engineering, Chongqing University, Chongqing, China
XIAOGUANG MAO, Key Laboratory of Software Engineering for Complex Systems, College of Computer, National University of Defense Technology, Changsha, China

Automatically transforming developers' natural language descriptions into source code has been a longstanding goal in software engineering research. Two types of approaches have been proposed in the literature to achieve this: code generation, which involves generating a new code snippet, and code search, which involves reusing existing code. However, despite existing efforts, the effectiveness of the state-of-the-art techniques remains limited. To seek for further advancement, our insight is that code generation and code search can help overcome the limitation of each other: the code generator can benefit from feedback on the quality of its generated code, which can be provided by the code searcher, while the code searcher can benefit from the additional training data augmented by the code generator to better understand code semantics. Drawing on this insight, we propose a novel approach that combines code generation and code search techniques using a generative adversarial network (GAN), enabling mutual improvement through the adversarial training. Specifically, we treat code generation and code search as the generator and discriminator in the GAN framework, respectively, and incorporate several customized designs for our tasks. We evaluate our approach in eight different settings, and consistently observe significant performance improvements for both code generation and code search. For instance, when using NatGen, a state-of-the-art code generator, as the generator and GraphCodeBERT, a

† Ming Wen and Yepang Liu are the corresponding authors.
This work was done when the first author was a visiting scholar at Southern University of Science and Technology.
Authors' addresses: Shangwen Wang, wangshangwen13@nudt.edu.cn, Key Laboratory of Software Engineering for Complex Systems, College of Computer, National University of Defense Technology, Changsha, China; Bo Lin, linbo19@nudt.edu.cn, Key Laboratory of Software Engineering for Complex Systems, College of Computer, National University of Defense Technology, Changsha, China; Zhensu Sun, zhensuuu@gmail.com, School of Computing and Information Systems, Singapore Management University, Singapore; Ming Wen, mwenaa@hust.edu.cn, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China; Yepang Liu, liuyp1@sustech.edu.cn, Research Institute of Trustworthy Autonoumous Systems and Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China; Yan Lei, yanlei@cqu.edu.cn, School of Big Data & Software Engineering, Chongqing University, Chongqing, China; Xiaoguang Mao, xgmao@nudt.edu.cn, Key Laboratory of Software Engineering for Complex Systems, College of Computer, National University of Defense Technology, Changsha, China.

2475-1421/2023/10-ART239
https://doi.org/10.1145/3622815

state-of-the-art code searcher, as the discriminator, we achieve a 32% increase in CodeBLEU score for code generation, and a 12% increase in mean reciprocal rank for code search on a large-scale Python dataset, compared to their original performances.

CCS Concepts: • **Software and its engineering → Reusability**; **Automatic programming**.

Additional Key Words and Phrases: Code Generation, Code Search, Generative Adversarial Network.

**ACM Reference Format:**
Shangwen Wang, Bo Lin, Zhensu Sun, Ming Wen, Yepang Liu, Yan Lei, and Xiaoguang Mao. 2023. Two Birds with One Stone: Boosting Code Generation and Code Search via a Generative Adversarial Network . *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 239 (October 2023), 30 pages. https://doi.org/10.1145/3622815

## 1 INTRODUCTION

Recommender systems in software engineering play a crucial role in improving developers' productivity [Luan et al. 2019; Miltner et al. 2019; Pelsmaeker et al. 2022; Raychev et al. 2014; Verbruggen et al. 2021; Zhang et al. 2022]. One of the main objectives of these systems is to effectively translate developers' intentions, which are usually specified in the form of natural language queries, into source code [Barke et al. 2023; Mohagheghi and Conradi 2007; Wang et al. 2023a]. This capability can significantly alleviate the burden of software development and facilitate important design and implementation decisions, such as choosing appropriate programming interfaces [James et al. 2020; Ko et al. 2004]. Developers, regardless of their different levels of programming proficiency, often raise queries of varying complexity about how to implement specific functionalities [Xu et al. 2022]. For example, a machine learning practitioner may want to change the dimension of a tensor (i.e., reshaping) and look for related solutions from online Q&A forums.[1]

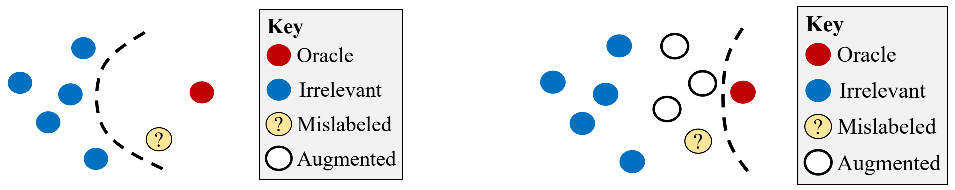
In the literature, there are mainly two types of techniques that can effectively transform developers' informally-specified queries into source code:

- **Code search.** In this type of approach, a code snippet is retrieved from a pre-collected codebase by mapping the semantic relevance between the code and the query [Bui et al. 2021; Sun et al. 2022; Yao et al. 2019]. Therefore, *the key ability of this type of approach is to accurately understand the semantics of diverse code snippets and then select the most relevant one.*
- **Code generation.** In this type of approach, code snippets are generated from scratch through the automatic learning of transformations from natural language to programming language. Generally, this is achieved via utilizing neural machine translation models, such as those described in the existing studies [Liu et al. 2020; Yin and Neubig 2017, 2018]. *The key ability of this type of approach is thus to generate code snippets that closely resemble those written by human developers.*

Over the years, considerable research efforts have been devoted into advancing the state of the art in these domains [Allamanis et al. 2018; Liu et al. 2021; Shin and Nam 2021; Wen et al. 2021]. Despite the promising results achieved, the effectiveness of existing techniques is still limited. Specifically, the previous study [Liu et al. 2020] found that deep learning based code generation approaches are usually unable to produce code that can pass test cases. Similarly, recent studies also showed that the latest large language models (LLMs) often struggle to generate semantically correct code [Chen et al. 2021; Li et al. 2022] and may even produce code with defects and vulnerabilities [Fan et al. 2023; Pearce et al. 2022]. Another study [Cambronero et al. 2019] observed that sophisticated code search models may not necessarily perform as well as a simple alternative, e.g., a bag-of-words approach. That is to say, it is necessary to further enhance the effectiveness of the code generation and code search techniques, in order to achieve satisfactory performance when applied in practice.

To that end, our insights come from the limitations of existing techniques:

[1]A related question has been viewed for 2K times over the years - https://stackoverflow.com/questions/45753153



**(a)** Normal situation. **(b)** Ideal situation.

**Fig. 1.** A conceptual illustration of normal vs. ideal decision boundaries of the code searcher.

- **Limitation for code generation.** Code generation techniques typically use the strategy of *teacher forcing* during training, where the model is trained to predict the next code token based on the oracle sequence at each time step [Chakraborty et al. 2022; Wang et al. 2021]. However, during inference, the model uses its own predicted outputs from previous time steps as input, resulting in the *exposure bias* problem [Bengio et al. 2015; Zhang et al. 2019] and potentially generating code that does not resemble human-written code. Informing the code generator about the quality of its generated code from a global perspective could be beneficial to address this issue, as this information can guide further improvements after the original training and enhance the code generator's key ability to generate code snippets that closely resemble human-written ones. Fortunately, **the code searcher can determine how well a code snippet fulfills an intended functionality** via calculating their semantic relevance, making it suitable for providing feedback to the code generator.
- **Limitation for code search.** Code search techniques are trained to differentiate between the developer-written code for a specific functionality (i.e., the oracle code) and a number of other semantically-irrelevant implementations (i.e., the non-oracle code). However, due to the limited quantity of code snippets used for training [Bui et al. 2021; Shi et al. 2022], when applied in practice, the large number of candidate code snippets can make it difficult for existing code searchers to distinguish between the oracle and non-oracle code, thus compromising their effectiveness in certain cases [Di Grazia and Pradel 2022; Liu et al. 2021]. To address this problem, a potential solution is to augment the non-oracle samples during training to help strengthen the key ability of the model on understanding code semantics [Shi et al. 2022]. A conceptual illustration is shown in Figure 1: in an ideal situation, the augmented data can help push the decision boundary of the learning model (i.e., the dashed black line) closer to the oracle code, thus avoiding potential mislabelling. In this regard, **the code generator can be promising to augment the training set of the code searcher**, as it can generate code that is distinct from existing non-oracle code and thus diversify the training data (the generated code can be categorized as non-oracle due to the gap that exists between the oracle and the generated code [Xu et al. 2022]).

Our analysis motivates us to leverage the complementary strengths of code generation and code search techniques to achieve mutual improvement. Taking the inspiration from the generative adversarial network (GAN) [Goodfellow et al. 2014], we can treat code generation and code search techniques as the generator and discriminator, respectively, where the former is trained to synthesize code snippets resembling those written by humans, and the latter is trained to distinguish between the generated and real-world code. In such a way, the discriminator evaluates the quality of the generated code and provides feedback to the generator, based on which the generator is trained to synthesize code snippets that resemble human-written ones to deceive the discriminator, thereby enhancing its ability. Meanwhile, the generated code can augment the non-oracle samples used for training the code searcher, thus diversifying the training data, reinforcing the code searcher to accurately understand the semantics of code snippets, and further improving its key ability. Prior studies have demonstrated that the adversarial training can strengthen the abilities of both the generator and discriminator [Radford et al. 2015; Salimans et al. 2016], and thus we are motivated to leverage such capabilities to further enhance both code generation and code search.

Based on this idea, in this paper, we propose an approach, CoCoGAN, which can simultaneously boost **Co**de generation and **Co**de search via a **G**enerative **A**dversarial **N**etwork. To apply the GAN framework on our tasks, we mainly face three challenges and accordingly take different solutions. First, the generator is not trained to produce a specific output in the vanilla GAN, while a code generator needs to produce code that accurately fulfills the intended functionality. To address this challenge, we utilize a **pairwise data** strategy where the input query to the generator and the input code to the discriminator are semantically matched (i.e., the code is written by developers which exactly aims at fulfilling the query). Additionally, the discriminator uses the query as the auxiliary guidance to distinguish between the real-world and generated code (**query guidance**), which enforces the generator to generate code fulfilling the intended functionality. Second, the vanilla GAN only works for continuous data, which makes the gradient undifferentiable when generating sequence data such as code tokens. To address this challenge, we utilize the policy gradient [Yu et al. 2017] and treat the generator's training as a reinforcement learning process, which enables updating the generator's parameters based on the **reward** value received from the discriminator. Third, during the adversarial training process, it is possible for the code searcher to forget previously learned knowledge on the code search task. To address this challenge, we employ a **replay-based lifelong learning** approach [Wang et al. 2019] during the adversarial training process, which continuously trains the code searcher to distinguish the semantics of different human-written code snippets.

In our experiments, we mainly investigate two state-of-the-art code generation techniques, i.e., CodeT5 [Wang et al. 2021] and NatGen [Chakraborty et al. 2022], and two state-of-the-art code search techniques, i.e., CodeBERT [Feng et al. 2020] and GraphCodeBERT [Guo et al. 2021]. We respectively treat the code generation technique as the generator and the code search technique as the discriminator, and integrate them into our CoCoGAN framework. We evaluate CoCoGAN on two large-scale datasets for Python and Java languages, each with 20K+ test queries, resulting in eight different experiment settings (2 generators $\times$ 2 discriminators $\times$ 2 programming languages). The experiment results demonstrate that across various settings, the adversarial training through CoCoGAN can consistently improve the performances of both the code generator and code searcher to a large extent, compared to those of the original models. For instance, when we use NatGen as the generator and GraphCodeBERT as the discriminator, we can achieve 32% and 30% increases in terms of the CodeBLEU value [Ren et al. 2020] for code generation, as well as 12% and 10% increases in terms of the mean reciprocal rank (MRR) [Gu et al. 2018] for code search, on the Python and Java datasets, respectively.

In summary, our study makes the following major contributions:

- **Significance:** We identify the complementary strengths of two effective ways for transforming developers' queries into source code, i.e., code generation and code search. Our study opens up a new direction for combining and improving these two types of techniques.
- **Approach:** We propose CoCoGAN, an adversarial network that holds the potential for simultaneously boosting the performances of both code generation and code search techniques.
- **Experiments:** We conduct extensive experiments in eight different settings, and find that our approach consistently improves the performances of both code generation and code search techniques. We also open source the artifacts of this study at **https://github.com/ShangwenWang/CoCoGAN** to facilitate replications and follow-up studies.

## 2 BACKGROUND

In this section, we provide fundamental background on code generation, code search, and generative adversarial networks.

### 2.1 Code Generation

Code generation aims at generating source code based on developers' requirements described in natural languages (NLs) [Liu et al. 2020]. Traditional approaches leverage formal methods to automatically generate source code [Harel et al. 1990; Whalen 2000], but the formal specifications are hard to create [Liu et al. 2020]. With the advances in deep learning, researchers propose to automatically learn the transformations from the requirements to source code. Specifically, the work by [Ling et al. 2016] proposes to treat code generation as a sequence-to-sequence translation process and builds a neural network that targets general-purpose programming languages like Python. Tranx [Yin and Neubig 2018] further takes the syntax structure of programs into consideration by making the network predict a sequence of grammar rules that can together compose the Abstract Syntax Tree (AST) of a program. The work by [Dong and Lapata 2018] explores the idea of using two decoders in the code generation task, where the first one aims at predicting a rough program *sketch* and the second one fills in the details.

More recently, pre-training techniques have received a lot of research attention due to their ability to address the data shortage problem faced by conventional supervised deep learning techniques (referred to as the *non-pre-training techniques*). Pre-training techniques perform self-supervised learning on a large amount of unlabelled data [Devlin et al. 2019; Radford et al. 2018], which enables them to leverage the power of big data. This has led to their outperformance of non-pre-training techniques on a number of code-related tasks [Zeng et al. 2022]. Among the existing pre-training techniques, CodeT5 [Wang et al. 2021], which adopts the text-to-text transfer transformer architecture, has been shown to achieve the state of the art on the code generation task [Niu et al. 2023; Zeng et al. 2022]. NatGen [Chakraborty et al. 2022] further continues the pre-training of the CodeT5 model with a customized task, in which the model learns to revise the given unnatural code (i.e., artificial code generated by deploying a set of meaning preserving transformations to existing code) and output a piece of natural code that keeps the original semantics.

### 2.2 Code Search

Instead of directly generating the source code, code search aims at helping developers retrieve some implementations that can serve as references for their development activities [Cambronero et al. 2019; McMillan et al. 2012; Sadowski et al. 2015; Xia et al. 2017]. Given a natural language query from the developer, such techniques search for the relevant code snippets from a large-scale code corpus. Traditionally, this process is mainly done by the information retrieval techniques such as keyword matching [Lv et al. 2015; McMillan et al. 2011]. However, these techniques are known to be suboptimal at capturing the semantic relations between the code and natural language queries [Gu et al. 2018]. Later on, researchers have proposed various deep-learning-based approaches to mitigate this limitation. For instance, DeepCS [Gu et al. 2018] jointly embeds code snippets and natural language descriptions into a high-dimensional vector space, where code snippets and queries can be matched according to their similarities. [Wan et al. 2019] design a multi-modal attention network that aggregates information from the token sequence, abstract syntax tree (AST), and control flow graph (CFG) for representing programs. CoaCor [Yao et al. 2019] explores to generate natural language annotations for each code snippet and use their similarities with the given query to refine the retrieval process. Sun *et al.* [Sun et al. 2022] model the code semantics through translating the code instructions into natural language descriptions with the help of specially designed heuristics.

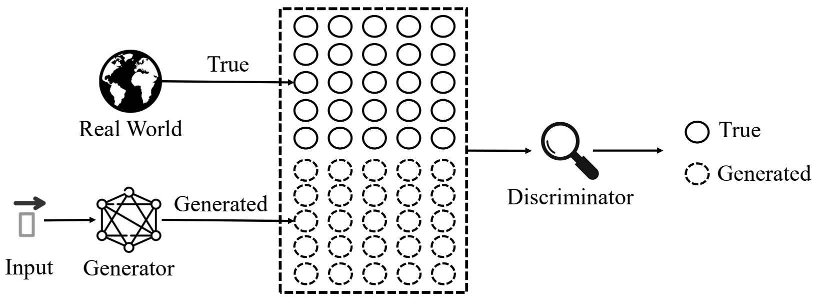


**Fig. 2.** The framework of a generative adversarial network.

Similar to recent advances in the code generation domain, pre-training techniques have also significantly improved code search techniques [Bui et al. 2021]. This can be attributed to the joint use of code and its corresponding comment (i.e., the natural language description of the code's functionality) for pre-training, which enables the model to understand the semantic information of the code. For instance, the widely-used pre-training task, Masked Language Modeling, requires the model to predict the randomly masked tokens in both the code and comment. Consequently, the state-of-the-art pre-training techniques, such as CodeBERT [Feng et al. 2020] and GraphCodeBERT [Guo et al. 2021], have been shown to outperform non-pre-training techniques significantly on the code search task [Zeng et al. 2022].

### 2.3 Generative Adversarial Network

The typical framework of the Generative Adversarial Network (GAN) [Goodfellow et al. 2014] is illustrated in Figure 2. It mainly consists of a generative network as the generator and a discriminative network as the discriminator, respectively. The generator first generates new samples based on the given inputs (usually random vectors), after which these generated samples are fed into the discriminator together with randomly-selected true samples from the real world. For each input sample, the discriminator will output a single scalar which represents the probability that the sample comes from the real world data rather than the generator, and the training process is based on the discriminator's outputs. Specifically, the generator aims at fooling the discriminator, and thus its training target is to maximize the probabilities assigned to those generated samples by the discriminator. In contrast, the discriminator tries to distinguish the true samples from the synthetic ones, and its training target is to maximize the probabilities assigned to the true samples while minimize the probabilities assigned to the generated samples at the same time. There is thus a type of adversarial relationship between the generator and the discriminator, and these two networks are trained and optimized iteratively.

## 3 MOTIVATION

### 3.1 Motivating Example

Figure 3a shows a method from the real-world project *pyxtuml* [2] on GitHub, which is a Python library for parsing, manipulating, and generating models. This project has been included in the *test set* of the well-known CodeSearchNet dataset [Husain et al. 2019]. The method is named as *serialize_instance* and its functionality can be described as "Serialize an instance from a metamodel", which is exactly the developer-written comment of this method. To achieve this target, developers first extract the meta information of the given instance through the method call `get_metaclass`

[2] https://github.com/xtuml/pyxtuml

```
1  # Oracle code
2  def serialize_instance(instance):
3      metaclass = get_metaclass(instance)
4      s = ''
5      for name, ty in metaclass.attributes:
6          value = getattr(instance, name)
7          s += '\n    '
8          s += serialize_value(value, ty)
9      s += '\n);\n'
10     return s
```

**(a)** The oracle code.

```
1  # Code generated by CodeT5
2  def serialize_metamodel(instance, metamodel):
3      if not isinstance(instance, metamodel.Metamodel):
4          raise TypeError("Expected Metamodel but got
5                          '{}'".format(type(instance)))
6      return serialize_metamodel(instance, metamodel)
```

**(b)** Code generated by CodeT5.

```
1  # Code retrieved by CodeBERT
2  def serialize_instances(metamodel):
3      s = ''
4      for inst in metamodel.instances:
5          s += serialize_instance(inst)
6      return s
```

**(c)** Code retrieved by CodeBERT.

**Fig. 3.** The developer-written code, the code generated by CodeT5, and the code retrieved by CodeBERT, for the query "Serialize an instance from a metamodel".

(cf. line 3), and then iteratively add these information into the string `s` within a loop (cf. lines 5 - 8). Finally, `s` is returned as the serialized result.

In Figures 3b and 3c, we show the results of two state-of-the-art code generation and code search techniques given the query "Serialize an instance from a metamodel" (the code generated by CodeT5 and the code retrieved by CodeBERT). These two pre-trained models are fine-tuned on the CodeSearchNet (CSN) dataset and the whole test set is used for the retrieval of CodeBERT. Based on the above example, we draw the following observations:

For code generation, in this example, the code generated by CodeT5 only performs a sanity check about the data type of the given instance and then recursively calls itself. The basic domain knowledge about programming language is that a recursive function should define the *terminating case* where the chain of recursion is broken, which should be known by even a novice programmer. However, such information is missed in the generated code, which means upon execution, the program will never stop because of the infinite loop. Consequently, compared with the oracle code, the generated code achieves a CodeBLEU value (a widely-used metric for code quality assessment [Ren et al. 2020]) of less than 10%, which is rather unsatisfactory. To summarize, this example shows that CodeT5 may produce low-quality and semantically-incorrect code that significantly differs from the human-written code. Unfortunately, as evaluated by recent replication studies [Niu et al. 2023; Zeng et al. 2022], CodeT5 achieves the state-of-the-art performance on code generation, meaning that code generated by other existing approaches may be less qualified.

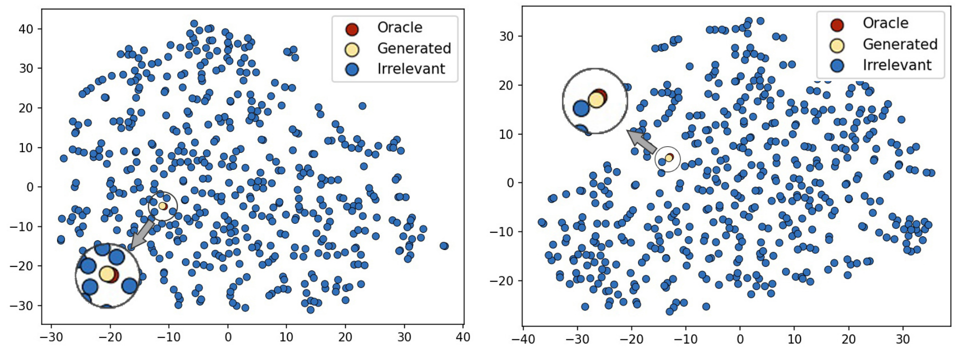
For code search, in this example, there is a subtle difference between the code retrieved by CodeBERT and the oracle code: the former utilizes a loop to deal with all the instances from a metamodel (cf. lines 4 - 5 in Figure 3c) and its functionality can be described as "Serialize **all** instances from a metamodel", while the latter only deals with a specific instance and its functionality, as introduced, is to "Serialize **an** instance from a metamodel". However, CodeBERT misunderstands the semantics of the retrieved method and ranks it at the top-1 position in the returned results, probably because the retrieved method and the oracle code have a high degree of overlapped tokens, such as `s` and `serialize`. In contrast, the oracle code is only ranked at the $4_{th}$ position, which means when applied in practice, developers will miss the opportunity to find the intended code from the top-3 recommendations and the user experience may thus be degraded [Wang et al. 2023b]. To summarize, this example shows that it may be hard for CodeBERT to accurately distinguish the semantics of code snippets in certain cases. Unfortunately, recent studies [Niu et al. 2023; Zeng et al. 2022] have also revealed that CodeBERT is the state-of-the-art code search technique, meaning that other existing approaches may even be less qualified on understanding code semantics.

## 3.2 Key Ideas

In the above examples, we have identified two key abilities that need to be strengthened for the current code generation and code search techniques: (1) generating high-quality code that resembles human-written code for code generator and (2) accurately understanding the semantics of code snippets for code searcher. Our key idea to enhance such key abilities is to utilize a generative adversarial network (GAN) in which the code generator plays the role of the generator whose generated code can enforce the code searcher to capture code semantics more precisely; and the code searcher serves as the discriminator, whose feedback on the code quality can enforce the code generator to generate code that resembles human-written code. The generator and the discriminator can be simultaneously strengthened during the adversarial training process. We next introduce the rationale of such a design.

From the perspective of the generator, the feedback from the discriminator forces it to become more skilled at generating code that resembles human-written code. Specifically, the generator aims to fool the discriminator into assigning high scores to its generated samples. In our context, the code generator aims to make the discriminator believe that its generated code is written by human developers. However, this is challenging as the discriminator has been trained with code written by human developers and has common sense knowledge about programming languages. For instance, the discriminator knows that a recursive function should define terminating cases. Therefore, the discriminator can easily identify any code that violates the coding practices of human developers. Consequently, to achieve its goal, the code generator should learn to generate code that does not contain fatal defects that contradict common coding practices of human developers. To summarize, with the aim of receiving positive feedback from the discriminator, the generator is reinforced to produce human-like code during the adversarial process, which enhances its key ability.

From the perspective of the discriminator, the code generated by the generator guides it to capture code semantics more accurately. Specifically, the aim of the discriminator is to distinguish the real samples from those generated by the generator. In our context, the code searcher needs to accurately identify the code that fulfills the requested functionality when disturbed by the generated code. This is non-trivial since (1) the generated code may be similar to the oracle in terms of the code tokens (e.g., in the shown example, the code generated by CodeT5 contains a number of overlapped tokens with the oracle such as `instance` and `metamodel`); and (2) during the adversarial process, the generator would become competent at generating code that resembles human-written code, which means the generated code may not violate common coding practices. As a result, making such distinctions by focusing merely on token similarity or contradictions to the common sense knowledge would be insufficient. The code searcher needs to be capable of capturing subtle differences between code semantics. For instance, in the example as shown in Figure 3, suppose the code generator now can generate a code snippet similar to the one retrieved by CodeBERT (without any contradiction to the common sense knowledge), the code searcher needs to understand that the presence of the loop in the code indicates that it deals with a number of instances. Generally, the progress of the generator in the adversarial process pushes the code searcher to better capture the code semantics, and ultimately, the key ability of the code searcher can be enhanced as well.



**(a)** Query: “zip a directory tree into an IO object”. **(b)** Query: “return a list of XML items”.

**Fig. 4.** Visualization of the vector representations of the code snippets.

## 3.3 Feasibility Analysis

A hypothesis of our approach is that during the adversarial training, the generated code from generator could help improve the code searcher. Our intuition is that, compared to the original non-oracle code in the training set, the generated code may exhibit greater semantic similarity to the oracle code, thereby pushing the decision boundary of the model closer to the oracle code, as illustrated in Figure 1. Intuitively, this could be achieved since (1) the generated code aims at fulfilling the same functionality as the oracle code, whereas other non-oracle code implements different functionalities; and (2) the generator may have already been well-trained (e.g., through fine-tuning) on the training set and therefore can generate code that is similar to the oracle. To test this hypothesis, we conduct an exploratory experiment on the *training set* of the CSN-Python dataset (the example shown in Figure 3 is out of the scope of this analysis since it is from the test set). Specifically, we embed the oracle and non-oracle code from the training set, as well as the generated code from the CodeT5 model fine-tuned on this dataset, into 768-dimensional vectors using the pre-trained CodeBERT model. We then check whether the generated code is closer to the oracle code in the vector space in terms of the cosine similarity. Our results show that, for approximately 70% of the queries, the generated code is closer to the oracle code than any of the existing non-oracle code. This indicates that our hypothesis holds: generally, the generated code is semantically similar to the oracle code, and the code searcher could become more competent at capturing such subtle differences by learning to distinguish between the generated and oracle code, thereby enhancing its key ability on understanding code semantics. It should be noted that the generator used in this exploratory experiment has not undergone any adversarial training yet. It is expected that through the adversarial training process, more code that is semantically similar to the oracle will be generated, thereby better strengthening the code searcher.

We also visualize the code vectors to help intuitively understand this phenomenon. Specifically, we use T-SNE [Van der Maaten and Hinton 2008] to reduce the dimensionality of the embedded vectors into 2D space. Figure 4 demonstrates the vectors of the oracle and non-oracle code, as well as the generated code, for two randomly selected queries. For better visualization, we randomly sample 500 irrelevant code snippets for each query. As we can see, the generated code is the closest one to the oracle code in the dimensionality-reduced space.

## 4 METHODOLOGY

In this section, we first introduce the vanilla generative adversarial network in detail and then delve into the key differences between our approach, `CoCoGAN`, and the vanilla generative adversarial network, which can be considered as the customized features of `CoCoGAN`.

**Algorithm 1:** The training process of GAN.

```
Input: the initial generator $G$, the initial discriminator $D$, the real-world data distribution $p_{real}(x)$, the input
       distribution $p_z(z)$.
Output: the well-trained generator and discriminator.
for number of training iterations do
    /* Training the discriminator.                                    */
    for $k$ steps do
        sample minibatch of $m$ samples from the real-world database $\{x^1, x^2, \ldots, x^m\}$
        sample minibatch of $m$ samples from the input distribution $\{z^1, z^2, \ldots, z^m\}$
        obtain the generated data from the generator $\{\widetilde{x}^1, \widetilde{x}^2, \ldots, \widetilde{x}^m\}$ where $\widetilde{x}^i = G(z^i)$
        update discriminator parameters $\theta_d$ to maximize the following:
            $\widetilde{V} = \frac{1}{m}\sum_{i=1}^{m} logD(x^i) + \frac{1}{m}\sum_{i=1}^{m} log(1 - D(\widetilde{x}^i))$
            $\theta_d \leftarrow \theta_d + \eta_d \nabla \widetilde{V}(\theta_d)$ where $\eta_d$ is the learning rate for the discriminator
    /* Training the generator.                                        */
    sample minibatch of $m$ samples from the input distribution $\{z^1, z^2, \ldots, z^m\}$
    update generator parameters $\theta_g$ to maximize the following:
        $\widetilde{V} = \frac{1}{m}\sum_{i=1}^{m} logD(G(z^i))$
        $\theta_g \leftarrow \theta_g + \eta_g \nabla \widetilde{V}(\theta_g)$ where $\eta_g$ is the learning rate for the generator
return G, D
```

## 4.1 Vanilla Generative Adversarial Network

A vanilla GAN consists of two neural networks, i.e., a generator $G$ with the parameters $\theta_g$ and a discriminator $D$ with the parameters $\theta_d$. Given the input $z$ which follows a certain distribution $p_z(z)$, the generator will output generated samples $G(z)$. These generated samples will then be fed into the discriminator together with samples from the real-world (denoted as $x$ that follows the real-world data distribution $p_{real}(x)$). For each input sample, the discriminator will output a single scalar which represents the probability of the input to come from the real world. For the generator, its aim is to fool the discriminator so that the generated samples can be assigned with high probabilities, and thus its training target is to maximize $logD(G(z))$; for the discriminator, its aim is to distinguish the real-world samples from those generated by the generator, and thus its training target is to maximize $logD(x)$ while minimizing $logD(G(z))$, which can also be considered as maximizing $logD(x)+log(1-D(G(z)))$. Consequently, the min-max adversarial loss for training the generator $G$ and the discriminator $D$ can be formulated as:

$$\min_{G}\max_{D} \mathbb{E}_{x\sim p_{real}(x)}[logD(x)] + \mathbb{E}_{z\sim p_z(z)}[log(1 - D(G(z)))] \tag{1}$$

The whole training process of GAN is illustrated in Algorithm 1. In each training iteration, the generator and the discriminator are separately trained. Typically, the discriminator is trained first using the generated samples from the generator, after which the generator is trained with the parameters of the discriminator ($\theta_d$) being fixed. Usually, in each iteration, the discriminator is optimized for multiple times (determined by a hyperparameter $k$) while the generator is optimized for only once. This decision is to ensure that the discriminator remains close to its optimal solution, which in turn provides valuable feedback to the generator [Goodfellow et al. 2014].

## 4.2 Customized Features of CoCoGAN

Figure 5 depicts the overview of our approach. To apply the GAN framework on the code generation and code search tasks, we mainly face three challenges and accordingly take different solutions for addressing such challenges. We highlight the differences between CoCoGAN and the vanilla GAN in the yellow parts as shown in Figure 5.

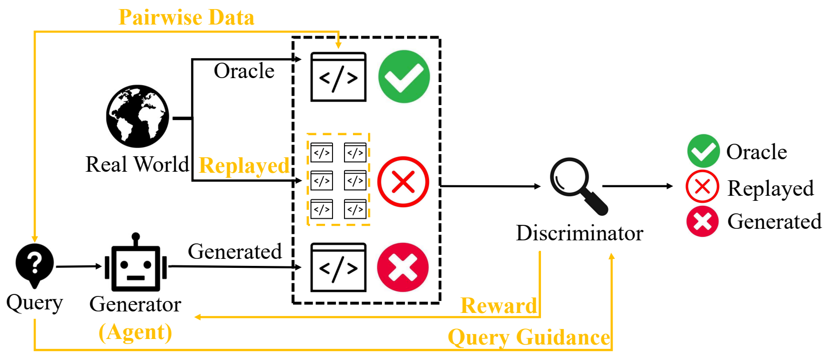


**Fig. 5.** The overview of our approach.

- **Challenge#1: Unpairwised training inputs.** In the vanilla GAN, the generator is not expected to produce a specific output given an input, since the real-world samples fed into the discriminator are randomly selected [Mirza and Osindero 2014]. However, in code generation, the code generator needs to produce the code snippet that can fulfill a specific functionality requested by a given natural language query. In CoCoGAN, to address this challenge, we adopt a **pairwise data** strategy where the real-world code fed into the discriminator and the query fed into the generator are semantically matched, and the query is further used to guide the prediction of the discriminator (**query guidance**).
- **Challenge#2: Undifferentiable gradients on sequence generation.** The vanilla GAN is typically used for generating continuous data, such as images, while the code generation task requires the generation of a sequence of discrete code tokens. This presents a challenge where the gradient of the generator's loss cannot be calculated through the conventional way in GAN. To address this challenge, we employ the policy gradient [Yu et al. 2017] and model the generator's training as a reinforcement learning process, in which the generator acts as an **agent** and the discriminator's feedback is treated as the **reward** signal to update the generator's parameters.
- **Challenge#3: Unmatched goal between the training target and the primary objective of the discriminator.** The discriminator of CoCoGAN is a well-trained code searcher whose primary objective is to perform the code search task, i.e., retrieving a code snippet from a large-scale corpus that fulfills a specific functionality. However, during the adversarial training process, the discriminator is tasked with distinguishing between the human-written and generated code. There is a notable difference between these two tasks: intuitively, retrieving a code snippet from a large-scale corpus needs to accurately capture the semantics of each code snippet, while distinguishing between the human-written and generated code is a binary action in nature. Therefore, the adversarial training may impact the code searcher's performance on the code search task by causing the model to forget previously-learned knowledge, a phenomenon known as the *catastrophic forgetting* problem [De Lange et al. 2021; Li and Hoiem 2017]. To address this challenge, we employ an effective **replay-based lifelong learning technique** [Wang et al. 2019] to maintain the capability of the discriminator on distinguishing different human-written code during the adversarial training.

*4.2.1 Pairwise data & Query guidance for addressing Challenge#1.* In the vanilla GAN, the real-world sample fed into the discriminator is randomly selected, and therefore has no correlation with the input to the generator. This lack of a specific target for the generator to aim for makes its output uncontrollable [Mirza and Osindero 2014]. As a consequence, the generator only needs to acquire knowledge about the high-level features of real-world samples to deceive the discriminator, as the discriminator has no way to verify detailed information about the generated sample (such as whether it belongs to a certain category or class). For example, if the generator is trained to generate images and the real-world image fed into the discriminator is a dog, the generator could successfully deceive the discriminator by generating other animals (such as a cat or a wolf) as long as the generated image appears to be human-like. However, the code generator needs to generate a code snippet that can fulfill the functionality requested by the query. That is to say, being an effective code generator requires the ability to generate semantically-correct code beyond simply avoiding making syntax errors. Inspired by the *Conditional GAN* [Mirza and Osindero 2014], in `CoCoGAN`, we take two steps to address this challenge. First, in contrast to the vanilla GAN, the discriminator in `CoCoGAN` receives a real-world code snippet that explicitly fulfills the desired functionality of the query, rather than a randomly selected sample. This ensures that the discriminator and generator are working with semantically matched pairwise data, allowing for more controllable generation. Additionally, the query is also fed to the discriminator to guide it in identifying the real-world code more accurately. This enables the discriminator to reference the query when distinguishing between real and generated code, thereby enforcing the generator to produce code that fulfills the intended functionality.

Formally, the generator $G$ takes as input a query $q$ (which now comes from the distribution of real-world queries used by developers $p_{query}(q)$) and outputs a generated code snippet $G(q)$. The discriminator is provided with both the query $q$ and the oracle code $o$ corresponding to the query (<$o$, $q$> is a piece of pairwise data) as inputs. It then tries to distinguish the oracle code from the generated one (i.e., maximizing $D(o, q)$ while minimizing $D(G(q), q)$), and the goal of the generator is to fool the discriminator (i.e., maximizing $D(G(q), q)$).

*4.2.2 Sequence-level policy gradient for addressing Challenge#2.* The vanilla GAN is mostly applied in computer vision tasks such as generating natural images [Denton et al. 2015], where the data is real-valued and continuous. However, the code snippet is composed of a sequence of discrete code tokens. Directly applying GAN on the sequence data is inappropriate, since the gradient of the discriminator's loss is used to guide the generator to slightly update its parameters, while there may be no token in the limited dictionary space corresponding to this slight change [Li et al. 2017; Yu et al. 2017]. More specifically, to generate a sequence, there is a sampling process for the generator at each time step, which means the code token sequence fed into the discriminator is not continuous data. This makes the whole "generator + discriminator" architecture (i.e., $D(G(z))$) not differentiable and the gradient ascent shown in the line 14 of Algorithm 1 not applicable.

To address the above challenge, in `CoCoGAN`, we borrow the idea of a variant of GAN, i.e., SeqGAN [Yu et al. 2017], to deal with the sequence data. Specifically, we use the policy gradient [Sutton et al. 1999] to avoid the differentiation difficulty for discrete data in the vanilla GAN. In such a way, the generator is treated as an agent of reinforcement learning (RL), the state is the generated code, and the discriminator is used to evaluate the generated code and feedback the reward for guiding the learning of the generator. The reward is estimated by the likelihood that the generated code would fool the discriminator. By doing so, the discriminator only needs to return a numeric value as the reward but does not need to participate in the calculation of gradient. Note that the gradient calculation of the discriminator is not affected by whether the input is sequence data or not.

The original SeqGAN approach [Yu et al. 2017] calculates the reward value for each decoding step to guide the prediction of the next token to be generated. However, this step-level reward calculation may not be appropriate for the code generation task because there can be different ways to implement a given functionality, which is known as the *program aliasing* phenomenon [Bunel et al. 2018; Sun et al. 2018]. To address this challenge, we adopt a *sequence-level* reward calculation strategy in our approach. Instead of focusing on individual decoding steps, the reward is calculated based on the entire generated token sequence, allowing the generator to consider the code generation process as a whole. This enables the generator to explore different ways of implementing the requested functionality, leading to more diverse code generation results.

Formally, given $m$ queries from the input distribution $\{q^1, q^2, \ldots, q^m\}$, the generator would generate $m$ code snippets $\{\widetilde{x}^1, \widetilde{x}^2, \ldots, \widetilde{x}^m\}$ where $\widetilde{x}^i = G(q^i)$. The expectation of the reward obtained by the generator can be calculated as:

$$\overline{\mathbb{R}}_{\theta_g} = \frac{1}{m} \sum_{i=1}^{m} D(\widetilde{x}^i, q^i) \cdot P_{\theta_g}(\widetilde{x}^i | q^i) \tag{2}$$

where $D(\widetilde{x}^i, q^i)$ denotes the reward obtained by the generator for generating $\widetilde{x}^i$, and $P_{\theta_g}(\widetilde{x}^i | q^i)$ denotes the probability of generating $\widetilde{x}^i$ given $q^i$.

The gradient of $\overline{\mathbb{R}}_{\theta_g}$ can thus be calculated as:

$$\nabla \overline{\mathbb{R}}_{\theta_g} = \frac{1}{m} \sum_{i=1}^{m} D(\widetilde{x}^i, q^i) \cdot \nabla P_{\theta_g}(\widetilde{x}^i | q^i) \tag{3}$$

Since the training of the generator is to maximize its reward value, we can now use gradient ascent to update its parameters.

*4.2.3 Replaying confusing exemplars for addressing Challenge#3.* Recall that the discriminator in CoCoGAN is a well-trained code searcher, which will be ultimately used for the code search task, i.e., retrieving a code snippet from a large-scale human-written corpus that fulfills a specific functionality. However, in the vanilla GAN, the discriminator is used to distinguish between real-world samples and generated samples during the adversarial training, i.e., distinguishing between the human-written code and the code generated by the generator. These two tasks have notable differences: from the perspective of task target, the former is to distinguish code snippets that implement various functionalities, while the latter is to distinguish between two code snippets that aim at fulfilling the identical functionality; moreover, from the perspective of task difficulty, the former requires accurate semantic understanding of each involved code snippet, while the latter is comparatively less complex as it only involves a binary classification problem. That is to say, the discriminator is firstly trained on one task, and then continuously trained on a new task, while still being expected to perform well on the old task, which is akin to the *lifelong learning* paradigm [De Lange et al. 2021; Li and Hoiem 2017]. However, this process can lead to *catastrophic forgetting* and the model may forget the knowledge learned from the old task. In the context of our approach, this means that the discriminator may not perform as well as before on the code search task after being trained to distinguish between human-written and generated code.

To address this challenge, we borrow the idea of a simple but effective lifelong learning technique, i.e., the replay-based approach [Gao et al. 2023; Wang et al. 2019], to alleviate the catastrophic forgetting problem. The key idea is to sample some exemplars from the old task for retraining during the learning of the new task, so that the previously-learned knowledge on the old task could be maintained in the model and thus the potential performance degradation on the old task can be avoided. Unlike randomly selecting such exemplars [Wang et al. 2019], in our approach, we select code snippets from the training set to serve as the replayed exemplars based on their token-level similarities to the oracle code. Specifically, to select the exemplars, we focus on the Jaccard similarity [Niwattanakul et al. 2013] between the token lists of the oracle code and the candidate code snippets. We refer to the selected snippets as *confusing exemplars* because they have similar tokens to the oracle code, which as observed in Figure 3, may easily confuse the code searcher. Specifically, given a query $q$, the generated code from the generator $G(q)$ and the

**Algorithm 2:** The training process of CoCoGAN.

```
Input: the code generator $G$, the code searcher $D$, the real-world query distribution $p_{query}(q)$, the code corpus $C$
       that contains the detailed implementation of each query.
Output: the adversarially-trained code generator and code searcher.
1  for number of training iterations do
2      /* Training the discriminator with lifelong learning.                                  */
3      for $k$ steps do
4          sample minibatch of $m$ queries from the real-world database $\{q^1, q^2, \ldots, q^m\}$
5          obtain the oracle code snippets of these queries $\{o^1, o^2, \ldots, o^m\}$
6          for each oracle code snippet $o^i$ do
7              obtain $l$ confusing exemplars $\{o^{i1}, o^{i2}, \ldots, o^{il}\}$ from $C$
8          obtain the generated code from the generator $\{\widetilde{x}^1, \widetilde{x}^2, \ldots, \widetilde{x}^m\}$ where $\widetilde{x}^i = G(q^i)$
9          update discriminator parameters $\theta_d$ to maximize the following:
10             $\widetilde{V} = \frac{1}{m}\sum_{i=1}^{m} logD(o^i, q^i) + \frac{1}{m}\sum_{i=1}^{m} log(1 - D(\widetilde{x}^i, q^i)) + \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{l} log(1 - D(o^{ij}, q^i))$
11             $\theta_d \leftarrow \theta_d + \eta_d \nabla \widetilde{V}(\theta_d)$ where $\eta_d$ is the learning rate for the discriminator
12     /* Training the generator with policy gradient.                                        */
13     sample minibatch of $m$ queries from the real-world database $\{q^1, q^2, \ldots, q^m\}$
14     obtain the generated code from the generator $\{\widetilde{x}^1, \widetilde{x}^2, \ldots, \widetilde{x}^m\}$
15     obtain the reward value of the generated code $D(\widetilde{x}^1, q^1), D(\widetilde{x}^2, q^2), \ldots, D(\widetilde{x}^m, q^m)$
16     update generator parameters $\theta_g$ to maximize the following:
17         $\mathbb{R} = \frac{1}{m}\sum_{i=1}^{m} D(\widetilde{x}^i, q^i) \cdot P_{\theta_g}(\widetilde{x}^i | q^i)$
18         $\theta_g \leftarrow \theta_g + \eta_g \nabla \mathbb{R}(\theta_g)$ where $\eta_g$ is the learning rate for the generator
19 return $G$, $D$
```

oracle code $o$ are fed into the discriminator, and the discriminator is trained to distinguish these two code snippets (i.e., maximizing $D(o, q)$ while minimizing $D(G(q), q)$). Beyond that, we further sample $l$ human-written code snippets which are lexically similar but semantically irrelevant to the given query from the training set $\{ir^1, ir^2, \ldots, ir^l\}$ and ask the discriminator to also distinguish the oracle code from these code snippets (i.e., minimizing $D(ir^i, q)$). By doing so, the capability of the discriminator to distinguish the semantics of different human-written code snippets is expected to be preserved. As a result, the final optimization objective of the discriminator during the adversarial training can be denoted as:

$$\max_{D} \mathbb{E}_{q \sim p_{query}(q)} [logD(o, q) - logD(G(q), q) - \sum_{i=1}^{l} logD(ir^i, q)] \tag{4}$$

## 4.3 The Whole Process

The complete training process of CoCoGAN is illustrated in Algorithm 2. The whole process works as follows: in each training iteration, we first sample $m$ queries from $p_{query}(q)$ and obtain their corresponding code snippets written by developers (i.e., the oracle code). Then, for each obtained oracle code snippet, we also sample $l$ confusing exemplars from the code corpus which are semantically irrelevant to the intended functionality. Such code snippets are served as the replayed samples, whose target is to retain the knowledge of the discriminator on the code search task. The discriminator is trained to maximize the score of the oracle code while simultaneously minimize the scores of the generated code from the generator and the replayed confusing exemplars (cf. lines 10 - 11). After training the discriminator, we use the policy gradient approach to train the generator. Specifically, after sampling $m$ queries and obtaining the generated code snippets from the generator, we obtain the reward value of each generated code snippet by querying the discriminator (cf. line 15). After that, the reward values are used to update the generator's parameters (cf. lines 17 - 18).

During such a process, there are mainly two hyperparameters that need to be determined, i.e., the number of steps of optimizing the discriminator ($k$) and the size of replayed samples for each query ($l$). To determine the value of $k$, we follow the setting of the vanilla GAN [Goodfellow et al. 2014] and set it to 1, which is the most efficient option. This setting has also been shown to help the GAN framework learn stably according to the previous study [Yu et al. 2017]. To determine the value of $l$, replaying all the code samples used to train the code searcher would be impractical due to its inefficiency. Therefore, we adopt the approach from the previous work [Gao et al. 2023] and set $l$ to 1% of the total training data, which provides a reasonable trade-off between effectiveness and efficiency.

## 5 EVALUATION

### 5.1 Research Questions

We have conducted several experiments to evaluate our approach CoCoGAN. Specifically, we seek to answer the following research questions:

**RQ1. Effectiveness of Code Generation.** To what extent can the adversarial training process of CoCoGAN help improve the code generation capability of the code generator? To answer this question, we compare the performances of the state-of-the-art code generators before and after the adversarial training process.

**RQ2. Effectiveness of Code Search.** To what extent can the adversarial training process of CoCoGAN help improve the code search capability of the code searcher? To answer this question, we compare the performances of the state-of-the-art code searchers before and after the adversarial training process.

**RQ3. Efficiency.** What is the time cost of the adversarial training for CoCoGAN? To answer this question, we evaluate the time consumption of CoCoGAN.

**RQ4. Sensitivity Analysis.** How does the lifelong learning strategy affect the overall performances of the code searchers? To answer this question, we exclude the lifelong learning strategy during the adversarial training process and evaluate the performances obtained by code searchers under such a setting.

### 5.2 Experiment Settings

*5.2.1 Evaluation datasets.* In this study, we use the widely-known CodeSearchNet (CSN) benchmark [Husain et al. 2019] for assessing the effectiveness of CoCoGAN. This dataset is mined from popular GitHub projects (in terms of the number of stars and forks). It contains the pairwise data <code, query> (i.e., the code functions and their corresponding comments) and upon released, it has been widely used in code generation and code search studies [Feng et al. 2020; Zeng et al. 2022; Zhao et al. 2022] by treating the comments as the developers' intents and the associated code as the oracle results. The rationale is that the comment usually summarises the main functionality of the code, making the code-comment pair close to actual use scenarios. Existing studies have shown that common queries from developers are similar to the comments (i.e., either being identical to the comment or by slightly prepending the comment with "how to"/"how do I") [Gu et al. 2018; Lv et al. 2015]. Therefore, in our study, we follow such a common experiment setting [Ling et al. 2020; Shuai et al. 2020; Wan et al. 2019; Wang et al. 2021; Xu et al. 2021]. In our study, to increase the generalizability of our results, we use two mostly-used datasets (i.e., the CSN-Python and CSN-Java). These two datasets have already been officially split into the training/validation/test sets and the detailed statistics of these two datasets are listed in Table 1. For evaluation purposes, when presented with a query, the code generator is tasked with directly generating code, while the code searcher is tasked with retrieving the corresponding oracle code from all the code snippets in the test set.

**Table 1.** The statistics of our evaluation datasets.

| Dataset | Training | Validation | Test |
|---|---|---|---|
| **CSN-Python** | 412,178 | 23,107 | 22,176 |
| **CSN-Java** | 394,471 | 15,328 | 26,909 |

*5.2.2 Code generators and code searchers.* We use the following two state-of-the-art code generators in CoCoGAN:

- **CodeT5** [Wang et al. 2021]: CodeT5 is a pre-trained model with the encoder-decoder architecture. One of its pre-training tasks is bimodal dual generation between natural language and programming language, which makes it capable of performing the generation tasks well. According to recent replication studies [Niu et al. 2023; Zeng et al. 2022], it achieves the optimal performances among existing pre-trained code models on generation tasks such as code generation.
- **NatGen** [Chakraborty et al. 2022]: Built on top of CodeT5, NatGen further uses an additional pre-training task in which the model learns to rewrite a piece of unnatural code into the form originally written by developers. It is expected that such a pre-training process enforces the model to capture the semantics of the input code and generate an output that closely resembles human-written code.

We use the following two state-of-the-art code searchers in CoCoGAN:

- **CodeBERT** [Feng et al. 2020]: CodeBERT is a pre-trained model that only has an encoder architecture. The pre-training tasks include Masked Language Modeling (where the model learns to recover randomly-masked tokens in the input sequence) and Replaced Token Detection (where the model learns to detect whether the tokens are replaced), both of which help the model build the semantics connection between natural language and programming language. As a result, the learned model is capable of performing the understanding tasks well such as code search.
- **GraphCodeBERT** [Guo et al. 2021]: GraphCodeBERT has identical model architecture to CodeBERT, i.e., a transformer-based encoder. Beyond the pre-training tasks of CodeBERT, it designs two customized tasks to incorporate code structure information: Edge Prediction (where the model learns to predict whether two variables contain the data flow relation) and Node Alignment (where the model learns to predict variable alignment across source code and data flow). The learned model can thus better understand code semantics by leveraging the structure information. Recent studies show that CodeBERT and GraphCodeBERT achieve the state-of-the-art performances on the code search task [Niu et al. 2023; Zeng et al. 2022].

It should be noted that such a study subject selection leads to totally eight detailed combinations in our study: 2 programming languages (Python & Java) $\times$ 2 code generators (CodeT5 & NatGen) $\times$ 2 code searchers (CodeBERT & GraphCodeBERT). We perform experiments under all the eight situations to better assess the generalizability of our CoCoGAN.

*5.2.3 Metrics.* To evaluate the effectiveness of code generation, we calculate the similarity between the generated code and the oracle code. Specifically, we use the following two metrics for calculating the similarity:

- **CodeBLEU** [Ren et al. 2020]: CodeBLEU is a widely-used metric for assessing the generated code [Chakraborty et al. 2022; Lu et al. 2021; Wang et al. 2021]. Its main idea is that unlike the pure texts, code snippets have certain structural and semantic information beyond the lexical information. Therefore, it compares the generated code with the oracle code from multiple perspectives including the token match (**TM** calculated by the standard BLEU value), the syntax match (**SM** calculated by the AST structure similarities), and the dataflow match (**DM** calculated by the data flow similarities). The final result is calculated as a combination of these similarities and thus provides a holistic view for the quality of the generated code. Readers can refer to the original study for more details about this metric [Ren et al. 2020].

- **CrystalBLEU** [Eghbali and Pradel 2022]: CrystalBLEU, a variant of BLEU [Papineni et al. 2002], is another recently-proposed metric for assessing the quality of generated code. The behind intuition is that even unrelated code snippets can share many common n-grams because of the convention of programming languages (such as `()` and `);`), which makes the conventional BLEU hard to distinguish semantically similar code. To address this challenge, CrystalBLEU first collects the information of commonly shared n-grams of a specific programming language from a large corpus (i.e., the training set of this language), and then reduces the noise caused by such n-grams during calculation. To apply this metric, we remove the top-100 most occurring n-grams as suggested by the original study [Eghbali and Pradel 2022].

Our evaluation of code search focuses on the ranking of the oracle code among all candidate code snippets in the test set for a specific query. We use the following metrics that are widely-used in this research domain [Sun et al. 2022; Wan et al. 2019]:

- **Mean Reciprocal Rank (MRR)** [Lv et al. 2015]: MRR is the average of the reciprocal ranks for the results of a set of queries $Q$. MRR provides a holistic perspective for evaluating the overall effectiveness and it is calculated as: $MRR = \frac{1}{|Q|}\sum_{q \in Q}\frac{1}{Rank_q}$, where $|Q|$ is the size of the query set, and $Rank_q$ is the rank of the code snippet which corresponds to the query $q$. Generally, the higher the MRR value is, the better the code search effectiveness is.
- **SuccessRate@k** [Gu et al. 2018]: SuccessRate@k measures the percentage of queries for which the corresponding code snippet could exist in the top k ranked results. It is an important metric because it can measure how many returned results could be manually checked by the developers before finding the relevant one. This metric is calculated as: $SuccessRate@k = \frac{1}{|Q|}\sum_{q \in Q}\delta(Rank_q \leq k)$, where $|Q|$ and $Rank_q$ denote the same meanings as above, $\delta(\cdot)$ is a function which returns 1 if the input is true and returns 0 otherwise. Generally, the higher the SuccessRate value is, the better the code search effectiveness is. The values of k are set to 1, 5, and 10, respectively, following a number of previous studies [Gu et al. 2018; Sun et al. 2022; Wan et al. 2019].

*5.2.4 Experiment details.* For the four code generators and code searchers in our evaluation, we reuse the source code as well as the values of hyper-parameters released by the original studies [Chakraborty et al. 2022; Feng et al. 2020; Guo et al. 2021; Wang et al. 2021] to perform our experiments. We first fine-tune the four pre-trained models on the CSN dataset for code generation and code search tasks, respectively, and then use the fine-tuned models to further perform the adversarial training process. This is because the previous study [Yu et al. 2017] finds that a well-trained discriminator is informative to help adjust the generator effectively and vice versa. After the adversarial training, we compare the effectiveness of the code generator/searcher on the test sets to that of the fine-tuned code generator/searcher (which can represent the state-of-the-art performances), to see to what extent the adversarial training helps boost the effectiveness of code generation/search.

The fine-tuning process works as follows. For code generators, the inputs and outputs of the model are the queries and corresponding code snippets, respectively. Specifically, at each time step $t$, the *teacher forcing* strategy is used and the model is trained to generate the oracle code token given the previous $t-1$ tokens in the oracle code. For code searchers, the query and code are concatenated as the input to the model, and the representation of a special token $[CLS]$ (the beginning token of the input) is used to measure the semantic relevance between the code and query. The training objective is to maximize the relevance score between a query and its associated code while minimize the relevance score between a query and its non-associated code.

**Table 2.** The performances of different code generators on the two evaluation datasets (in %).

| Code Generator | CSN-Python | | | | | CSN-Java | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | TM | SM | DM | CodeBLEU | CrystalBLEU | TM | SM | DM | CodeBLEU | CrystalBLEU |
| CodeT5 | 18.5 | 16.1 | 39.2 | 21.2 | 8.6 | 12.7 | 23.2 | 37.3 | 20.5 | 7.1 |
| CoCoGAN$_{CodeT5+CodeBERT}$ | 18.3 | 35.9 | 45.2 | 28.6(↑35%) | 14.4(↑67%) | 15.0 | 30.6 | 46.7 | 25.8(↑26%) | 8.5(↑20%) |
| CoCoGAN$_{CodeT5+GraphCodeBERT}$ | 18.7 | 36.4 | 47.0 | 29.4(↑39%) | 14.9(↑73%) | 15.6 | 32.2 | 45.9 | 26.2(↑28%) | 8.7(↑23%) |
| NatGen | 15.4 | 22.7 | 33.8 | 21.6 | 7.8 | 13.2 | 23.3 | 38.0 | 20.8 | 7.9 |
| CoCoGAN$_{NatGen+CodeBERT}$ | 17.0 | 34.6 | 45.9 | 27.9(↑29%) | 13.4(↑72%) | 16.1 | 32.2 | 46.6 | 26.6(↑28%) | 9.2(↑16%) |
| CoCoGAN$_{NatGen+GraphCodeBERT}$ | 18.2 | 35.7 | 44.9 | 28.4(↑32%) | 14.2(↑82%) | 16.7 | 33.4 | 46.6 | 27.0(↑30%) | 9.4(↑19%) |

As for the adversarial training, for each batch, we successively update the parameters of the discriminator and generator for one time, as introduced in Section 4. The hyper-parameters of the adversarial training are in consistent with those of the fine-tuning process. For instance, the batch size is set to 32 and the learning rate is set to $2e^{-5}$ for both the generator and discriminator. In the adversarial training, we adopt an early stopping strategy: we set the epochs to 30 but the training process will be stopped if the performance of the code generator on the validation set does not increase for more than three consecutive epochs, following the previous study [Chakraborty et al. 2022]. We conjecture that if the generator cannot be further strengthened, it means the reward from the discriminator cannot further guide the training, which may indicate that the discriminator itself cannot better distinguish the generated and oracle code.

All the experiments were performed on a server with 4 Nvidia GeForce RTX 4090 GPUs with 32GB memory.

## 5.3 Experiment Results

*5.3.1 Effectiveness of code generation.* Table 2 lists the results of different code generators. We observe that the adversarial training process can enhance the effectiveness of different code generators to a large extent. For instance, on the Python dataset, the adversarial training leads to an increase of up to around 40% in terms of CodeBLEU values and around 70% in terms of CrystalBLEU values. Specifically, when CodeT5 is adversarially trained with GraphCodeBERT, its CodeBLEU value is climbed from 21.2% to 29.4%, an increase of 39%; when NatGen is adversarially trained with GraphCodeBERT, its CrystalBLEU value is boosted from 7.8% to 14.2%, an increase of 82%. Similarly, we observe an increase of approximately 30% in terms of CodeBLEU values and 20% in terms of CrystalBLEU values on the Java dataset. We also conduct the Wilcoxon Signed-Rank Tests [Wilcoxon 1945] to analyze the statistical significance of the differences and the results show that the improvements are all statistically significant with all the p-values less than 0.001. We note that generally the CrystalBLEU values achieved on the Java dataset are lower than those achieved on the Python dataset. A potential reason is that the Java code contains more tokens: on average, a Java code snippet contains 113 tokens while the number for a Python code snippet is only 48. Since CrystalBLEU is a metric only focusing on token overlaps, it could be comparatively trivial to achieve high scores on the Python code.

We also list the values of TM, SM, and DM in Table 2 to investigate the improvements from different perspectives. We find that the biggest improvements for code generators are from the same aspect on the two datasets, i.e., to generate correct code syntax. For instance, on the Python dataset, when CodeT5 is adversarially trained with GraphCodeBERT, its SM value is increased from 16.1% to 36.4%, an improvement of around 126%. Similarly, on the Java dataset, when NatGen is adversarially trained with GraphCodeBERT, its SM value is increased from 23.3% to 33.4%, an improvement of around 43%. Besides SM, we note that the DM values of the code generators also increase significantly, which indicates that the code generators become more competent at generating correct data flow relations after the adversarial training. In contrast, the TM values of the code generators do not increase to a large extent. This phenomenon reflects the difference between

**Table 3.** The performances of different code searchers on the two evaluation datasets.

| **Code Searcher** | **CSN-Python** | | | | **CSN-Java** | | | |
|---|---|---|---|---|---|---|---|---|
| | MRR | SR@1 | SR@5 | SR@10 | MRR | SR@1 | SR@5 | SR@10 |
| CodeBERT | 0.669 | 0.563 | 0.802 | 0.857 | 0.522 | 0.420 | 0.646 | 0.707 |
| CoCoGAN$_{CodeT5+CodeBERT}$ | 0.714(↑7%) | 0.622 | 0.829 | 0.883 | 0.557(↑7%) | 0.470 | 0.653 | 0.710 |
| CoCoGAN$_{NatGen+CodeBERT}$ | 0.721(↑8%) | 0.626 | 0.837 | 0.889 | 0.559(↑7%) | 0.474 | 0.662 | 0.713 |
| GraphCodeBERT | 0.682 | 0.577 | 0.812 | 0.865 | 0.534 | 0.432 | 0.659 | 0.721 |
| CoCoGAN$_{CodeT5+GraphCodeBERT}$ | 0.751(↑10%) | 0.665 | 0.860 | 0.905 | 0.580(↑9%) | 0.496 | 0.673 | 0.725 |
| CoCoGAN$_{NatGen+GraphCodeBERT}$ | 0.763(↑12%) | 0.678 | 0.863 | 0.908 | 0.589(↑10%) | 0.502 | 0.681 | 0.737 |

**Table 4.** The time consumption of CoCoGAN under different experiment settings (in hours).

| Dataset | Models | Generator Fine-tuning | Discriminator Fine-tuning | Adversarial Training |
|---|---|---|---|---|
| CSN-Python | CoCoGAN$_{CodeT5+CodeBERT}$ | 18 | 6 | 26 |
| | CoCoGAN$_{CodeT5+GraphCodeBERT}$ | 18 | 7 | 30 |
| | CoCoGAN$_{NatGen+CodeBERT}$ | 18 | 6 | 26 |
| | CoCoGAN$_{NatGen+GraphCodeBERT}$ | 18 | 7 | 30 |
| CSN-Java | CoCoGAN$_{CodeT5+CodeBERT}$ | 37 | 9 | 40 |
| | CoCoGAN$_{CodeT5+GraphCodeBERT}$ | 37 | 12 | 46 |
| | CoCoGAN$_{NatGen+CodeBERT}$ | 37 | 9 | 49 |
| | CoCoGAN$_{NatGen+GraphCodeBERT}$ | 37 | 12 | 47 |

this metric and the CrystalBLEU metric: the former focuses on all the generated tokens while the latter removes commonly-observed tokens before the calculation. Therefore, an improvement in terms of the CrystalBLEU metric does not necessarily result in an improvement in terms of the TM, and vice versa.

*5.3.2 Effectiveness of code search.* Table 3 lists the results of different code searchers. We note that generally, the performances of code searcher can be improved by around 10% through the adversarial training in terms of the MRR values. For instance, when adversarially trained with NatGen, the MRRs of GraphCodeBERT on the Python and Java datasets are increased from 0.682 to 0.763 and 0.534 to 0.589, respectively, resulting in improvement ratios of 12% and 10%. Similarly, the *SuccessRate@k* values are also consistently increased. Specifically, after adversarially trained with NatGen, GraphCodeBERT can successfully rank the oracle code at top-1 position for around 70% and 50% cases on the Python and Java datasets, respectively, which substantially outperforms the vanilla GraphCodeBERT model (i.e., ranking the oracle at top-1 for around 57% and 43% cases).

Based on the results presented in Tables 2 and 3, it can be observed that code generation and code search techniques can achieve better effectiveness when they are trained with high-performing techniques of the other type, respectively. For instance, on the Python dataset, NatGen achieves higher performance when trained with GraphCodeBERT than when trained with CodeBERT, resulting in CodeBLEU values of 28.4% and 27.9%, respectively. Therefore, it is advisable to involve advanced techniques of both types when using CoCoGAN to achieve optimal results.

*5.3.3 Efficiency.* Table 4 demonstrates the time costs of different CoCoGAN variants. In addition to the cost of the adversarial training process, we also list those of the fine-tuning processes for the code generators and code searchers for comparison. Note that the two code generators have an identical fine-tuning time cost since they have an identical model architecture, which results in the same amount of parameters, and the fine-tuning process is performed for a fixed number of epochs without the early stopping mechanism [Wang et al. 2021].

We observe that the time cost of the adversarial training approximately equals to the total cost of fine-tuning the generator and discriminator. For instance, on the Python dataset, the time costs of fine-tuning NatGen and CodeBERT are 18 and 6 hours, respectively, while the adversarial training takes around 26 hours. On the Java dataset, fine-tuning the generator and discriminator even takes a longer time than the adversarial training, which indicates that the time consumption of CoCoGAN is controlled in a reasonable scale. Given that the adversarial training process is a one-time task:

**Table 5.** The MRR values achieved by different code searchers w/ and w/o lifelong learning (LLL).

| Dataset | Models | w/ LLL | w/o LLL |
|---|---|---|---|
| CSN-Python | CoCoGAN$_{CodeT5+CodeBERT}$ | 0.714 | 0.678(↓5%) |
| | CoCoGAN$_{NatGen+CodeBERT}$ | 0.721 | 0.679(↓6%) |
| | CoCoGAN$_{CodeT5+GraphCodeBERT}$ | 0.751 | 0.691(↓8%) |
| | CoCoGAN$_{NatGen+GraphCodeBERT}$ | 0.763 | 0.705(↓8%) |
| CSN-Java | CoCoGAN$_{CodeT5+CodeBERT}$ | 0.557 | 0.528(↓5%) |
| | CoCoGAN$_{NatGen+CodeBERT}$ | 0.559 | 0.531(↓5%) |
| | CoCoGAN$_{CodeT5+GraphCodeBERT}$ | 0.580 | 0.551(↓5%) |
| | CoCoGAN$_{NatGen+GraphCodeBERT}$ | 0.589 | 0.567(↓4%) |

once trained, the code generator and code searcher can be applied in practice for any query, we thus consider that our CoCoGAN is user-friendly in terms of the efficiency.

We also observe that both the fine-tuning and adversarial training processes generally take longer on the Java dataset compared to the Python dataset, which may suggest that models converge more slowly on Java code. It is also important to note that there is a trade-off between the effectiveness and efficiency. Future studies could explore the possibility of increasing the values of the hyper-parameters $k$ and $l$ in Algorithm 2 to achieve better effectiveness in code generation and search, albeit at the cost of efficiency.

*5.3.4 Sensitivity Analysis.* The MRR values of the code searchers when the lifelong learning strategy is excluded during the adversarial training are shown in Table 5. We observe that excluding the lifelong learning strategy results in a performance degradation of over 5% for the code searchers across various settings. However, it is worth noting that the degraded performance is still better than that of the vanilla code searcher. For instance, on the Python dataset, when GraphCodeBERT is trained with CodeT5, its degraded MRR value is 0.691, while its original MRR value is 0.682 (cf. Table 3). These findings suggest that (1) the adversarial training process indeed helps the code searcher better understand code semantics, and (2) the lifelong learning strategy further prevents the model from forgetting some previously-learned knowledge and helps it achieve the optimal performance.

## 6 DISCUSSION

### 6.1 Human Evaluation

The ultimate goal of code generators is to generate the source code for developers. Although we have demonstrated that code generators trained through our CoCoGAN framework are good at generating code similar to the oracle, in this section, we further conduct a human evaluation to assess the effectiveness of CoCoGAN.

**Setup.** As a demonstration, we assess the code generation effectiveness of CodeT5 on the Python language before and after the adversarial training process, when CodeBERT is used as the discriminator. Specifically, we randomly select 100 queries from the CSN-Python dataset and collect the generated code from CodeT5 and CoCoGAN$_{CodeT5+CodeBERT}$. Therefore, we obtain 200 programs for evaluation. We recruit five computer science Ph.D students who are not co-authors of this paper and each of them has more than five years programming experience with Python. For the identical query, we randomly list the two examples of generated code to the participants. Following the existing study [Li et al. 2023], each participant is asked to rate each generated code snippet from the three aspects: (1) **correctness** which reflects whether the code satisfies the given query, (2) **quality** which reflects whether the code does not contain bad code smell, and (3) **maintainability** which reflects whether the code has good readability, on a 5-point Likert scale (1 for poor, 2 for

**Table 6.** The results of human evaluation.

| | Approach | Average | Std. |
|---|---|---|---|
| Correctness | CodeT5 | 3.5 | 1.3 |
| | CoCoGAN$_{CodeT5+CodeBERT}$ | 3.7 | 1.1 |
| Quality | CodeT5 | 3.7 | 1.2 |
| | CoCoGAN$_{CodeT5+CodeBERT}$ | 4.2 | 0.9 |
| Maintainability | CodeT5 | 4.1 | 0.9 |
| | CoCoGAN$_{CodeT5+CodeBERT}$ | 4.3 | 0.7 |

```
# Code generated by CodeT5
def read(self, n):
    result = self.data[self.pos:self.pos + n]
    self.pos += n
    return result

# Code generated by adversarially trained CodeT5
def read(self, n=None):
    if n is None:
        return None
    if isinstance(n, int):
        result = self.data[self.pos:self.pos + n]
        self.pos += n
        return result
    else:
        return None
```

**Fig. 6.** The code generated by Vanilla CodeT5 and the code generated by CodeT5 after adversarial training, for the query "Read n bits from the stream".

marginal, 3 for acceptable, 4 for good, and 5 for excellent). The participants are also allowed to search the Internet for encountered unfamiliar concepts.

**Results.** The results of the human evaluation are shown in Table 6. Our CoCoGAN is better than the vanilla CodeT5 in all the three aspects. Specifically, the CodeT5 model after the adversarial training increases the correctness from 3.5 to 3.7, the code quality from 3.7 to 4.2, and the maintainability from 4.1 to 4.3, respectively. Also, our statistical test results show that the p-values comparing the scores obtained by CoCoGAN and CodeT5 are all less than 0.005, which shows that the improvements are statistically significant.

Our human evaluation indicates that one significant advantage of our adversarial training perceived by the programmers is its capability to generate code that has less code smells (i.e., increasing the score from 3.7 to 4.2). Figure 6 gives a detailed example where the two code generators are asked to generate code for the query "Read n bits from the stream". Generally, the two code snippets adopts the similar way to fulfill the main functionality, which is to retrieve the data from a list and return the result. However, the code from the vanilla CodeT5 model assumes that the input argument `n` is already an integer, while the code produced by CoCoGAN performs some sanity checks about the data type of `n` and returns `None` if `n` is not an integer. This case shows that CoCoGAN possesses the capability to perform data validation, a critical ability in fault tolerance [Hayes and Offutt 2006]. Indeed, in our human evaluation, the five participants rate a score of 4.6 towards the quality of the generated code on average, which is a comparatively high value.

## 6.2 Case Analysis

In this section, we provide detailed examples to show how our CoCoGAN helps improve the effectiveness of both the code generator and code searcher. As a showcase, the examples are selected from the experiment setting CoCoGAN$_{CodeT5+CodeBERT}$ on the Python language.

```
1  # Oracle code
2  def _get_num_cpus():
3      try:
4          return os.sysconf("SC_NPROCESSORS_ONLN")
5      except ValueError:
6          num = 0
7          f = open('/proc/cpuinfo', 'r')
8          try:
9              lines = f.readlines()
10         finally:
11             f.close()
12         for line in lines:
13             if line.startswith('processor'):
14                 num += 1
15     if num == 0:
16         raise RuntimeError("can't determine")
17     return num
18
19
```

**(a)** The oracle code.

```
1  # Code generated by vanilla CodeT5
2  def cpu_count():
3      try:
4          with open('/proc/cpuinfo') as f:
5              return int(f.read())
6      except IOError:
7          return 0
8
9  # Code generated by adversarially trained CodeT5
10 def cpu_count():
11     num = 0
12     try:
13         with open('/proc/cpuinfo') as f:
14             for line in f:
15                 if line.startswith('processor'):
16                     num += 1
17     except IOError:
18         return 0
19     return num
```

**(b)** The code generated by CodeT5.

**Fig. 7.** The developer-written code, the code generated by Vanilla CodeT5, and the code generated by CodeT5 after adversarial training, for the query "Return the number of CPUs on the system".

**Case 1.** In the first case, we focus on demonstrating how the code generators become more competent at generating code snippets that resemble human-written ones. Figure 7 shows such an example. The intention here is to obtain the number of CPUs on the working system. To achieve it, developers use different approaches: they first refer to `SC_NPROCESSORS_ONLN`, the macro in the "unistd.h" header file in Linux system, to get the number of CPUs that are currently available by the system; if this fails, they further refer to the "/proc/cpuinfo" file which contains information about the CPUs installed on the system. We observe that the vanilla CodeT5 has some common sense knowledge, since it realizes that the task could be finished by referring to the "/proc/cpuinfo" file. Nonetheless, it lacks more detailed knowledge about this file and thus utilizes it in an incorrect way. The fact is that the "/proc/cpuinfo" file contains one or more paragraphs, each describing a CPU in terms of the CPU model, speed, cache size, etc. Therefore, reading the whole file as an integer (like the code in line 5 in Figure 7b) will lead to a `ValueError`. After the adversarial training, CodeT5 realizes that the target file should be utilized in another way and generates a code snippet that resembles the developer-written one (i.e., by counting the number of lines starting with `processor`).

**Case 2.** In the second case, we focus on demonstrating how the code searchers become more competent at capturing code semantics after the adversarial training. Figure 8 shows an example where the vanilla CodeBERT fails to accurately retrieve the correct code for a given query but our CoCoGAN succeeds. The intended functionality is "plot two digits frequency counts using matplotlib". Given an input `f2`, a two-dimensional matrix that records the frequency data, the oracle code draws the image by calling the `plt.matshow` function, which is widely-used to plot a matrix as a color-coded image. The vanilla CodeBERT inaccurately ranks another code snippet at the top position as shown in the figure. The retrieved code calls the `plt.plot` method to create a line plot so that its input parameter is with only one dimension. Actually, its primary purpose can be described as "plot one digit frequency counts using matplotlib". While the two intentions are similar in terms of their lexical meaning (i.e., they are only different with respect to the number of digits), they require different implementations. This is because, in addition to the differences in the API calls as introduced, the number of digits also affects the labels of the resulting image. The original code sets both the x and y labels as two digits, while the retrieved code only sets the x label as a digit. Because there are many overlapped tokens between the two code snippets, such as

```python
# Oracle code
def plot_two_digit_freqs(f2):
    ax = plt.matshow(f2)
    plt.colorbar()
    for i in range(10):
        for j in range(10):
            plt.text(i-0.2, j+0.2, str(j)+str(i))
    plt.ylabel('First digit')
    plt.xlabel('Second digit')
    return ax

# Code retrieved by CodeBERT
def plot_one_digit_freqs(f1):
    ax = plt.plot(f1,'bo-')
    plt.xlabel('Digit')
    plt.ylabel('Count')
    return ax
```

**Fig. 8.** The developer-written code and the code retrieved by CodeBERT, for the query "Plot two digits frequency counts using matplotlib".

xlabel, ylabel, and ax, the vanilla CodeBERT is unable to distinguish between them and can be easily confused. Fortunately, after undergoing the adversarial training, CodeBERT is able to better differentiate between the two code snippets and correctly ranks the oracle code at the top position.

## 6.3 The Contribution of Fine-Tuning

We aim to explore to what extent the fine-tuning process contributes to the final performance of CoCoGAN, i.e., how does the effectiveness of CoCoGAN depend on the accuracy of the initially given code generation model? To this end, we run the evaluation with different initial models, which are trained for a different amount of time. Originally, the CodeT5 and NatGen models were fine-tuned for 20 epochs. To address the raised question, we retrain two distinct models that have undergone varying degrees of fine-tuning: one model is fine-tuned for 10 epochs, while the other is not fine-tuned at all on the code generation task. After that, we have totally three initial models on hand (the one with 20 epochs of fine-tuning, the one with 10 epochs of fine-tuning, and the one without fine-tuning). We are thus able to answer the raised question by comparing their effectiveness after the adversarial training process.

We use CoCoGAN$_{CodeT5+CodeBERT}$ on the Python dataset as a showcase. Results show that (1) if the code generator is not fine-tuned at all, its CodeBLEU value after the adversarial training is only 15.0%, and (2) if the code generator is not fine-tuned adequately (i.e., 10-epoch fine-tuning), its CodeBLEU value after the adversarial training is 23.6%, only slightly higher than that of the adequately fine-tuned code generator (i.e., 21.2%). Such results indicate that the fine-tuning process, which provides a reasonable initialization of the model, contributes to the final performance of CoCoGAN to a large extent.

## 6.4 The Quality of the Code Generated by the Initial Code Generator

Typically, if the GAN framework is perfectly trained, then samples from the generator will look the same as the real data. This means that the discriminator cannot be further improved because there will be no supervision for it on distinguishing code semantics. Therefore, a raised question is how often is the code generated by the initially given code generator so different from the oracle code that they could be distinguished by the discriminator?

We recall that in the CodeBLEU paper [Ren et al. 2020], the authors performed a human evaluation where 10 participants rated the generated code from CodeGPT, a state-of-the-art code generator, on a 5-point Likert scale. Results show that the average score obtained by CodeGPT is 3.125, which is near the neutral degree but far away from satisfactory (the score of 4 or 5). This suggests that the current state-of-the-art code generators are generally unable to produce code that is highly semantically similar to the oracle, although there may be some cases where they can generate the qualified code. As a result, the code generated by the code generator can be used to strengthen the key ability of the code searcher. That could be the reason for the improvement of the code searcher during the adversarial training process.

## 6.5 Large Language Models

Recently, various large language models (LLMs) have been proposed to facilitate developers' development activities through code generation, such as ChatGPT[3] and GPT-4[4]. To quantitatively understand the code generation effectiveness of such models, we randomly select 2K queries from our evaluation dataset (1K from the CSN-Python dataset and 1K from the CSN-Java dataset) and task ChatGPT with generating corresponding method code. To perform this experiment, we use the ChatGPT API (accessed on March 27, 2023) with the prompts to the model in the form of "Assume that you are a Python/Java programmer. Please write a Python/Java function that...", followed by the query contents. The first sentence is to prepare ChatGPT for the code generation task while the second one is the detailed requirement. Results show that on average, ChatGPT achieves the CodeBLEU values of 22.1% and 26.5% on the Python and Java laguages, respectively. Such performances are higher than those of the vanilla NatGen but lower than those of the adversarially-trained NatGen. One possible explanation for NatGen's superior performance could be its fine-tuning and adversarial training specifically designed for the code generation task, whereas ChatGPT is optimized for artificial general intelligence (AGI) and not specifically for code generation. However, the target of our study is not to show that our approach outperforms ChatGPT in code generation. Instead, we aim to propose an adversarial training framework that can boost the performance of code generators. We have demonstrated the effectiveness of our framework on two state-of-the-art code generators and, theoretically, our approach can be applied to any code generator that faces the *exposure bias* problem, including ChatGPT. Hopefully, in the future, the performance of LLMs on code generation could also be improved in such a way.

## 6.6 Threats to Validity

We have identified the following threats to validity:

**Selection of study subjects.** Beyond the code generators and code searchers used in this study, there are still a number of alternatives in the literature [Gu et al. 2018; Lu et al. 2021; Wan et al. 2019; Yin and Neubig 2018]. It is unclear to what extent our CoCoGAN can help boost their performances. However, this threat is mitigated considering that our study subjects are representative works in these research domains, which have been demonstrated to achieve state-of-the-art performances on code generation and code search tasks [Niu et al. 2023; Zeng et al. 2022].

**Applying study subjects on our dataset.** Our study reproduces four state-of-the-art pre-trained models for code. To mitigate the reproduction bias, we have carefully fine-tuned the pre-trained models on our evaluation datasets by reusing the code and hyper-parameters released by the original studies, and the models from the best-performing checkpoints are used for evaluation.

[3]https://chat.openai.com/
[4]https://openai.com/product/gpt-4

**Selection of dataset.** Our study requires a large-scale dataset to ensure the adversarial training is adequately performed. To this end, we choose to use the CodeSearchNet dataset which contains hundreds of thousands of data samples and has been widely-used in the research domains of both code generation and code search [Feng et al. 2020; Zeng et al. 2022; Zhao et al. 2022]. We apply our approach on Java and Python code because these two programming languages are widely used nowadays (e.g., Python is widely used for data science and machine learning, and Java is widely used for mobile applications and enterprise applications). [5] We thus cannot claim that our approach is generic for all programming languages. However, the key idea of CoCoGAN is general and can be applied on arbitrary programming language. We leave evaluating CoCoGAN on more datasets and other programming languages as our future work.

**Assumption.** One assumption of CoCoGAN is that all code generated by code generators during the adversarial training is non-oracle (i.e., we deem that they are not semantically identical to the oracle code). This decision is made based on our observation that, even after adversarial training, code generators are unable to produce code that is lexically identical to the oracle code on the training sets for both Python and Java datasets. However, it is possible that the generated code is semantically identical to the oracle even if it is not lexically identical, and dynamically adjusting the labels of the generated code during the adversarial training process is a potential direction for future work.

## 7 RELATED WORK

### 7.1 Combination of Code Generation and Code Search

The code generation and code search are two effective ways to transform natural language descriptions into source code. The previous work [Xu et al. 2022] conducted a user study where participants were asked to implement specific functionalities with the help of the code generation or code search techniques, and demonstrated the complementarity between these two types of techniques: developers prefer code generation over code search on some tasks and vice versa on other tasks. A recent empirical study [Wang et al. 2023a] compared ten well-known code generation and code search techniques on a large-scale query dataset and also observed the complementarity between code generation and code search techniques. These findings motivate us to explore a way for combining such two types of techniques. However, previous researches within these two domains are mainly separately studied and the only existing way to combine these two types of techniques is to first retrieve a code snippet given a natural language query, and then generate the final code based on the retrieval results [Hashimoto et al. 2018; Hayati et al. 2018; Li et al. 2023]. Such approaches mainly have two limitations that restrict their application scenarios. First, they usually assume that the input and output of the method is already known and the method is partially written [Hashimoto et al. 2018], while general code generation techniques aim at generating code from scratch. Second, their effectiveness heavily relies on the existence of code similar to the oracle in the retrieval corpus, so that they are usually evaluated on simple code snippets such as the Hearthstone (HS) dataset [Ling et al. 2016] which is composed of code snippets implementing card descriptions, while general code generation techniques require the ability to generate complex code snippets such as the code in our evaluation dataset CodeSearchNet, which are collected from real-world projects. In this study, we propose to combine code generation and code search to overcome the limitation of each other: the code generator requires feedback on the quality of its generated code, which can be provided by the code searcher, and the code searcher requires more data for training, which can be generated by the code generator. We respectively treat the code generator and code searcher as the generator and discriminator in a GAN framework and

[5] https://www.tiobe.com/tiobe-index/

simultaneously boost their performances through the adversarial training process. The resulted code generator can generate complex method-level code from scratch.

### 7.2 GAN in the Software Engineering Domain

In recent years, the GAN framework has been applied on diverse software engineering tasks due to its superior ability on generating samples that are similar to real-world data. ACTGAN [Bao et al. 2019] automatically generates software configurations for helping system users better configure a large number of parameters. During the adversarial training, the generator is forced to seek important features hidden in good configuration samples. GUIGAN [Zhao et al. 2021] alleviates the burden of designers by automatically generating Graphical User Interface (GUI) designs. It formulates the task as selecting a sequence of GUI components from existing mobile apps to compose new GUI designs and also exploits the SeqGAN framework in their approach for generating sequence data. CGAN4FL [Lei et al. 2023] synthesizes program coverage information of failing tests and integrates the generated failing tests into the original test suite to compensate for the lack of failing tests, which adversely affects the effectiveness of fault localization techniques. The most obvious difference between our study and the aforementioned ones is that they all focus on utilizing the power of the GAN framework to generate samples but the discriminator is only used to strengthen the generator, while we simultaneously boost the performances of both code generation and code search techniques through the GAN framework (i.e., the generator and discriminator are strengthened together).

### 7.3 GAN in Other Research Domains

Upon proposed, the GAN framework has been applied on diverse generation tasks [Xia et al. 2022]. To name a list, BigGAN [Brock et al. 2019] applies orthogonal regularization to the generator to make the training process of GAN stable, and achieves promising results on generating clear images. CycleGAN [Zhu et al. 2017] implements a cycle mechanism which enables the transformation between two domains. It thus can be used to change the style of an image into a specific type. By mapping a sequence of random vectors to a sequence of video frames, MoCoGAN [Tulyakov et al. 2018] can successfully generate videos. Building on these studies, we leverage the powerful generation capabilities of the generator from the GAN framework for code generation.

## 8 CONCLUSION

In this study, we introduce CoCoGAN, an approach that combines code generation and code search techniques to overcome their limitations and enhance their performances. By treating the code generator and code searcher as the generator and discriminator in a GAN framework, we enable the code generator to receive feedback on the quality of its generated code from the code searcher, while also generating additional training data for the code searcher. Our experiments demonstrate that our approach consistently improves the performances of both code generation and code search on eight different settings. We believe our approach holds potential in advancing the state of the art in code generation and code search, and could lead to more efficient and effective software development.

## ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China (No.2022YFA1005101), the National Natural Science Foundation of China No.61932021, No.62002125, and No.62272072, as well as the Young Elite Scientists Sponsorship Program by CAST (Grant No.2021QNRC001).

## REFERENCES

Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *Comput. Surveys* 51, 4 (2018), 81:1–81:37. https://doi.org/10.1145/3212695

Liang Bao, Xin Liu, Fangzheng Wang, and Baoyin Fang. 2019. ACTGAN: automatic configuration tuning for software systems with generative adversarial networks. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 465–476.

Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. 7, OOPSLA1, Article 78 (apr 2023), 27 pages. https://doi.org/10.1145/3586030

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. *Advances in neural information processing systems* 28 (2015).

Andrew Brock, Jeff Donahue, and Karen Simonyan. 2019. Large Scale GAN Training for High Fidelity Natural Image Synthesis. In *International Conference on Learning Representations*.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 511–521.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. 2018. Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. In *International Conference on Learning Representations (ICLR)*.

Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 964–974.

Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar Devanbu, and Baishakhi Ray. 2022. NatGen: Generative pre-training by "Naturalizing" source code. In *Proceedings of the 30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

Matthias De Lange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Aleš Leonardis, Gregory Slabaugh, and Tinne Tuytelaars. 2021. A continual learning survey: Defying forgetting in classification tasks. *IEEE transactions on pattern analysis and machine intelligence* 44, 7 (2021), 3366–3385.

Emily L Denton, Soumith Chintala, Rob Fergus, et al. 2015. Deep generative image models using a laplacian pyramid of adversarial networks. *Advances in neural information processing systems* 28 (2015).

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*. 4171–4186. https://doi.org/10.18653/v1/n19-1423

Luca Di Grazia and Michael Pradel. 2022. Code Search: A Survey of Techniques for Finding Code. *arXiv preprint arXiv:2204.02765* (2022).

Li Dong and Mirella Lapata. 2018. Coarse-to-Fine Decoding for Neural Semantic Parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 731–742.

Aryaz Eghbali and Michael Pradel. 2022. CrystalBLEU: precisely and efficiently measuring the similarity of code. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

Shuzheng Gao, Hongyu Zhang, Cuiyun Gao, and Chaozheng Wang. 2023. Keeping Pace with Ever-Increasing Data: Towards Continual Learning of Code Intelligence Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. *Advances in Neural Information Processing Systems* 27 (2014).

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *ICLR*.

David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. 1990. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on software engineering* 16, 4 (1990), 403–414.

Tatsunori B Hashimoto, Kelvin Guu, Yonatan Oren, and Percy S Liang. 2018. A retrieve-and-edit framework for predicting structured outputs. *Advances in Neural Information Processing Systems* 31 (2018).

Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-Based Neural Code Generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*.

Jane Huffman Hayes and Jeff Offutt. 2006. Input validation analysis and testing. *Empirical Software Engineering* 11 (2006), 493–522.

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

Michael B James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. 2020. Digging for fold: synthesis-aided API discovery for Haskell. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.

Amy J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 199–206.

Yan Lei, Tiantian Wen, Huan Xie, Lingfeng Fu, Chunyan Liu, Lei Xu, and Hongxia Sun. 2023. Mitigating the Effect of Class Imbalance in Fault Localization Using Context-aware Generative Adversarial Network. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*.

Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. SkCoder: A Sketch-based Approach for Automatic Code Generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.

Jiwei Li, Will Monroe, Tianlin Shi, Sébastien Jean, Alan Ritter, and Dan Jurafsky. 2017. Adversarial Learning for Neural Dialogue Generation. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. 2157–2169.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

Zhizhong Li and Derek Hoiem. 2017. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence* 40, 12 (2017), 2935–2947.

Chunyang Ling, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2020. Adaptive deep code search. In *Proceedings of the 28th International Conference on Program Comprehension*. 48–59.

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Fumin Wang, and Andrew Senior. 2016. Latent Predictor Networks for Code Generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 599–609.

Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John Grundy. 2021. Opportunities and challenges in code search tools. *ACM Computing Surveys (CSUR)* 54, 9 (2021), 1–40.

Hui Liu, Mingzhu Shen, Jiaqi Zhu, Nan Niu, Ge Li, and Lu Zhang. 2020. Deep Learning Based Program Generation from Requirements Text: Are We There Yet? *IEEE Transactions on Software Engineering* 01 (2020), 1–1.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code recommendation via structural code search. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28.

Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 260–270.

Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*. 111–120.

Collin McMillan, Negar Hariri, Denys Poshyvanyk, Jane Cleland-Huang, and Bamshad Mobasher. 2012. Recommending source code for use in rapid software prototypes. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 848–858.

Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the fly synthesis of edit suggestions. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

Mehdi Mirza and Simon Osindero. 2014. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784* (2014).

Parastoo Mohagheghi and Reidar Conradi. 2007. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering* 12, 5 (2007), 471–516.

Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An Empirical Comparison of Pre-Trained Models of Source Code. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.

Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. 2013. Using of Jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, Vol. 1. 380–384.

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*. 311–318.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

Daniël AA Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. 2022. Language-parametric static semantic code completion. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–30.

Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*. 419–428.

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

Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 191–201.

Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. 2016. Improved techniques for training gans. *Advances in neural information processing systems* 29 (2016).

Ensheng Shi, Wenchao Gub, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2022. Enhancing Semantic Code Search with Multimodal Contrastive Learning and Soft Data Augmentation. *arXiv preprint arXiv:2204.03293* (2022).

Jiho Shin and Jaechang Nam. 2021. A Survey of Automatic Code Generation from Natural Language. *Journal of Information Processing Systems* 17, 3 (2021), 537–555.

Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Improving code search with co-attentive representation learning. In *Proceedings of the 28th International Conference on Program Comprehension*. 196–207.

Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph Lim. 2018. Neural program synthesis from diverse demonstration videos. In *International Conference on Machine Learning*. PMLR, 4790–4799.

Weisong Sun, Chunrong Fang, Yuchen Chen, Guanhong Tao, Tingxu Han, and Quanjun Zhang. 2022. Code Search based on Context-aware Code Translation. In *Proceedings of the 44th International Conference on Software Engineering*.

Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems* 12 (1999).

Sergey Tulyakov, Ming-Yu Liu, Xiaodong Yang, and Jan Kautz. 2018. Mocogan: Decomposing motion and content for video generation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1526–1535.

Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

Gust Verbruggen, Vu Le, and Sumit Gulwani. 2021. Semantic programming by example with pre-trained models. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–25.

Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 13–25.

Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. 2023b. Practitioners' Expectations on Code Completion. *arXiv preprint arXiv:2301.03846* (2023).

Hong Wang, Wenhan Xiong, Mo Yu, Xiaoxiao Guo, Shiyu Chang, and William Yang Wang. 2019. Sentence embedding alignment for lifelong relation extraction. In *Annual Conference of the North American Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics (ACL).

Shangwen Wang, Mingyang Geng, Bo Lin, Zhensu Sun, Ming Wen, Yepang Liu, Li Li, Tegawendé F. Bissyandé, and Xiaoguang Mao. 2023a. Natural Language to Code: How Far are We?. In *Proceedings of the 31st ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. 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*. 8696–8708.

Fengcai Wen, Emad Aghajani, Csaba Nagy, Michele Lanza, and Gabriele Bavota. 2021. Siri, write the next method. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 138–149.

Michael W Whalen. 2000. High-integrity code generation for state-based formalisms. In *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*. IEEE, 725–727.

F. Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.

Weihao Xia, Yulun Zhang, Yujiu Yang, Jing-Hao Xue, Bolei Zhou, and Ming-Hsuan Yang. 2022. Gan inversion: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022).

Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empirical Software Engineering* 22, 6 (2017), 3149–3185.

Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–47.

Ling Xu, Huanhuan Yang, Chao Liu, Jianhang Shuai, Meng Yan, Yan Lei, and Zhou Xu. 2021. Two-Stage Attention-Based Model for Code Search with Textual and Structural Features. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 342–353.

Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. Coacor: Code annotation for code retrieval with reinforcement learning. In *The world wide web conference*. 2203–2214.

Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 440–450.

Pengcheng Yin and Graham Neubig. 2018. TRANX: A Transition-based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 7–12.

Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. 2017. Seqgan: Sequence generative adversarial nets with policy gradient. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 31.

Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An Extensive Study on Pre-trained Models for Program Understanding and Generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM.

Wen Zhang, Yang Feng, Fandong Meng, Di You, and Qun Liu. 2019. Bridging the Gap between Training and Inference for Neural Machine Translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 4334–4343.

Yuhao Zhang, Yasharth Bajpai, Priyanshu Gupta, Ameya Ketkar, Miltiadis Allamanis, Titus Barik, Sumit Gulwani, Arjun Radhakrishna, Mohammad Raza, Gustavo Soares, et al. 2022. Overwatch: learning patterns in code edit sequences. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 395–423.

Junchen Zhao, Yurun Song, Junlin Wang, and Ian G Harris. 2022. GAP-Gen: Guided Automatic Python Code Generation. *arXiv preprint arXiv:2201.08810* (2022).

Tianming Zhao, Chunyang Chen, Yuanning Liu, and Xiaodong Zhu. 2021. Guigan: Learning to generate gui designs using generative adversarial networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 748–760.

Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. 2017. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*. 2223–2232.

Received 2023-04-14; accepted 2023-08-27