CoaCor: Code Annotation for Code Retrieval with Reinforcement Learning

Ziyu Yao
The Ohio State University
yao.470@osu.edu

Jayavardhan Reddy Peddamail
The Ohio State University
peddamail.1@osu.edu

Huan Sun
The Ohio State University
sun.397@osu.edu

ABSTRACT

To accelerate software development, much research has been performed to help people understand and reuse the huge amount of available code resources. Two important tasks have been widely studied: *code retrieval*, which aims to retrieve code snippets relevant to a given natural language query from a code base, and *code annotation*, where the goal is to annotate a code snippet with a natural language description. Despite their advancement in recent years, the two tasks are mostly explored separately. In this work, we investigate a novel perspective of *Code annotation for Code retrieval* (hence called "CoaCor"), where a code annotation model is trained to generate a natural language annotation that can represent the semantic meaning of a given code snippet and can be leveraged by a code retrieval model to better distinguish relevant code snippets from others. To this end, we propose an effective framework based on reinforcement learning, which explicitly encourages the code annotation model to generate annotations that can be used for the retrieval task. Through extensive experiments, we show that code annotations generated by our framework are much more detailed and more useful for code retrieval, and they can further improve the performance of existing code retrieval models significantly.[1]

CCS CONCEPTS

• **Information systems → Novelty in information retrieval**; **Summarization**; • **Software and its engineering**; • **Computing methodologies →** *Reinforcement learning*; *Markov decision processes*; Neural networks;

KEYWORDS

Code Annotation; Code Retrieval; Reinforcement Learning

ACM Reference Format:
Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. CoaCor: Code Annotation for Code Retrieval with Reinforcement Learning. In *Proceedings of the 2019 World Wide Web Conference (WWW '19), May 13–17, 2019, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3308558.3313632

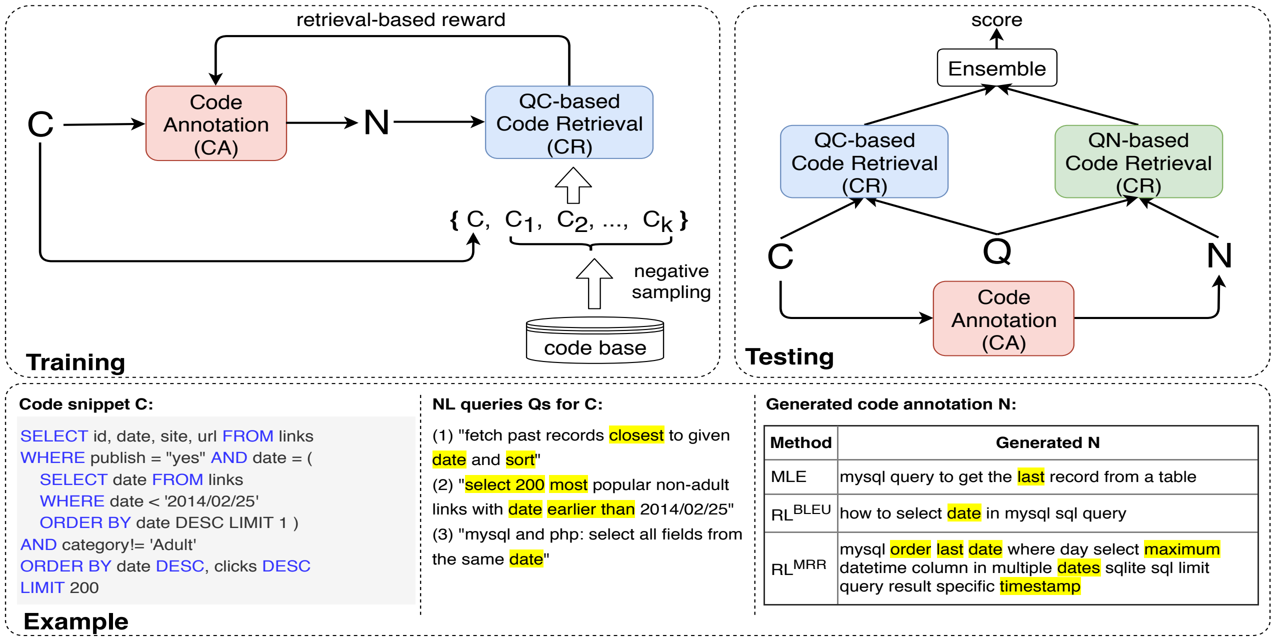
1 INTRODUCTION

Software engineering plays an important role in modern society. Almost every aspect of human life, including health care, education, transportation and web security, depends on reliable software [1]. Unfortunately, developing and maintaining large code bases are very costly. Understanding and reusing billions of lines of code in online open-source repositories can significantly speed up the software development process. Towards that, *code retrieval* (CR) and *code annotation* (CA) are two important tasks that have been widely studied in the past few years [1, 13, 19, 21, 58], where the former aims to retrieve relevant code snippets based on a natural language (NL) query while the latter is to generate natural language descriptions to describe what a given code snippet does.

Most existing work [2, 13, 19, 22, 58, 60] study either code annotation or code retrieval individually. Earlier approaches for code retrieval drew inspiration from the information retrieval field [14, 17, 23, 32] and suffered from surface form mismatches between natural language queries and code snippets [6, 34]. More recently, advanced deep learning approaches have been successfully applied to both code retrieval and code annotation [1, 2, 9, 13, 19–22, 31, 58, 60]. For example, the code retrieval model proposed by Gu et al. [13] utilized two deep neural networks to learn the vector representation of a natural language query and that of a code snippet respectively, and adopted cosine similarity to measure their matching degree. For code annotation, Iyer et al. [21] and Hu et al. [19] utilized encoder-decoder models with an attention mechanism to generate an NL annotation for a code snippet. They aim to generate annotations similar to the human-provided ones, and therefore trained the models using the standard maximum likelihood estimation (MLE) objective. For the same purpose, Wan et al. [58] trained the code annotation model in a reinforcement learning (RL) framework with reward being the BLEU score [41], which measures n-gram matching precision between the currently generated annotation and the human-provided one.

In this work, we explore a novel perspective - code annotation *for* code retrieval (CoaCor), which is to generate an NL annotation for a code snippet so that the generated annotation *can be used for code retrieval* (i.e., *can represent the semantic meaning of a code snippet and distinguish it from others w.r.t. a given NL query in the code retrieval task*). As exemplified by [56], such an annotation can be taken as the representation of the corresponding code snippet, based on which the aforementioned lexical mismatch issue in a naive keyword-based search engine can be alleviated. A similar idea of improving retrieval by adding extra annotations to items is also explored in document retrieval [47] and image search [61]. However, most of them rely on humans to provide the annotations. Intuitively, our perspective can be interpreted as one type of *machine-machine collaboration*: On the one hand, the NL annotation generated by the code annotation model can serve as a second view of a code snippet (in addition to its programming content) and can be utilized to match with an NL query in code retrieval. On the other hand, with

[1]Code available at https://github.com/LittleYUYU/CoaCor.



Code snippet C:

```
SELECT id, date, site, url FROM links
WHERE publish = "yes" AND date = (
   SELECT date FROM links
   WHERE date < '2014/02/25'
   ORDER BY date DESC LIMIT 1 )
AND category!= 'Adult'
ORDER BY date DESC, clicks DESC
LIMIT 200
```

NL queries Qs for C:

(1) "fetch past records closest to given date and sort"

(2) "select 200 most popular non-adult links with date earlier than 2014/02/25"

(3) "mysql and php: select all fields from the same date"

Generated code annotation N:

Method	Generated N
MLE	mysql query to get the last record from a table
RL^{BLEU}	how to select date in mysql sql query
RL^{MRR}	mysql order last date where day select maximum datetime column in multiple dates sqlite sql limit query result specific timestamp

Figure 1: Our CoaCor framework: (1) *Training phase.* A code annotation model is trained via reinforcement learning to maximize retrieval-based rewards given by a QC-based code retrieval model (pre-trained using <NL query, code snippet> pairs). (2) *Testing phase.* Each code snippet is first annotated by the trained CA model. For the code retrieval task, given query Q, a code snippet gets two scores - one matching Q with its code content and the other matching Q with its code annotation N, and is ranked by a simple ensemble strategy$^+$. (3) *Example.* We show an example of a code snippet and its associated multiple NL queries in our dataset. The code annotation generated by our framework (denoted as RL^{MRR}) is much more detailed with many keywords *semantically aligned* with Qs, when compared with CA models trained via MLE or RL with BLEU rewards (RL^{BLEU}). $^+$ We simply use a weighted combination of the two scores, and other ensemble strategies can also apply here.

a goal to facilitate code retrieval, the code annotation model can be stimulated to produce rich and detailed annotations. Unlike existing work [19–21, 58], our goal is *not* to generate an NL annotation as close as possible to a human-provided one; hence, the MLE objective or the BLEU score as rewards for code annotation will not fit our setting. Instead, we design a novel rewarding mechanism in an RL framework, which guides the code annotation model directly based on how effectively the currently generated code annotation distinguishes the code snippet from a candidate set.

Leveraging collaborations and interactions among machine learning models to improve the task performance has been explored in other scenarios. Goodfellow et al. [12] proposed the Generative Adversatial Nets (GANs), where a generative model produces difficult examples to fool a discriminative model and the latter is further trained to conquer the challenge. He et al. [15] proposed another framework called *dual learning*, which jointly learns two dual machine translation tasks (e.g., En -> Fr and Fr -> En). However, none of the existing frameworks are directly applicable to accomplish our goal (i.e., to train a code annotation model for generating annotations that can be utilized for code retrieval).

Figure 1 shows our reinforcement learning-based CoaCor framework. In the training phase, we first train a CR model based on <natural language query, code snippet> (QC) pairs (referred to as QC-based CR model). Then given a code snippet, the CA model generates a sequence of NL tokens as its annotation and receives a reward from the trained CR model, which measures how effectively the generated annotation can distinguish the code snippet from others. We formulate the annotation generation process as a Markov Decision Process [5] and train the CA model to maximize the received reward via an advanced reinforcement learning [53] framework called *Advantage Actor-Critic* or *A2C* [36]. Once the CA model is trained, we use it to generate an NL annotation *N* for each code snippet *C* in the code base. Therefore, for each QC pair we originally have, we can derive a QN pair. We utilize the generated annotation as a second view of the code snippet to match with a query and train another CR model based on the derived QN pairs (referred to as QN-based CR model). In the testing phase, given an NL query, we rank code snippets by combining their scores from both the QC-based as well as QN-based CR models, which utilize both the programming content as well as the NL annotation of a code snippet.

On a widely used benchmark dataset [21] and a recently collected large-scale dataset [63], we show that the automatically generated annotation can significantly improve the retrieval performance. More impressively, without looking at the code content, the QN-based CR model trained on our generated code annotations obtains a retrieval performance comparable to one of the state-of-the-art

QC-based CR models. It also surpasses other QN-based CR models trained using code annotations generated by existing CA models.

To summarize, our major contributions are as follows:

- First, we explored a novel perspective of generating useful code annotations *for* code retrieval. Unlike existing work [19, 21, 58], we do not emphasize the n-gram overlap between the generated annotation and the human-provided one. Instead, we examined the real usefulness of the generated annotations and developed a machine-machine collaboration paradigm, where a code annotation model is trained to generate annotations that can be used for code retrieval.
- Second, in order to accomplish our goal, we developed an effective RL-based framework with a novel rewarding mechanism, in which a code retrieval model is directly used to formulate rewards and guide the annotation generation.
- Last, we conducted extensive experiments by comparing our framework with various baselines including state-of-the-art models and variants of our framework. We showed significant improvements of code retrieval performance on both a widely used benchmark dataset and a recently collected large-scale dataset.

The rest of this paper is organized as follows. Section 2 introduces the background on code annotation and code retrieval tasks. Section 3 gives an overview of our proposed framework, with algorithm details followed in Section 4. Experiments are shown in Section 5. Finally, we discuss related work and conclude in Section 6 and 7.

2 BACKGROUND

We adopt the same definitions for code retrieval and code annotation as previous work [9, 21]. Given a natural language query Q and a set of code snippet candidates $\mathbb{C}$, *code retrieval* is to retrieve code snippets $C^* \in \mathbb{C}$ that can match with the query. On the other hand, given a code snippet C, *code annotation* is to generate a natural language (NL) annotation N^* which describes the code snippet appropriately. In this work, we use *code search* and *code retrieval* interchangeably (and same for *code annotation/summary/description*).

Formally, for a training corpus with <natural language query, code snippet> pairs, e.g., those collected from Stack Overflow [51] by [21, 63], we define the two tasks as:

Code Retrieval (CR): Given an NL Query Q, a model F_r will be learnt to retrieve the highest scoring code snippet $C^* \in \mathbb{C}$.

$$C^* = \underset{C \in \mathbb{C}}{\operatorname{argmax}}\, F_r(Q, C) \tag{1}$$

Code Annotation (CA): For a given code snippet C, the goal is to generate an NL annotation N^* that maximizes a scoring function F_a:

$$N^* = \underset{N}{\operatorname{argmax}}\, F_a(C, N) \tag{2}$$

Note that one can use the same scoring model for F_r and F_a as in [9, 21], but for most of the prior work [13, 19, 20, 58], which consider either code retrieval or code annotation, researchers usually develop their own models and objective functions for F_r or F_a. In our work, we choose two vanilla models as our base models for CR and CA, but explore a novel perspective of how to train F_a so that it can generate NL annotations that can be used for code retrieval. This perspective is inspired by various machine-machine collaboration mechanisms [15, 28, 55, 59] where one machine learning task can help improve another.

3 FRAMEWORK OVERVIEW

In this section, we first introduce our intuition and give an overview of the entire framework, before diving into more details.

3.1 Intuition behind CoaCor

To the best of our knowledge, previous code annotation work like [19–21, 58] focused on getting a large n-gram overlap between generated and human-provided annotations. However, it is still uncertain (and non-trivial to test) how helpful the generated annotations can be. Driven by this observation, we are the first to examine the real usefulness of the generated code annotations and how they can help a relevant task, of which we choose code retrieval as an example.

Intuitively, CoaCor can be interpreted as a *collaboration mechanism* between code annotation and code retrieval. On the one hand, the annotation produced by the CA model provides a second view of a code snippet (in addition to its programming content) to assist code retrieval. On the other hand, when the CA model is trained to be useful for the retrieval task, we expect it to produce richer and more detailed annotations, which we verify in experiments later.

3.2 Overview

The main challenge to realize the above intuition lies in how to train the CA model effectively. Our key idea to address the challenge is shown in Figure 1.

We first train a base CR model on <natural language query, code snippet> (QC) pairs. Intuitively, a QC-based CR model ranks a code snippet C by measuring how well it matches with the given query Q (in comparison with other code snippets in the code base). From another point of view, a well-trained QC-based CR model can work as a measurement on whether the query Q describes the code snippet C precisely or not. Drawing inspiration from this view, we propose using the trained QC-based CR model to determine whether an annotation describes its code snippet precisely or not and thereby, train the CA model to generate rich annotations to maximize the retrieval-based reward from the CR model. Specifically, given a code snippet C, the CA model generates a sequence of NL words as its annotation N. At the end of the sequence, we let the trained QC-based CR model use N to search for relevant code snippets from the code base. If C can be ranked at top places, the annotation N is treated as well-written and gets a high reward; otherwise, a low reward will be returned. We formulate this generation process as the Markov Decision Process [5] and train the CA model with reinforcement learning [53] (specifically, the *Advantage Actor-Critic* algorithm [36]) to maximize the retrieval-based rewards it can receive from the QC-based CR model. We elaborate the CR and CA model details as well as the RL algorithm for training the CA model in Section 4.1 ∼ 4.3.

Once the CA model is trained, in the testing phase, it generates an NL annotation N for each code snippet C in the code base. Now for each <NL query, code snippet> pair originally in the datasets,

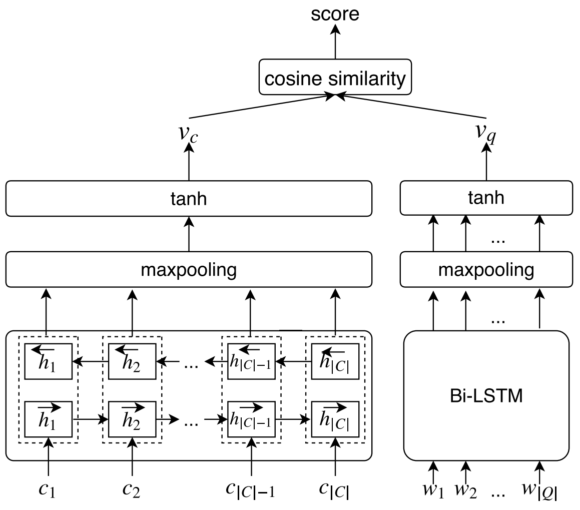


Figure 2: The base code retrieval model encodes the input code snippet C = $(c_1, c_2, ..., c_{|C|})$ and NL query Q = $(w_1, w_2, ..., w_{|Q|})$ into a vector space and outputs a similarity score.

we derive an <NL query, code annotation> (QN) pair and train another CR model based on such QN pairs. This QN-based CR model complements the QC-based CR model, as they respectively use the annotation and programming content of a code snippet to match with the query. We finally combine the matching scores from the two CR models to rank code snippets for a given query.

Note that we aim to outline a general paradigm to explore the perspective of code annotation for code retrieval, where the specific model structures for the two CR models and the CA model can be instantiated in various ways. In this work, we choose one of the simplest and most fundamental deep structures for each of them. Using more complicated model structures will make the training more challenging and we leave it as future work.

4 CODE ANNOTATION FOR CODE RETRIEVAL

Now we introduce model and algorithm details in our framework.

4.1 Code Retrieval Model

Both QC-based and QN-based code retrieval models adopt the same deep learning structure as the previous CR work [13]. Here for simplicity, we only illustrate the QC-based CR model in detail, and the QN-based model structure is the same except that we use the generated annotation on the code snippet side.

As shown in Figure 2, given an NL query $Q = w_{1..|Q|}$ and a code snippet $C = c_{1..|C|}$, we first embed the tokens of both code and NL query into vectors through a randomly initialized word embedding matrix, which will be learned during model training. We then use a bidirectional Long Short-Term Memory (LSTM)-based Recurrent Neural Network (RNN) [10, 11, 18, 35, 48, 64] to learn the token representation by summarizing the contextual information from both directions. The LSTM unit is composed of three multiplicative gates. At every time step t, it tracks the state of sequences by controlling how much information is updated into the new hidden state h_t and memory cell g_t from the previous state h_{t-1}, the previous memory cell g_{t-1} and the current input x_t. On the code side, x_t is the embedding vector for c_t, and on the NL query side, it is the embedding vector for w_t. At every time step t, the LSTM hidden state is updated as:

$$\begin{aligned} i &= \sigma(\mathbf{W}_i h_{t-1} + \mathbf{U}_i x_t + b_i) \\ f &= \sigma(\mathbf{W}_f h_{t-1} + \mathbf{U}_f x_t + b_f) \\ o &= \sigma(\mathbf{W}_o h_{t-1} + \mathbf{U}_o x_t + b_o) \\ g &= \tanh(\mathbf{W}_g h_{t-1} + \mathbf{U}_g x_t + b_g) \\ g_t &= f \odot g_{t-1} + i \odot g \\ h_t &= o \odot \tanh(g_t) \end{aligned}$$

where σ is the element-wise sigmoid function and $\odot$ is the element-wise product. $\mathbf{U}_i, \mathbf{U}_f, \mathbf{U}_g, \mathbf{U}_o$ denote the weight matrices of different gates for input x_t and $\mathbf{W}_i, \mathbf{W}_f, \mathbf{W}_g, \mathbf{W}_o$ are the weight matrices for hidden state h_t, while b_i, b_f, b_g, b_o denote the bias vectors. For simplicity, we denote the above calculation as below (the memory cell vector g_{t-1} is omitted):

$$h_t = \text{LSTM}(x_t, h_{t-1}) \tag{3}$$

The vanilla LSTM's hidden state h_t takes information from the past, knowing nothing about the future. Our CR model instead incorporates a bidirectional LSTM [48] (i.e., Bi-LSTM in Figure 2), which contains a forward LSTM reading a sequence X from start to end and a backward LSTM which reads from end to start. The basic idea of using two directions is to capture past and future information at each step. Then the two hidden states at each time step t are concatenated to form the final hidden state h_t.

$$\begin{aligned} \overrightarrow{h_t} &= \text{LSTM}(x_t, \overrightarrow{h_{t-1}}) \\ \overleftarrow{h_t} &= \text{LSTM}(x_t, \overleftarrow{h_{t+1}}) \\ h_t &= [\overrightarrow{h_t}, \overleftarrow{h_t}] \end{aligned}$$

Finally, we adopt the commonly used max pooling strategy [24] followed by a *tanh* layer to get the embedding vector for a sequence of length T.

$$v = \tanh(\text{maxpooling}([h_1, h_2, ..., h_T])) \tag{4}$$

By applying the above encoding algorithm, we encode the code snippet C and the NL query Q into v_c and v_q, respectively. Similar to Gu et al. [13], to measure the relevance between the code snippet and the query, we use cosine similarity denoted as $cos(v_q, v_c)$. The higher the similarity, the more related the code is to the query.

Training. The CR model is trained by minimizing a ranking loss similar to [13]. Specifically, for each query Q in the training corpus, we prepare a triple of $<Q, C, C^->$ as a training instance, where C is the correct code snippet that answers Q and C^- is a negative code snippet that does not answer Q (which is randomly sampled from the entire code base). The ranking loss is defined as:

$$\mathcal{L}(\theta) = \sum_{<Q,C,C^->} max(0, \epsilon - cos(v_q, v_c) + cos(v_q, v_{c^-})) \tag{5}$$

where θ denotes the model parameters, ϵ is a constant margin, and v_q, v_c and v_{c^-} are the encoded vectors of Q, C and C^- respectively.

Essentially, the ranking loss is a kind of hinge loss [46] that promotes the cosine similarity between Q and C to be greater than that between Q and C^- by at least a margin ϵ. The training leads the model to project relevant queries and code snippets to be close in the vector space.

4.2 Code Annotation Model

Formally, given a code snippet C, the CA model computes the probability of generating a sequence of NL tokens $n_{1..|N|} = (n_1, n_2, ..., n_{|N|})$ as its annotation N by:

$$P(N|C) = P(n_1|n_0, C) \prod_{t=2}^{|N|} P(n_t|n_{1..t-1}, C) \tag{6}$$

where n_0 is a special token "<START>" indicating the start of the annotation generation, $n_{1..t-1} = (n_1, ..., n_{t-1})$ is the partially generated annotation till time step t-1, and $P(n_t|n_{1..t-1}, C)$ is the probability of producing n_t as the next word given the code snippet C and the generated $n_{1..t-1}$. The generation stops once a special token "<EOS>" is observed.

In this work, we choose the popular sequence-to-sequence model [52] as our CA model structure, which is composed of an encoder and a decoder. We employ the aforementioned bidirectional LSTM-based RNN structure as the encoder for code snippet C, and use another LSTM-based RNN as the decoder to compute Eqn. (6):

$$h_t^{\text{dec}} = \text{LSTM}(n_{t-1}, h_{t-1}^{\text{dec}}), \forall t = 1, ..., |N|$$

where h_t^{dec} is the decoder hidden state at step t, and h_0^{dec} is initialized by concatenating the last hidden states of the code snippet encoder in both directions. In addition, a standard global attention layer [33] is applied in the decoder, in order to attend to important code tokens in C:

$$\tilde{h}_t^{\text{dec}} = \tanh(\mathbf{W}_\alpha [v_{\text{attn}}, h_t^{\text{dec}}])$$

$$v_{\text{attn}} = \sum_{t'=1}^{|C|} \alpha_{t'} \, h_{t'}^{\text{enc}}$$

$$\alpha_{t'} = \text{softmax}((h_{t'}^{\text{enc}})^T h_t^{\text{dec}})$$

where $\alpha_{t'}$ is the attention weight on the t'-th code token when generating the t-th word in the annotation, and $\mathbf{W}_\alpha$ is a learnable weight matrix. Finally, the t-th word is selected based on:

$$P(n_t|n_{0..t-1}, C) = \text{softmax}(\mathbf{W}\tilde{h}_t^{\text{dec}} + b) \tag{7}$$

where $\mathbf{W} \in R^{|V_n| \times d}, b \in R^{|V_n|}$ project the d-dim hidden state $\tilde{h}_t^{\text{dec}}$ to the NL vocabulary of size $|V_n|$.

4.3 Training Code Annotation via RL

4.3.1 Code Annotation as Markov Decision Process. Most previous work [9, 19, 21] trained the CA model by maximizing the log-likelihood of generating human annotations, which suffers from two drawbacks: (1) The *exposure bias* issue [4, 44, 45]. That is, during training, the model predicts the next word given the ground-truth annotation prefix, while at testing time, it generates the next word based on previous words generated by itself. This mismatch between training and testing may result in error accumulation in testing phase. (2) More importantly, maximizing the likelihood is not aligned with our goal to produce annotations that can be useful for code retrieval.

To address the above issues, we propose to formulate code annotation within the reinforcement learning (RL) framework [53]. Specifically, the annotation generation process is viewed as a Markov Decision Process (MDP) [5] consisting of four main components:

State. At step t during decoding, a state s_t maintains the source code snippet and the previously generated words $n_{1..t-1}$, i.e., $s_t = \{C, n_{1..t-1}\}$. In particular, the initial decoding state $s_0 = \{C\}$ only contains the given code snippet C. In this work, we take the hidden state vector $\tilde{h}_t^{dec}$ as the vector representation of state s_t, and the MDP is thus processing on a continuous and infinite state space.

Action. The CA model decides the next word (or *action*) $n_t \in V_n$, where V_n is the NL vocabulary. Thus, the action space in our formulation is the NL vocabulary.

Reward. As introduced in Section 3, to encourage it to generate words useful for the code retrieval task, the CA model is rewarded by a well-trained QC-based CR model based on whether the code snippet C can be ranked at the top positions if using the generated *complete* annotation N as a query. Therefore, we define the reward at each step t as:

$$r(s_t, n_t) = \begin{cases} \text{RetrievalReward}(C, n_{1..t}) & \text{if } n_t = \text{<EOS>} \\ 0 & \text{otherwise} \end{cases}$$

where we use the popular ranking metric Mean Reciprocal Rank [57] (defined in Section 5.2) as the RetrievalReward(C, N) value. Note that, in this work, we let the CR model give a valid reward only when the annotation generation stops, and assign a zero reward to intermediate steps. However, other designs such as giving rewards to a partial generation with the reward shaping technique [4] can be reasonable and explored in the future.

Policy. The policy function $P(n_t|s_t)$ takes as input the current state s_t and outputs the probability of generating n_t as the next word. Given our definition about state s_t and Eqn. (7),

$$P(n_t|s_t) = P(n_t|n_{1..t-1}, C) = \text{softmax}(\mathbf{W}\tilde{h}_t^{dec} + b) \tag{8}$$

Here, the policy function is stochastic in that the next word can be sampled according to the probability distribution, which allows action space exploration and can be optimized using policy gradient methods [53].

The objective of the CA model training is to find a policy function $P(N|C)$ that maximizes the expected accumulative future reward:

$$\max_\phi \mathcal{L}(\phi) = \max_\phi \mathbb{E}_{N \sim P(\cdot|C;\phi)}[R(C, N)] \tag{9}$$

where ϕ is the parameter of P and $R(C, N) = \sum_{t=1}^{|N|} r(s_t, n_t)$ is the accumulative future reward (called "return").

The gradient of the above objective is derived as below.

$$\begin{aligned} \nabla_\phi \mathcal{L}(\phi) &= \mathbb{E}_{N \sim P(\cdot|C;\phi)}[R(C, N) \nabla_\phi \log P(N|C;\phi)] \\ &= \mathbb{E}_{n_{1..|N|} \sim P(\cdot|C;\phi)}[\sum_{t=1}^{|N|} R_t(s_t, n_t) \nabla_\phi \log P(n_t|n_{1..t-1}, C;\phi)] \end{aligned} \tag{10}$$

where $R_t(s_t, n_t) = \sum_{t' \geq t} r(s_{t'}, n_{t'})$ is the return for generating word n_t given state s_t.

4.3.2 Advantage Actor-Critic for Code Annotation. Given the gradient in Eqn. (10), the objective in Eqn. (9) can be optimized by policy gradient approaches such as REINFORCE [62] and Q-value Actor-Critic algorithm [4, 54]. However, such methods may yield very high variance when the action space (i.e., the NL vocabulary) is large and suffer from biases when estimating the return of rarely taken actions [39], leading to unstable training. To tackle the challenge, we resort to the more advanced *Advantage Actor-Critic* or *A2C* algorithm [36], which has been adopted in other sequence generation tasks [39, 58]. Specifically, the gradient function in Eqn. (10) is replaced by an *advantage* function:

$$\nabla_\phi \mathcal{L}(\phi) = \mathbb{E}[\sum_{t=1}^{|N|} A_t \nabla_\phi \log P(n_t|n_{1..t-1}, C; \phi)] \quad (11)$$
$$A_t = R_t(s_t, n_t) - V(s_t)$$
$$V(s_t) = \mathbb{E}_{\hat{n}_t \sim P(\cdot|n_{1..t-1}, C)}[R_t(s_t, \hat{n}_t)]$$

where $V(s_t)$ is the state value function that estimates the future reward given the current state s_t. Intuitively, $V(s_t)$ works as a *baseline* function [62] to help the model assess its action n_t more precisely: When advantage A_t is greater than zero, it means that the return for taking action n_t is better than the "average" return over all possible actions, given state s_t; otherwise, action n_t performs worse than the average.

Following previous work [36, 39, 58], we approximate $V(s_t)$ by learning another model $V(s_t; \rho)$ parameterized by ρ, and the RL framework thus contains two components: the policy function $P(N|C; \phi)$ that generates the annotation (called "Actor"), and the state value function $V(s_t; \rho)$ that approximates the return under state s_t (called "Critic"). Similar to the actor model (i.e., the CA model in Section 4.2), we train a separate attention-based sequence-to-sequence model as the critic network. The critic value is finally computed by:

$$V(s_t; \rho) = \mathbf{w}_\rho^T (\tilde{h}_t^{dec})_{crt} + b_\rho \quad (12)$$

where $(\tilde{h}_t^{dec})_{crt} \in R^d$ is the critic decoder hidden state at step t, and $\mathbf{w}_\rho \in R^d$, $b_\rho \in R$ are trainable parameters that project the hidden state to a scalar value (i.e., the estimated state value).

The critic network is trained to minimize the Mean Square Error between its estimation and the true state value:

$$\min_\rho \mathcal{L}(\rho) = \min_\rho \mathbb{E}_{N \sim P(\cdot|C)}[\sum_{t=1}^{|N|} (V(s_t; \rho) - R(C, N))^2] \quad (13)$$

The entire training procedure of CoaCor is shown in Algorithm 1.

4.4 Generated Annotations for Code Retrieval

As previously shown in Figure 1, in the testing phase, we utilize the generated annotations to assist the code retrieval task. Now we detail the procedure in Algorithm 2. Specifically, for each <NL query, code snippet> pair (i.e., QC pair) in the dataset, we first derive an <NL query, code annotation> pair (i.e., QN pair) using the code annotation model. We then build another code retrieval (CR) model based on the QN pairs in our training corpus. In this work, for simplicity, we choose the same structure as the QC-based CR model (Section 4.1) to match QN pairs. However, more advanced methods for modeling the semantic similarity between two NL sentences (e.g., previous work on NL paraphase detection [16, 26]) are applicable and can be explored as future work.

The final matching score between query Q and code snippet C combines those from the QN-based and QC-based CR model:

$$\text{score}(Q, C) = \lambda * cos(v_q, v_n) + (1 - \lambda) * cos(v_q, v_c) \quad (14)$$

where v_q, v_c, v_n are the encoded vectors of Q, C, and the code annotation N respectively. $\lambda \in [0, 1]$ is a weighting factor for the two scores to be tuned on the validation set.

Algorithm 1 : Training Procedure for CoaCor.

```
Input: <NL query, code snippet> (QC) pairs in training set, number of iterations $E$.
1: Train a base code retrieval model based on QC pairs, according to Eqn. (5).
2: Initialize a base code annotation model ($\phi$) and pretrain it via MLE according to Eqn. (7), using $Q$ as the desired $N$ for $C$.
3: Pretrain a critic network ($\rho$) according to Eqn. (13).
4: for $iteration = 1$ to $E$ do
5:     Receive a code snippet $C$.
6:     Sample an annotation $N \sim P(\cdot|C;\phi)$ according to Eqn. (8).
7:     Receive the final reward $R(C, N)$.
8:     Update the code annotation model ($\phi$) using Eqn. (11).
9:     Update the critic network ($\rho$) using Eqn. (13).
10: end for
```

Algorithm 2 : Generated Annotations for Code Retrieval.

```
Input: NL query $Q$, code snippet candidate $C$.
Output: The matching score, score($Q, C$).
1: Receive score$_1(Q, C) = cos(v_q, v_c)$ from a QC-based code retrieval model.
2: Generate a code annotation $N \sim P(\cdot|C;\phi)$ via greedy search, according to Eqn. (8).
3: Receive score$_2(Q, C) = cos(v_q, v_n)$ from a QN-based code retrieval model.
4: Calculate score($Q, C$) according to Eqn. (14).
```

5 EXPERIMENTS

In this section, we conduct extensive experiments and compare our framework with various models to show its effectiveness.

5.1 Experimental Setup

Dataset. (1) We experimented with the **StaQC** dataset presented by Yao et al. [63]. The dataset contains 119,519 SQL <question title, code snippet> pairs mined from Stack Overflow [51], making itself the largest-to-date in SQL domain. In our code retrieval task, the question title is considered as the NL query Q, which is paired with the code snippet C to form the QC pair. We randomly selected 75% of the pairs for training, 10% for validation (containing 11,900 pairs), and the left 15% for testing (containing 17,850 pairs). As mentioned in [63], the dataset may contain multiple code snippets for the same NL query. We examine the dataset split to ensure that alternative relevant code snippets for the same query would not be sampled as negative code snippets when training the CR model. For pretraining the CA model, we consider the question title as a code annotation N and form QN pairs accordingly. **(2)** Iyer et al. [21] collected two small sets of SQL code snippets (called "**DEV**" and "**EVAL**" respectively) from Stack Overflow for validation and testing. In addition to the originally paired question title, each code snippet is manually annotated by two different NL descriptions. Therefore, in total, each set contains around 100 code snippets with three NL descriptions (resulting in around 300 QC pairs). We use them as additional datasets for model comparison.[2] Following [21], QC pairs occuring in DEV and EVAL set or being used as negative code snippets by them are removed from the StaQC training set.

Data Preprocessing. We followed [21] to perform code tokenization, which replaced table/column names with placeholder tokens and numbered them to preserve their dependencies. For text tokenization, we utilized the "word_tokenize" tool in the NLTK toolkit [7]. All code tokens and NL tokens with a frequency of less than 2 were replaced with an <UNK> token, resulting in totally 7726 code tokens and 7775 word tokens in the vocabulary. The average lengths of the NL query and the code snippet are 9 and 60 respectively.

5.2 Evaluation

We evaluate a model's retrieval performance on four datasets: the validation (denoted as "StaQC-val") and test set (denoted as "StaQC-test") from StaQC [63], and the DEV and EVAL set from [21]. For each <NL query Q, code snippet C> pair (or QC pair) in a dataset, we take C as a positive code snippet and randomly sample K negative code snippets from all others except C in the dataset,[3] and calculate the rank of C among the $K + 1$ candidates. We follow [9, 21, 63] to set $K = 49$. The retrieval performance of a model is then assessed by the Mean Reciprocal Rank (MRR) metric [57] over the entire set $\mathcal{D} = \{(Q_1, C_1), (Q_2, C_2), ..., (Q_{|\mathcal{D}|}, C_{|\mathcal{D}|})\}$:

$$MRR = \frac{1}{|\mathcal{D}|} \sum_{i=1}^{|\mathcal{D}|} \frac{1}{Rank_i}$$

where $Rank_i$ is the rank of C_i for query Q_i. The higher the MRR value, the better the code retrieval performance.

5.3 Methods to Compare

In order to test the effectiveness of our CoaCor framework, we compare it with both existing baselines and our proposed variants.

Existing Baselines. We choose the following state-of-the-art code retrieval models, which are based on QC pairs, for comparison.

- Deep Code Search (DCS) [13]. The original DCS model [13] adopts a similar structure as Figure 2 for CR in Java domain. To learn the vector representations for code snippets, in addition to code tokens, it also considers features like function names and API sequences, all of which are combined into a fully connected layer. In our dataset, we do not have these features, and thus slightly modify their original model to be the same as our QC-based CR model (Figure 2).
- CODE-NN [21]. CODE-NN is one of the state-of-the-art models for both code retrieval and code annotation. Its core component is an LSTM-based RNN with an attention mechanism, which models the probability of generating an NL sentence conditioned on a given code snippet. For code retrieval, given an NL query, CODE-NN computes the likelihood of generating the query as an annotation for each code snippet and ranks code snippets based on the likelihood.

QN-based CR Variants. As discussed in Section 4.4, a trained CA model is used to annotate each code snippet C in our datasets with an annotation N. The resulting <NL query Q, code annotation N> pairs can be used to train a QN-based CR model. Depending on how we train the CA model, we have the following variants:

- QN-MLE. Similar to most previous work [9, 19, 21], we simply train the CA model in the standard MLE manner, i.e., by maximizing the likelihood of a human-provided annotation.
- QN-RL$^{\text{BLEU}}$. As introduced in Section 1, Wan et al. [58] proposed to train the CA model via reinforcement learning with BLEU scores [41] as rewards. We compare this variant with our rewarding mechanism.
- QN-RL$^{\text{MRR}}$. In our CoaCor framework, we propose to train the CA model using retrieval rewards from a QC-based CR model (see Section 3). Here we use the MRR score as the retrieval reward.

Since CODE-NN [21] can be used for code annotation as well, we also use its generated code annotations to train a QN-based CR model, denoted as "QN-CodeNN".[4]

Ensemble CR Variants. As introduced in Section 4.4, we tried ensembling the QC-based CR model and the QN-based CR model to improve the retrieval performance. We choose the DCS structure as the QC-based CR model as mentioned in Section 4.1. Since different QN-based CR models can be applied, we present the following 4 variants: (1) QN-MLE + DCS, (2) QN-RL$^{\text{BLEU}}$ + DCS, (3) QN-RL$^{\text{MRR}}$ + DCS, and (4) QN-CodeNN + DCS, where QN-MLE, QN-RL$^{\text{BLEU}}$, QN-RL$^{\text{MRR}}$, and QN-CodeNN have been introduced.

5.4 Implementation Details

Our implementation is based on Pytorch [42]. For CR models, we set the embedding size of words and code tokens to 200, and chose batch size in {128, 256, 512}, LSTM unit size in {100, 200, 400} and dropout rate [50] in {0.1, 0.35, 0.5}. A small, fixed ϵ value of 0.05 is used in all the experiments. Hyper-parameters for each model were chosen based on the DEV set. For CODE-NN baseline, we followed Yao et al. [63] to use the same model hyper-parameters as the original paper, except that the dropout rate is tuned in {0.5, 0.7}. The StaQC-val set was used to decay the learning rate and the best model parameters were decided based on the retrieval performance on the DEV set.

For CA models, the embedding size of words and code tokens and the LSTM unit size were selected from {256, 512}. The dropout rate is selected from {0.1, 0.3, 0.5} and the batch size is 64. We updated model parameters using the Adam optimizer [25] with learning rate 0.001 for MLE training and 0.0001 for RL training. The maximum length of the generated annotation is set to 20. For CodeNN, MLE-based and RL$^{\text{BLEU}}$-based CA models, the best model parameters

[2]Previous work [9, 21] used only one of the three descriptions while we utilize all of them to enrich and enlarge the datasets for a more reliable evaluation.
[3]For DEV and EVAL, we use the same negative examples as [21].

[4]There is another recent code annotation method named DeepCom [19]. We did not include it as baseline, since it achieved a similar performance as our MLE-based CA model (see Table 3) when evaluated with the standard BLEU script by [21].

Table 1: The main code retrieval results (MRR). * denotes significantly different from DCS [13] in one-tailed t-test ($p < 0.01$).

Model	DEV	EVAL	StaQC-val	StaQC-test
Existing (QC-based) CR Baselines				
DCS [13]	0.566	0.555	0.534	0.529
CODE-NN [21]	0.530	0.514	0.526	0.522
QN-based CR Variants				
QN-CodeNN	0.369	0.360	0.336	0.333
QN-MLE	0.429	0.411	0.427	0.424
QN-RL$^{\text{BLEU}}$	0.426	0.402	0.386	0.381
QN-RL$^{\text{MRR}}$ (ours)	0.534	0.512	0.516	0.523
Ensemble CR Variants				
QN-CodeNN + DCS	0.566	0.555	0.534	0.529
QN-MLE + DCS	0.571	0.561	0.543	0.537
QN-RL$^{\text{BLEU}}$ + DCS	0.570	0.559	0.541	0.534
QN-RL$^{\text{MRR}}$ + DCS (ours)	0.582*	**0.572***	0.558*	0.559*
QN-RL$^{\text{MRR}}$ + CODE-NN (ours)	**0.586***	0.571*	**0.575***	**0.576***

were picked based on the model's BLEU score on DEV, while for RL$^{\text{MRR}}$-based CA model, we chose the best model according to its MRR reward on StaQC-val. For RL models, after pretraining the actor network via MLE, we first pretrain the critic network for 10 epochs, then jointly train the two networks for 40 epochs. Finally, for ensemble variants, the ensemble weight λ in all variants is selected from $0.0 \sim 1.0$ based on its performance on DEV.

5.5 Results

To understand our CoaCor framework, we first show several concrete examples to understand the differences between annotations generated by our model and by baseline/variant models, and then focus on two research questions (RQs):

- **RQ1 (CR improves CA)**: Is the proposed retrieval reward-driven CA model capable of generating rich code annotations that can be used for code retrieval (i.e., can represent the code snippet and distinguish it from others)?
- **RQ2 (CA improves CR)**: Can the generated annotations further improve existing QC-based code retrieval models?

5.5.1 Qualitative Analysis. Table 2 presents two examples of annotations generated by each CA model. Note that we do not target at human language-like annotations; rather, we focus on annotations that can describe/capture the functionality of a code snippet. In comparison with baseline CA models, our proposed RL$^{\text{MRR}}$-based CA model can produce more concrete and precise descriptions for corresponding code snippets. As shown in Example 1, the annotation generated by RL$^{\text{MRR}}$ covers more conceptual keywords semantically aligned with the three NL queries (e.g., "average", "difference", "group"), while the baseline CODE-NN and the variants generate short descriptions covering a very limited amount of conceptual keywords (e.g., without mentioning the concept "subtracting").

We also notice that our CA model can generate different forms of a stem word (e.g., "average", "avg" in Example 1), partly because the retrieval-based reward tends to make the generated annotation semantically aligned with the code snippet and these diverse forms of words can help strengthen such semantic alignment and benefit the code retrieval task when there are various ways to express user search intent.

5.5.2 Code Retrieval Performance Evaluation. Table 1 shows the code retrieval evaluation results, based on which we discuss **RQ1** and **RQ2** as below:

RQ1: To examine whether or not the code annotations generated by a CA model can represent the corresponding code snippet in the code retrieval task, we analyze its corresponding QN-based CR model, which retrieves relevant code snippets by matching the NL query Q with the code annotation N generated by this CA model. Across all of the four datasets, our proposed QN-RL$^{\text{MRR}}$ model, which is based on a retrieval reward-driven CA model, achieves the best results and outperforms other QN-based CR models by a wide margin of around $0.1 \sim 0.2$ absolute MRR. More impressively, its performance is already on a par with the CODE-NN model, which is one of the state-of-the-art models for the code retrieval task, even though it understands a code snippet solely based on its annotation and without looking at the code content. This demonstrates that the code annotation generated by our proposed framework can reflect the semantic meaning of each code snippet more precisely.

To further understand whether or not the retrieval-based reward can serve as a better reward metric than BLEU (in terms of stimulating a CA model to generate useful annotations), we present the BLEU score of each CA model in Table 3.[5] When connecting this table with Table 1, we observe an inverse trend: For the RL$^{\text{BLEU}}$ model which is *trained for* a higher BLEU score, although it can improve the MLE-based CA model by more than 2% absolute BLEU on three sets, it harms the latter's ability on producing useful code annotations (as revealed by the performance of QN-RL$^{\text{BLEU}}$ in Table 1, which is worse than QN-MLE by around 0.04 absolute MRR on StaQC-val and StaQC-test). In contrast, our proposed RL$^{\text{MRR}}$ model, despite getting the lowest BLEU score, is capable of generating annotations useful for the retrieval task. This is mainly because that

[5]BLEU is evaluated with the script provided by Iyer et al. [21]: https://github.com/sriniiyer/codenn/blob/master/src/utils/bleu.py.

Table 2: Two examples of code snippets and their annotations generated by different CA models. "Human-provided" refers to (multiple) human-provided NL annotations or queries. Words semantically aligned between the generated and the human-provided annotations are highlighted.

Model	Annotation
Example 1 from EVAL set	
SQL Code	SELECT col3, Format(Avg([col2]-[col1]),"hh:mm:ss") AS TimeDiff FROM Table1 GROUP BY col3;
Human-provided	(1) find the average time in hours , mins and seconds between 2 values and show them in groups of another column (2) group rows of a table and find average difference between them as a formatted date (3) ms access average after subtracting
CODE-NN	how do i get the average of a column in sql?
MLE	how to get average of the average of a column in sql
RL^{BLEU}	how to average in sql query
RL^{MRR}	average avg calculating difference day in access select distinct column value sql group by month mysql format date function?
Example 2 from StaQC-test set	
SQL Code	SELECT Group_concat(DISTINCT(p.products_id)) AS comma_separated, COUNT(DISTINCT p.products_id) AS product_count FROM ...
Human-provided	how to count how many comma separated values in a group_concat
CODE-NN	how do i get the count of distinct rows?
MLE	mysql query to get count of distinct values in a column
RL^{BLEU}	how to count in mysql sql query
RL^{MRR}	group_concat count concatenate distinct comma group mysql concat column in one row rows select multiple columns of same id result

BLEU score calculates surface form overlaps while the retrieval-based reward measures the semantically aligned correspondences.

These observations imply an interesting conclusion: *Compared with BLEU, a (task-oriented) semantic measuring reward, such as our retrieval-based MRR score, can better stimulate the model to produce detailed and useful generations.* This is in line with the recent discussions on whether the automatic BLEU score is an appropriate evaluation metric for generation tasks or not [30, 40]. In our work, we study the potential to use the performance of a relevant model to guide the learning of the target model, which can be generalized to many other scenarios, e.g., conversation generation [27], machine translation [4, 44], etc.

RQ2: We first inspect whether the generated code annotations can assist the base code retrieval model (i.e., DCS) or not by comparing several ensemble CR variants. It is shown that, by simply combining the matching scores from QN-RL^{MRR} and DCS with a weighting factor, our proposed model is able to significantly outperform the DCS model by 0.01 ∼ 0.03 and the CODE-NN baseline by 0.03 ∼ 0.06 consistently across all datasets, showing the advantage of utilizing code annotations for code retrieval. Particularly, the best performance is achieved when the ensemble weight $\lambda = 0.4$ (i.e., 0.4 weight on the QN-based CR score and 0.6 on the QC-based CR score), meaning that the model relies heavily on the code annotation to achieve better performance.

Table 3: The BLEU score of each code annotation model.

Model	DEV	EVAL	StaQC-val	StaQC-test
CODE-NN [21]	17.43	16.73	8.89	8.96
MLE	18.99	19.87	10.52	10.55
RL^{BLEU}	21.12	18.52	12.72	12.78
RL^{MRR}	8.09	8.52	5.56	5.60

In contrast, QN-CodeNN, QN-MLE and QN-RL^{BLEU} can hardly improve the base DCS model, and their best performances are all achieved when the ensemble weight $\lambda = 0.0 \sim 0.2$, indicating little help from annotations generated by CODE-NN, MLE-based and BLEU-rewarded CA. This is consistent with our conclusions to RQ1.

We also investigate the benefit of our generated annotations to other code retrieval models (besides DCS) by examining a baseline "QN-RL^{MRR} + CODE-NN", which combines QN-RL^{MRR} and CODE-NN (as a QC-based CR model) to score a code snippet candidate. As mentioned in Section 5.3, CODE-NN scores a code snippet by the likelihood of generating the given NL query when taking this code snippet as the input. Since the score is in a different range from the cosine similarity given by QN-RL^{MRR}, we first rescale it by taking its log value and dividing it by the largest absolute log score among all code candidates. The rescaled score is then combined with the cosine similarity score from QN-RL^{MRR} following Eqn. (14). The result is shown in the last row of Table 1. It is very impressive that, with the help of QN-RL^{MRR}, the CODE-NN model can be improved by ≥ 0.05 absolute MRR value across all test sets.

In summary, through extensive experiments, we show that our proposed framework can generate code annotations that are much more useful for building effective code retrieval models, in comparison with existing CA models or those trained by MLE or BLEU-based RL. Additionally, the generated code annotations can further improve the retrieval performance, when combined with existing CR models like DCS and CODE-NN.

6 DISCUSSION

In this work, we propose a novel perspective of using a relevant downstream task (i.e., code retrieval) to guide the learning of a target task (i.e., code annotation), illustrating a novel machine-machine collaboration paradigm. It is shown that the annotations generated by the RL^{MRR} CA model (trained with rewards from the DCS model) can boost the performance of the CODE-NN model, which was not involved in any stage of the training process. It is interesting to explore more about machine-machine collaboration mechanisms, where multiple models for either the same task or relevant tasks can be utilized in tandem to provide different views or effective rewards to improve the final performance.

In terms of training, we also experimented with directly using a QN-based CR model or an ensemble CR model for rewarding the

CA model. However, these approaches do not work well, since we do not have a rich set of QN pairs as training data in the beginning. Collecting paraphrases of queries to form QN pairs is non-trivial, which we leave to the future.

Finally, our CoaCor framework is applicable to other programming languages, such as Python and C#, extension to which is interesting to study as future work.

7 RELATED WORK

Code Retrieval. As introduced in Section 1, code retrieval has been studied widely with information retrieval methods [14, 17, 23, 32, 56] and recent deep learning models [3, 13, 21]. Particularly, Keivanloo et al. [23] extracted abstract programming patterns and their associated NL keywords from code snippets in a code base, with which a given NL query can be projected to a set of associated programming patterns facilitating code content-based search. Similarly, Vinayakarao et al. [56] built an entity discovery system to mine NL phrases and their associated syntactic patterns, based on which they annotated each line of code snippets with NL phrases. Such annotations were utilized to improve NL keyword-based search engines. Different from these work, we construct a neural network-based code annotation model to describe the functionality of an entire code snippet. Our code annotation model is explicitly trained to produce meaningful words that can be used for code search. In our framework, the code retrieval model adopts a similar deep structure as the Deep Code Search model proposed by Gu et al. [13], which projects a NL query and a code snippet into a vector space and measures the cosine similarity between them.

Code Annotation. Code annotation/summarization has drawn a lot of attention in recent years. Earlier works tackled the problem using template-based approaches [37, 49] and topic n-grams models [38] while the recent techniques [2, 19–22, 31] are mostly built upon deep neural networks. Specifically, Sridhara et al. [49] developed a software word usage model to identify action, theme and other arguments from a given code snippet, and generated code comments with templates. Allamanis et al. [2] employed convolution on the input tokens to detect local time-invariant and long-range topical attention features to summarize a code snippet into a short, descriptive function name-like summary. Most related to our work are [19] and [58], which utilized sequence-to-sequence networks with attention over code tokens to generate natural language annotations. They aimed to generate NL annotations as close as possible to human-provided annotations for human readability, and hence adopted the Maximum Likelihood Estimation (MLE) or BLEU score optimization as the objective. However, our goal is to generate code annotations *which can be used for code retrieval*, and therefore we design a retrieval-based reward to drive our training.

Deep Reinforcement Learning for Sequence Generation. Reinforcement learning (RL) [53] has shown great success in various tasks where an agent has to perform multiple actions before obtaining a reward or when the metric to optimize is not differentiable. The sequence generation tasks, such as machine translation [4, 39, 44], image captioning [45], dialogue generation [27] and text summarization [43], have all benefitted from RL to address the *exposure bias* issue [4, 44, 45] and to directly optimize the model towards a certain metric (e.g., BLEU). Particularly, Ranzato et al. [44] were among the first to successfully apply the REINFORCE algorithm [62] to train RNN models for several sequence generation tasks, indicating that directly optimizing the metric used at test time can lead to significantly better models than those trained via MLE. Bahdanau et al. [4] additionally learned a *critic* network to better estimate the return (i.e., future rewards) of taking a certain action under a specific state, and trained the entire generation model via the *Actor-Critic* algorithm [54]. We follow Nguyen et al. [39] and Wan et al. [58] to further introduce an *advantage* function and train the code annotation model via the *Advantage Actor-Critic* algorithm [36], which is helpful for reducing biases from rarely taken actions. However, unlike their work, the reward in our framework is based on the performance on a different yet relevant task (i.e., code retrieval), rather than the BLEU metric.

Machine-Machine Collaboration via Adversarial Training and Dual/Joint Learning. Various kinds of machine-machine collaboration mechanisms have been studied in many scenarios [12, 15, 28, 55, 59]. For example, Goodfellow et al. [12] proposed the Generative Adversarial Nets (GANs) framework, where a generative model generates images to fool a discriminative classifier, and the latter is further improved to distinguish the generated from the real ones. He et al. [15] proposed the dual learning framework and jointly optimized the machine translation from English to French and from French to English. Li et al. [29] trained a paraphrase generator by rewards from a paraphrase evaluator model. In the context of code retrieval and annotation, Chen and Zhou [9] and Iyer et al. [21] showed that their models can be used directly or with slight modification for both tasks, but their training objective only considered one of the two tasks. All these frameworks are not directly applicable to achieve our goal, i.e., training a code annotation model to generate rich NL annotations that can be used for code search.

8 CONCLUSION

This paper explored a novel perspective of generating code annotations *for* code retrieval. To this end, we proposed a reinforcement learning-based framework (named "CoaCor") to maximize a retrieval-based reward. Through comprehensive experiments, we demonstrated that the annotation generated by our framework is more detailed to represent the semantic meaning of a code snippet. Such annotations can also improve the existing code content-based retrieval models significantly. In the future, we will explore other usages of the generated code annotations, as well as generalizing our framework to other tasks such as machine translation.

ACKNOWLEDGMENTS

This research was sponsored in part by the Army Research Office under cooperative agreements W911NF-17-1-0412, NSF Grant IIS1815674, Fujitsu gift grant, and Ohio Supercomputer Center [8]. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notice herein.

REFERENCES

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

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*. 2091–2100.

[3] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *International Conference on Machine Learning*. 2123–2132.

[4] Dzmitry Bahdanau, Philemon Brakel, Kelvin Xu, Anirudh Goyal, Ryan Lowe, Joelle Pineau, Aaron Courville, and Yoshua Bengio. 2016. An actor-critic algorithm for sequence prediction. *arXiv preprint arXiv:1607.07086* (2016).

[5] Richard Bellman. 1957. A Markovian decision process. *Journal of Mathematics and Mechanics* (1957), 679–684.

[6] Ted J Biggerstaff, Bharat G Mitbander, and Dallas E Webster. 1994. Program understanding and the concept assignment problem. *Commun. ACM* 37, 5 (1994), 72–82.

[7] Steven Bird and Edward Loper. 2004. NLTK: the natural language toolkit. In *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*. Association for Computational Linguistics, 31.

[8] Ohio Supercomputer Center. 1987. Ohio Supercomputer Center. http://osc.edu/ark:/19495/f5s1ph73.

[9] Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 826–831.

[10] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. 1999. Learning to forget: Continual prediction with LSTM. (1999).

[11] Christoph Goller and Andreas Kuchler. 1996. Learning task-dependent distributed representations by backpropagation through structure. In *Neural Networks, 1996., IEEE International Conference on*, Vol. 1. IEEE, 347–352.

[12] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems*. 2672–2680.

[13] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 933–944.

[14] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 842–851.

[15] Di He, Yingce Xia, Tao Qin, Liwei Wang, Nenghai Yu, Tieyan Liu, and Wei-Ying Ma. 2016. Dual learning for machine translation. In *Advances in Neural Information Processing Systems*. 820–828.

[16] Hua He and Jimmy Lin. 2016. Pairwise word interaction modeling with deep neural networks for semantic similarity measurement. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 937–948.

[17] Emily Hill, Manuel Roldan-Vega, Jerry Alan Fails, and Greg Mallet. 2014. NL-based query refinement and contextualized code search results: A user study. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 34–43.

[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[19] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep Code Comment Generation. In *Proceedings of the 2017 26th IEEE/ACM International Conference on Program Comprehension*. ACM.

[20] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing Source Code with Transferred API Knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 2269–2275. https://doi.org/10.24963/ijcai.2018/314

[21] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 2073–2083.

[22] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 135–146.

[23] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 664–675.

[24] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

[25] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[26] Wuwei Lan and Wei Xu. 2018. Neural Network Models for Paraphrase Identification, Semantic Textual Similarity, Natural Language Inference, and Question Answering. *arXiv preprint arXiv:1806.04330* (2018).

[27] Jiwei Li, Will Monroe, Alan Ritter, Michel Galley, Jianfeng Gao, and Dan Jurafsky. 2016. Deep reinforcement learning for dialogue generation. *arXiv preprint arXiv:1606.01541* (2016).

[28] Yikang Li, Nan Duan, Bolei Zhou, Xiao Chu, Wanli Ouyang, Xiaogang Wang, and Ming Zhou. 2018. Visual question generation as dual task of visual question answering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 6116–6124.

[29] Zichao Li, Xin Jiang, Lifeng Shang, and Hang Li. 2018. Paraphrase Generation with Deep Reinforcement Learning. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 3865–3878.

[30] Chia-Wei Liu, Ryan Lowe, Iulian V Serban, Michael Noseworthy, Laurent Charlin, and Joelle Pineau. 2016. How not to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation. *arXiv preprint arXiv:1603.08023* (2016).

[31] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A neural architecture for generating natural language descriptions from source code changes. *arXiv preprint arXiv:1704.04856* (2017).

[32] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. 2015. Query expansion via wordnet for effective code search. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 545–549.

[33] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025* (2015).

[34] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. 2012. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering* 38, 5 (2012), 1069–1087.

[35] Larry Medsker and Lakhmi C Jain. 1999. *Recurrent neural networks: design and applications*. CRC press.

[36] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*. 1928–1937.

[37] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 23–32.

[38] Dana Movshovitz-Attias and William W Cohen. 2013. Natural language models for predicting programming comments. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, Vol. 2. 35–40.

[39] Khanh Nguyen, Hal Daumé III, and Jordan Boyd-Graber. 2017. Reinforcement Learning for Bandit Neural Machine Translation with Simulated Human Feedback. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. 1464–1474.

[40] Jekaterina Novikova, Ondřej Dušek, Amanda Cercas Curry, and Verena Rieser. 2017. Why we need new evaluation metrics for nlg. *arXiv preprint arXiv:1707.06875* (2017).

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

[42] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[43] Romain Paulus, Caiming Xiong, and Richard Socher. 2017. A deep reinforced model for abstractive summarization. *arXiv preprint arXiv:1705.04304* (2017).

[44] Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2015. Sequence level training with recurrent neural networks. *arXiv preprint arXiv:1511.06732* (2015).

[45] Steven J Rennie, Etienne Marcheret, Youssef Mroueh, Jarret Ross, and Vaibhava Goel. 2017. Self-critical sequence training for image captioning. In *CVPR*, Vol. 1. 3.

[46] Lorenzo Rosasco, Ernesto De Vito, Andrea Caponnetto, Michele Piana, and Alessandro Verri. 2004. Are loss functions all the same? *Neural Computation* 16, 5 (2004), 1063–1076.

[47] Falk Scholer, Hugh E Williams, and Andrew Turpin. 2004. Query association surrogates for web search. *Journal of the American Society for Information Science and Technology* 55, 7 (2004), 637–650.

[48] Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing* 45, 11 (1997), 2673–2681.

[49] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 43–52.

[50] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.

[51] Stack Overflow. 2018. Stack Overflow. https://stackoverflow.com/.

[52] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

[53] Richard S Sutton and Andrew G Barto. 1998. *Introduction to reinforcement learning*. Vol. 135. MIT press Cambridge.

[54] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. 2000. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*. 1057–1063.

[55] Duyu Tang, Nan Duan, Tao Qin, Zhao Yan, and Ming Zhou. 2017. Question answering and question generation as dual tasks. *arXiv preprint arXiv:1706.02027* (2017).

[56] Venkatesh Vinayakarao, Anita Sarma, Rahul Purandare, Shuktika Jain, and Saumya Jain. 2017. ANNE: Improving Source Code Search Using Entity Retrieval Approach. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining (WSDM '17)*. ACM, New York, NY, USA, 211–220. https://doi.org/10.1145/3018661.3018691

[57] Ellen M Voorhees et al. 1999. The TREC-8 Question Answering Track Report.. In *Trec*, Vol. 99. 77–82.

[58] 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*. ACM, 397–407.

[59] Jun Wang, Lantao Yu, Weinan Zhang, Yu Gong, Yinghui Xu, Benyou Wang, Peng Zhang, and Dell Zhang. 2017. Irgan: A minimax game for unifying generative and discriminative information retrieval models. In *Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval*. ACM, 515–524.

[60] Xiaoran Wang, Yifan Peng, and Benwen Zhang. 2018. Comment Generation for Source Code: State of the Art, Challenges and Opportunities. *arXiv preprint arXiv:1802.02971* (2018).

[61] Wikipedia contributors. 2019. Google Image Labeler — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Google_Image_Labeler&oldid=881738511 [Online; accessed 4-February-2019].

[62] Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8, 3-4 (1992), 229–256.

[63] Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. 2018. StaQC: A Systematically Mined Question-Code Dataset from Stack Overflow. *arXiv preprint arXiv:1803.09371* (2018).

[64] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).