IRCoCo: Immediate Rewards-Guided Deep Reinforcement Learning for Code Completion

BOLUN LI, Shandong Normal University, China
ZHIHONG SUN, Shandong Normal University, China
TAO HUANG, Shandong Normal University, China
HONGYU ZHANG, Chongqing University, China
YAO WAN, Huazhong University of Science and Technology, China
GE LI, Peking University, China
ZHI JIN*, Peking University, China
CHEN LYU*[†], Shandong Normal University, China

Code completion aims to enhance programming productivity by predicting potential code based on the current programming context. Recently, pre-trained language models (LMs) have become prominent in this field. Various approaches have been proposed to fine-tune LMs using supervised fine-tuning (SFT) techniques for code completion. However, the inherent *exposure bias* of these models can cause errors to accumulate early in the sequence completion, leading to even more errors in subsequent completions. To address this problem, deep reinforcement learning (DRL) is an alternative technique for fine-tuning LMs for code completion, which can improve the generalization capabilities and overall performance. Nevertheless, integrating DRL-based strategies into code completion faces two major challenges: 1) The dynamic nature of the code context requires the completion model to quickly adapt to changes, which poses difficulties for conventional DRL strategies that focus on delayed rewarding of the final code state. 2) It is difficult to evaluate the correctness of partial code, thus the reward redistribution-based strategies cannot be adapted to code completion. To tackle these challenges, we propose IRCoCo, a code completion-specific DRL-based fine-tuning framework. This framework is designed to provide immediate rewards as feedback for detecting dynamic context changes arising from continuous edits during code completion. With the aid of immediate feedback, the fine-tuned LM can gain a more precise understanding of the current context, thereby enabling effective adjustment of the LM and optimizing code completion in a more refined manner. Experimental results demonstrate that fine-tuning pre-trained LMs with IRCoCo leads to significant improvements in the code completion task, outperforming both SFT-based and other DRL-based baselines.

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

---

*Zhi Jin and Chen Lyu are the corresponding authors.
[†]This work was done when Chen Lyu was a visiting scholar at Peking University.

Authors' addresses: Bolun Li, School of Information Science and Engineering, Shandong Normal University, Jinan, China, libolun118@gmail.com; Zhihong Sun, School of Information Science and Engineering, Shandong Normal University, Jinan, China, 2022021002@stu.sdnu.edu.cn; Tao Huang, School of Information Science and Engineering, Shandong Normal University, Jinan, China, 2022317095@stu.sdnu.edu.cn; Hongyu Zhang, Chongqing University, Chongqing, China, hyzhang@cqu.edu.cn; Yao Wan, Huazhong University of Science and Technology, Wuhan, China, wanyao@hust.edu.cn; Ge Li, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Beijing, Peking University, China, lige@pku.edu.cn; Zhi Jin, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Beijing, Peking University, China, zhijin@pku.edu.cn; Chen Lyu, School of Information Science and Engineering, Shandong Normal University, Jinan, China, lvchen@sdnu.edu.cn.

---

1 INTRODUCTION

Intelligent code completion can significantly boost the productivity of software developers by
offering automated suggestions of subsequent code elements, based on the programming contexts
(e.g., the already typed partial code). It has evolved into a key feature within contemporary integrated
development environments (IDEs), exemplified by Visual Studio Code and its Copilot extension,
and IntelliJ IDEA with its IntelliSense feature. Based on the degree of code completion, we classify
contemporary code completion tools into two distinct categories: 1) token-level code completion,
which centers on predicting individual tokens like function names, variable names, keywords,
and operators within the code context [1–4]; and 2) line-level code completion, designed to tackle
the completion of multiple tokens, including function or class definitions, or the completion of
multi-word expressions within intricate statements [5–7]. In this paper, we narrow our research
scope to the latter one, which presents a more demanding challenge.

Recently, we have witnessed remarkable achievements in code generation, exemplified by the
outstanding performance of pre-trained language models (LMs) as demonstrated by tools like
GitHub Copilot [8] and Amazon's CodeWhisperer [9]. However, pre-training an LM on a vast
code corpus remains a time-consuming and computationally demanding endeavor, rendering it
impractical for academic and small business environments with limited computing resources [10].
Furthermore, subscribing to a code completion service may raise serious concerns about privacy
leakage for many organizations. For instance, recent reports have revealed three incidents of data
leakage at Samsung Electronics [11] when using online code completion services, e.g., ChatGPT.
These concerns underscore a pressing need to develop a localized and customized code completion
model based on fine-tuning techniques for personal use.

To tackle the aforementioned concerns, many approaches based on supervised fine-tuning (SFT)
have been proposed to refine LMs to the specific task of code completion, thereby enhancing their
effectiveness within authentic code completion contexts [7, 12]. In the SFT of a code completion
model, we typically refine the parameters of models by maximizing the log-likelihood of the
subsequent ground-truth code token, also referred to as "teacher-forcing". We argue that the
"teacher-forcing" strategy may suffer the *exposure bias* issue. This issue emerges because, during
the training phase, models are consistently exposed to ground-truth sequences. However, in the
testing phase without ground truth, these models must predict based on their prior outputs, causing
potential discrepancies between training and testing (i.e., *exposure bias*). Over time, this *exposure
bias* would lead to accumulating errors in the testing phase, hindering the model from generating
tokens that might be contextually appropriate but had a lower likelihood of being chosen during
the training process [13, 14].

To mitigate the issue of *exposure bias* inherent in "teacher-forcing", deep reinforcement learn-
ing (DRL), which utilizes a reward function to guide the model towards optimal completion
sequences during training, is developed. DRL, rather than sequentially predicting tokens, blends
exploration and exploitation. Through its dual-network design, with the "actor" offering local-
ized token predictions and the "critic" giving global feedback on potential state outcomes, DRL
refines decision-making. This combined strategy ensures that the model can recognize and utilize
contextually appropriate tokens, even those with lower probabilities that might be overlooked by
using the actor network in isolation. Thus, the issue of *exposure bias* can be resolved. On some

related code intelligence tasks, such as natural language to code (NL2Code), DRL-based models show more promising results. For instance, Shojaee et al. [15] proposed the PPOCoder, which is a fine-tuned model, guided by the reward signals derived from unit tests conducted as code generation is completed. We refer to the reward obtained upon completion of code generation as a delayed reward. Le et al. [16] proposed CodeRL, which obtains a delayed reward based on whether the generated code can pass the unit tests. This delayed reward is subsequently redistributed to individual-generated tokens, reflecting their significance in achieving positive unit test outcomes.

Motivated by the aforementioned insights, this paper introduces an innovative DRL-based alignment method specifically designed for code completion. To maximize the potential of the DRL reward mechanism and guide the code completion model toward precise predictions, two pivotal challenges arise: 1) handling of dynamic intents and 2) evaluation of partial code.

Challenge 1: Handling of Dynamic Intents. The intents for the code completion task are determined by the context of the code in the file currently being edited. As edits are made during the completion process, the context code changes, giving these intents a dynamic nature. It is difficult for existing delayed reward-based strategies, such as PPOCoder, to provide accurate feedback on such dynamically changing intents. Such strategies predominantly rely on the evaluative outcomes of the final generated code as a form of rewarding feedback to the environment. As a result, such an approach fails to capture the nuanced alterations in the context presented by intermediate code fragments during the completion process. These nuanced changes can perturb the code completion model's precise semantic understanding of the current code context and exert a substantial influence on the model's ensuing decisions.

Challenge 2: Evaluation of Partial Code. Code completion typically produces partial code based on the local context. Since such code does not always offer complete functionality, it is challenging to directly perform unit testing to verify its correctness. Thus, reward redistribution-based strategies, such as those adopted by CodeRL [16], which rely on the analysis of unit test results of the generated code, cannot be adapted to code completion. Moreover, benchmark datasets designated for code completion (e.g., Py150 [17] and Java Corpus [18]) often lack test cases, which further exacerbates the challenge of evaluating partial code.

These challenges lead to a substantial gap between the capabilities the code completion model aspires to achieve and the support contemporary DRL methodologies offer. To mitigate this gap, we propose IRCoCo (**I**mmediate **R**ewards-Guided Deep Reinforcement Learning for **Co**de **Co**mpletion), a DRL-based alignment mechanism that is model-agnostic, devised specifically for the unique demands of code completion tasks. **First**, to tackle *Challenge 1*, we formulate an immediate rewards-guided DRL alignment architecture. This architecture offers real-time environmental feedback corresponding to the changing context during the code completion process. Such immediate feedback enables the model to more finely discern shifts in both intents and code semantics. Consequently, the model can iteratively adjust its generative strategy, thereby outputting code snippets that are in alignment with the current, most up-to-date intents. **Second**, acknowledging that code completion tasks often lack overt functional feedback mechanisms, such as unit tests, we design a novel reward shaping method based on the beneficial evaluation of subsequent code fragment completion to mitigate *Challenge 2*. The core motivation behind this approach is to evaluate the validity of the tokens generated at a given time step for the accurate completion of subsequent code fragments. Within this dynamically evolving context, the immediate reward accurately gauges the extent to which the current token influences the ultimate code completion outcome. This immediate feedback facilitates more effective strategy learning for the model, harmonizes the balance between exploration and exploitation, and further refines the precision of subsequent token generation. This, in turn, accommodates the fluid nature of code completion contexts and enhances the accuracy of the code completion.

In this paper, we develop IRCoCo utilizing the actor-critic framework and examine the performance across six widely adopted pre-trained LMs. These LMs serve as the actor to perform code completion, while two quality evaluators for generated code, trained using BLEU [19] and Edit-Sim [32] metrics, act as the critic to produce immediate rewards; these components are incorporated within the IRCoCo framework. We assess IRCoCo's performance using two comprehensive datasets for Python and Java, culminating in 28 distinct experimental configurations (7 pre-trained LMs × 2 quality evaluators × 2 programming languages). The experimental outcomes indicate that across various configurations, IRCoCo consistently enhances the efficacy of code completion models. For instance, when employing CodeGPT as the actor and utilizing BLEU to train the code completion quality evaluator as the critic, the Edit-Sim scores demonstrate an improvement of 7.9 % and 1.6 % for Python and Java datasets, respectively, compared to the SFT method. Similarly, the EM scores exhibit significant enhancements of 40.2 % and 4.2 %, while the BLEU-4 scores see increases of 14.7 % and 1.6 %, compared to the SFT method. Additionally, the CodeBLEU [20] score witness improvements of 7.9 % and 0.6 % in the same datasets, compared to the SFT method.

In summary, this paper makes the following major contributions.

- **Significant Problem.** We provide a comprehensive analysis of the characteristics of code completion tasks and identify effective ways to apply SFT and DRL-based alignment mechanisms to code completion. Our study introduces a novel, targeted fine-tuning paradigm specifically tailored for code completion tasks.
- **Novel Approach.** We introduce IRCoCo, an immediate rewards-guided DRL framework with great potential to improve the performance of pre-trained LMs on code completion.
- **Extensive Experiments.** We perform extensive experiments using six open-source pre-trained LMs on the Py150 and Java Corpus datasets. Our empirical findings show that our methodology yields models with significantly improved performance across various metrics, including Edit-Sim, EM, CodeBLEU, and BLEU, thereby substantially improving code completion capabilities.

2 MOTIVATION

2.1 A Motivating Example

In Figure 1, we use an example to illustrate the motivation of our work. Figure 1a shows a function `render_to_response` that is randomly sampled from the Py150 dataset [17]. This function operates by selecting the appropriate rendering class according to request parameter values, utilizing that class to generate a response object, and ultimately returning the object. Figures 1b and 1c present the code completion results generated by CodeGPT using two distinct training strategies: SFT and DRL with delayed reward. These results are generated based on an incomplete code fragment, specifically the code spanning lines 1 to 11 as depicted in Figure 1a. From these examples, we can derive the following observations.

As shown in Figure 1b, it is clear to see that the code completed by CodeGPT based on SFT strategy deviates significantly from the reference code. In SFT, the model is fine-tuned using the "teacher-forcing" strategy. However, this approach introduces significant discrepancies between the training and inference phases, resulting in the emergence of the *exposure bias* issue during inference. To illustrate, when the anticipated output is `return`, the model may produce the `resp` object instead. Consequently, this initial error will propagate to subsequent completions, influencing the incorrect invocation of the `.write()` function, even its relevance to the current context is negligible.

In terms of the DRL-based strategy with delayed rewards, as shown in Figure 1c, CodeGPT, when trained using this approach, occasionally exhibits errors. While we can see a performance advancement compared to that achieved using the SFT strategy, and the model seemingly comprehends the intended action following the `return`, it still lacks complete accuracy. More specifically, in the

```
1  def render_to_response(self, context, **response_kwargs):
2      if self.request.GET.get('request_param', 'html') = 'html':
3          render_class = self.csv_response_class
4          response_kwargs.setdefault("file_name_param", "default.html")
5      else:
6          render_class = self.response_class
7          resp = render_class(request=self.request,
8                          template=self.get_template_names(),
9                          context=context,
10                         content_type=self.get_content_type(),
11                         **response_kwargs)
12     return resp     // Reference code.
```

(a) The source code.

```
1  // Code completed by CodeGPT (SFT).
2  resp.write('**response_kwargs')
```

(b) Code completed by CodeGPT (SFT).

```
1  // Code completed by CodeGPT (DRL with delayed rewards).
2  return render_class
```

(c) Code completed by CodeGPT (DRL w/ delayed rewards).

Fig. 1. The developer-written code, completed by CodeGPT trained by SFT, completed by CodeGPT trained by DRL w/ delayed rewards.

reference code, the render_class object is assigned to a variable named resp; nevertheless, the model fails to accurately capture this modification. The error could be ascribed to the limitations inherent in traditional DRL methods, which provide feedback to the model only after it completes the entire sequence. As a result, the model fails to allocate rewards during intermediate states, possibly leading to erroneous decisions, especially when feedback is delayed until the end of the generated sequence. Furthermore, the repetitive occurrence of the render_class in the subsequent code may divert the model's attention, causing it to overlook more optimal solutions.

2.2 Key Ideas

From the example above, we identify two essential features a proficient code completion model needs to possess, serving as the inspiration for our proposed approach: 1) the ability to mitigate error accumulation during code completion; 2) the capability to perceive the latest context changes. The key idea of our approach centers around the development of a DRL framework guided by immediate rewards, based on the actor-critic framework [21]. In this framework, we assign the role of the actor network to the code completion model, typically an LM, which undergoes fine-tuning through DRL, with the goal of alleviating the inherent *exposure bias* in SFT, thereby mitigating error accumulation during code completion. Concurrently, we develop and train a critic network to assess the quality of code completion. This critic network provides immediate rewards for each code token generated by the actor network, effectively converting the sparse feedback in DRL, resulting from delayed rewards, into dense feedback. This ensures the timely integration of the latest contextual information, thereby minimizing delays in learning. The code completion model is further refined during the policy gradient optimization phase inherent in DRL. Next, we will detail the fundamentals of this designed strategy.

The interaction between actor and critic networks can be analogized to the relationship between a student and a teacher. In this analogy, the actor assumes the role of the student, while the critic serves as the teacher. From the actor's perspective, the reward signals from the critic play a pivotal role in enhancing its ability to discern the ever-evolving context in code completion, enabling the actor to make prompt adjustments based on immediate feedback. Specifically, the actor feeds each completed token to the critic, which subsequently assigns a score to assess the quality of the token in the current context. This scoring mechanism provides valuable insights to the actor, aiding in the refinement of its token completion strategy.

From the critic's perspective, the task is to assign a reasonable reward to every token produced by the actor. Nevertheless, crafting effective immediate rewards for each generated code token poses a significant challenge. Previously, CodeRL [16] derived reward signals based on the code's ability to pass unit tests. Subsequently, the model quantifies the impact of each generated token on the overall outcome of the code's unit test results and reallocates the delayed reward across individual

tokens proportionally to their respective contributions. However, employing this approach in code completion is impractical for two main reasons: 1) The constraint arises from the frequent absence of test cases in datasets for code completion, rendering code completions incapable of deriving reward signals from unit tests. 2) There is currently no established and effective method for assessing the contribution of a specific portion of code within the dynamic context of code completion. The primary insight regarding the critic is that by furnishing a justifiable reward for each token generated by the actor, the actor can gain a heightened sensitivity to the dynamically evolving context throughout its training. This empowers the actor to assess the effectiveness of the generated code segment in shaping the final outcome. Adopting such an approach facilitates a more refined balance between exploration and exploitation, thereby enhancing the actor's performance.

2.3 Feasibility Analysis

A foundational premise of our approach is that the immediate reward given by the critic to each token generated by the actor serves as feedback, indicating if the token aids in the subsequent code completion. Acquiring these rewards poses a considerable challenge; however, in contrast to code generation, code completion involves shorter sequences and is not contingent upon the code's execution outcome. This distinction substantially alleviates the complexity and cost associated with calculating these immediate rewards. Considering the inherent features of code completion, we propose to utilize the widely adopted evaluation metrics (e.g., BLEU and Edit-Sim) to design methods for assessing immediate rewards. Specifically, we suggest assigning higher rewards to tokens if the completion results generated by the actor, based on the current token, exhibit closer alignment with the correct outcomes. This encourages the actor to promptly adjust its strategy, promoting the generation of tokens with an increased likelihood of yielding correct completions.

In addition, we examine the influence of this score on guiding the model towards generating accurate subsequent completion sequences. Experimental results demonstrate that our method surpasses those DRL methods with delayed rewards. On the Py150 dataset, our proposed immediate rewards demonstrate notable improvements across various metrics. Specifically, we observe average enhancements of 3.32 %, 4.56 %, 3.75 %, and 2.81 % in Edit-Sim, EM, BLEU-4, and CodeBLEU, respectively (see Table 3). These results affirm the efficacy of the designed immediate rewards, showcasing their simplicity and effectiveness in significantly enhancing the performance of pre-trained LMs in code completion tasks.

3 IRCOCO

The IRCoCo framework utilizes a combination of SFT and DRL to fine-tune the code completion model, aiming to improve accuracy and ensure both syntactic and semantic correctness of the generated code. Figure 2 shows an overview of the IRCoCo framework. The first step involves utilizing SFT method to fine-tune the pre-trained LM, which serves as the actor network for sampling synthetic samples. Subsequently, an evaluator model is trained as the critic network, responsible for evaluating the synthetic samples and returning reward scores. Finally, the LM (actor) is jointly optimized using SFT and DRL. In the following subsections, we will provide a detailed description of each component in the IRCoCo framework.

3.1 Code Completion Task

The goal of the code completion task studied in this paper is to predict the subsequent code fragments given a partial code context until the end-of-sequence special token </s> is generated. Specifically, given a partial code sequence $X = \{x_1, x_2, \ldots, x_k\}$, the task is to predict the following code fragments $Y = \{y_1, y_2, \ldots, y_n\}$ using an LM p for code completion. Formally, the code completion task can be
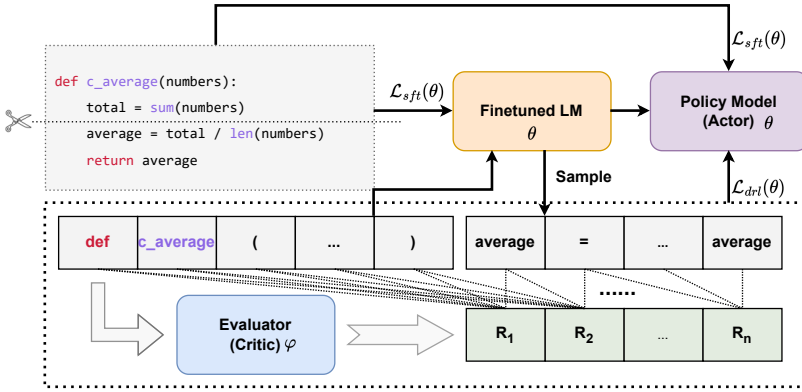
Fig. 2. **Overview of the IRCoCo using the Actor-Critic Framework.** First, the actor network samples synthetic samples. These samples are generated token by token and are sequentially added to the end of the incomplete code fragment. Afterward, they are rewarded by the critic. Leveraging these immediate rewards, the strategy is refined by integrating the IRCoCo framework, which employs a joint fine-tuning approach using SFT and DRL.

formulated as:

$$p\left(Y \mid X\right) = \prod_{t=1}^{n} p\left(y_t \mid y_{1:t-1}, X\right) . \tag{1}$$

3.2 Supervised Fine-Tuning-Based Model Training

Typically, code completion is modeled as a sequence-to-sequence task whose goal is to map the input sequence X to the output sequence $Y = \{y_1, y_2, \ldots, y_n\}$, where each token y_i is sampled from the vocabulary ($\mathcal{V}$) of code. During the training period, the code completion model aims to minimize the cross entropy between the generated code and the reference code, based on the following training loss:

$$\mathcal{L}_{sft}(\theta) = - \sum_t \log p_\theta(Y \mid X) = - \sum_t \log\left[p_\theta\left(y_t \mid y_{1:t-1}, X\right)\right] , \tag{2}$$

where y_t is the output of each decoding step t, and θ is the model parameter.

3.3 Quality Evaluator for Generated Code

In the IRCoCo framework, the evaluator assesses the quality of code completions given an incomplete code fragment, enabling code completion models to meticulously sense dynamically changing contextual requirements. Drawing inspiration from [22], we adopt the same Transformer-based GPT-2 model as the primary architecture for the evaluator, due to its pre-training on a large-scale corpus and demonstrated success in NL generation tasks. To enhance the efficiency and performance of the evaluator, we restrict the model parameters to 16 million and incorporate a linear head layer into the Transformer-based GPT-2 architecture. To evaluate the similarity between the generated code fragments and target code, we employ BLEU and Edit-Sim respectively as the optimization metric for the evaluator.

Our basic idea is to align the reward score distribution predicted by the evaluator with the discrepancy between the LM's completions and the reference code, as measured by the BLEU or Edit-Sim metrics. With such a training approach, the evaluator is capable of scoring any fragment of code. This score directly reveals the likelihood that, starting from the last token of the incomplete
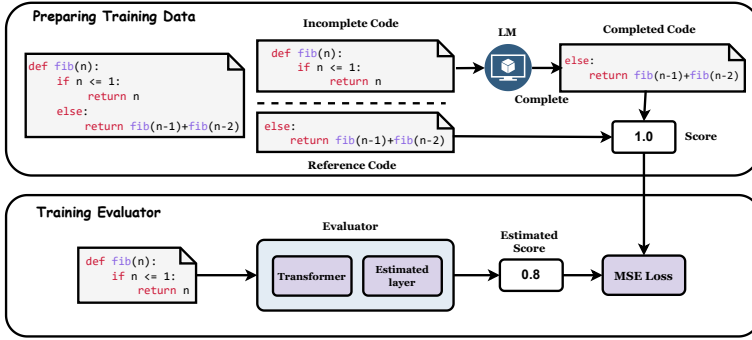
Fig. 3. **Overview of the Evaluator.** Training the evaluator first requires preparing training data. In the training data preparation phase, we randomly split the complete code to obtain the incomplete code and reference code fragments. After that, we pass the incomplete code fragment through the LM to obtain the completed code and compute the score *s*. Finally, we pair the incomplete code fragment with the score *s* to obtain the training data. In the training phase, we will obtain the score *s'* by the evaluator, and the training goal is to minimize the MSE loss of *s* and *s'*.

code fragment, the completion matches the reference code. A higher score indicates a greater expectation that the current token will lead to a correct completion. Unlike SFT, the optimization objective for each immediate reward focuses on the anticipated benefit from the entire completion, not just the local gain from the next token. With the guidance of this reward mechanism, the LM's prediction capabilities undergo continual refinement, aiming to bolster the likelihood of generating precise code. Figure 3 illustrates the training framework of the quality evaluator using BLEU as an optimization metric.

Preparing the Training Data. To train the quality evaluator, the first step involves obtaining training data. The effectiveness and performance of the evaluator are contingent on the quality and diversity of the training data. Thus, it is imperative to obtain representative code fragments from a large-scale codebase to use as training data. To obtain the necessary training data, we randomly divide a given complete code fragment C into two parts. The first portion is treated as an incomplete code fragment C_x, while the second part serves as the corresponding reference code fragment C_y. Next, the incomplete code fragment C_x is inputted into the fine-tuned LM, which generates the completed code fragment C_g. The generated code fragment C_g is then compared to the reference code fragment C_y, and the accuracy is calculated to obtain the score *s*. By following this process, we can obtain the datasets for training the evaluator of code completion quality after pairing each incomplete code fragment C_x with its corresponding score *s*.

Training the Evaluator. We utilize the standard GPT-2 architecture for our model. To process the training data (C_x, s), we employ a multi-headed attention mechanism in each Transformer block to enhance the linguistic representation by aggregating the previous output. Subsequently, we transform the output of the multiple attention heads into the final score *s'* using a linear layer:

$$h_0 = H_x W_{embedding} + W_{position}, \tag{3}$$

$$h_l = \text{TransformerBlock}\left(h_{l-1}\right), l \in [1, L], \tag{4}$$

$$s' = \text{Linear}\left(h_l\right), \tag{5}$$

where H_x is the context vector of the incomplete code fragment C_x; $W_{embedding}$ is the token embedding matrix, $W_{position}$ is the position embedding matrix, and h_0 is the hidden state vector representation of the incomplete code fragment C_x; L is the number of layers of the transformer

block, and h_l is the hidden state vector representation of the incomplete code fragment C_x in the L-th layer of the model; $Linear$ is the linear layer. s' is the score of the evaluator output.

The code completion quality evaluator minimizes the mean-square error between s' and s as the final training goal:

$$\text{MSE}(s, s') = \frac{1}{N} \sum_{i=1}^{N} (s'_i - s_i)^2 . \tag{6}$$

3.4 Reinforcement Learning-Based Alignment of Generated Code

To tackle the *exposure bias* issue, several studies [15, 23] have attempted to utilize DRL to train code completion models. Nevertheless, current DRL methods are prone to the delayed reward problem. The optimal policy may require multiple steps before attaining the maximum reward, leading to difficulties for the model to determine the optimal policy and a tendency to converge to local optimal solutions. For this reason, we propose to incorporate immediate rewards into DRL. Specifically, the code completion generation process is viewed as a Markov Decision Process (MDP) [24] consisting of four main components:

State. During each time step t of the decoding process, the state s_t consists of the incomplete code fragment X and the word $\hat{y}_{1:t-1}$ generated earlier in the decoding process, i.e., $s_t = \{X, \hat{y}_{1:t-1}\}$. In the initial state of decoding, the state s_t consists of only the incomplete code fragment X, i.e., $\{X\}$. We use the hidden state vector h_t as the vector representation under state s_t.

Action. In our scenario, the task involves predicting the subsequent code token by sampling a token ($\hat{y}_t$) from the vocabulary ($\mathcal{V}$) at each time step. We conceptualize the task of predicting the next token as the action of sampling a token from a predefined vocabulary (action space).

Reward. Rewards are used to evaluate whether the completed code facilitates the generation of subsequent completions. In this study, we aim to incentivize the code completion model to produce tokens that facilitate subsequent completions. To achieve this, we provide rewards to the model based on an evaluator (i.e., critic) that has been trained for this purpose. Therefore, we define the reward for each time step t as:

$$r(s_t, \hat{y}_t) = \begin{cases} Q_\varphi(X; \hat{y}_{1:t-1}, \hat{y}_t) & \text{if } \hat{y}_t \neq \text{</s>} \\ Q_\varphi(X; \hat{y}_{1:t-1}) & \text{if } \hat{y}_t = \text{</s>} \end{cases} \tag{7}$$

where Q_φ is the trained evaluator. In particular, in this work, we give a valid reward for each generated time step t of the code completion model and give the same reward for the final end-of-sequence special token </s> as for the previous token of the generation.

Policy. The policy function $p_\theta(\hat{y}_t|s_t)$ takes the current state s_t as input and outputs $\hat{y}_t$ as the probability of the next completion token. In this work, we use the policy gradient [21] method to optimize the policy function. The definition of the policy function $p_\theta(\hat{y}_t|s_t)$ is as follows:

$$p_\theta(\hat{y}_t|s_t) = p_\theta(\hat{y}_t|\hat{y}_{1:t-1}, X) = softmax(Kh_t + b) , \tag{8}$$

where K is the weight parameter of the model, b is the bias vector, and h_t is the hidden state at time step t.

The parameters of the code completion model θ can be thought of as stochastic strategies and the goal of model training is to find a policy network $p_\theta(\hat{Y}|X)$ to minimize negative expected returns:

$$\mathcal{L}_{drl}(\theta) = -\mathbb{E}_{\hat{Y} \sim p_\theta}\left[r(X, \hat{Y})\right] , \tag{9}$$

where $\hat{Y} = (\hat{y}_1, \ldots, \hat{y}_t)$ represents a sequence of synthetic samples, with each code token $\hat{y}_t$ being sampled by the code completion model at decoding time step t. According to the DRL algorithm and

---

Algorithm 1: The training process of IRCoCo

Input : A set of incomplete code and reference code pairs (X, Y), along with the pre-trained LM p.
Output: The parameters of the model after fine-tuning θ.

---

1 Fine-tuning of actor p_θ using Eq. (2);
2 Sampling synthetic samples $\hat{Y}$ using Eq. (8);
3 Training critic Q_φ using Eq. (6);
4 **for** *number of epochs until convergence* **do**
5   **for** $(x, y) \subset (X, Y)$ *and* $(x, \hat{y}) \subset (X, \hat{Y})$ **do**
6   **repeat**
7   // Calculate Reward
8   Compute $r(s_t, \hat{y}_t)$ using Eq. (7)
9   // Calculate Loss
10             $\mathcal{L}_{sft}(\theta) \leftarrow -\sum_t \log p_\theta(Y \mid X) = -\sum_t \log [p_\theta(y_t \mid y_{1:t-1}, X)]$
11             $\mathcal{L}_{drl}(\theta) \leftarrow -\mathbb{E}_{\hat{Y} \sim p_\theta}[r(X, \hat{Y})]$
12             $\mathcal{L}(\theta) \leftarrow \mathcal{L}_{sft}(\theta) + \mathcal{L}_{drl}(\theta)$
13   // Updata Model Parameters
14             $\theta \leftarrow \theta - \nabla_\theta \mathcal{L}(\theta)$
15   **until** *number of samples*;
16   **end**
17 **end**

---

policy gradient definition, we define the gradient $\nabla_\theta \mathcal{L}_{drl}(\theta)$ of the non-differentiable regression reward function r as:

$$\nabla_\theta \mathcal{L}_{drl}(\theta) \approx -\mathbb{E}_{\hat{Y} \sim p_\theta}\left[r(X, \hat{Y})\nabla_\theta \log p_\theta(\hat{Y} \mid X)\right]$$

$$\approx -\mathbb{E}_{\hat{Y} \sim p_\theta}\left[\sum_t r(X, \hat{Y})\nabla_\theta \log p_\theta(\hat{y}_t | \hat{y}_{1:t-1}, X)\right]. \tag{10}$$

During the training process, we adopt a hybrid learning method based on SFT-based fine-tuning and DRL-based alignment, setting this as our final training objective:

$$\mathcal{L}(\theta) = \mathcal{L}_{sft}(\theta) + \mathcal{L}_{drl}(\theta). \tag{11}$$

SFT and DRL-based alignment have distinct advantages, with SFT enabling supervised learning through large-scale data and DRL allowing autonomous learning through intelligent trial and error. By integrating these two learning methods, we can leverage their strengths to enhance the model's performance and generalization. The ultimate loss function is determined by summing the losses of SFT-based and DRL-based alignment.

Algorithm 1 presents the pseudo-code of training IRCoCo. For reinforcement learning data, we use the policy network p to sample one synthetic sample $(X, \hat{Y})$ for each source code pair (X, Y). Subsequently, we compute immediate reward based on the reward function defined in Section 3.4 to estimate the payoff. Finally, we update the gradients based on the corresponding loss functions.

4 EXPERIMENTAL SETUP

We have conducted several experiments to evaluate IRCoCo. Specifically, we seek to answer the following Research Questions (RQs).

Table 1. Statistical analysis of the two datasets.

Dataset	Examples	Average tokens of inputs	Average token of outputs	Average lines of code
Py150	50,000	96.9	9.06	11.6
Java Corpus	8,268	111.9	10.4	8.2

- **RQ1: Effectiveness of Code Completion.** To what extent can the training process of IRCoCo help improve the capabilities of code completion models? To answer this question, we compare the performance of pre-trained LMs before and after integration with IRCoCo.
- **RQ2: Validity of Immediate Rewards.** Are immediate rewards in the IRCoCo framework effective? To answer this question, we compare it with delayed reward-based DRL and several rule-based reward construction methods.
- **RQ3: Impact of Different Model Learning Objectives.** Are the learning objectives for joint training in IRCoCo effective? To answer this question, we compare IRCoCo with SFT-only and DRL-only training modes.
- **RQ4: Quantitative Analysis.** How does IRCoCo effective across varying token counts? To answer this question, we evaluate completions across different numbers of tokens.
- **RQ5: Qualitative Analysis.** How realistic is the predictive power of IRCoCo? To answer this question, we conduct a qualitative analysis study on IRCoCo.

4.1 Evaluation Datasets

In our experiments, we adopt the Py150 dataset and the Java Corpus dataset, both widely used to evaluate code completion tasks. The Py150 dataset [17] comprises 150,000 code files in Python 2, which are partitioned into a training set of 100,000 files and a test set of 50,000 files. The Java Corpus dataset [18] comprises almost 30,000 Java files, which are divided into a training set of 12,934 files and a test set of 8,268 files. Following the data preprocessing approach in the CodeXGLUE benchmark [25], we normalize the 200 most commonly used strings and the 30 most commonly used numeric characters by special tokens such as <STR LIT:utf-8> and <NUM LIT>. As the original Py150 and Java Corpus datasets consist of complete code snippets, and given our objective to perform line-level code completion, we randomly cut the code data to reflect the diversity in real-world application scenarios. In our work, we randomly divide a code fragment into two parts: the first half served as an incomplete code fragment, and the second half comprised 10 tokens that are designated as reference code. During DRL-based alignment, we organize the data in a format identical to that of the APPS program synthesis benchmark [26], whereby each incomplete code is paired with one reference code and one sampled example. Our aim is to train the model to complete the next 10 tokens. The statistics of the data are presented in Table 1.

4.2 Baselines

We utilize the following pre-trained LMs, which are widely used in code completion [1, 6, 25, 27, 28], as the underlying base models to evaluate their performance both prior to and following the integration of the IRCoCo framework.

- **GPT-2 [**124 M **&** 1.5 B**]:** GPT-2 [29] is a pre-trained LM that harnesses the Transformer architecture. It undergoes pre-training via large-scale unsupervised learning and demonstrates robust performance on generative tasks, such as question answering and code completion.
- **CodeGPT [**124 M**]:** CodeGPT [25] is a Transformer-based code completion model with the same architecture and training objectives as GPT-2. It comprises a 12-layer Transformer decoder and is pre-trained using the Python and Java corpus of the CodeSearchNet [30] datasets. Through this

pre-training, the model acquires a comprehensive understanding of code structure and syntax rules, enabling it to automatically generate code.

- **CodeGPT-adapt [**124 M**]:** CodeGPT-adapt [25] is a domain adaptive model, which is trained on the code corpus with the same vocabulary as GPT-2 as a starting point, and inherits the natural language understanding capability of GPT-2.
- **CodeGen [**350 M**]:** CodeGen [31] uses a standard Transformer-based autoregressive LM with a next-token prediction LM as a learning objective, trained on NL and PL datasets, and has demonstrated excellent performance in the field of program synthesis.
- **StarCoder [**164 M**]:** StarCoder [27] is based on the GPT architecture, which is obtained after training based on the licensed data on GitHub, and we use StarCoder [164M] in our experiments, which has the same architecture as StarCoder.
- **CodeT5+ [**220 M**]:** CodeT5+ [28] is an encoder-decoder-based masked language model, utilizing diverse training tasks and a simple yet effective pre-training method in its pre-training phase, offering enhanced support for program comprehension and code completion compared to CodeT5.

4.3 Evaluation Metrics

Following previous studies [5, 6, 25], we employ Edit Similarity (Edit-Sim) [32], BLEU-4 [19] similarity metric, CodeBLEU [20] metric, and Exact Match Accuracy (EM) [32] to evaluate IRCoCo.

4.4 Implementation Details

For the six baselines considered in our work, we follow existing works [25, 28, 33] that experiment on CodeXGLUE and fine-tune the pre-trained LM in Huggingface [34]. Subsequently, these fine-tuned models undergo the DRL-based alignment process. The experiments involving GPT-2 [1.5 B] are conducted using an Nvidia GeForce RTX A6000 GPU with 48 GB memory. Other experiments are performed on a server with two Nvidia GeForce RTX 3090 GPUs with 22 GB memory.

The process of SFT is elaborated as follows. For the decoder-only model, both the input and output consist of complete code fragments. In contrast, for the encoder-decoder model, the input comprises incomplete code fragments, while the output represents the corresponding code to be completed. Specifically, at each time step t, a "teacher-forcing" strategy is used and the next correct code token is generated based on the first $t - 1$ tokens in the reference code.

For the code completion quality evaluator (i.e., critic network), we train two evaluators using the BLEU and Edit-Sim indicators respectively to provide rewards for the code completion model. Specifically, we use the Transformer-based GPT-2 model, setting the number of layers to 4, the number of heads to 4, the embedding size of code token to 256, and the epochs to 30.

In terms of DRL, we update the parameters of the code completion model (i.e., actor network) once for each batch. The hyperparameters employed during DRL-based alignment are consistent with those used in the SFT-based process. Specifically, we have defined the experimental parameters as follows: a batch size of 2, a learning rate set at $2 \times e^{-5}$, and a total of 10 epochs.

5 EXPERIMENTAL RESULTS

5.1 Effectiveness of Code Completion (RQ1)

Table 2 presents the performance of IRCoCo with various base pre-trained LMs on two evaluation datasets. It is evident that IRCoCo significantly enhances code completion performance, regardless of whether the evaluator is trained with BLEU or Edit-Sim metrics. For instance, in the Py150 dataset, there is an increase of approximately 7 % in Edit-Sim scores, around 40 % in EM scores, around 6 % in CodeBLEU scores, and close to 10 % in BLEU-4 scores. More specifically, when CodeGPT undergoes training using the code completion quality evaluator guided by the BLEU metric, its

Table 2. Performance of IRCoCo with various base pre-trained LMs on two evaluation datasets (in %).

Model	BLEU (Evaluator)								Edit-Sim (Evaluator)							
	Py150				Java Corpus				Py150				Java Corpus			
	Edit-Sim	EM	BLEU-4	CodeBLEU	Edit-Sim	EM	BLEU-4	CodeBLEU	Edit-Sim	EM	BLEU-4	CodeBLEU	Edit-Sim	EM	BLEU-4	CodeBLEU
GPT-2 [124M]	59.37	9.21	35.31	40.81	57.42	4.95	32.56	42.25	59.37	9.21	35.31	40.81	57.42	4.95	32.56	42.25
GPT-2 [124M]+IRCoCo	63.65	13.93	39.96	43.70	58.58	5.40	33.12	43.68	63.87	14.12	40.87	43.83	58.37	5.33	33.06	43.86
Relative Improvement	(↑7.2%)	(↑51.2%)	(↑13.2%)	(↑7.1%)	(↑2.0%)	(↑9.1%)	(↑1.7%)	(↑3.4%)	(↑7.6%)	(↑53.3%)	(↑15.7%)	(↑6.2%)	(↑1.7%)	(↑7.7%)	(↑1.5%)	(↑3.8%)
GPT-2 [1.5B]	65.62	14.45	41.93	44.76	58.14	6.24	34.24	43.47	65.62	14.45	41.93	44.76	58.14	6.24	34.24	43.47
GPT-2 [1.5B]+IRCoCo	66.90	16.37	43.32	45.91	59.51	6.69	35.02	43.98	66.50	16.26	43.25	45.81	59.26	6.53	34.87	43.76
Relative Improvement	(↑2.0%)	(↑13.3%)	(↑3.3%)	(↑2.6%)	(↑2.4%)	(↑7.2%)	(↑2.3%)	(↑1.2%)	(↑1.3%)	(↑12.5%)	(↑3.1%)	(↑2.3%)	(↑1.9%)	(↑4.6%)	(↑1.8%)	(↑0.6%)
CodeGPT	60.66	15.65	38.10	42.42	58.98	12.06	36.32	43.92	60.66	15.65	38.10	42.42	58.98	12.06	36.32	43.92
CodeGPT+IRCoCo	65.44	21.94	43.71	45.79	59.94	12.57	36.89	44.19	65.93	22.32	43.96	45.53	59.91	12.77	36.94	44.32
Relative Improvement	(↑7.9%)	(↑40.2%)	(↑14.7%)	(↑7.9%)	(↑1.6%)	(↑4.2%)	(↑1.6%)	(↑0.6%)	(↑8.7%)	(↑42.6%)	(↑15.4%)	(↑7.3%)	(↑1.6%)	(↑5.9%)	(↑1.7%)	(↑0.9%)
CodeGPT-adapt	63.08	13.10	39.31	43.27	58.87	5.57	33.53	42.97	63.08	13.10	39.31	43.27	58.87	5.57	33.53	42.97
CodeGPT-adapt+IRCoCo	63.68	14.02	40.15	43.63	59.15	5.84	33.68	44.02	64.05	14.32	40.86	43.96	59.32	6.05	33.98	43.75
Relative Improvement	(↑1.0%)	(↑7.0%)	(↑2.1%)	(↑0.8%)	(↑0.5%)	(↑4.8%)	(↑0.4%)	(↑2.4%)	(↑1.5%)	(↑9.3%)	(↑4.0%)	(↑1.6%)	(↑0.8%)	(↑8.6%)	(↑1.3%)	(↑1.8%)
CodeGen	59.45	10.18	35.53	40.90	59.33	11.52	35.96	43.08	59.45	10.18	35.53	40.90	59.33	11.52	35.96	43.08
CodeGen+IRCoCo	64.55	14.46	41.31	43.08	60.31	12.40	36.99	44.06	64.03	14.21	40.97	42.86	60.17	12.22	36.87	43.81
Relative Improvement	(↑8.6%)	(↑42.0%)	(↑16.3%)	(↑5.3%)	(↑1.7%)	(↑7.6%)	(↑2.9%)	(↑2.3%)	(↑7.7%)	(↑39.6%)	(↑15.3%)	(↑4.8%)	(↑1.4%)	(↑6.1%)	(↑2.5%)	(↑1.7%)
StarCoder	61.37	16.44	39.11	43.11	59.94	11.88	36.24	43.56	61.37	16.44	39.11	43.11	59.94	11.88	36.24	43.56
StarCoder+IRCoCo	64.02	20.38	42.06	44.76	61.63	13.15	37.96	44.38	64.37	20.73	42.32	45.49	61.08	12.96	37.41	44.71
Relative Improvement	(↑4.3%)	(↑24.0%)	(↑7.5%)	(↑3.8%)	(↑2.8%)	(↑10.7%)	(↑4.8%)	(↑1.8%)	(↑4.9%)	(↑26.1%)	(↑8.2%)	(↑5.5%)	(↑1.9%)	(↑9.1%)	(↑3.3%)	(↑2.6%)
CodeT5+	55.81	5.33	32.99	38.71	53.42	4.61	31.49	36.85	55.81	5.33	32.99	38.71	53.42	4.61	31.49	36.85
CodeT5+ +IRCoCo	56.97	6.25	33.74	39.96	54.11	4.98	32.06	38.61	57.33	6.46	33.98	39.21	54.58	5.13	32.34	39.86
Relative Improvement	(↑2.1%)	(↑17.3%)	(↑2.3%)	(↑3.2%)	(↑1.3%)	(↑8.0%)	(↑1.8%)	(↑4.8%)	(↑2.7%)	(↑21.2%)	(↑3.0%)	(↑1.3%)	(↑2.2%)	(↑11.3%)	(↑2.7%)	(↑8.2%)

Edit-Sim score rises from 60.66 % to 65.44 %, marking an increment of 7.9 %. Concurrently, its EM score ascends from 15.65 % to 21.94 %, a notable surge of 40.2 %, while the BLEU-4 score elevates from 38.10 % to 43.71 %, a rise of 14.7 %. Additionally, the CodeBLEU metric shows a significant improvement, climbing from 42.42 % to 45.79 %, an increase of 7.9 %. In the Java Corpus dataset, we observe enhancements of about 2 % in Edit-Sim, roughly 8 % in EM scores, and nearly 2 % in BLEU-4 scores. Furthermore, there is an approximate increase of 2.3 % in the CodeBLEU scores. It is worth noting that metrics recorded on Java dataset are generally inferior to those on Python dataset. This disparity can be attributed to the lengthier nature of Java code; on average, Java code snippets comprise 112 tokens, in contrast to Python's 97 tokens.

One surprising finding is that CodeGPT-adapt without adding the IRCoCo framework outperforms CodeGPT. However, after adding the IRCoCo framework, the metrics of CodeGPT-adapt+IRCoCo are lower than that of CodeGPT+IRCoCo. This is attributed to the inherent properties of the pre-trained LM itself. CodeGPT-adapt is a domain adaptive model, and the primary objective of domain adaptive models is to transfer learning between different data distributions and are highly sensitive to changes in the data distribution. On the other hand, SFT and DRL-based alignment have different objectives, which may result in insignificant performance improvements for the model. Overall, incorporating the pre-trained LM into the IRCoCo framework generally improves its performance in code generation. This suggests that utilizing the immediate rewards provided by the IRCoCo framework enables the model to acquire better strategies.

> **Answer to RQ1:** Our results indicate that incorporating the pre-trained LM into the IRCoCo framework generally improves its performance in code generation. This suggests that utilizing the immediate rewards provided by the IRCoCo framework enables the model to acquire better strategies, leading to improved effectiveness.

5.2 Validity of Immediate Rewards (RQ2)

To evaluate the effectiveness of the immediate rewards, we investigate three different reward shaping strategies. For delayed rewards (DR), we allocate rewards exclusively at the end of code completion (i.e., after the generation of the end-of-sequence token </s>), assigning 0 to intermediate

Table 3. Comparative results of our proposed method with delayed rewards (DR), linearly attenuating (LA) rewards, and 0-1 (0-1) based rewards.

Model	BLEU (Evaluator)								Edit-Sim (Evaluator)							
	Py150				Java Corpus				Py150				Java Corpus			
	Edit-Sim	EM	BLEU-4	CodeBLEU	Edit-Sim	EM	BLEU-4	CodeBLEU	Edit-Sim	EM	BLEU-4	CodeBLEU	Edit-Sim	EM	BLEU-4	CodeBLEU
GPT-2 [124M] (DR)	57.05	9.05	35.55	40.57	56.92	5.27	32.98	43.02	57.65	9.43	35.81	40.68	57.82	5.21	32.90	42.74
GPT-2 [124M] (LA)	62.57	13.15	39.22	43.62	57.59	5.29	32.30	43.11	62.57	13.15	39.22	43.62	57.59	5.29	32.30	43.11
GPT-2 [124M] (0-1)	58.34	8.14	33.33	40.09	57.11	4.46	32.38	42.93	58.34	8.14	33.33	40.09	57.11	4.46	32.38	42.93
GPT-2 [124M]+IRCoCo	63.65	13.93	39.96	43.70	58.58	5.40	33.12	43.68	63.87	14.12	40.87	43.83	58.37	5.33	33.06	43.86
GPT-2 [1.5B] (DR)	65.11	14.26	41.88	44.64	57.62	5.97	33.96	43.18	65.02	14.13	41.79	44.42	57.47	5.91	33.84	43.06
GPT-2 [1.5B] (LA)	65.68	15.37	42.08	45.18	58.46	6.37	34.44	43.62	65.68	15.37	42.08	45.18	58.46	6.37	34.44	43.62
GPT-2 [1.5B] (0-1)	66.16	15.61	42.33	45.30	58.17	6.21	34.28	43.53	66.16	15.61	42.33	45.30	58.17	6.21	34.28	43.53
GPT-2 [1.5B]+IRCoCo	66.90	16.37	43.32	45.91	59.51	6.69	35.02	43.98	66.50	16.26	43.25	45.81	59.26	6.53	34.87	43.76
CodeGPT (DR)	62.12	17.38	39.92	42.98	52.07	7.42	34.29	42.16	62.30	17.52	40.08	43.14	53.26	8.05	34.93	42.89
CodeGPT (LA)	64.45	21.03	42.84	45.03	59.18	11.72	36.01	44.02	64.45	21.03	42.84	45.03	59.18	11.72	36.01	44.02
CodeGPT (0-1)	63.18	17.11	38.94	42.60	58.46	11.28	35.34	43.77	63.18	17.11	38.94	42.60	58.46	11.28	35.34	43.77
CodeGPT+IRCoCo	65.44	21.94	43.71	45.79	59.94	12.57	36.89	44.19	65.93	22.32	43.96	45.53	59.91	12.77	36.94	44.32
CodeGPT-adapt (DR)	62.78	13.14	39.28	43.09	58.40	5.53	33.32	43.37	63.46	13.48	39.60	43.33	58.24	5.41	33.28	43.12
CodeGPT-adapt (LA)	63.34	13.60	39.55	42.94	58.50	5.26	32.94	43.37	63.34	13.60	39.55	42.94	58.50	5.26	32.94	43.37
CodeGPT-adapt (0-1)	62.91	11.91	38.89	42.15	58.61	5.03	32.46	42.75	62.91	11.91	38.89	42.15	58.61	5.03	32.46	42.75
CodeGPT-adapt+IRCoCo	63.68	14.02	40.15	43.63	59.15	5.84	33.68	44.02	64.05	14.32	40.86	43.96	59.32	6.05	33.98	43.75
CodeGen (DR)	60.33	10.71	36.24	40.61	56.84	9.61	34.71	41.15	60.06	10.55	36.09	40.37	57.22	9.88	34.89	41.83
CodeGen (LA)	62.46	12.89	38.41	41.32	58.92	11.10	35.34	42.66	62.46	12.89	38.41	41.32	58.92	11.10	35.34	42.66
CodeGen (0-1)	61.44	11.26	37.18	41.43	59.16	11.86	36.03	42.74	61.44	11.26	37.18	41.43	59.16	11.86	36.03	42.74
CodeGen+IRCoCo	64.55	14.46	41.31	43.08	60.31	12.40	36.99	44.06	64.03	14.21	40.97	42.86	60.17	12.22	36.87	43.81
StarCoder (DR)	61.51	16.56	39.23	42.78	57.56	10.08	35.17	42.63	61.86	16.85	39.55	42.91	56.44	9.53	34.61	42.01
StarCoder (LA)	62.77	17.01	39.86	44.16	60.44	12.23	36.89	43.85	62.77	17.01	39.86	44.16	60.44	12.23	36.89	43.85
StarCoder (0-1)	61.96	15.48	38.77	43.45	60.81	11.67	36.02	44.02	61.96	15.48	38.77	43.45	60.81	11.67	36.02	44.02
StarCoder+IRCoCo	64.02	20.38	42.06	44.76	61.63	13.15	37.96	44.38	64.37	20.73	42.32	45.49	61.08	12.96	37.41	44.71
CodeT5+(DR)	55.42	5.11	32.69	38.14	53.51	4.20	31.38	36.47	55.16	5.22	32.52	38.26	53.71	4.58	31.30	36.05
CodeT5+(LA)	55.13	5.01	31.77	38.23	53.29	4.44	31.08	37.17	55.13	5.01	31.77	38.23	53.29	4.44	31.08	37.17
CodeT5+ (0-1)	54.36	4.28	30.24	37.66	52.66	3.23	30.26	35.46	54.36	4.28	30.24	37.66	52.66	3.23	30.26	35.46
CodeT5+ +IRCoCo	56.97	6.25	33.74	39.96	54.11	4.98	32.06	38.61	57.33	6.46	33.98	39.21	54.58	5.13	32.34	39.86

tokens. For immediate rewards, our first approach employs a linear attenuation (LA) based rule [16], which decays rewards based on token position, ranging from time steps $t = 1$ to $t = T$. The second strategy involves a binary (0-1) reward system, where we evaluate the completion status of each token. If the generated token is exactly aligned with the reference code token, a reward of 1 is awarded, otherwise a value of 0 is assigned. The experimental results are shown in Table 3.

From the table, it is evident that IRCoCo consistently surpasses the three reward design methods across all metrics. For instance, in the Py150 dataset, when employing CodeGPT as the code completion model and a quality evaluator trained by BLEU, IRCoCo's performance exceeds that of DR by approximately 3.5 % in Edit-Sim, 4.5 % in EM, 3 % in CodeBLEU, and 4 % in BLEU-4. This indicates that, in contrast to the delayed reward-based approach, IRCoCo can adeptly discern dynamically shifting contextual demands. Furthermore, compared to the other two immediate reward configurations, IRCoCo exhibits enhancements across multiple metrics. This underscores the potential of rewards derived from our code completion quality evaluator in motivating the model to generate superior subsequent completion sequences.

> **Answer to RQ2:** Our results indicate that IRCoCo can adeptly discern dynamically shifting contextual demands. Furthermore, compared to the other two immediate reward configurations, IRCoCo exhibits enhancements across multiple metrics. This underscores the validity of immediate rewards.

5.3 Impact of Different Model Learning Objectives (RQ3)

In code completion, SFT primarily aims to maximize the log-likelihood of the next correct code. By contrast, DRL seeks to maximize the reward signal by utilizing a policy-based approach. Due to the

Table 4. **Results with different learning objectives.** 'LM' indicates that LM has completed 5 rounds of SFT. '$LM + \mathcal{L}_{sft}$' is an additional 5 rounds of SFT based on 'LM'; '$LM + \mathcal{L}_{drl}$' is 10 rounds of DRL-based alignment based on 'LM'; '$LM + \mathcal{L}_{sft} + \mathcal{L}_{drl}$' is a 10-round joint training of SFT and DRL-based alignment based on 'LM'.

Py150																
Model	LM				$LM + \mathcal{L}_{sft}$				$LM + \mathcal{L}_{drl}$				$LM + \mathcal{L}_{sft} + \mathcal{L}_{drl}$			
	Edit-Sim	EM	BLEU-4	CodeBLEU	Edit-Sim	EM	BLEU-4	CodeBLEU	Edit-Sim	EM	BLEU-4	CodeBLEU	Edit-Sim	EM	BLEU-4	CodeBLEU
GPT-2 [124M]	59.37	9.21	35.31	40.81	59.48	9.00	35.37	40.69	58.15	8.86	34.98	40.26	**63.65**	**13.93**	**39.96**	**43.70**
GPT-2 [1.5B]	65.62	14.45	41.93	44.76	62.36	12.23	39.45	42.82	63.74	12.51	39.87	44.06	**66.90**	**16.37**	**43.32**	**45.91**
CodeGPT	60.66	15.65	38.10	42.42	58.47	12.57	35.30	41.17	58.72	12.72	38.82	43.10	**65.44**	**21.94**	**43.71**	**45.79**
CodeGPT-adapt	63.08	13.10	39.31	43.27	60.84	10.75	36.67	41.73	61.51	10.96	37.30	42.34	**63.68**	**14.02**	**40.15**	**43.63**
CodeGen	59.45	10.18	35.53	40.90	58.33	9.52	34.88	40.22	58.82	9.81	35.16	41.43	**64.55**	**14.46**	**41.31**	**43.08**
StarCoder	61.37	16.44	39.11	43.11	59.42	11.41	35.96	41.77	60.09	11.82	36.26	42.43	**64.02**	**20.38**	**42.06**	**44.76**
CodeT5+	55.81	5.33	32.99	38.71	54.62	5.10	32.31	38.34	54.84	5.01	32.24	38.92	**56.97**	**6.25**	**33.74**	**39.96**
Java Corpus																
Model	LM				$LM + \mathcal{L}_{sft}$				$LM + \mathcal{L}_{drl}$				$LM + \mathcal{L}_{sft} + \mathcal{L}_{drl}$			
	Edit-Sim	EM	BLEU-4	CodeBLEU	Edit-Sim	EM	BLEU-4	CodeBLEU	Edit-Sim	EM	BLEU-4	CodeBLEU	Edit-Sim	EM	BLEU-4	CodeBLEU
GPT-2 [124M]	57.42	4.95	32.56	42.25	56.33	4.49	31.90	42.03	52.68	4.63	32.55	42.55	**58.58**	**5.40**	**33.12**	**43.68**
GPT-2 [1.5B]	58.14	6.24	34.24	43.47	57.27	5.86	34.00	42.71	56.89	5.47	33.88	42.35	**59.51**	**6.69**	**35.02**	**43.98**
CodeGPT	58.98	12.06	36.32	43.92	57.55	11.35	35.25	43.23	55.98	10.56	34.71	42.97	**59.94**	**12.57**	**36.89**	**44.19**
GodeGPT-adapt	58.87	5.57	33.53	42.97	57.24	5.17	32.88	42.66	55.38	4.80	30.88	41.78	**59.15**	**5.84**	**33.68**	**44.02**
CodeGen	59.33	11.52	35.96	43.08	58.66	11.03	35.24	42.71	59.14	11.41	35.66	43.29	**60.31**	**12.40**	**36.99**	**44.06**
StarCoder	59.94	11.88	36.24	43.56	59.18	11.03	35.75	43.22	58.68	10.71	35.18	42.91	**61.63**	**13.15**	**37.96**	**44.38**
CodeT5+	53.42	4.61	31.49	36.85	53.17	4.46	31.05	35.73	52.76	4.09	30.88	35.02	**54.11**	**4.98**	**32.06**	**38.61**

different learning goals, we conduct experiments using various combinations of $\mathcal{L}_{sft}$ and $\mathcal{L}_{drl}$ for the model, and the experimental results are shown in Table 4. Notably, the pre-trained LM in the table has been fine-tuned for 5 epochs using SFT (as indicated in the 'LM' column).

As shown in Table 4, when the model is further fine-tuned using only $\mathcal{L}_{sft}$, the loss of the model is further reduced during training. However, the performance of the model on the test set decreases, which is due to the overfitting of the model during training. Secondly, when conducting experiments solely with $\mathcal{L}_{drl}$, we encounter the issue of vanishing gradients during fine-tuning. This phenomenon aligns with the observations in [14, 16, 21]. As a result, the performance of the model eventually decreases. However, the performance of the model is further improved when the model is fine-tuned using a combination of $\mathcal{L}_{sft}$ and $\mathcal{L}_{drl}$. First and foremost, SFT concentrates on deciphering the inherent patterns and structures within data, predominantly utilizing vast quantities of labeled datasets. In contrast, DRL is designed to adapt through interactions with its environment, with the goal of optimizing a set reward metric. By merging these two approaches, the model is equipped to navigate dynamic contexts while also capturing the inherent patterns within the data. Hence, when relying exclusively on SFT, the model typically learns based on the loss sourced from labeled data. However, when integrated with DRL, the model draws insights from a more comprehensive feedback system, encompassing aspects such as reward signals, which could lead to improved performance.

> **Answer to RQ3:** The experimental results show that SFT has a positive facilitating effect on the training of DRL, and the hybrid training strategy of SFT and DRL can alleviate the gradient vanishing problem encountered during the training of DRL, thus effectively improving the performance of the pre-trained LM.

5.4 Quantitative Analysis (RQ4)

We analyze the model's completion performance for different numbers of tokens on the Py150 and Java Corpus datasets. Given that we employ BLEU-4, we do not report the BLEU metric for
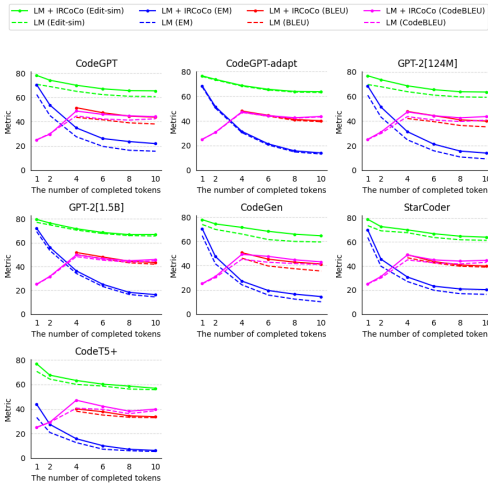
Fig. 4. Comparison of the IRCoCo framework under different numbers of tokens (Py150 dataset).
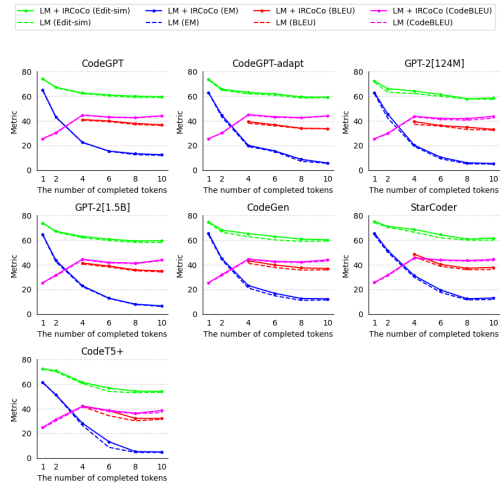
Fig. 5. Comparison of the IRCoCo framework under different numbers of tokens (Java Corpus dataset).

token counts less than 4 in our experiments. The results of these experiments are illustrated in Figures 4 and 5. These figures clearly demonstrate that IRCoCo outperforms the pre-trained LMs across different code completion lengths for both the Py150 and the Java Corpus. Particularly notable is IRCoCo's consistent superiority over the pre-trained LMs in cases involving longer completion lengths. This finding suggests that the immediate rewards mechanism enables the model to effectively incorporate the already completed information into its predictions. Furthermore, compared to Python, the improvement brought by IRCoCo when applied to Java has decreased. This limited enhancement can be attributed to Java's exhaustive type system and its intricate syntax, which gives rise to complex coding patterns. Conversely, Python tends to be more streamlined. Given this degree of complexity, DRL necessitates a larger volume of data for effective adaptation and learning. Unfortunately, the Py150 dataset is about seven times the size of the Java Corpus dataset, and this discrepancy suggests that there are fewer samples of reinforcement learning data for training in the Java language compared to the Python language, leading to IRCoCo's inferior performance on the Java Corpus compared to its performance on Py150.

> **Answer to RQ4:** The experimental results show that the pre-trained LM after integrating the IRCoCo framework outperforms the pre-trained LM for different code complement lengths, suggesting that the immediate reward mechanism enables the model to efficiently incorporate already completed information into its predictions.

5.5 Qualitative Analysis (RQ5)

Although statistical metrics offer valuable insights, they might not fully reflect the model's predictive capabilities. Thus, we conduct a qualitative assessment of the code generated by IRCoCo. For this analysis, we adopt CodeGPT as the base model and supplement it with diverse techniques to provide a more detailed comparison. Figures 6a and 6b display cases of successful code completions, while Figures 6c and 6d highlight instances where the model failed to generate the desired code.

From our observations, it is evident that when augmented with IRCoCo, CodeGPT exhibits enhanced accuracy in completing incomplete code fragments. In the context of the Py150 dataset,

```
1  // Uncompleted Code (Py150):
2  from base import db, Base
3  from cluster import Cluster
4  Class Proxy (Base):
5      __tablename__ = '<STR_LIT>'
6      host = db.Column (db.String (<NUM_LIT:255>), nullable = False)
7      port = db.Column (db.Integer, nullable = False)
8      eru_container_id = db.Column (db.String (<NUM_LIT:64>), index = True)
9      cluster_id = db.Column (db.ForeignKey (Cluster.id), index  ----->>>
10 --------------------------------------------Completed code--------------
11 // CodeGPT: ))) <EOL> def __init__(self
12 -----------------------------------------------------------------------
13 // CodeGPT (Delayed reward): = False ) <EOL> host = db.
14 -----------------------------------------------------------------------
15 // CodeGPT (Linear attenuation): = True ) <EOL> proxy_id = db.
16 -----------------------------------------------------------------------
17 // CodeGPT+IRCoCo : = True ) <EOL> suppress_alert = db.
18 -----------------------------------------------------------------------
19 // Reference: = True ) <EOL> suppress_alert = db.
```

(a) Correct completions (Python)

```
1  // Uncompleted Code (Java Corpus):
2  package com.asakusafw.windgate.hadoopfs;
3  import java.text.MessageFormat;
4  import java.util.ResourceBundle;
5  import com.asakusafw.windgate.core.WindGateLogger;
6  public class HadoopFsLogger ----->>>
7  --------------------------------------------Completed code--------------
8  // CodeGPT: extends WindGateLogger { private static final Logger
9  -----------------------------------------------------------------------
10 // CodeGPT (Delayed reward): extends WindGateLogger { private static final
                                        MessageFormat
11 -----------------------------------------------------------------------
12 // CodeGPT (Linear attenuation): extends WindGateLogger { private static
                                        final String; }
13 -----------------------------------------------------------------------
14 // CodeGPT+IRCoCo : extends WindGateLogger { private static final
                                    ResourceBundle; }
15 -----------------------------------------------------------------------
16 // Reference: extends WindGateLogger { private static final ResourceBundle; }
```

(b) Correct completions (Java)

```
1  // Uncompleted Code (Py150):
2  import numpy as np
3  from brainstorm.describable import Describable
4  Class Scorer(Describable):
5      (more...)
6      def aggregate (errors):
7          errors = np.array (errors)
8          assert errors.ndim == <NUM_LIT:2> and errors.shape [ <NUM_LIT:1> ]
           == <NUM_LIT:2>  ----->>>
9  --------------------------------------------Completed code--------------
10 // CodeGPT: <EOL> scores = Scorer() <EOL> scores.
11 -----------------------------------------------------------------------
12 // CodeGPT (Delayed reward): <EOL> scores = Score()
13 -----------------------------------------------------------------------
14 // CodeGPT (Linear attenuation): <EOL> scores = np.array(errors)
15 -----------------------------------------------------------------------
16 // CodeGPT+IRCoCo : <EOL> scores = np.zeros((errors.
17 -----------------------------------------------------------------------
18 // Reference: <EOL> return np.sum(errors[:,
```

(c) Error completions (Python)

```
1  // Uncompleted Code (Java Corpus):
2  package com.asakusafw.modelgen.emitter;
3  import java.io.DataInput;
4  import java.io.DataOutput;
5  (more...)
6  import com.asakusafw.utils.java. model.syntax.Annotation;
7  import com.asakusafw.utils.java. model.syntax.Attribute;
8  import com.asakusafw.utils.java. model.syntax.Block;
9  import com.asakusafw.utils.java  ----->>>
10 --------------------------------------------Completed code--------------
11 // CodeGPT: .util.Arrays; import com.asak
12 -----------------------------------------------------------------------
13 // CodeGPT (Delayed reward): .util.List; import com.asak
14 -----------------------------------------------------------------------
15 // CodeGPT (Linear attenuation): .model.syntax.Annotation; import com.
16 -----------------------------------------------------------------------
17 // CodeGPT+IRCoCo: .model.syntax.Type; import com.
18 -----------------------------------------------------------------------
19 // Reference: .model.syntax.Expression; import com.
```

(d) Error completions (Java)

Fig. 6. Examples of CodeGPT completion on Py150 and Java Corpus datasets.

the predictions of CodeGPT were notably inaccurate. When introduced to delayed rewards, the model generated code snippets that, although closer to the reference, were not exact matches. In experiments that utilize LA rewards, we find that the model began to complete `= True )` `<EOL>`, which shows that the model has developed towards the correct method, but the subsequent variable name `proxy_id` is wrong, indicating that rule-based immediate rewards are still not effective enough. Interestingly, the inclusion of IRCoCo's immediate rewards leads to perfect code completion, suggesting the significant role our novel immediate rewards play in guiding the model toward the correct strategy. When using the Java Corpus dataset, we find that the baseline CodeGPT does not perform quite well, its suggestions ceased after the *Logger* prompt. The completions with delayed and LA rewards both fall short of the reference by one keyword. In contrast, when using the immediate rewards from IRCoCo, the model completes the code correctly. This indicates that our immediate reward mechanism allows the model to adjust its strategy during exploration based on the anticipated benefits of future completions, thus enhancing the accuracy of code completion. In contrast, other immediate reward mechanisms, such as the LA reward scheme with a fixed decay coefficient, cannot accurately perceive the expected returns of future correct completions. Moreover, the coarse-grained (0-1) reward scheme struggles to capture the fine-grained expected returns in code completion tasks. Contrastingly, as evident from Figures 6c and 6d, even when our proposed methodology is not entirely accurate, the results are close enough to the reference that developers may find that they can be used with only minor modifications.

These observations underline the effectiveness of incorporating IRCoCo's immediate rewards into DRL, steering the model to discern the optimal strategy. This is largely because our immediate rewards guide the model away from incorrect intermediate solutions and towards best practices

during training. Additionally, the combined power of SFT and DRL-based alignment, when trained jointly, leverages the strengths of both paradigms, leading to high-quality code.

> **Answer to RQ5:** The results show that IRCoCo's overall predictions of the model are in the right direction in both datasets, although complementary errors still occur, suggesting that the immediate rewards gradually steer the model away from the wrong solution and towards the right strategy.

6 DISCUSSION

Why use the current selection of these metrics and why not pass@k? In code generation, generating complete and highly accurate code poses significant challenges. Nevertheless, code completion offers a viable approach that does not require the model to achieve a 100 % completion rate for correct code. In most cases, the objective is to generate code fragments that are as accurate or similar as possible, allowing programmers to utilize them with minor modifications. To evaluate the code completion model, we have selected Edit-Sim and BLEU as similarity metrics, along with the accuracy metric to demonstrate the model's performance. As the current code completion datasets lack unit test cases, we have not opted to employ pass@k [35] as the evaluation metric.

Why some relevant models (e.g., CodeRL) cannot do the line-level code completion we are concerned with? We opted not to draw comparisons with CodeRL [16] due to several factors. Initially, code completion differs fundamentally from code generation. Our study emphasizes line-level code completion, while CodeRL is geared towards generating code from NL descriptions to produce code sequences. Next, the reward mechanism employed by CodeRL does not readily apply to code completion. CodeRL deploys a critic model using an encoder-decoder architecture that provides dense reward signals to the rendered complete code. Under this framework, the encoder receives an NL description, and the decoder outputs complete code that fulfills the functional requirements. In the realm of code generation, the NL description remains unchanged. Conversely, in code completion, the "context", which is the code awaiting completion, evolves with each completed token. Furthermore, CodeRL's rewards are based on unit tests. However, since the code completion task does not usually result in creating complete functional code, there are no unit test cases in the datasets. When inspecting code generation datasets endowed with unit tests, such as APPS [26], it is evident that the code therein addresses programming contests and is not tailored for code completion objectives.

7 THREATS TO VALIDITY

We have identified the following two threats to the validity.

Limitations of the Evaluation Metrics. Through our experiments, we observe that certain code completion results are presented in the form of "b==a" while the actual value is in the "a==b" format. Although these two expressions are semantically equivalent in terms of correctness, the way evaluation metrics are computed classifies the completion result as incorrect. In this study, we employ four automated evaluation metrics to assess code completion quality. Despite these metrics inevitably facing the aforementioned issue, we opt for popular and widely adopted metrics to align our evaluations closely with current research directions. Thus, for future work, besides using standard metrics, integrating human evaluations is essential to alleviate this threat.

Metrics for Code Completion Quality Evaluator. Since manually constructing datasets for training code completion quality evaluators is impractical, we utilized automated evaluation metrics,

namely BLEU and Edit-sim, to build our datasets for experiments. However, such exact match-based metrics overlook the semantics of code, which might undermine their reliability in measuring code completion quality. The research community has previously probed the limitations of these metrics [36, 37], yet they remain a formidable challenge. For quality evaluators in code completion, the acceptance rate of completions stands as the most desirable metric [22]. Obtaining such data, however, is challenging. Given that in the industrial domain, code completions are often short and do not need to be entirely correct—just requiring minor modifications by the programmer—we opted for similarity metrics that align closely with the ideal acceptance rate. To address these concerns, we call upon a broader participation of researchers to further delve into and advance studies in this realm.

8 RELATED WORK

Code Completion. The code completion task is a critical task in the field of software engineering, and early code completion methods typically employ rule-based heuristic approaches and statistical-based techniques. The heuristic rule-based approach [38, 39] for code completion involves manually writing rules to generate code suggestions. However, this approach requires a significant amount of manual labor and expertise and may struggle to cope with complex code completion tasks. The statistical-based approach [18, 40] for code completion employs N-gram LMs to model source code and learn code completion models by statistically analyzing large amounts of code data. However, this approach requires high-quality and high-quantity data and may struggle to handle semantic and contextual information in the code.

With the continuous development of deep learning techniques, SFT-based LMs have made significant strides in code completion. Liu et al. [41] proposed a neural network-based approach to generate more accurate completion suggestions by learning contextual information about the code. Li et al. [42] proposed CCTEST, a code completion framework for testing and repairing, which repairs the code after completion as the final output. Li et al. [2] proposed a pointer hybrid network to solve the out of vocabulary (OOV) problem, which improved the generalization ability and robustness of the model. In addition to the above-mentioned approach of treating source code sequences as code token sequences, some researchers have proposed transforming source code into an abstract syntax tree to predict the next node of the tree. This approach can better capture the semantic and structural information in the code, thus improving the effectiveness and quality of code completion. With the recent rise of pre-trained LMs, several research efforts have begun to utilize these models to solve the code completion problem [25]. This approach leverages the linguistic representation capabilities learned from large amounts of data to achieve the code completion task by fine-tuning on the datasets. Although these methods have good performance at first, *exposure bias* still needs to be addressed in SFT.

Sequence Generation via Reinforcement Learning. Related to code completion is the field of sequence generation, where DRL methods are often used to address the impact of *exposure bias* present in SFT. In a previous study on DRL, Ranzato et al. [14] directly used the final optimized indicators, BLEU and ROUGE, as reward signals and then used DRL algorithm to optimize network parameters in machine translation tasks. Wan et al. [43, 44] trained a generative model in the code summarization task using the actor-critic algorithm and a reward function consisting of BLEU metrics. Reed et al. [45] used the DRL algorithm to train policy networks in image generation tasks by simulating the drawing process using the policy gradient method. Recently, Chen et al. [46] proposed Decision Transformer, a method that transforms the DRL problem into a conditional sequence modeling problem by converting state and action sequences into inputs for a conditioned LM, which is then trained using the Transformer architecture to achieve the DRL task. In the field

of code completion, Shojaee et al. [15] proposed a novel reward function that combines discrete compiler feedback with the syntactic and semantic match scores between the generated code and the executable target. This approach aims to reduce the sparsity of the reward function by combining multiple metrics to better guide the generation of code that more closely aligns with the correct target.

Traditional reward designs in DRL have focused on optimizing evaluation metrics directly after sequence generation or relying on the functional correctness of unit tests for assignment. However, such delayed reward mechanisms often result in slow model convergence and a higher likelihood of getting trapped in a local optimal solution. In the context of code completion, unit tests cannot be used as a direct reward signal since the generated code may not necessarily form a complete code fragment. Therefore, it is crucial to devise a mechanism that provides immediate rewards for the sequence generation process.

Large Language Models (LLMs). At present, several closed-source models, including Chat-GPT [47] and GPT-4 [48], as well as open-source models such as CodeGen [31], Code Llama [49], CodeT5+ [28], and StarCoder [27], have showcased impressive capabilities in NL2Code tasks. Our primary objective is not to surpass these LLMs. Instead, we introduce a fine-tuning mechanism tailored for LM-based code completion. Theoretically, this mechanism can be applied across models constrained by exposure bias and delayed reward challenges.

9 CONCLUSION

We present IRCoCo, a DRL framework tailored for code completion, which collaboratively employs SFT fine-tuning and DRL-based alignment to augment pre-trained LMs. Specifically, we introduce a code completion quality evaluator to provide immediate rewards for the code completion generation process. We evaluate IRCoCo in combination with pre-trained LMs, and our comprehensive analysis demonstrates that IRCoCo can improve the performance of pre-trained LMs. In essence, IRCoCo is an innovative framework that leverages immediate rewards to improve pre-trained LMs' performance on the code completion task through DRL.

Data Availability. All experimental data and source code used in this paper are available at https://github.com/Libolun-star/IRCoCo.

ACKNOWLEDGMENTS

REFERENCES

[1] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 150–162. IEEE, 2021.

[2] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. Code completion with neural attention and pointer networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 4159–25, 2018.

[3] Yanlin Wang and Hui Li. Code completion by modeling flattened abstract syntax trees as graphs. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 14015–14023, 2021.

[4] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2727–2735, 2019.

[5] Wenhan Wang, Sijie Shen, Ge Li, and Zhi Jin. Towards full-line code completion with neural language models. *arXiv preprint arXiv:2009.08603*, 2020.

[6] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. Reacc: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6227–6240, 2022.

[7] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. Codefill: Multi-token code completion by jointly learning from structure and naming sequences. In *Proceedings of the 44th International Conference on Software Engineering*, pages 401–412, 2022.

[8] Nat Friedman. Introducing github copilot: your ai pair programmer. *URL https://github. blog/2021-06-29-introducing-github-copilot-ai-pair-programmer*, 2021.

[9] C Amazon. Ai code generator—amazon codewhisperer, 2023.

[10] Yizhan Huang, Yichen Li, Weibin Wu, Jianping Zhang, and Michael R Lyu. Do not give away my secrets: Uncovering the privacy issue of neural code completion tools. *arXiv preprint arXiv:2309.07639*, 2023.

[11] samsung-engineers-sensitive-data-chatgpt-warnings-ai-use-workplace. https://www.darkreading.com/vulnerabilities-threats/samsung-engineers-sensitive-data-chatgpt-warnings-ai-use-workplace, 2023. [Online; accessed 1-April-202].

[12] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 473–485, 2020.

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

[14] Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. Sequence level training with recurrent neural networks. In *4th International Conference on Learning Representations, ICLR 2016*, 2016.

[15] Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*, 2023.

[16] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

[17] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747, 2016.

[18] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th working conference on mining software repositories (MSR)*, pages 207–216. IEEE, 2013.

[19] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

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

[21] Dzmitry Bahdanau, Philemon Brakel, Kelvin Xu, Anirudh Goyal, Ryan Lowe, Joelle Pineau, Aaron Courville, and Yoshua Bengio. An actor-critic algorithm for sequence prediction. In *International Conference on Learning Representations*, 2016.

[22] Zhensu Sun, Xiaoning Du, Fu Song, Shangwen Wang, Mingze Ni, and Li Li. Don't complete it! preventing unhelpful code completion for productive and sustainable neural code completion systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 324–325. IEEE, 2023.

[23] Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Junjie Shan, Caishuang Huang, Wei Shen, Xiaoran Fan, Zhiheng Xi, et al. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint arXiv:2402.01391*, 2024.

[24] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[25] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 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)*, 2021.

[26] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[27] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

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

[29] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

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

[31] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2022.

[32] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443, 2020.

[33] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, 2021.

[34] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[35] 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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[36] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. Out of the bleu: how should we assess quality of the code generation models? *Journal of Systems and Software*, 203:111741, 2023.

[37] Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. Reassessing automatic evaluation metrics for code summarization tasks. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1105–1116, 2021.

[38] Martin Stubenschrott. A context sensitive code completion system for the c and c++ programming languages., 2005.

[39] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 213–222, 2009.

[40] Nicolas Bettenburg, Meiyappan Nagappan, and Ahmed E Hassan. Towards improving statistical modeling of software engineering data: think locally, act globally! *Empirical Software Engineering*, 20:294–335, 2015.

[41] Chang Liu, Xin Wang, Richard Shin, Joseph E Gonzalez, and Dawn Song. Neural code completion. 2016.

[42] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Shuai Wang, and Cuiyun Gao. Cctest: Testing and repairing code completion systems. *arXiv preprint arXiv:2208.08289*, 2022.

[43] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 397–407, 2018.

[44] Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, S Yu Philip, and Guandong Xu. Reinforcement-learning-guided source code summarization using hierarchical attention. *IEEE Transactions on software Engineering*, 48(1):102–119, 2020.

[45] Scott E Reed, Zeynep Akata, Santosh Mohan, Samuel Tenka, Bernt Schiele, and Honglak Lee. Learning what and where to draw. *Advances in neural information processing systems*, 29, 2016.

[46] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34:15084–15097, 2021.

[47] OpenAI. Chatgpt. https://OpenAI.com/blog/ChatGPT, 2022. Accessed: 2024-02-20.

[48] OpenAI OpenAI. Gpt-4 technical report. Mar 2023.

[49] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.