CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning

Anonymous Author(s)
Affiliation
Address
email

Abstract

Program synthesis or code generation aims to generate a program that satisfies a problem specification. Recent approaches using large-scale pretrained language models (LMs) have shown promising results, yet they have some critical limitations. In particular, they often follow a standard supervised fine-tuning procedure to train a code generation model from natural language problem descriptions and ground-truth programs only. Such paradigm largely ignores some important but potentially useful signals in the problem specification such as unit tests, which thus results in poor performance when solving complex unseen coding tasks. To address the limitations, we propose "CodeRL", a new framework for program synthesis tasks through pretrained LMs and deep reinforcement learning (RL). Specifically, during training, we treat the code-generating LM as an actor network, and introduce a critic network that is trained to predict the functional correctness of generated programs and provide dense feedback signals to the actor. During inference, we introduce a new generation procedure with a critical sampling strategy that allows a model to automatically regenerate programs based on feedback from example unit tests and critic scores. For the model backbones, we extended the encoder-decoder architecture of CodeT5 with enhanced learning objectives, larger model sizes and better pretraining data. Our method not only achieves new SOTA results on the challenging APPS benchmark, but also shows strong zero-shot transfer capability with new SOTA results on the simpler MBPP benchmark.

1 Introduction

Considering program synthesis as a sequence-to-sequence task, pretrained language models (LMs) [Hendrycks et al., 2021, Chen et al., 2021a, Austin et al., 2021] can be adapted to receive input sequence as problem specification in natural language and generate a sequence of codes as the output program (See Figure 1, right, for an example). While these models achieve promising results, especially in basic programming tasks [Chen et al., 2021a, Austin et al., 2021], we observe that they still fail to generate codes to solve complex problems [Hendrycks et al., 2021, Li et al., 2022].

There are two main limitations. First, current models are trained using a conventional next-token prediction (NTP) objective which maximizes the next ground-truth token likelihood. As noted in NLP domains Bengio et al. [2015], Ranzato et al. [2016], training models only with next-token prediction objective in a "teacher-forcing" manner often leads to accumulating errors during test time when tokens are generated by conditioning on previously sampled tokens, not the ground-truth tokens. This issue becomes more serious in the domain of program synthesis, where token-matching scores such

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

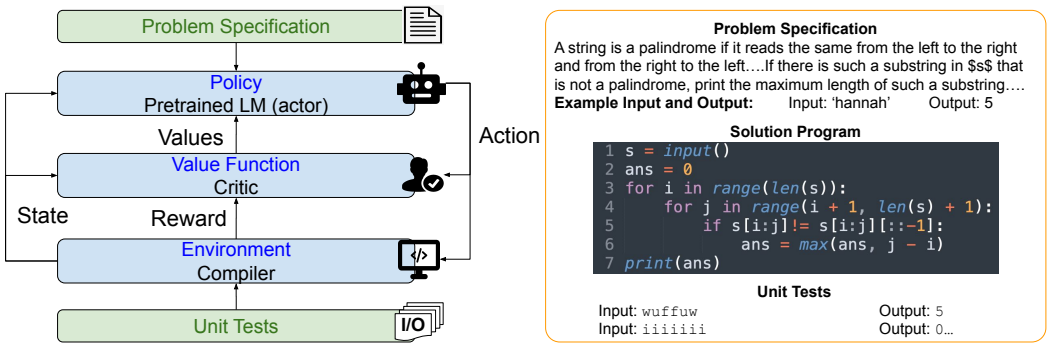

Figure 1: A high-level overview of our CodeRL framework for program synthesis (Left) and an example program synthesis task (Right)

as BLEU [Papineni et al., 2002, Ren et al., 2020] are more appropriate in partial program synthesis tasks (i.e. code completion) [Husain et al., 2019] but have failed to measure the functional correctness of complete programs [Hendrycks et al., 2021, Chen et al., 2021a]. Training only with NTP objective is hence, not ideal to tackle full program generation to solve programming problems.

Secondly, current models fail to utilize the potential meaningful signals from unit tests, which directly determine the model performance by the functional correctness of programs. Current approaches neglect this important signal during model optimization as well as generation procedure. During optimization, unit tests could be factored into learning objectives to match the final goal of generating semantically correct programs. During inference, since unit tests are often parts of problem description (i.e. example unit tests), they are potentially powerful to further improve output programs.

To address the above issues, we introduce "CodeRL", a new framework to improve pretrained LMs for program synthesis tasks through reinfocement learning (See Figure 1, left). Specifically, we propose a training strategy that optimizes pretrained LMs for program synthesis tasks in an actor-critic approach [Konda and Tsitsiklis, 1999, Sutton et al., 1999]. We treat the pretrained LM as an actor network and synthetically sample sequences from this actor, including both correct and incorrect programs. These program samples are passed to a critic model which is trained as an error predictor to assess the functional correctness of these samples. We use the token-level hidden states extracted from the learned critic model to estimate the values/scores of output tokens of these synthetic samples. The actor network is then finetuned on these synthetic samples weighted by their critic scores. During inference, we introduce a new generation procedure that involves example unit tests and a critic to filter and select sub-sequences. These sub-sequences are utilized as seeds which condition the model to resample new tokens and obtain new output programs. This approach allows the model to automatically refine output programs based on their functional correctness during test time.

We extend CodeT5 with better pretraining strategies as the foundation model for CodeRL. Our comprehensive experiments show that our models can achieve SOTA performance on the challenging APPS benchmark [Hendrycks et al., 2021]. Specifically, our models reach more than 2% *pass@1*, 6% *pass@5*, and 19% *pass@1000*. Since our RL method is model-agnostic, we apply it to various large-scale models and achieve consistent performance gains. We further test its zero-shot transfer ability on a simpler MBPP benchmark [Austin et al., 2021], where it sets a new SOTA result of 63.0% *pass@80* over a finetuned GPT-137B's 61.4%. We release the improved CodeT5-large (770M) model which outperforms many pretrained LMs of much larger sizes.

2 Related Work

Program Synthesis. Program synthesis tasks can date back as early as the early adoption of machine learning research Waldinger and Lee [1969], Manna and Waldinger [1971]. Earlier tasks include problem specifications in the form of input-output (IO) examples [Summers, 1977, Gulwani et al., 2012] and synthesis methods are limited to probabilistic approaches [Liang et al., 2010] or

simple programming concepts [Joulin and Mikolov, 2015, Kurach et al., 2015]. As deep learning methods became popular, later approaches adopt neural models to induce output programs, assuming an inductive bias given large number of program samples [Parisotto et al., 2016, Balog et al., 2016, Devlin et al., 2017]. More recently, we witnessed the emergence of program synthesis tasks in which output programs are extended to general-purpose programming languages [Yin and Neubig, 2017, Xu et al., 2018, Chen et al., 2021a] and program specifications are fully described in natural English text [Hendrycks et al., 2021, Austin et al., 2021, Poesia et al., 2022]. These extensions have encouraged a rising number of applications of pretrained language models (LMs) to program synthesis to exploit the contextual representations learned from massive data of codes and natural languages [Feng et al., 2020, Clement et al., 2020, Wang et al., 2021, Wang and Komatsuzaki, 2021, Chen et al., 2022]. Despite impressive results in basic programming problems and initial commercial deployment[1], existing models still perform poorly against complex problems such as those from programming competitions on Codeforces [Hendrycks et al., 2021, Li et al., 2022].

Reinforcement Learning for Sequence Generation. Related to the program synthesis tasks are research domains of sequence generation, in which RL approaches have demonstrated remarkable achievements. In these domains, RL approaches are used to exploit signals from non-differentiable metrics of the task at hand. Earlier work such as [Ranzato et al., 2016] adopts this strategy with REINFORCE algorithm [Williams, 1992] to directly optimize models for sequence-based test metrics such as BLEU [Papineni et al., 2002] and ROUGE [Lin, 2004] scores for translation models. In the same domain, Bahdanau et al. [2016] introduced an actor-critic framework [Sutton, 1984, Konda and Tsitsiklis, 1999]. In visual captioning domains, Rennie et al. [2017], Wang et al. [2018] proposed to use RL to optimize image captioning models using variants of CIDEr scores [Vedantam et al., 2015]. Alternatively, Ren et al. [2017] derived a new goal-oriented return estimate using visual-semantic embedding. Johnson et al. [2017], Trivedi et al. [2021] introduce program generation as an auxiliary task to learn interpretable policies in question-answering and synthetic navigation tasks.

Different from prior domains, in program synthesis, Austin et al. [2021], Chen et al. [2021a], Li et al. [2022] demonstrated very low correlation between token-based similarity metrics and functional correctness of programs. Hence, it is not trivial to define an appropriate optimization goal in this domain. We propose to exploit unit test signals, which directly exhibit functional correctness of programs, during both - model optimization and test-time generation stages. More related to our work are RL-based program synthesis [Guu et al., 2017, Bunel et al., 2018, Liang et al., 2018, Zhong et al., 2018] and execution-guided synthesis approaches [Ellis et al., 2019, Chen et al., 2021b]. However, these are limited to programming languages defined within a specific application domain only.

3 CodeRL

3.1 Program Synthesis Task

Following a sequence-to-sequence approach, the program synthesis task contains a problem description as an input sequence D and an output sequence of program $\hat{W} = (\hat{w}_1, ..., \hat{w}_T), \hat{w}_t \in \mathcal{V}$ [2] that can solve the problem. The output at each decoding step t is a distribution over the vocabulary $\mathcal{V}$, computed by the softmax function $\hat{w}_t \sim \mathrm{softmax}(\mathrm{Linear}(s_t))$ where s_t is the contextual hidden state at decoding step t. Conventionally, during train time, model parameters, θ, are learned by maximizing the likelihood of the ground-truth reference programs. Denoting $W = (w_1, ...w_T)$ as the ground-truth program, the objective is to minimize the cross-entropy loss:

$$\mathcal{L}_{ce}(\theta) = -\sum_t \log p_\theta(W|D) = -\sum_t \log[p_\theta(w_t|w_{1:t-1}, D)] \quad (1)$$

where the conditional probability p_θ is parameterized following the above softmax function. During test time, models generate sequences of programs by autoregressively sampling token $\hat{w}_t$ from the distribution $p_\theta(.|\hat{w}_{1:t-1}, D)$. Models are evaluated against unit tests corresponding to the problem.

[1] `https://copilot.github.com/`

[2] For simplicity, we use T as the notation of sequence length for all sequences which can actually be variable.

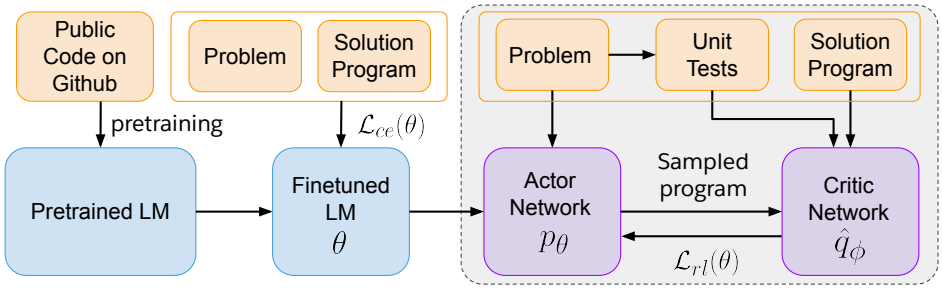

Figure 2: Overview of our actor-critic framework to optimize pretrained LMs for program synthesis

Each test includes a pair of input and ground-truth output. In real-world program synthesis tasks [Hendrycks et al., 2021], *example unit tests* are often given as parts of the problem specification.

3.2 Pretraining Language Models on Code

We adopt Transformer models as the backbone of our program synthesis systems. Specifically, this paper extends the CodeT5 model [Wang et al., 2021] as a foundation model for CodeRL.

CodeT5. CodeT5 [Wang et al., 2021] is a multi-lingual code-aware language model pretrained on large-scale source code corpora curated from Github. With a unified encoder-decoder architecture, CodeT5 achieves state-of-the-art performance in a wide range of code intelligence tasks in the CodeXGLUE benchmark [Lu et al., 2021] including both code understanding and generation tasks.

Improving Pretraining Data. We enlarge the Python pretraining dataset using the recently released large-scale Github Code dataset[3]. We filter the dataset by keeping only the code with licenses that at least permit academic use ("mit", "apache-2", "bsd-3-clause", "bsd-2- 126 clause", "cc0-1.0", "unlicense", "isc"). The resulting Python dataset (GCPY) has 10.5B tokens and is 10x larger than the CodeSearchNet (CSN) corpus [Husain et al., 2019] used in the original CodeT5 [Wang et al., 2021].

Improving Pretraining Objective. While pretraining tasks in CodeT5 like masked span prediction (MSP) benefit code understanding tasks, they have a large discrepancy with program synthesis objectives. To mitigate this gap, we introduce a pretraining task of next-token prediction (NTP) into CodeT5. Specifically, we uniformly sample a pivot location for each code sample, then pass the content preceding the pivot to the encoder and remaining to the decoder. To control the length of input and output sequences, we restrict the pivot within 10% to 90% of the original sequence.

3.3 Program Synthesis as an RL Problem

We propose to formulate the Program Synthesis as an RL problem and apply an actor-critic RL approach to improve the performance of a pretrained LM by exploiting the unit test signals in both model optimization (See Fig. 2) and generation procedure (See Fig. 3).

More formally, we can view the learned parameters of an LM model, θ as a stochastic *policy*, which decides an *action* as the prediction of each token. Following each action, an LM model updates its hidden state representations which are used by the policy to determine the next action in the next decoding step. At the end of the generation episode (i.e. an <*endoftext*> token is observed), the LM model receives a *return* r measured by the functional correctness of the generated program. The goal of RL finetuning is to minimize the expected return:

$$\mathcal{L}_{rl}(\theta) = -\mathbb{E}_{W^s \sim p_\theta}[r(W^s)] \tag{2}$$

where $W^s = (w_1^s, ..., w_T^s)$ is a synthetic sample in which each token w_t^s is sampled by the LM model at decoding time step t. Following the REINFORCE algorithm [Williams, 1992, Sutton and Barto, 2018] and policy gradient theorem [Sutton et al., 1999] we can define an estimate of the gradient

[3] `https://huggingface.co/datasets/lvwerra/github-code`

$\nabla_\theta L(\theta)$ of the non-differentiable return r as:

$$\begin{aligned}\nabla_\theta \mathcal{L}_{rl}(\theta) &\approx -\mathbb{E}_{W^s \sim p_\theta}[r(W^s)\nabla_\theta \log p_\theta(W^s|D)] \\ &\approx -\mathbb{E}_{W^s \sim p_\theta}[r(W^s)\sum_t \nabla_\theta \log p_\theta(w_t^s|w_{1:t-1}^s, D)]\end{aligned} \tag{3}$$

Defining Return by Unit Test Signals. For each sample sequence W^s, the return r can be defined heuristically by checking its functional correctness. We pass generated programs together with the corresponding unit tests to a compiler. From the outputs of the tests, we can determine the return r:

$$r(W^s) = \begin{cases} \text{-1.0} & \text{, if } W^s \text{ cannot be compiled (i.e. compile error)} \quad (4) \\ \text{-0.6} & \text{, if } W^s \text{ cannot be executed with unit tests (i.e. runtime error)} \quad (5) \\ \text{-0.3} & \text{, if } W^s \text{ failed any unit test} \quad (6) \\ \text{+1.0} & \text{, if } W^s \text{ passed all unit tests} \quad (7) \end{cases}$$

However, in related domains such as text-to-SQL research [Zhong et al., 2018, Xu et al., 2018], we note that this approach to estimate returns can lead to unstable training process with high variance of the gradient estimate following Eq. (3) with mini-batches in training.

Return with a Baseline. In order to alleviate this variance, we adopt a "baseline" [Sutton and Barto, 2018]. Specifically, we use a greedy decoding strategy as a baseline and any generated samples that outperform this baseline are given positive return estimation, and negative return estimation otherwise. This relative normalization technique allows models to explore imperfect programs, as long as their returns are better than the baseline's. Given a training sample, we denote the return of the baseline $r(W^b)$ and the expected gradient is computed as:

$$\nabla_\theta \mathcal{L}_{rl}(\theta) \approx -\mathbb{E}_{W^s \sim p_\theta}[(r(W^s) - r(W^b))\sum_t \nabla_\theta \log p_\theta(w_t^s|w_{1:t-1}^s, D)] \tag{8}$$

Note that at each decoding step t, our greedy decoding baseline is independent from the action w_t^s and hence, the expected gradient term $\nabla_\theta \mathcal{L}_{rl}(\theta)$ from Eq. (3) remains the same in Eq. (8).

Intermediate Return by Critic as Error Predictor. We observe that the above gradient estimate is only based on a final return at the end of the decoding process. However, programs often follow fixed syntactical rules in which a single token such as an additional white-space can render a program erroneous. Therefore, Eq. (8) becomes too restrictive. A straightforward solution is to use token-based similarity scores [Papineni et al., 2002, Ren et al., 2020]) between each subsequence $W_{1:t}^s$ and the ground truth. However, code matching is not an ideal return measure due to its poor correlation with program correctness [Hendrycks et al., 2021, Chen et al., 2021a, Austin et al., 2021] which can only be measured against fully complete programs.

Alternatively, we introduce a *critic* model. The critic model is parameterized as a neural network with parameters ϕ that receives inputs as the problem description D and a sampled program $W^s = \{w_1^s, \ldots, w_T^s\}$. The critic is trained to infer the unit test outcome; one of $\{\text{CompileError}, \text{RuntimeError}, \text{FailedTest}, \text{PassedTest}\}$ as described in the return definitions in Eq. (4) to (7). The training objective of the critic ϕ can be expressed as:

$$\mathcal{L}_{critic}(\phi) = -\log p_\phi(u|W^s, D) \tag{9}$$

where u denotes the ground-truth unit test outcome given by the compiler. We use Transformer models of smaller sizes than the actor model as the base architecture for the critic model. The contextual hidden states of the program tokens $(h_1, \ldots, h_T)$ obtained from the critic model decoder are max-pooled along the sequence length dimension $h^{\text{pool}} = \text{Pooling}(h_1, \ldots, h_T)$. The critic's prediction on the unit test outcome is computed as $\hat{u} = \text{softmax}(\text{Linear}(h^{\text{pool}}))$.

Given a learned critic, we use the probability distribution $\hat{v}_t = \text{softmax}(\text{Linear}(h_t))$ to estimate the token-level value $\hat{q}$ of w_t^s *in relation to* the ground-truth unit test output (note that we use the token level contextual representation h_t here, before the pooling operation). Specifically, $\hat{q}_\phi(w_t^s) = \hat{v}_t[u]$ where $\hat{v}[.]$ denotes the probability of a specific unit test outcome from the four possible ones. We use

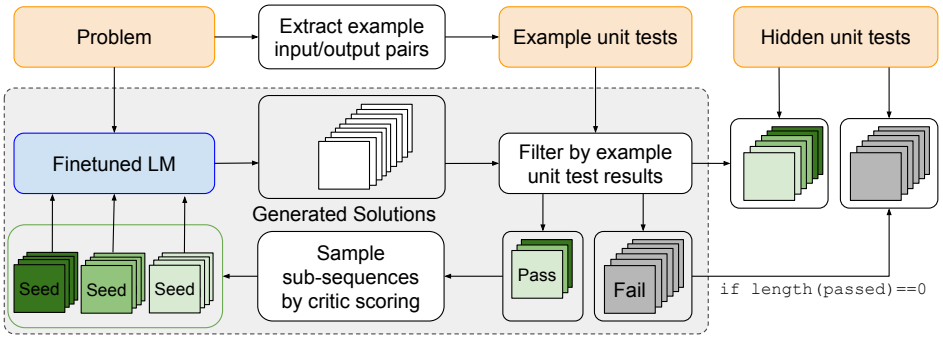

Figure 3: Overview of our Critic Sampling (CS) approach for program synthesis during inference

this estimate to train the actor LM model with intermediate returns:

$$\nabla_\theta \mathcal{L}_{rl}(\theta) \approx -\mathbb{E}_{W^s \sim p_\theta}[(r(W^s) - r(W^b)) \sum_t \hat{q}_\phi(w_t^s) \nabla_\theta \log p_\theta(w_t^s | w_{1:t-1}^s, D)] \tag{10}$$

Generating Programs with Example Unit Tests and Critic. We leverage the unit tests provided in the input problem description to improve the generation procedure during inference too (see Figure 3 for an overview). For each problem, we generate N programs, out of which we only select programs that pass example tests (leading to a set $\mathcal{F}$) and filter out the rest. To improve sample quality, we perform another round of generation where we use sub-sequences from these filtered samples as prompts (or "seed" sequences) to the actor LM. We employ a separate critic model (ϕ_{test}) to guide our choice of sub-sequences from these filtered samples. This critic model is trained with a similar objective as Eq. (9), but in a binary classification setup with $\{\text{FailedTest}, \text{PassedTest}\}$ labels.

Let $W^{\text{filter}} = \{w_1, \ldots, w_T\}$ denote a generated sample that passes the example unit tests. We use the critic model to assign a value to each token $\hat{q}_{\phi_{\text{test}}}(w_t) = p_{\phi_{\text{test}}}(\hat{u} = \text{PassedTest} | w_{1:t}, D)$ corresponding to the critic's predicted probability of the sub-sequence till t passing the unit tests. We split the sequence at position t_{max} corresponding to the highest critic assigned value and use the left split as the seed for the next stage. If this seed sequence till t_{max} contains a token with $p_{\phi_{\text{test}}}(\text{FailedTest}) > p_{\phi_{\text{test}}}(\text{PassedTest})$, we further chop it at this token by removing tokens on the right. This is done to pick prompts that are likely to generate successful programs in the next round.

We use these seeds to initialize and condition the (actor) LM to resample new tokens till we encounter the <*endoftext*> token. In this round, each seed sequence can be stacked $N/|\mathcal{F}|$ times for upsampling. This results in the same number of output programs N. We call this generation procedure as "Critic Sampling" (CS). We use mini-batch generating to improve efficiency during inference and employ nucleus sampling with a batch size of $N = 200$. While we do incur additional cost to re-sample using the seed sequences, we are only required to generate partial programs in the re-generation stage, making this stage less expensive than conventional generating procedures.

4 Experiments

4.1 Experimental Setups and Datasets

Pretraining Setup. We pretrain a CodeT5-large model (770M) from scratch following T5-large's architecture [Raffel et al., 2020]. We follow the pretraining setups in CodeT5 [Wang et al., 2021] with the modifications as proposed in §3.2. We evaluate this new pretrained CodeT5 model on CodeXGLUE [Lu et al., 2021] and achieve new SOTA results (See the Supplementary).

APPS Benchmark. We choose the challenging APPS program synthesis benchmark [Hendrycks et al., 2021], as it has large coding problems of varying difficulties collected from multiple coding websites. It includes training and test splits, each of which has 5000 samples of programming tasks with diverse levels of difficulty, including "Introductory", "Interview", and "Competition" levels. Each sample includes 20 unit tests on average to validate the functional correctness of programs.

Table 1: Results on APPS: "Intro": introductory, "Inter": interview, "Comp": competition-level tasks

(a) Performance by *pass@k* with $k = \{1, 5, 1000\}$

Model	Size	*pass@1*				*pass@5*				*pass@1000*			
		Intro	Inter	Comp	All	Intro	Inter	Comp	All	Intro	Inter	Comp	All
Codex	12B	4.14	0.14	0.02	0.92	9.65	0.51	0.09	2.25	25.02	3.70	3.23	7.87
AlphaCode	1B	-	-	-	-	-	-	-	-	17.67	5.24	7.06	8.09
GPT3	175B	0.20	0.03	0.00	0.06	-	-	-	-	-	-	-	-
GPT2	0.1B	1.00	0.33	0.00	0.40	2.70	0.73	0.00	1.02	-	-	-	-
GPT2	1.5B	1.30	0.70	0.00	0.68	3.60	1.03	0.00	1.34	-	-	-	-
GPT-Neo	2.7B	3.90	0.57	0.00	1.12	5.50	0.80	0.00	1.58	-	-	-	-
GPT-J	6B	5.60	1.00	0.50	1.82	9.20	1.73	1.00	3.08	-	-	-	-
CodeRL+CodeT5	770M	**6.77**	**1.80**	**0.69**	**2.57**	**15.27**	**4.48**	**2.36**	**6.21**	**38.10**	**14.33**	**15.70**	**19.36**

(b) Performance by *n@k* with k up to 50000 and $n = \{1, 5\}$

Model	Size	k	*1@k*				*5@k*			
			Intro	Inter	Comp	All	Intro	Inter	Comp	All
Codex	12B	1000	**22.78**	2.64	3.04	6.75	**24.52**	3.23	3.08	7.46
AlphaCode	1B	1000	-	-	-	-	14.36	5.63	4.58	7.17
AlphaCode	1B	10000	-	-	-	-	18.18	8.21	6.65	9.89
AlphaCode	1B	50000	-	-	-	-	20.36	**9.66**	7.75	11.42
CodeRL+CodeT5	770M	1000	16.52	**6.16**	**4.15**	**7.83**	24.49	8.58	**7.82**	**11.61**

Finetuning Setup. Due to the potential large number of trajectories (i.e. $\mathcal{V}^T$) to generate a sequence and the unstable feedback loop between actor and critic [Lillicrap et al., 2015, Wang et al., 2018], we applied imitation learning to first warm-start a pretrained LM model with $\mathcal{L}_{ce}$ only for up to 10 epochs. We then sampled sequences of program from this actor network to train the critic while keeping the parameters of the actor network frozen. For experiments with CodeT5 actor models, we use the CodeT5-small architecture for the critic model, and GPT2-small critic architecture when the actor models are GPT variants. After training the critic, we then apply both $\mathcal{L}_{ce}$ and $\mathcal{L}_{rl}$ with equal weights to finetune the actor network.

Evaluation. We follow [Hendrycks et al., 2021, Chen et al., 2021a] and evaluate the models using the *pass@k* metric, which is the percentage of problems solved by using *k* generated programs per problem. We also follow Li et al. [2022] and use *n@k* metric which only considers a subset of *n* candidates from *k* generated programs per problem. The subset of n candidates are typically selected by a filtering method by passing generated programs through *example tests* given as part of the problem description [Chen et al., 2021a, Li et al., 2022].

For more details of experimental setup, please refer to the Supplementary Material.

4.2 Experimental Results on APPS

Baselines. As reported by Hendrycks et al. [2021], we compared our models with several baselines, including GPT2 [Radford et al., 2019], GPT-Neo [Black et al.], and GPT3 [Brown et al., 2020]. We also compare the results with Codex [Chen et al., 2021a] and AlphaCode [Li et al., 2022]. Note that by default, results of pretrained LMs (except for Codex and GPT3) are from models finetuned on APPS using the standard loss $\mathcal{L}_{ce}$ only. In our ablations, since CodeRL is model-agnostic, we can also integrate it with GPT variants such as GPT-J [Wang and Komatsuzaki, 2021] and GPT-Neo.

Overall Results. Firstly, Table 1a shows that the CodeRL with the CodeT5 model can achieve significant performance gains, outperforming many pretrained LMs of much larger sizes. Specifically, our approach achieved new SOTA results of 2.57% *pass@1*, 6.21% *pass@5*, and 19.36% *pass@1000*. Table 1b shows that when evaluating on a subset of filtered code samples, our CodeRL+CodeT5 can achieve SOTA results of 7.83% *1@k* and 11.61% *5@k*. Note that while CodeRL incurs additional computation cost during inference with CS, our approach only requires much lower k to achieve comparable performance with other models. Specifically, with $k = 1000$ only, our model performance is as good as AlphaCode with much larger generation budget of $k = 50000$.

4.3 Ablation Studies

In this section, for a fair comparison between variants of return estimates and learning objectives, we report the results of *pass@k* where $k = \{1, 5\}$ with beam search decoding. For larger k, we report the results with and without CS procedure.

Impacts of Return Estimates. Table 2 show the results of CodeT5-770M trained by different approaches to estimate returns of code samples. Overall, we report that the CodeRL objective with relative token-level return estimates by our critic model (Model D) can achieve the best performance on *pass@1* and *pass@5*. Secondly, we note that using absolute returns without a baseline (Model B) could lead to the most performance drop, as this approach heavily penalizes all incorrect samples (even though they might still be better than a naive baseline). Thirdly, without a critic model, simply assigning identical rewards to all tokens in a code sample (Model A) is disadvantageous as these return estimates are too restrictive to be used as feedback signals for RL training. Finally, we experimented with a distance-based critic which assumes that token values decay linearly from $t = 1$ to $t = T$ (Model C). The lower performance suggests the benefit of training a critic network to compute the returns.

Table 2: Ablation results with variants of return estimates

#	W^b	$\hat{q}_\phi$	*pass@1*				*pass@5*			
			Intro	Inter	Comp	All	Intro	Inter	Comp	All
A	✓	-	4.60	1.10	0.20	1.62	7.10	1.57	0.40	2.44
B	-	✓	4.00	0.87	0.20	1.36	5.60	1.30	0.20	1.94
C	✓	dist.	4.90	1.03	0.20	1.64	7.80	1.60	0.30	2.58
D	✓	✓	**6.20**	**1.50**	**0.30**	**2.20**	**9.39**	**1.90**	**0.42**	**3.10**

Impacts of Learning Objectives. Table 3 shows the results with different combinations of $\mathcal{L}_{ce}$ and $\mathcal{L}_{rl}$. We experiment with using only $\mathcal{L}_{rl}$ and note the problem of vanishing gradients during finetuning [Ranzato et al., 2016, Bahdanau et al., 2016]. Secondly, we note that by using only $\mathcal{L}_{ce}$ for further finetuning, despite improvement in losses during training time, the model performance indeed degrades during test time. We expect these models are overfitting to the training data. Interestingly, a naive approach of $\mathcal{L}_{ce}$ with synthetic samples W^s, all of which are treated as correct codes with $r(W^s) = 1$, still leads to some performance improvement with GPT-Neo on *pass@5* (but not in other cases). Finally, we found that using both $\mathcal{L}_{ce}$ and $\mathcal{L}_{rl}$ results in a more consistent performance improvement overall on *pass@1* and *pass@5* for the GPT-Neo and CodeT5 models.

Table 3: Ablation results with different learning objectives

$\mathcal{L}_{ce}$	$\mathcal{L}_{rl}$	*pass@1*				*pass@5*			
		Intro	Inter	Comp	All	Intro	Inter	Comp	All
GPT-Neo									
-	-	3.90	0.57	0.00	1.12	5.50	0.80	0.00	1.58
✓	-	2.70	**0.90**	0.10	1.10	5.00	1.43	0.30	1.92
✓(+W^s)	-	2.90	0.80	**0.30**	1.12	5.20	**1.57**	**0.40**	2.06
-	✓	3.30	0.80	0.20	1.18	5.30	**1.57**	0.20	2.04
✓	✓	**4.70**	0.73	**0.30**	**1.44**	**6.58**	1.54	0.18	**2.28**
CodeT5-770M									
-	-	**6.60**	1.03	0.30	2.00	8.80	1.67	**0.70**	2.90
✓	-	4.60	0.93	0.10	1.50	7.00	1.37	0.20	2.26
✓(+W^s)	-	5.10	1.10	0.40	1.76	8.30	1.43	**0.70**	2.66
-	✓	5.00	0.90	**0.50**	1.64	7.60	1.53	0.60	2.56
✓	✓	6.20	**1.50**	0.30	**2.20**	**9.39**	**1.90**	0.42	**3.10**

Impact of Critic Sampling. Table 4 shows the ablation results of critical sampling (CS) during inference. Overall, we found positive impact of CS for improving *pass@200* and *pass@1000* metrics. Interestingly, we observe that CS does not provide a significant gain on the *n@k* metric. Note that $n@k$ measures the solving rate among the subset $\mathcal{F}$ filtered from k samples. As CS will technically increase the size of this subset, the *n@k* metric will consider exponentially larger number of options of n samples than before. This will normalize $n@k$ by a larger pool of n candidate set, resulting in less impact of CodeRL on the results. We recommend additional post-processing steps such as candidate ranking [Cobbe et al., 2021] to improve the *n@k* performance.

Table 4: Ablation results of critic sampling

Metric	Approach	Intro	Inter	Comp	All
pass@200	without CS	26.79	8.73	7.60	12.12
	with CS	29.10	9.67	9.50	13.52
pass@1000	without CS	35.30	13.33	13.60	17.78
	with CS	38.10	14.33	15.70	19.36
1@1000	without CS	16.27	6.00	4.27	7.71
	with CS	16.52	6.16	4.15	7.83

Impacts of Pretraining Approaches for CodeT5. Table 5 reports the results of CodeT5 with different configurations of model sizes, pretraining data, and pretraining objectives.

For a fair comparison, all models are only finetuned with $\mathcal{L}_{ce}$ on APPS. As observed in prior work [Chen et al., 2021a, Austin et al., 2021], scaling up the number of model parameters or the size of the pretraining data can significantly improve model performance of downstream synthesis tasks. We also find that enhancing the pretraining objectives with next token prediction (NTP) is vital for generation tasks, surpassing just masked span prediction (MSP) from the original CodeT5.

Table 5: Ablation results of CodeT5 pretrained model variants

Size	Data	Task	*pass@1*				*pass@5*			
			Intro	Inter	Comp	All	Intro	Inter	Comp	All
60M	CSN	MSP	1.40	0.67	0.00	0.68	2.60	0.87	0.10	1.06
220M	CSN	MSP	2.50	0.73	0.00	0.94	3.30	1.10	0.10	1.34
770M	CSN	MSP	3.60	0.90	0.20	1.30	4.30	1.37	0.20	1.72
770M	+GCPY	MSP	4.30	**1.10**	0.20	1.56	5.60	1.47	0.30	2.06
770M	+GCPY	+NTP	**6.60**	1.03	**0.30**	**2.00**	**8.80**	**1.67**	**0.70**	**2.90**

4.4 Zero-shot Evaluation on MBPP Benchmark

Finally, we test the zero-shot transfer ability of CodeRL on another smaller and simpler program synthesis benchmark MBPP [Austin et al., 2021]. Table 6 reports the results of our CodeRL+CodeT5 on MBPP benchmark compared with finetuned GPT models of up to 137B size. Our CodeRL+CodeT5 (ZS) was trained on APPS and then evaluated on MBPP in a zero-shot setting. We observe that CodeRL with CodeT5 of a much smaller model size yields surprisingly good zero-shot performance, setting a new SOTA result of 63.0% *pass@80* over GPT-137B's 61.4% *pass@80*. This validates the strong zero-shot transfer ability of CodeRL for unseen tasks. More analysis is included in Supplementary.

Table 6: MBPP benchmark results

Model	Size	*pass@80*
GPT	224M	7.2
GPT	422M	12.6
GPT	1B	22.4
GPT	4B	33.0
GPT	8B	40.6
GPT	68B	53.6
GPT	137B	61.4
CodeRL+CodeT5 (ZS)	770M	**63.0**

5 Limitations and Broader Impacts

One limitation of our approach is the computation cost of training critic model to estimate returns in addition to the original LM (actor network). However, in practice, we found that training a good critic model does not require large-scale models to attain a decent performance. For instance, a finetuned critic model initialized from a pretrained GPT-2 (small) can achieve over 75% error prediction accuracy on synthetic samples.

Program synthesis can lead to substantial positive social impacts, e.g., transforming future software developing tools, increasing productivity of developers, and improving accessibility and quality of programming courses. Yet, some risks and bias issues are still worth considering before deploying such models at scale. For example, training data from public github code repos may contain vulnerabilities and the resulting synthesis models may generate programs with weak security measures [Hammond Pearce et al., 2021].

6 Conclusion

We present CodeRL, a novel framework for program synthesis, using deep reinforcement learning to improve pretrained LMs, by exploiting unit test signals in both training and inference stages. Specifically, we introduce an actor-critic training approach to optimize pretrained LMs with dense feedback signals on synthetic code samples. During inference, we propose a new generation procedure with critical sampling, which enables the model to automatically regenerate programs based on feedback from unit tests and critic scores. We integrate CodeRL with the improved CodeT5-large model (770M) and achieve new SOTA results on both the APPS and MBPP benchmarks, surpassing the prior SOTA by massive pretrained LMs of much larger model sizes. Our comprehensive analysis shows that CodeRL achieved consistent improvement upon the conventional pretrained LMs for code generation tasks. CodeRL is a general framework that integrates pretrained LMs and RL holistically for program synthesis, and can be extended and improved in various ways. For example, it can be easily integrated with other better pretrained LMs and can be improved with more fine-grained feedback from the environment, such as feedback received from a static code analyzer.

References

J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

D. Bahdanau, P. Brakel, K. Xu, A. Goyal, R. Lowe, J. Pineau, A. Courville, and Y. Bengio. An actor-critic algorithm for sequence prediction. *arXiv preprint arXiv:1607.07086*, 2016.

M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. *Advances in neural information processing systems*, 28, 2015.

S. Black, G. Leo, P. Wang, C. Leahy, and S. Biderman. Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow, march 2021. *URL https://doi. org/10.5281/zenodo*, 5297715.

T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

R. Bunel, M. Hausknecht, J. Devlin, R. Singh, and P. Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=H1Xw62kRZ`.

M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021a.

Q. Chen, J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu, and C. Le Goues. VarCLR: Variable semantic representation pre-training via contrastive learning. In *International Conference on Software Engineering*, ICSE '22, 2022.

X. Chen, D. Song, and Y. Tian. Latent execution for neural program synthesis beyond domain-specific languages. *Advances in Neural Information Processing Systems*, 34, 2021b.

C. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan. PyMT5: multi-mode translation of natural language and python code with transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9052–9065, Online, Nov. 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.728. URL `https://aclanthology.org/2020.emnlp-main.728`.

K. Cobbe, V. Kosaraju, M. Bavarian, J. Hilton, R. Nakano, C. Hesse, and J. Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR, 2017.

K. Ellis, M. Nye, Y. Pu, F. Sosa, J. Tenenbaum, and A. Solar-Lezama. Write, execute, assess: Program synthesis with a repl. *Advances in Neural Information Processing Systems*, 32, 2019.

Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, Nov. 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL `https://aclanthology.org/2020.findings-emnlp.139`.

S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

K. Guu, P. Pasupat, E. Liu, and P. Liang. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1051–1062, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1097. URL `https://aclanthology.org/P17-1097`.

B. A. Hammond Pearce, B. Tan, B. Dolan-Gavitt, and R. Karri. An empirical cybersecurity evaluation of github copilot's code contributions. *arXiv preprint arXiv:2108.09293*, 2021.

D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436, 2019.

J. Johnson, B. Hariharan, L. Van Der Maaten, J. Hoffman, L. Fei-Fei, C. Lawrence Zitnick, and R. Girshick. Inferring and executing programs for visual reasoning. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2989–2998, 2017.

A. Joulin and T. Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. *Advances in neural information processing systems*, 28, 2015.

V. Konda and J. Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

K. Kurach, M. Andrychowicz, and I. Sutskever. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.

Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

C. Liang, M. Norouzi, J. Berant, Q. V. Le, and N. Lao. Memory augmented policy optimization for program synthesis and semantic parsing. *Advances in Neural Information Processing Systems*, 31, 2018.

P. Liang, M. I. Jordan, and D. Klein. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 639–646, 2010.

T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

C.-Y. Lin. Rouge: A package for automatic evaluation of summaries. *Text Summarization Branches Out*, 2004.

S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *NeurIPS Datasets and Benchmarks*, 2021.

Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

E. Parisotto, A.-r. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

G. Poesia, A. Polozov, V. Le, A. Tiwari, G. Soares, C. Meek, and S. Gulwani. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=KmtVD97J43e.

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020.

M. Ranzato, S. Chopra, M. Auli, and W. Zaremba. Sequence level training with recurrent neural networks. In Y. Bengio and Y. LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.06732.

S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

Z. Ren, X. Wang, N. Zhang, X. Lv, and L.-J. Li. Deep reinforcement learning-based image captioning with embedding reward. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 290–298, 2017.

S. J. Rennie, E. Marcheret, Y. Mroueh, J. Ross, and V. Goel. Self-critical sequence training for image captioning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7008–7024, 2017.

P. D. Summers. A methodology for lisp program construction from examples. *Journal of the ACM (JACM)*, 24(1):161–175, 1977.

R. S. Sutton. *Temporal credit assignment in reinforcement learning*. PhD thesis, University of Massachusetts Amherst, 1984.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

D. Trivedi, J. Zhang, S.-H. Sun, and J. J. Lim. Learning to synthesize programs as interpretable and generalizable policies. *Advances in Neural Information Processing Systems*, 34:25146–25163, 2021.

R. Vedantam, C. Lawrence Zitnick, and D. Parikh. Cider: Consensus-based image description evaluation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4566–4575, 2015.

R. J. Waldinger and R. C. Lee. Prow: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*, pages 241–252, 1969.

B. Wang and A. Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

X. Wang, W. Chen, J. Wu, Y.-F. Wang, and W. Y. Wang. Video captioning via hierarchical reinforcement learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4213–4222, 2018.

Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP (1)*, pages 8696–8708. Association for Computational Linguistics, 2021.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

X. Xu, C. Liu, and D. Song. SQLNet: Generating structured queries from natural language without reinforcement learning, 2018. URL `https://openreview.net/forum?id=SkYibHlRb`.

P. Yin and G. Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1041. URL `https://aclanthology.org/P17-1041`.

V. Zhong, C. Xiong, and R. Socher. Seq2SQL: Generating structured queries from natural language using reinforcement learning, 2018. URL `https://openreview.net/forum?id=Syx6bz-Ab`.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] See Section 3 and 4
   (b) Did you describe the limitations of your work? [Yes] See Section 5.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See Section 5.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] See the Supplementary.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See the Supplementary.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No] As it is very expensive to experiment with large-scale language models, we did not try different random seeds due to the limitation of computation resources.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See the configurations in the Supplementary.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] See the Supplementary.
   (b) Did you mention the license of the assets? [Yes] See the Supplementary.
   (c) Did you include any new assets either in the supplemental material or as a URL? [No] We do not curate any new dataset in this paper. We will release the code and models.

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] All datasets evaluated in our experiments are publicly available for use.

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] The data we are using are code samples from public programming competitions which do not include personally identifiable information or offensive content.

5. If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]