Compilable Neural Code Generation with Compiler Feedback

Xin Wang[1⋆◇], **Yasheng Wang**[2⋆], **Yao Wan**[4], **Fei Mi**[2], **Yitong Li**[2,3],
Pingyi Zhou[2], **Jin Liu**[1✉], **Hao Wu**[5], **Xin Jiang**[2], **Qun Liu**[2]
[1]School of Computer Science, Wuhan University, China
[2]Huawei Noah's Ark Lab, [3]Huawei Technologies Co., Ltd.
[4]School of Computer Sci. & Tech., Huazhong University of Science and Technology, China
[5]School of Information Science and Engineering, Yunnan University, China
{xinwang0920, jinliu}@whu.edu.cn, wanyao@hust.edu.cn, haowu@ynu.edu.cn
{wangyasheng, feimi2, liyitong3, zhoupingyi, Jiang.Xin, qun.liu}@huawei.com

Abstract

Automatically generating compilable programs with (or without) natural language descriptions has always been a touchstone problem for computational linguistics and automated software engineering. Existing deep-learning approaches model code generation as text generation, either constrained by grammar structures in decoder, or driven by pre-trained language models on large-scale code corpus (e.g., CodeGPT, PLBART, and CodeT5). However, few of them account for compilability of the generated programs. To improve compilability of the generated programs, this paper proposes CompCoder, a three-stage pipeline utilizing compiler feedback for compilable code generation, including language model fine-tuning, compilability reinforcement, and compilability discrimination. Comprehensive experiments on two code generation tasks demonstrate the effectiveness of our proposed approach, improving the success rate of compilation from 44.18 to 89.18 in code completion on average and from 70.3 to 96.2 in text-to-code generation, respectively, when comparing with the state-of-the-art CodeGPT.

1 Introduction

Automated code generation (or program synthesis) has attracted much attention over the past few years (Lu et al., 2021), because of its potential to improve the productivity of developers, as well as to speed up the software development (Parvez et al., 2021; Wang et al., 2021). In the life cycle of software development, different types of code generation tasks are desired, including code completion (Liu et al., 2020b,a), text-to-code generation (Hashimoto et al., 2018), program translation (Chen et al., 2018), and program repair (Yasunaga and Liang, 2021).

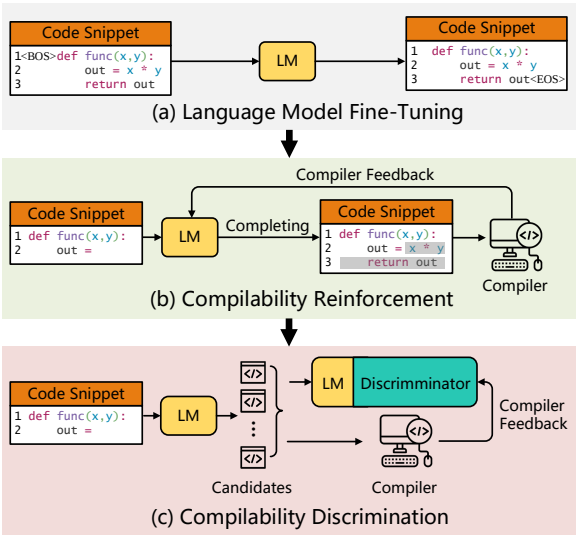


Figure 1: An illustration of Python code completion by CompCoder, utilizing the compiler feedback with three stages.

Recently, much effort has been made to advance the development of code generation (Li et al., 2018), using different logical forms of code, such as the abstract syntax tree (AST) (Kim et al., 2021; Yin and Neubig, 2017; Rabinovich et al., 2017), sketch (Nye et al., 2019) and graph (Yasunaga and Liang, 2020). Benefiting from the strong power of pre-training techniques (Devlin et al., 2019; Wang et al., 2021a) in natural language processing, several attempts have been made towards pre-training a language model on large-scale code corpus for code generation, such as CodeGPT (Lu et al., 2021), PLBART (Ahmad et al., 2021), and CodeT5 (Wang et al., 2021b).

However, to the best of our knowledge, most deep-learning approaches for code generation are still difficult to guarantee the compilability of the generated code, resulting in non-compilable code. For example, Chen et al. (2021) found that up to 67%-97% of patches generated by the most advanced deep-learning-based models are non-

⋆ Equal contribution.
◇ Work done while this author was an intern at Huawei Noah's Ark Lab.
✉ Correspondence author.

Findings of the Association for Computational Linguistics: ACL 2022, pages 9 - 19
May 22-27, 2022

compilable. We think this is because they generally do not directly optimize the compilability for code generation. The generation of non-compilable code will waste the time of programmers, as well as seriously reduce the trust and satisfaction of developers with the model. To improve the compilability of the generated code, some works attempt to repair the synthesized program which fails to compile (Kulal et al., 2019; Yasunaga and Liang, 2020, 2021). Recently, Korbak et al. (2021) attempt to directly generate compilable code using an energy model with compilability constraints.

This paper focuses on the task of compilable neural code generation. Different from previous works, we use compilability signals in two ways and design a novel method to jointly train the discriminator and generator for compilable code generation. Concretely, we propose COMPCODER, a novel three-stage pipeline utilizing compiler feedback for compilable code generation, including language model fine-tuning, compilability reinforcement, and compilability discrimination. Figure 1 shows an example of Python code completion by COMPCODER, which utilizes the compiler feedback in two ways. In Figure 1(b), we use the compiler feedback to optimize the generator. In Figure 1(c), we use the discriminator to check if the results generated by the generator can be successfully compiled. The joint training of the generator and discriminator significantly improves the compilability of the generated code.

Overall, the key contributions of this paper are as follows:

- We use compilability signals in two ways and design a novel method to jointly train the generator and discriminator for compilable code generation, called COMPCODER. We refine a pre-trained code generator using reinforcement learning and jointly learn a discriminator to enforce the generator to correct its own mistakes.
- Comprehensive experiments on two code generation tasks demonstrate the effectiveness of COMPCODER. It boosts the average compilation rate of CodeGPT from 44.18 to 89.18 in the code completion task and from 70.3 to 96.2 in the text-to-code generation task.

2 Preliminary

In this section, we set out notations for task formulation, as well as some preliminaries of compiler feedback. Let $s \in \mathcal{S}$ denote a given input, which can be a piece of partial code, natural-language description, or buggy program. Let $t \in \mathcal{T}$ denote the generated source code. Formally, the problem of code generation can be formulated as learning a mapping f between the input space and target code space, i.e. $f : \mathcal{S} \rightarrow \mathcal{T}$. In this paper, we investigate two specific code generation tasks, code completion and text-to-code generation, conditioned on different inputs.

Code Completion Let $c = \{c_1, c_2, \ldots, c_{|c|}\}$ denote a sequence of code tokens for program c, where $|c|$ denotes the length of the code. We use notation $c_{1:\,m} \in \mathcal{S}$ to refer to the previous code snippet $\{c_1, c_2, \ldots, c_m\}$ and notation $c_{m+1:\,|c|} \in \mathcal{T}$ to represent the subsequent code snippet $\{c_{m+1}, \ldots, c_{|c|}\}$. The code completion task can be defined as generating the subsequent (t) code token sequence $c_{m+1:\,|c|}$, given the previous (s) code sequence $c_{1:\,m}$.

Text-to-Code Generation Different from code completion, text-to-code generation aims to generate a whole program based on natural language description. Let $d = \{d_1, d_2, \ldots, d_{|d|}\}$ refer to a sequence of natural-language tokens. The text-to-code generation task can be defined as generating source code $c = t \in \mathcal{T}$, given the corresponding natural language description $d = s \in \mathcal{S}$.

Compiler Feedback As the whole program c is generated, no matter from partial code snippets or natural-language descriptions, we feed it into a compiler to test whether it can be compiled successfully. Formally, we define the the compiler feedback as:

$$\textit{feedback} = \mathbb{1}_{\text{Compiler}}(c)\,, \tag{1}$$

where the compiler feedback is a binary value (compilable or non-compilable), and c denotes the code snippet fed into the compiler. As for the task of text-to-code generation, we simply feed the generated code t into the compiler, i.e., $c = t$. As for the task of code completion, we concatenate the partial code with generated code as a whole program, i.e., $c = [s; t]$, where ; is the concatenation operation.

3 COMPCODER

Figure 2 shows the overall architecture of COMPCODER on the code completion task, which covers three stages, i.e., language model fine-tuning (Stage 1), compilability reinforcement (Stage 2) and compilability discrimination (Stage 3). In the following

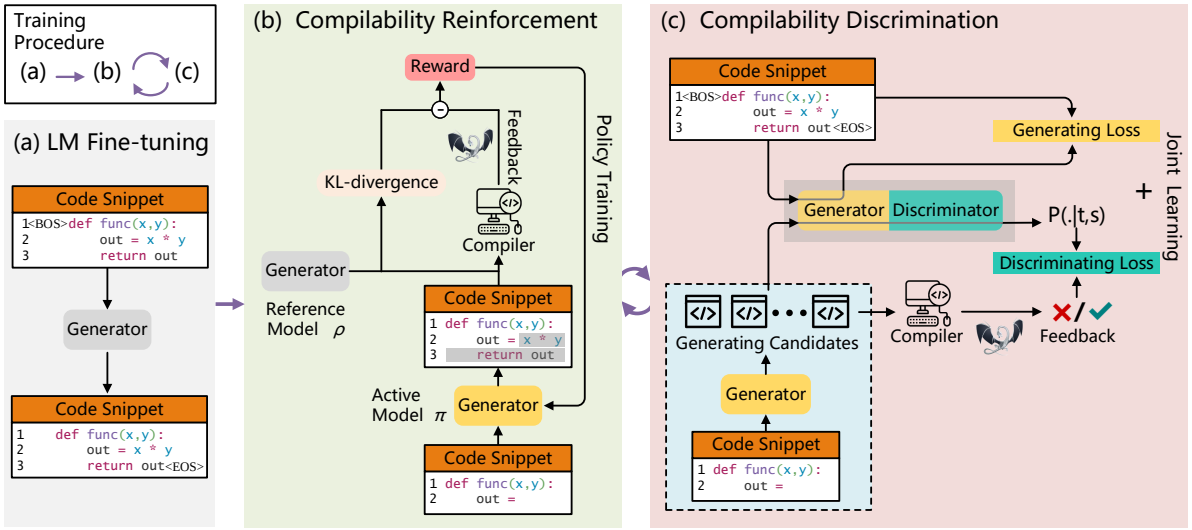


Figure 2: An illustration of our proposed three-stage pipeline for Python code completion. (a) We first fine-tune the generator based on pre-trained language models. (b) We take the compiler feedback into account as a reward via reinforcement learning. (c) We design a compilability discriminator which is jointly trained with the generator, to enforce the generator to correct its own mistakes. Stages 2 and 3 are performed alternately.

subsections, we will elaborate on each stage one by one. We alternately perform Stages 2 and 3, as described in Section 3.4.

3.1 Stage 1: Language Model Fine-Tuning

As shown in Figure 2(a), we adopt CodeGPT as the generator, which uses GPT-2 (Radford et al., 2019) as the starting point and is continually pre-trained on the large-scale code corpus. Our generator is then fine-tuned on the target task to minimize the cross-entropy loss:

$$\mathcal{L}_{\mathrm{G}} = -\frac{1}{|\mathcal{M}|}\sum_{i}^{|\mathcal{M}|}\sum_{j}^{|\mathcal{V}|} Y_{ij}\log P_{ij}\,, \qquad (2)$$

where $\mathcal{M}$ denotes the set of the generated code tokens, $\mathcal{V}$ represents the vocabulary, Y_{ij} denotes the label of the code token i in class j, and P_{ij} is the predicted probability of token i in class j.

During training, the generator takes $x = \{\texttt{<BOS>}, c, \texttt{<EOS>}\}$ as the input in the code completion task, and $x = \{d, \texttt{<BOS>}, c, \texttt{<EOS>}\}$ as input in the text-to-code generation task, correspondingly. Special tokens <BOS> and <EOS> indicate the start and end symbols of code sequences. After several epochs of supervised fine-tuning on the target task dataset, we save the trained generator, which will be used in the next stage.

3.2 Stage 2: Compilability Reinforcement

Reinforcement Learning (RL) is a method of learning the optimal policy by obtaining reward signals from the real environment (Sutton and Barto, 1998; Wan et al., 2018). As shown in Figure 2(b), we use the fine-tuned generator ρ (after Stage 1) as the *reference model*. Then we initialize a policy $\pi = \rho$. Given an input sequence $s \in \mathcal{S}$, our goal is to find a policy π that generates an output sequence $t \in \mathcal{T}$ with the objective of maximizing the compilability-based reward. We use RL (specifically PPO2 version of Proximal Policy Optimization (Schulman et al., 2017)) to directly optimize the expected reward as:

$$\mathbb{E}_{\pi}\left[r\right] = \mathbb{E}_{s\sim\mathcal{S},t\sim\pi(.|s)}\left[r(s,t)\right], \qquad (3)$$

where the policy π is rewarded by the compiler (Eq. 1), r is the reward function. We define $r(s,t) = 1.0$ iff the code can be compiled by the program compiler and $r(s,t) = -1.0$ otherwise.

It is worth mentioning that code compilability constraints can be strong or weak. *Strong constraint* is defined that a long piece of code snippet may not be correctly compiled if a certain token is changed. And *weak constraint* means a blank string consisting of whitespace characters can be correctly compiled by the compiler. Concretely, in the text-to-code generation task, if the generator generates a string composed of whitespace characters, the compiler will consider it as a good case. In the code completion task, if the previous code snippet is compilable, the generator can fool the compiler easily. The RL is good at making use of this, resulting in the generated code can be com-

Figure 3: An example of code completion. We mask the last five tokens of the code and let the generator complete them. Some minor mistakes prevent four candidates from being correctly compiled by the program compiler.

piled, but seriously deviating from the generation likelihood objective.

To avoid *active model* π being too far away from *reference model* ρ, we add a Kullback-Leibler (KL) penalty with expectation, e.g., $\beta\mathrm{KL}(\pi, \rho)$ (Ziegler et al., 2019). Therefore, the modified reward will be reformulated as follows:

$$r(s,t) = r(s,t) - \beta \log \frac{\pi(t|s)}{\rho(t|s)}\,, \tag{4}$$

where β is a constant, which plays the role of an entropy bonus, preventing the policy from moving too far from the range where r is valid.

To alleviate the imbalance between the reward term and the KL penalty term and improve the stability of training, we use autoregressive fine-tuning (Causal Language Modeling) (Radford et al., 2019) to make the KL penalty term fluctuate within a small range after RL training. This fine-tuning process incorporates a compilability-aware discriminator that will be introduced in the next stage.

3.3 Stage 3: Compilability Discrimination

Figure 3 shows an example of code completion. We mask the last five tokens of a Python function and ask the generator to complete them. The generator generates five candidates with high probabilities. Some minor mistakes prevent four of them from being successfully compiled. We hope the generator can have more perception power to explicitly distinguish compilable and non-compilable code generated by itself. Therefore, at this stage, we design a compilability-aware discriminator to deal with this issue.

Concretely, we add a discriminator (a two-layer MLP equipped with the $\tanh$ activation function between layers) after the final hidden layer of the generator. As shown in Figure 2(c), given the input sequence (s), we perform beam search on the generator to generate top-k candidates (t). Each entire code $c \in \mathcal{Q}$ ($c = [s;t]$ in the code completion task) is labeled by the program compiler as positive (1) or negative (0), depending on whether it can be successfully compiled (see Eq. 1).

We use the hidden representation of the last token (<EOS>) as the final representation of the entire code c. Finally, the hidden representation of the last token (<EOS>) is fed into the discriminator for prediction:

$$h_{\texttt{<EOS>}} = \mathrm{CodeGPT}(s,t)\,, \tag{5}$$
$$h'_{\texttt{<EOS>}} = \mathrm{Discriminator}(h_{\texttt{<EOS>}})\,, \tag{6}$$
$$P(.|t,s) = \mathrm{softmax}(h'_{\texttt{<EOS>}})\,, \tag{7}$$

where $h_{\texttt{<EOS>}}$ denotes the representation of the last token <EOS>. The training loss of the discrimination process can be defined as:

$$\mathcal{L}_{\mathrm{D}} = -\frac{1}{|\mathcal{Q}^+ \cup \mathcal{Q}^-|}\left[\sum_{c\in\mathcal{Q}^+} \log\ P(1|t,s) + \sum_{c\in\mathcal{Q}^-} \log\ P(0|t,s)\right], \tag{8}$$

where $\mathcal{Q}^+$ and $\mathcal{Q}^-$ represent positive and negative sets respectively. The parameters of the generator and discriminator will be jointly updated.

At this stage, we jointly train the generator and discriminator, including a generating objective (*to learn the generator only*) and a discriminating objective (*to learn the generator and discriminator together*), as shown in Figure 2(c). The joint training loss is defined as follows:

$$\mathcal{L} = \mathcal{L}_{\mathrm{G}} + \mathcal{L}_{\mathrm{D}}\,. \tag{9}$$

3.4 Overall Pipeline

Training Procedure We perform an interactive training procedure. Concretely, except that the first epoch contains Stages 1, 2, and 3, each subsequent epoch only consists of Stages 2 and 3. We update the *reference model* (at Stage 2), and *candidates* in Stage 3 is generated on the training dataset, which is time consuming, so we update the *candidates* in a preset frequency.

For better understanding, Stage 2 improves the compilability of generated code, Stage 3 distinguishes the compilable and non-compilable code generated by itself. Stage 2 and 3 refine each other and improve the performance iteratively, which is a basic idea of this training procedure. We think that the generator with high compilability (after Stage 2) facilitates the learning of the discriminator (discriminating objective at Stage 3). The autoregressive fine-tuning (generating objective at Stage 3) helps the KL penalty term (at Stage 2) fluctuate in a small range, improving the stability of RL training. At Stage 3, the discriminating objective is optimized by learning the generator and discriminator together, which makes the generator have more perception power to distinguish compilable and non-compilable code.

Inference Procedure The model inference consists of two stages. Given an input sequence (s), we perform the beam search on the generator to generate top-k candidates. The code (c in Eq. 1) with the highest compilability probability evaluated by the discriminator will be selected. Then the output (t) can be obtained as the final result.

4 Experiment Setup

4.1 Evaluation Tasks and Datasets

We conduct experiments on two tasks: code completion and text-to-code generation. To investigate the compilability of the generated code, we need to preserve the indentation and newline operations in code. We also need to make sure that the code and its version belong to the scope of the compiler. Existing datasets on both of the two tasks usually do not serve these considerations. For convenience, we choose Python for experiments, as it is very popular and used in many projects. We conduct all experiments based on Python 3 environment and adopt the `codeop`[1] module to simulate the program compiler. We remove code that could not be compiled correctly by the compiler.

Code Completion For the code completion task, we use the Python corpus in CodeSearchNet (Husain et al., 2019). We want to study the compilability of long enough code, while longer code means higher computational overhead. Therefore, we extract $50k$ compilable Python methods (Python 3 version) with eclectic token lengths ranging from 64 to 96. We randomly select $45k$ samples for training and the remaining $5k$ samples for testing. We mask a different number of tokens at the tail of the source code and let the model complete.

Text-to-Code Generation For the text-to-code generation task, we adopt the AdvTest dataset (Lu et al., 2021), which contains 251,820 text and Python code pairs. We only need code in Python 3 version. We expect code token lengths to range from 128 to 170, a moderate length, and text token lengths to be at least more than 5, containing sufficient semantics. Finally, we extract about $41k$ text-code pairs. We randomly select $40k$ text-code pairs for training, and the remaining $1k$ text-code pairs for testing.

4.2 Evaluation Metrics

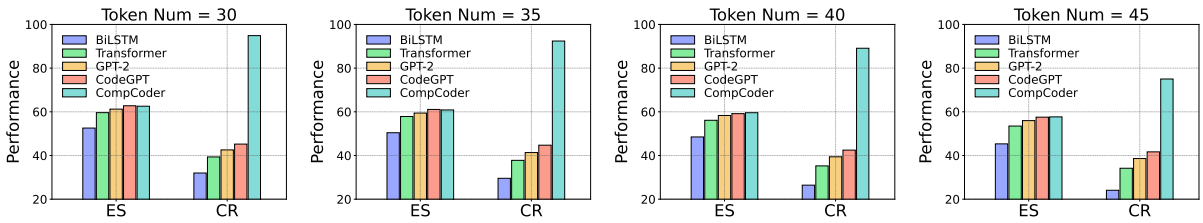
To evaluate the quality of the generated code, we adopt two widely-used evaluation metrics: Levenshtein *Edit Similarity (ES)* (Svyatkovskiy et al., 2020; Lu et al., 2021) and *Compilation Rate (CR)* (Kulal et al., 2019). Levenshtein Edit Similarity measures the number of single-character edits required to transform one string into another. It is a critical evaluation metric for the code generation scenario, as it measures the effort required for the developer to correct the code. Compilation Rate measures how many code can be correctly compiled by the program compiler. For both of these metrics, bigger values indicate better performance.

4.3 Baseline Methods

We compare our approach with various state-of-the-art models in the code completion task and the text-to-code generation task:

- **BiLSTM** is a Seq2Seq model based on Bidirectional LSTM with an attention mechanism (Luong et al., 2015).
- **Transformer** (Vaswani et al., 2017) is the base architecture of CodeGPT. We use 6-layer Transformer decoder to conduct experiments.

[1] https://docs.python.org/3.6/library/codeop.html



(a) (b) (c) (d)

Figure 4: : Results in the code completion task (completing 30, 35, 40, 45 tokens respectively) evaluating with Edit Similarity (ES) and Compilation Rate (CR) metrics, using the CodeSearchNet-Python dataset.

- **GPT-2** (Radford et al., 2019) is an autoregressive pre-trained model trained on large-scale text corpus.
- **CodeGPT** (Lu et al., 2021) is pre-trained with source code corpus on the basis of GPT-2 vis causal language modeling objective (Radford et al., 2019).
- **PLBART** (Ahmad et al., 2021) is based on the BART (Lewis et al., 2020) architecture, which is pre-trained on large-scale Java and Python corpora via denoising autoencoding.
- **CodeT5** (Wang et al., 2021b) is based on the T5 (Raffel et al., 2020) architecture, which employs denoising sequence-to-sequence pre-training on multiple programming languages.

4.4 Implementation Details

In the code completion task, we set the learning rate as $1.5e$-5, the batch size as 32, the maximum fine-tuning epoch as 20, the maximum code sequence length as 96. We mask different numbers of code tokens (25, 30, 35, 40, and 45) and ask the model to complete them. We set the minimum generation length as 25, 30, 35, 40, and 45 accordingly. In the text-to-code generation task, we set the learning rate as $1.5e$-5, the batch size as 16, the maximum fine-tuning epoch as 20, the maximum text and code sequence length as 32 and 170. We set the minimum generation length as 96 (the generated code is slightly shorter than the ground-truth is allowed). In these two tasks, the generated sequence consisting of whitespace characters will be considered as a bad case.

We use the Adam optimizer to update model parameters. We train our model on the basis of CodeGPT checkpoint[2]. Our model is trained on 2 NVIDIA Tesla V100 with 32GB memory. We employ the same tokenizer as CodeGPT. To train the policy π, we use the PPO2 version of Proximal Policy Optimization (Schulman et al., 2017). In each epoch, we only randomly select 5% training data for the stability of RL training (Stage 2). In other stages (Stages 1 and 3), we use the full training data. To generate *candidates* (at Stage 3), we set the beam size as 5 in beam search. For efficiency, we update the *candidates* every 5 epochs.

[2] https://huggingface.co/microsoft/CodeGPT-small-py-adaptedGPT2

Models	ES	CR
BiLSTM	55.32	36.34
Transformer	61.47	40.22
GPT-2	63.02	43.26
CodeGPT	64.47	46.84
CompCoder	**64.53**	**94.48**

Table 1: Results in the code completion task (completing 25 tokens) evaluating with Edit Similarity (ES) and Compilation Rate (CR) metrics, using the CodeSearchNet-Python dataset.

5 Results and Analysis

5.1 Code Completion

Table 1 shows the results of the code completion task. We mask 25 tokens at the tail of code functions and ask the generation model to complete. We can observe that: (1) The code generated by existing autoregressive models has a low Compilation Rate. CodeGPT and GPT-2 only achieve 46.84 and 43.26 scores respectively on the Compilation Rate, which means that more than half of the code generated by them cannot be correctly compiled by the program compiler. (2) CompCoder significantly improves the Compilation Rate. It obtains 94.48 scores on the Compilation Rate, which is 47.64 points higher than the closest competitor

(CodeGPT). (3) When our approach significantly improves the Compilation Rate, it does not sacrifice the fluency of the generated code. CompCoder obtains a comparable and even slightly better Edit Similarity score than other baselines, indicating that it effectively preserves the code fluency.

Figure 4 presents more results of the code completion task in the setting of completing 30, 35, 40, and 45 tokens. CompCoder still effectively improves the Compilation Rate when generating longer code. As the completion length increases, our approach outperforms CodeGPT by 49.66, 47.68, 46.64, and 33.36 points in the setting of completing 30, 35, 40, and 45 tokens, respectively. On average, our approach outperforms CodeGPT by *45* points across a different number of tokens for the task of code completion.

Models	ES	CR
BiLSTM	54.86	48.7
Transformer	57.47	55.6
GPT-2	60.54	63.3
CodeGPT	61.82	70.3
PLBART	62.43	71.9
CodeT5	62.58	73.1
CompCoder	**62.74**	**96.2**

Table 2: Results in the text-to-code generation task evaluating with Edit Similarity (ES) and Compilation Rate (CR), using the AdvTest dataset.

5.2 Text-to-Code Generation

Table 2 presents the results of the text-to-code generation task. We could see that: (1) CompCoder significantly outperforms all other models w.r.t. the Compilation Rate. E.g., CompCoder achieves 23.1 points and 24.3 points improvements when compared with PLBART and CodeT5 respectively. (2) Compared to code completion task (Table 1), all models in the text-to-code generation task have relatively higher Compilation Rate. One of the main reasons we think may be: code completion requires the generated code to be constrained by the existing (previous) code, which is a much stronger restriction than the text-to-code generation.

5.3 Ablation Study

We compare several simplified versions of our model to understand contributions of different components, including the Reinforcement Learning (RL) component and the discriminator's effect for both model training (D_{train}) and model inference (D_{test}). As a case study, we take the code completion task as an example in the setting of completing 25 tokens and present the results in Table 3.

Models	ES	CR
(1) CodeGPT	64.47	46.84
(2) w/ D_{train}	65.46	64.88
(3) w/ RL	64.71	76.48
(4) w/ RL+D_{train}	64.43	83.14
(5) w/ D_{train}+D_{test}	65.24	81.96
(6) w/ RL+D_{train}+D_{test} (Ours)	64.53	94.48

Table 3: Ablation study in the code completion task in the setting of completing 25 code tokens.

Several meaningful observations can be drawn: **First**, both RL (Row 2) and D_{train} (Row 3) effectively increase the code Compilation Rate of the generation model (CodeGPT in Row 1), which confirms that the two components we designed can indeed improve the ability of the generator for compilable code generation. **Second**, applying RL and D_{train} together (Row 4) further improves the Compilation Rate over their individual contributions. **Third**, using the discriminator to select the output during model inference stage (D_{test}) is beneficial. It further boosts the Compilation Rate of vanilla "D_{train}" by 17.08% (Row 5 v.s. Row 2) and boosts "RL+D_{train}" by 11.34% (Row 6 v.s. Row 4). **Forth**, these three components (RL, D_{train}, D_{test}) that effectively improve the Compilation Rate do not compromise the generation capability measured by the Edit Similarity.

5.4 Case Study

To better understand the effectiveness of our proposed approach, we present two cases for code completion and text-to-code generation tasks respectively. For both CodeGPT and CompCoder, we present top-1 result in Figure 5. For code completion, we observe that CodeGPT can not complete code with high quality (non-compilable), while CompCoder can complete the code well, and it is *exactly the same* for the reference solution. For text-to-code generation, we observe that although both models can not generate exactly the same code as the reference solution, CompCoder generates a compilable code at the function level. These results reveal the effectiveness of our proposed approach for compilable code generation.

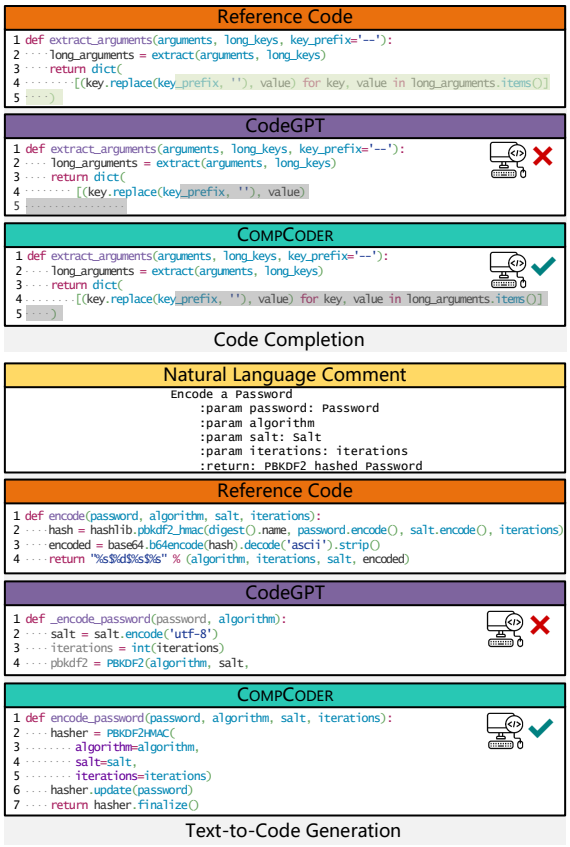


Figure 5: Case study for code completion and text-to-code generation tasks.

6 Related Work

Neural Code Generation With the rapid development of Deep Learning (DL), some researchers attempt to use DL for code generation tasks. Liu et al. (2020a) proposed a neural architecture for code completion task with multi-task learning based on the architecture of Transformer-XL Dai et al. (2019) and BiLSTM (Schuster and Paliwal, 1997). Kim et al. (2021) presented several ways of feeding the code structure to Transformer (Vaswani et al., 2017) and further improved the accuracy of the code prediction (next token prediction) task. Wei et al. (2019) adopted an encoder-decoder architecture and utilized the relations between code generation and code summarization to improve the performance of both tasks. Yasunaga and Liang (2021) proposed a new training approach for program repair. They used the critic to check a fixer's output on real bad inputs and add good outputs to the training data, and trains a breaker to generate realistic bad code from good code. Yasunaga and Liang (2020) used compiler error messages to repair programs. They designed a program-feedback graph and then applied a graph neural network on top to model the reasoning process. Many unlabeled programs are used for program repair with self-supervised learning.

Benefiting from the strong power of pre-training techniques (Devlin et al., 2019; Wang et al., 2021a) in natural language processing, such as GPT (Radford and Narasimhan, 2018), BART (Lewis et al., 2020), and T5 (Raffel et al., 2020), some recent works attempt to pre-train language models on the corpus of source code for code generation. Lu et al. (2021) proposed CodeGPT follows the architecture of GPT-2 (Radford et al., 2019), which is pre-trained with a causal language modeling (CLM) objective on large-scale source code. Ahmad et al. (2021) proposed PLBART follows the architecture of BART (Lewis et al., 2020), which is pre-trained on Java and Python functions paired with code comments via denoising autoencoding. Wang et al. (2021b) proposed CodeT5 based on the T5 (Raffel et al., 2020) architecture, which employs denoising sequence-to-sequence pre-training on multiple programming languages.

Reinforced Text Generation Reinforcement learning (Sutton and Barto, 1998) has shown great success in various tasks. It focuses on how agents ought to take actions in an environment to maximize the cumulative reward, is well suited for decision-making tasks. Ranzato et al. (2016) were among the first to apply REINFORCE algorithm (Williams, 1992) to train recurrent neural networks on sequence generation tasks, suggesting that directly optimizing the metric used at the test phase can lead to better results. Chen and Bansal (2018) proposed a hybrid extractive-abstractive architecture with policy-based reinforcement learning. They used an extractor agent to select salient sentences and then employed an abstractor network to rewrite these extracted sentences. Wan et al. (2018); Wang et al. (2022) incorporated the tree structure and sequential content of code snippets and designed a deep reinforcement learning framework optimized by the metric of BLEU to improve the performance of the code summarization task. Yao et al. (2019) proposed a reinforcement learning framework, which encourages the code annotation model to generate annotations that can be used for code retrieval tasks. Korbak et al. (2021) proposed an energy-based model with an imposed constraint of generating only compilable sequences to improve compilation rates of generated code.

7 Conclusion and Future Work

In this paper, we use the compilability signals in two ways and design a novel method to jointly train the generator and discriminator for compilable code generation, called CompCoder. Comprehensive experiments on two code generation tasks demonstrate the effectiveness of CompCoder, improving the average compilation rate of state-of-the-art CodeGPT from 44.18 to 89.18 in the code completion task and from 70.3 to 96.2 in the text-to-code generation task.

This work presents our preliminary attempt to generate compilable code. Yet, considering the compilation rate is not the whole story as it still cannot guarantee the correctness of generated code. As a future work, we would like to utilize unit tests to evaluate the code correctness towards building more useful code generation models.

Acknowledgements

This work is supported by National Natural Science Foundation of China under Grant No. 61972290. Yao Wan is partially supported by National Natural Science Foundation of China under Grant No. 62102157. Hao Wu is supported by National Natural Science Foundation of China under Grant No. 61962061.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 2655–2668. Association for Computational Linguistics.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 2552–2562.

Yen-Chun Chen and Mohit Bansal. 2018. Fast abstractive summarization with reinforce-selected sentence rewriting. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pages 675–686. Association for Computational Linguistics.

Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Trans. Software Eng.*, 47(9):1943–1959.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 2978–2988. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

Tatsunori B. Hashimoto, Kelvin Guu, Yonatan Oren, and Percy Liang. 2018. A retrieve-and-edit framework for predicting structured outputs. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 10073–10083.

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

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 150–162. IEEE.

Tomasz Korbak, Hady ElSahar, Marc Dymetman, and Germán Kruszewski. 2021. Energy-based models for code generation under compilability constraints. *arXiv preprint arXiv: 2106.04985*.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. 2019. Spoc: Search-based pseudocode to code. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 11883–11894.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 7871–7880. Association for Computational Linguistics.

Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code completion with neural attention and pointer networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 4159–4165. ijcai.org.

Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. 2020a. A self-attentional neural architecture for code completion with multi-task learning. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*, pages 37–47. ACM.

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

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pages 1412–1421. The Association for Computational Linguistics.

Maxwell I. Nye, Luke B. Hewitt, Joshua B. Tenenbaum, and Armando Solar-Lezama. 2019. Learning to infer program sketches. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 4861–4870. PMLR.

Md. Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, pages 2719–2734. Association for Computational Linguistics.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 1139–1149. Association for Computational Linguistics.

Alec Radford and Karthik Narasimhan. 2018. Improving language understanding by generative pre-training.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

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

Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2016. Sequence level training with recurrent neural networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv: 1707.06347*.

Mike Schuster and Kuldip K. Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE Trans. Signal Process.*, 45(11):2673–2681.

Richard S. Sutton and Andrew G. Barto. 1998. Introduction to reinforcement learning.

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

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

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 397–407. ACM.

Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip S. Yu, and Guandong Xu. 2022. Reinforcement-learning-guided source code summarization using hierarchical attention. *IEEE Transactions on Software Engineering*, 48(1):102–119.

Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang.

2021. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv: 2108.04556*.

Xin Wang, Pingyi Zhou, Yasheng Wang, Xiao Liu, Jin Liu, and Hao Wu. 2021a. Servicebert: A pre-trained model for web service tagging and recommendation. In *Service-Oriented Computing - 19th International Conference, ICSOC 2021, Virtual Event, November 22-25, 2021, Proceedings*, volume 13121 of *Lecture Notes in Computer Science*, pages 464–478. Springer.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021b. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 6559–6569.

Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8:229–256.

Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. Coacor: Code annotation for code retrieval with reinforcement learning. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pages 2203–2214. ACM.

Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 10799–10808. PMLR.

Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 11941–11952. PMLR.

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, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 440–450. Association for Computational Linguistics.

Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul F. Christiano, and Geoffrey Irving. 2019. Fine-tuning language models from human preferences. *CoRR*, abs/1909.08593.