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

Hung Le[*], **Yue Wang**[*], **Akhilesh Deepak Gotmare**, **Silvio Savarese**, **Steven C.H. Hoi** [†]
Salesforce Research
https://github.com/salesforce/CodeRL

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 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

[*]Equal contribution.
[†]Corresponding authors: {hungle, shoi}@salesforce.com

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

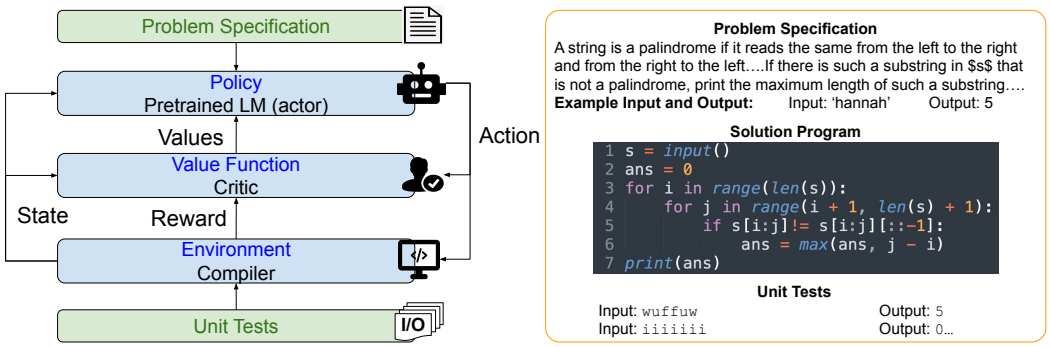

Figure 1: **An example program synthesis task (Right):** Each task is defined by a problem specification in natural language, often containing example input and output pairs. The expected output is a program to be checked for functional correctness against some unit tests. **A high-level overview of our CodeRL framework for program synthesis (Left):** we treat a pretrained code language model (LM) as a stochastic policy, code generations as actions, and rewards can be estimated based on the unit test results of output programs from the compiler (environment).

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 reinforcement 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 model to filter and select sub-sequences. These sub-sequences are utilized as seeds that 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

2.1 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 a 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]. Recently, Nijkamp et al. [2022] proposed a conversational program synthesis approach with large pretrained language models. Despite impressive results in basic programming problems and initial commercial deployment[3], existing models still perform poorly against complex problems such as those from programming competitions on Codeforces [Hendrycks et al., 2021, Li et al., 2022].

Program Synthesis in Visual Context. Another related line of research is program synthesis in computer vision domains such as images and videos. Early papers such as [Kulkarni et al., 2015, Yang et al., 2015] introduce inverse graphics networks to infer visual properties such as pose, shape, and lighting, of visual objects. Wu et al. [2017], Liu et al. [2019], Ellis et al. [2018] study the problem of image rendering, which transforms an image to structured and compact representations, i.e. *scene programs*. Tian et al. [2019] extends the prior work to render 3D shapes from images through *shape programs*, containing features to capture geometric and structural priors. Ganin et al. [2018] introduces an RL-based approach to render realistic images through high-level graphics programs. Sun et al. [2018]introduces program synthesis from demonstration synthetic videos to summarize the behaviors of the objects in the videos.

While this line of research has remarkable impacts on applications such as image/video editing, captioning, and extrapolating, these approaches are limited to programs of domain-specific languages defined for visual objects. For instance, in [Sun et al., 2018], programming language contains basic functions for object perception, action, and control flows. In our work, we focus on program synthesis from natural language problem specifications and the output programs are in general-purpose languages such as Python. This type of programming task can range from basic programming problems to competition-level programming tasks that require a high level of problem-solving skills.

2.2 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 the 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] `https://copilot.github.com/`

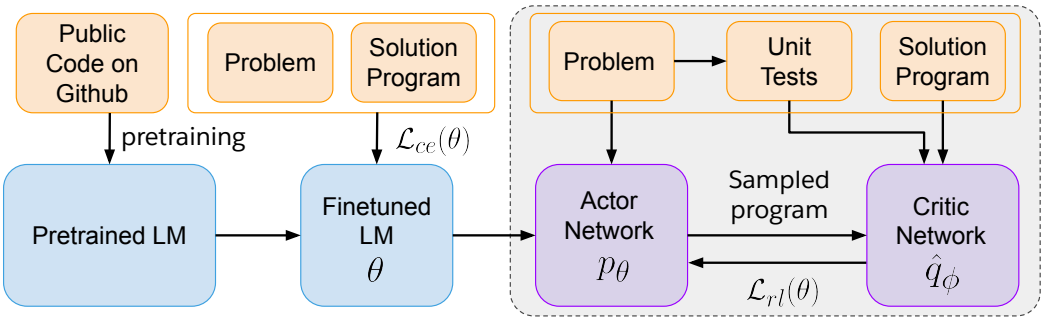

Figure 2: **Overview of our actor-critic framework to optimize pretrained LMs for program synthesis:** We treat the LM as an actor network and sample synthetic samples from this actor. Another neural network is trained as a critic model to evaluate these synthetic samples based on their probabilities of passing unit tests. The returns are estimated based on critic scores and finally factored into the learning objective $\mathcal{L}_{rl}$ to finetune the actor LM network using synthetic samples.

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 $\hat{W} = (\hat{w}_1, ..., \hat{w}_T), \hat{w}_t \in \mathcal{V}$ [4] 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 $\hat{w}_t \sim \text{softmax}(\text{Linear}(s_t))$ 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. 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[5]. We have compiled public, non-personal information from GitHub consisting of permissively licensed Python code ("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].

[4]For simplicity, we use T as the notation of sequence length for all sequences which can actually be variable.
[5]`https://huggingface.co/datasets/lvwerra/github-code`

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 the 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 Figure 2) and generation procedure (see Figure 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 $\nabla_\theta L(\theta)$ 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 an 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 character can render a program erroneous. Therefore, Eq. (8) becomes too restrictive. A straightforward solution

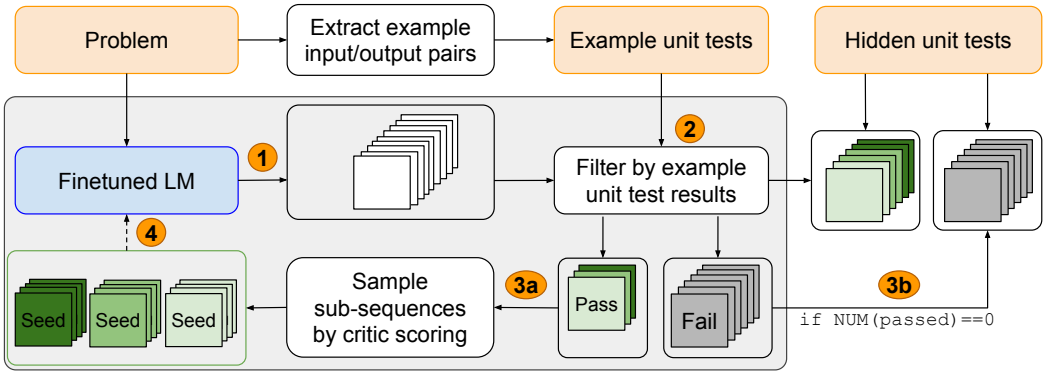

Figure 3: **Overview of our Critic Sampling (CS) approach:** (1) For each test problem, we first use finetuned LM to generate a set of solution programs. (2) From the problem description, we extract example unit tests and test against generated solutions. (3) If there are any passed solutions, we pass them to the critic model to sample sub-sequences. (4) The sub-sequences from (3) are used as seed sequences to condition the LM to regenerate solution programs, repeating the steps from (1) onward.

is to use token-based similarity scores [Papineni et al., 2002, Ren et al., 2020]) between each subsequence $W^s_{1:t}$ 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 (see Appendix A for an overview). The critic model is parameterized as a neural network with parameters ϕ that receives inputs as the problem description D and a sampled program $W^s = \{w^s_1, \ldots, w^s_T\}$. 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^s_t *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, $\hat{q}_\phi(w^s_t) = \hat{v}_t[u]$ where $\hat{v}[.]$ denotes the probability of a specific unit test outcome from the four possible ones. We use this estimate to train the actor LM model with intermediate returns:

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

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 and Appendix B for step-by-step explanation). 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 "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 costs 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 Appendix D).

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.

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 programs 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 the experimental setup, please refer to Appendix C.

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 a much larger generation budget of $k = 50000$.

Table 1: **Results on APPS:** Overall, CodeRL can bring the performance gains on CodeT5 models and achieves new SOTA on both *pass@k* and *n@k* metrics. "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	25.00	9.27	8.80	12.32
GPT-Neo	2.7B	3.90	0.57	0.00	1.12	5.50	0.80	0.00	1.58	27.90	9.83	11.40	13.76
GPT-J	6B	5.60	1.00	0.50	1.82	9.20	1.73	1.00	3.08	35.20	13.15	13.51	17.63
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**

Table 2: **Ablation results with variants of return estimates:** CodeT5 model that is trained with return estimates using a baseline (W_b) and a critic-based return estimates ($\hat{q}_\theta$) can achieve the best performance. "dist." indicates a rule-based approach that estimates returns following a linear decay by token positions from $t = 1$ to $t = T$.

#	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**

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 the 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 rather than relying on rule-based approaches.

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

Table 3: **Ablation results with different learning objectives:** We experiment with both CodeT5 and GPT-Neo with different combinations of cross-entropy loss $\mathcal{L}_{ce}$ and reinforcement learning loss $\mathcal{L}_{rl}$. Note that these losses are applied on models that are already warm-started with conventional cross-entropy losses for up to 10 epochs.

$\mathcal{L}_{ce}$	$\mathcal{L}_{rl}$	*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**

Table 4: **Ablation results of Critic Sampling (CS):** Overall, using CS can lead to higher passing rates as program generations are conditioned on seed sequences filtered by their test results.

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

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.

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. In addition, the positive impacts of critic sampling on *pass@1* and *pass@5* are indicated by comparing the results of Table 1a and Table 2 (row D, in which we only used conventional beam search decoding without critic sampling). We can observe that, using critic sampling, model performance increases from 2.2% *pass@1* (3.1% *pass@5*) to 2.57% *pass@1* (6.21% *pass@5*). Interestingly, from Table 4, 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 an 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.

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 the 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.

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 5: **Ablation results of CodeT5 pretrained model variants:** We report the results of models pretrained on different configurations by model size, pretraining data, and pretraining task. CSN: CodeSearchNet, GCPY: Github Code Python, MSP: Masked Span Predition, NTP: Next Token Prediction. For a fair comparison, all models are finetuned only with $\mathcal{L}_{ce}$ on APPS.

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**

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.

For additional experiments, analysis, and qualitative results, please refer to Appendix D.

Table 6: **Results on MBPP:** we test the zero-shot transfer ability of CodeRL. CodeRL+CodeT5 (ZS) which was trained on APPS with $\mathcal{L}_{rl}$ and evaluated on MBPP [Austin et al., 2021] in a zero-shot setting, achieves new SOTA.

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 the productivity of developers, and improving the 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, D. Ritchie, A. Solar-Lezama, and J. Tenenbaum. Learning to infer graphics programs from hand-drawn images. *Advances in neural information processing systems*, 31, 2018.

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`.

Y. Ganin, T. Kulkarni, I. Babuschkin, S. A. Eslami, and O. Vinyals. Synthesizing programs for images using reinforced adversarial learning. In *International Conference on Machine Learning*, pages 1666–1675. PMLR, 2018.

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.

T. D. Kulkarni, W. F. Whitney, P. Kohli, and J. Tenenbaum. Deep convolutional inverse graphics network. *Advances in neural information processing systems*, 28, 2015.

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.

Y. Liu, J. Wu, Z. Wu, D. Ritchie, W. T. Freeman, and J. B. Tenenbaum. Learning to describe scenes with programs. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=SyNPk2R9K7.

I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *ICLR (Poster)*. OpenReview.net, 2019.

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.

E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

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.

S.-H. Sun, H. Noh, S. Somasundaram, and J. Lim. Neural program synthesis from diverse demonstration videos. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4790–4799. PMLR, 10–15 Jul 2018. URL `https://proceedings.mlr.press/v80/sun18a.html`.

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.

Y. Tian, A. Luo, X. Sun, K. Ellis, W. T. Freeman, J. B. Tenenbaum, and J. Wu. Learning to infer and execute 3d shape programs. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=rylNH20qFQ`.

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.

J. Wu, J. B. Tenenbaum, and P. Kohli. Neural scene de-rendering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

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`.

J. Yang, S. E. Reed, M.-H. Yang, and H. Lee. Weakly-supervised disentangling with recurrent transformations for 3d view synthesis. *Advances in neural information processing systems*, 28, 2015.

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 Appendix D and code at `https://github.com/salesforce/CodeRL`

(b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Appendix C.

(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 Appendix C.

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 Appendix C.

(b) Did you mention the license of the assets? [Yes] See Appendix C.

(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]

A Overview of Critic Model

Figure 4 shows an overview of our critic model. In our CodeRL framework, besides the actor LM network θ, we introduce a critic model that is trained as an error predictor and parameterized as a neural network with parameters ϕ. The critic receives as inputs a problem description D and a corresponding synthetic program W^S sampled from the actor. The critic is required to output a prediction of the unit test outcome of the input program. We define 4 possible outcomes u: *CompileError*, *RuntimeError*, *FailedTest*, and *PassedTest*. The critic model is trained by minimizing the following loss:

$$\mathcal{L}_{critic}(\phi) = -\log p_\phi(u|W^s, D) \tag{11}$$

The ground-truth outcome of a synthetic sample is obtained by passing it to the unit tests corresponding to the problem. Note that since our critic model is applied in a supervised learning environment with available ground truth, we also use the training samples from the original dataset with ground truth output $u =$ *PassedTest* to train the critic.

The learned hidden state representations of program tokens when passed through the critic are then used to measure their return estimates for our RL optimization objective. The return estimates are incorporated as intermediate returns at decoding steps to compute the expected gradient of the actor network $\nabla_\theta \mathcal{L}_{rl}(\theta)$.

B Critic Sampling Procedure

Refer to Algorithm 1 for a step-by-step explanation of our critic sampling procedure.

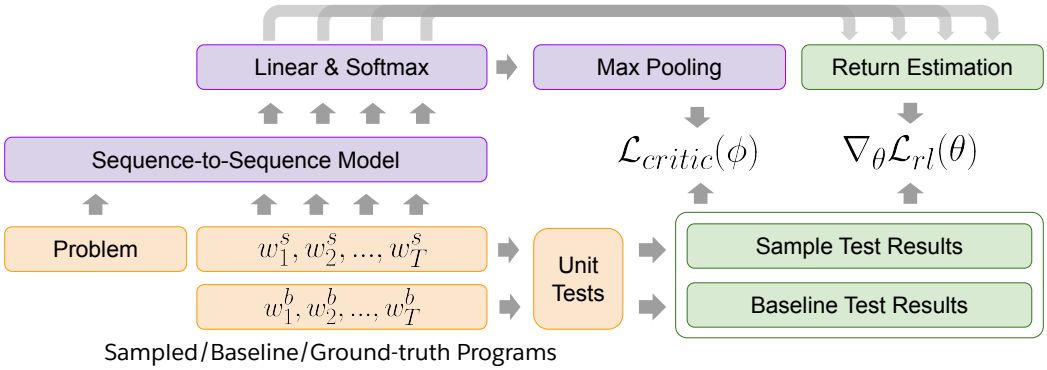

Figure 4: **Overview of our critic model:** The critic receives problem specifications and programs as inputs. For each program, the critic predicts one of four possible test outcomes: $\{\text{CompileError}, \text{RuntimeError}, \text{FailedTest}, \text{PassedTest}\}$. The learned hidden state representations are then used to estimate the returns of synthetic samples to finetune the actor network. To improve and stabilize the training process, baseline programs are used to compute relative return estimates.

C Additional Experimental Setup Details

Pretraining Setup. For CodeT5, we adopt the code-specific tokenizer as described by Wang et al. [2021]. Note that we employ 6 programming languages (PLs) in CodeSearchNet [Husain et al., 2019] (CSN) instead of 8 PLs in CodeT5 as C/C# datasets are not publicly available. We employ only the pretraining task of masked span prediction (MSP) in CodeT5 and hence, we do not have to parse programs into abstract syntax trees (ASTs) to obtain the identifier information. This preprocessing step was required in other original pretraining tasks like masked identifier prediction [Wang et al., 2021]. To further speed up training, we concatenate data samples to batch size 512 for pretraining with MSP and the resulting number of tokens is 1.1B.

APPS Benchmark. We follow the same preprocessing step as in [Hendrycks et al., 2021] to formulate the input sequences from problem descriptions. APPS consists of 10,000 coding problems with a 50-50 train-test split. Each problem is accompanied by 23.2 correct Python programs and 21.2 unit tests on average. The average length per problem is 293.2 words and the average length per program is 18.0 lines. The dataset is categorized into three levels of difficulty: Introductory (3639, train/test=2639/100), Interview (5000, train/test=2000/3000), and Competition (1361, train/test=361/1000). Similarly to [Hendrycks et al., 2021], we employ the strict accuracy metric to evaluate the functional correctness of a program, where it is counted as correct if it can pass all the unit tests corresponding to the problem.

MBPP Benchmark. We additionally include another smaller and simpler Python program synthesis dataset called MBPP [Austin et al., 2021] (Mostly Basic Programming Problems) for evaluation. The dataset contains 974 instances with 374/90/500 instances for training/validation/testing respectively and 10 reserved for few-shot learning. The problems are typically short, usually one sentence of natural language descriptions each. Each problem is accompanied by 1 correct solution (6.8 lines of code on average) and 3 unit tests in the form of `assert` statements for validating the functional correctness. Unlike APPS, unit tests in MBPP are not hidden and are explicitly incorporated into the source sequences for program synthesis models. This might encourage models to be overfitting to these `assert` statements via hard-coding an if-expression very occasionally. However, for a fair comparison with the baselines, we construct the source sequences in the same way as in prior work. Specifically, we adopt the same prompt format as in [Austin et al., 2021] to prepare the input sequence as: problem descriptions + "Your code should satisfy these tests:" + 3 assert statements.

Finetuning Setup. Following [Bahdanau et al., 2016], since our RL method is applied in a supervised learning task, in addition to synthetic programs, we also use the ground-truth programs of training samples to train the critic. These samples are considered perfect programs and always have a label of *PassedTest*. To optimize the LM actor network, in practice, following previous work

Algorithm 1: Critic sampling procedure to generate programs

Input: Problem D, Language model θ, Critic model ϕ
Output: A set of N generated solution programs

begin
  `/* step 1: generate program solutions by LM */`
  a set of output solutions $\mathcal{S} \longleftarrow \emptyset$
  for N **times do**
    `/* use sampling decoding to generate programs */`
    Program $\hat{W}_i \longleftarrow$ Sampling $p_\theta(W_i|D) \longleftarrow$ Sampling $p_\theta(w_t^i|w_{1:t-1}^i, D)$
    $\mathcal{S} \longleftarrow \mathcal{S} \cup \{\hat{W}_i\}$
  `/* step 2: extract example unit tests from problem */`
  example unit tests $\mathcal{I} \longleftarrow \text{ExtractExampleInputOutput}(D)$
  `/* step 3: filter for passed programs and pass to the critic model to select sub-sequences */`
  a set of filtered programs $\mathcal{F} \leftarrow \emptyset$
  for each generated program $\hat{W}_i \in \mathcal{S}$ **do**
    Testoutcomes $u_i \longleftarrow \text{RunTests}(\hat{W}_i, \mathcal{I})$
    if $u_i =$ "PassedTest" **then**
      $\mathcal{F} \longleftarrow \mathcal{F} \cup \{\hat{W}_i\}$
  `/* If no passed samples, return all current programs for evaluation */`
  if $|\mathcal{F}| = 0$ **then return** $\mathcal{S}$
  upsampling param $N^{'} = N/|\mathcal{F}|$
  a set of output solutions $\mathcal{S}^{'} \longleftarrow \emptyset$
  for each filtered program $\hat{W}_i \in \mathcal{F}$ **do**
    subsequence $W_i^{sub} \longleftarrow$ Sampling $p_\phi(u_i = \text{PassedTest}|\hat{W}_i, D)$
      $\longleftarrow$ Sampling $p_\phi(u_i = \text{PassedTest}|\hat{w}_{1:t-1}, D)$
    `/* step 4: subsequence as seeds to regenerate programs */`
    length of subsequence $m \longleftarrow |W_i^{sub}|$
    for $N^{'}$ **times do**
      $\hat{W}_j \longleftarrow$ Sampling $p_\theta(W_j|W_i^{sub}, D) \longleftarrow$ Sampling $p_\theta(w_t^j|w_{m:t-1}^j, W_i^{sub}, D)$
      $\mathcal{S}^{'} \longleftarrow \mathcal{S}^{'} \cup \{\hat{W}_j\}$
  `/* return the regenerated programs for evaluation */`
  return $\mathcal{S}^{'}$

[Bahdanau et al., 2016, Rennie et al., 2017, Wang et al., 2018], in each training optimization step, we can simply approximate the expected gradient with a single sample $W_s \sim p_\theta$:

$$\nabla_\theta \mathcal{L}_{rl}(\theta) \approx -(r(W^s) - r(W^b)) \sum_t \hat{q}_\phi(w_t^s) \nabla_\theta \log p_\theta(w_t^s|w_{1:t-1}^s, D) \tag{12}$$

Configurations. For pretraining, we perform our experiments on a Kubernetes with 16 A100-40G GPUs on Google Cloud Platform and the total pretraining duration is around 21 days. In the first pretraining stage with MSP, we employ a corruption rate of 15%, a peak learning rate (LR) of 2e-4, and a batch size of 2048. We pretrain on CSN for 150 epochs (10 days) and then on GCPY for 10 epochs (5 days). For the second stage pretraining with NTP, we adopt a peak LR of 1e-4 and a batch size of 256, and pretrain for 10 epochs (6 days). We set the maximum length to 768 and 600 for source and target sequences respectively for this objective. For all experiments, we employ an AdamW optimizer [Loshchilov and Hutter, 2019] with a 0.05 weight decay and a linear decay LR scheduler with a warmup step of 1000.

For finetuning on APPS, we adopt a batch size of 64 and warmup LR from 0 to 2e-5 for the first 500 steps and polynomially (power=0.5) decay to 1e-5 until the end of 10 epochs, which takes around 30

Table 7: **CodeT5 results on CodeXGLUE:** Code-to-Text generation results (smoothed BLEU-4) of CodeT5 pretrained with larger data, improved learning objectives, and larger model size

Model	Ruby	JavaScript	Go	Python	Java	PHP	Overall
RoBERTa	11.17	11.90	17.72	18.14	16.47	24.02	16.57
CodeBERT	12.16	14.90	18.07	19.06	17.65	25.16	17.83
DOBF	-	-	-	18.24	19.05	-	-
PLBART	14.11	15.56	18.91	19.30	18.45	23.58	18.32
CoTexT	14.02	14.96	18.86	19.73	19.06	24.58	18.55
CodeT5-small	14.87	15.32	19.25	20.04	19.92	25.46	19.14
CodeT5-base	15.24	16.16	19.56	20.01	20.31	26.03	19.55
CodeT5-large	**15.58**	**16.17**	**19.69**	**20.57**	**20.74**	**26.49**	**19.87**

hours on one A100 GPU. We set the maximum source and target sequence length to 600 and 512 respectively.

To train the critic model (either a GPT2-small or a CodeT5-base), we train the models with synthetic sampled programs for up to 10 epochs. We found that in this phase, the critic model usually converges quite earlier than the 10-epoch training limit. Once the critic model is trained, we continue to finetune the actor language model for 10 more epochs using both the conventional NTP loss and RL-based loss.

For MBPP, in finetuning experiments with CodeT5, due to its small training set, we finetune the models for 60 epochs with a constant LR of 2e-5 and a batch size of 32, which takes less than 30 mins on one A100. We set its maximum source and target length to 382 and 306 respectively.

Critic Sampling. Note that in CS, many rounds of regeneration can be run to keep improving the output programs. However, in practice, we found that one round of regeneration is sufficient to obtain good programs. In the initial generation, if no samples pass example unit tests, we do not apply critic sampling and simply use all available programs for evaluation.

D Additional Experimental Results

D.1 CodeXGLUE Benchmark Results

To validate the effectiveness of our simplified pretraining strategies of CodeT5-large, we extensively evaluate it on a variety of generation tasks in CodeXGLUE [Lu et al., 2021], including code-to-text generation (i.e. summarization, see Table 7), text-to-code generation (see Table 8), and code-to-code generation (i.e., code translation and code refinement, see Table 9). Different from APPS [Hendrycks et al., 2021] and MBPP [Austin et al., 2021], we follow the default similarity-based evaluation metrics in the CodeXGLUE benchmark, including BLEU [Papineni et al., 2002] and CodeBLEU [Ren et al., 2020], and exact match (EM) scores. Table 7, 8, and 9 show that our simplified pretrained CodeT5-large sets new SOTA results on a large majority of the tasks, and hence, can be served as a better foundation model for other code-related generation tasks. Note that in these experiments, we employ the conventional finetuning objective with $\mathcal{L}_{ce}$ and there might be potential to improve the performance further with our CodeRL framework.

D.2 MBPP Benchmark Results

Following Austin et al. [2021], we adopt temperature sampling to generate multiple candidate solutions. We empirically find that CodeT5 benefits from a higher temperature of 1.2 (less greedy decoding or more diverse) than their GPT's temperature of 0.5 on this benchmark.

Table 10 reports the *pass@80* and *pass@1000* results for both finetuned and zero-shot settings. For baselines, we compared with GPT models with sizes ranging from 224M to 137B [Austin et al., 2021], which are pretrained on 2.93B web documents (13.8M containing source code) using standard language modeling objective. Results of GPT models are obtained from the original authors. From the comparison among various CodeT5 variants, we again confirm that larger model sizes and pretraining data, and better pretraining objective of NTP all lead to a performance boost. Particularly,

Table 8: **CodeT5 results on CodeXGLUE:** Text-to-Code generation results of CodeT5 pretrained with larger data, improved learning objectives, and larger model size

Model	EM	BLEU-4	CodeBLEU
GPT-2	17.35	25.37	29.69
CodeGPT-2	18.25	28.69	32.71
CodeGPT-adapted	20.10	32.79	35.98
PLBART	18.75	36.69	38.52
CoTexT	20.10	37.40	40.14
UniXcoder	22.60	38.23	-
CodeT5-small	21.55	38.13	41.39
CodeT5-base	22.30	40.73	43.20
CodeT5-large	**22.65**	**42.66**	**45.08**

Table 9: **CodeT5 results on CodeXGLUE:** Code-to-Code generation results of CodeT5 pretrained with larger data, improved learning objectives, and larger model size

Model	Java to C#		C# to Java		Refine Small		Refine Medium	
	BLEU-4	EM	BLEU-4	EM	BLEU-4	EM	BLEU-4	EM
Naive copy	18.54	0.00	18.69	0.00	78.06	0.00	90.91	0.00
Roborta (code)	77.46	56.10	71.99	57.90	77.30	15.90	90.07	4.10
CodeBERT	79.92	59.00	72.14	58.00	77.42	16.40	91.07	5.20
GraphCodeBERT	80.58	59.40	72.64	58.80	**80.02**	17.30	**91.31**	9.10
PLBART	83.02	64.60	78.35	65.00	77.02	19.21	88.50	8.98
CoTexT	-	-	-	-	77.79	21.03	88.40	13.11
NSEdit	-	-	-	-	71.06	**24.04**	85.72	13.87
CodeT5-small	82.98	64.10	79.10	65.60	76.23	19.06	89.20	10.92
CodeT5-base	**84.03**	65.90	**79.87**	66.90	77.43	21.61	87.64	13.96
CodeT5-large	83.56	**66.00**	79.77	**67.00**	77.38	21.70	89.22	**14.76**

our CodeT5-770M yields a *pass@80* of 46.8%, surpassing GPT-8B's 40.6% with a much smaller model size. In addition, we find CodeT5 models finetuned on APPS can achieve a surprisingly good zero-shot performance on MBPP with a *pass@80* of 60.2% and further improved to 63.0% with the help of CodeRL, which even outperforms the largest GPT-137B's performance of 61.4%. This indicates APPS is a comprehensive program synthesis benchmark and CodeT5+CodeRL models trained on it are able to generalize to other simpler coding tasks. If we further increase the budget of attempts up to 1000, all models witness a consistent and significant boost in solving rate, especially our CodeT5+CodeRL yielding a new SOTA result of 81.8% *pass@1000*.

A common concern about transfer learning is that the source (APPS) and target (MBPP) tasks might overlap in their training data, which could result in the source model tending to memorize these substantially similar data when applied to the target task. To address this concern, we analyze how many lines of code appear in both the training set of APPS and the programs of MBPP following Austin et al. [2021]. For this analysis, we discard code comments and normalize the whitespaces for each line, and then exclude lines that appear more than twice anywhere in MBPP, as these are likely to be common Python keywords such as `return` and `break`.

Figure 5 illustrates the number of absolute duplicated lines (Left) and the relative fraction of duplicated lines (Right) in the MBPP programs. As can be seen, the overlap between APPS and MBPP seems to be minimal. Only 12.6% MBPP programs have more than half of their lines matched somewhere in the APPS training data. Besides, more than half (514 out of 974) of programs have a zero overlap and 90.9% have only no more than 3 lines overlapping with the APPS training set. If we further require the lines to be consecutive, there are no more than 2 consecutive duplicated lines.

D.3 APPS Benchmark Results on Competition-level Tasks

Figure 6 shows the results of *pass@k* and *n@k* with k ranging from 1 to 200 and $n = \{1, 5\}$, for CodeRL+CodeT5 and CodeT5 only. We choose to investigate a subset of the APPS test split, which contains the test samples of the highest difficulty level (i.e. competition programming tasks). Since CodeRL is model-agnostic, we also integrate it with GPT-J [Wang and Komatsuzaki, 2021] and report

Table 10: **Ablation results on MBPP:** Ablation results of CodeRL with different CodeT5 model variants with different sizes, pretraining data and objectives on MBPP. CodeT5† is finetuned on APPS and evaluated on MBPP in a zero-shot setting.

Model	Size	Data	Objective	*pass@80*	*pass@1000*
GPT finetuned results					
GPT	224M	Web Doc	LM	7.2	-
GPT	422M	Web Doc	LM	12.6	-
GPT	1B	Web Doc	LM	22.4	-
GPT	4B	Web Doc	LM	33.0	-
GPT	8B	Web Doc	LM	40.6	-
GPT	68B	Web Doc	LM	53.6	-
GPT	137B	Web Doc	LM	61.4	-
CodeT5 finetuned results					
CodeT5	60M	CSN	MSP	19.2	36.2
CodeT5	220M	CSN	MSP	24.0	42.8
CodeT5	770M	CSN	MSP	32.4	47.8
CodeT5	770M	+GCPY	MSP	34.6	51.6
CodeT5	770M	+GCPY	+NTP	46.8	66.2
CodeRL zero-shot results					
CodeT5†	770M	+GCPY	+NTP	60.2	78.4
+CodeRL	770M	+GCPY	+NTP	63.0	81.8

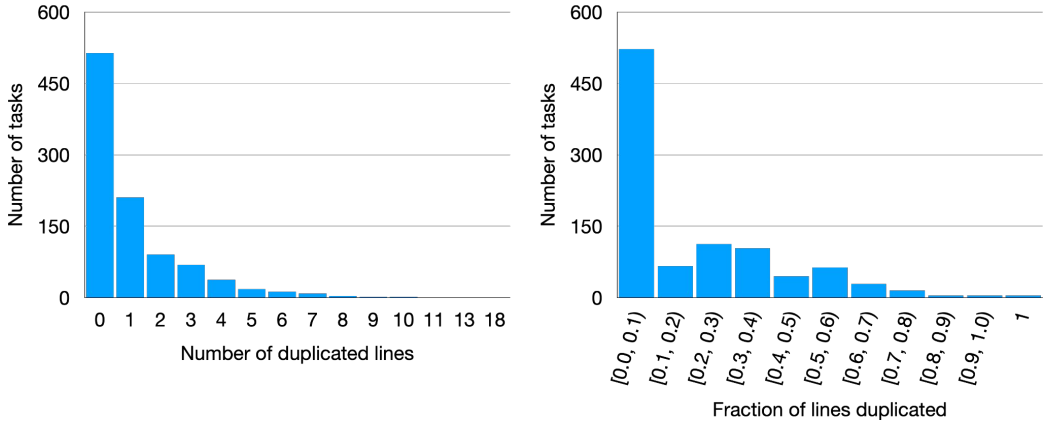

Figure 5: **Analysis of duplicated lines between MBPP and APPS:** The overlap of data between APPS and MBPP appears to be minimal, with only 12.6% MBPP programs with $> 50\%$ lines duplicated in APPS training data.

the results. To focus on the impact of our RL optimization, during test time, we compare models using only nucleus sampling and without the CS procedure. Figure 6 shows that the performance gains are quite consistent on both GPT-J and CodeT5. In particular, as k increases, the performance gain of CodeRL is more significant on both GPT-J and CodeT5 models. We attribute these gains to the CodeRL learning objective $\mathcal{L}_{rl}$ that encourages models to explore code solutions drawn from the model's sampling distribution. During test time with an increasing k sampling budget, models are allowed to generate diverse code solutions and the impact of $\mathcal{L}_{rl}$ becomes more significant.

D.4 CodeT5 Ablation by Training Epochs

Figure 7 shows the performance of CodeT5 model variants by finetuning epochs and by difficulty levels of programming tasks. Note that in these experiments, we only need to compare among CodeT5 model variants by pretraining strategies, and hence, only engage $\mathcal{L}_{ce}$ (imitation learning) in the finetuning stage on APPS. Consistent with our prior analysis (see Section 4 of the main paper), enhancing both pretraining data (with larger data of GCPY) and pretraining objectives (with NTP objective) improves model performance across training epochs in general. Moreover, as noted by our analysis of learning objectives, only using $\mathcal{L}_{ce}$ often leads to overfitting performance, typically after

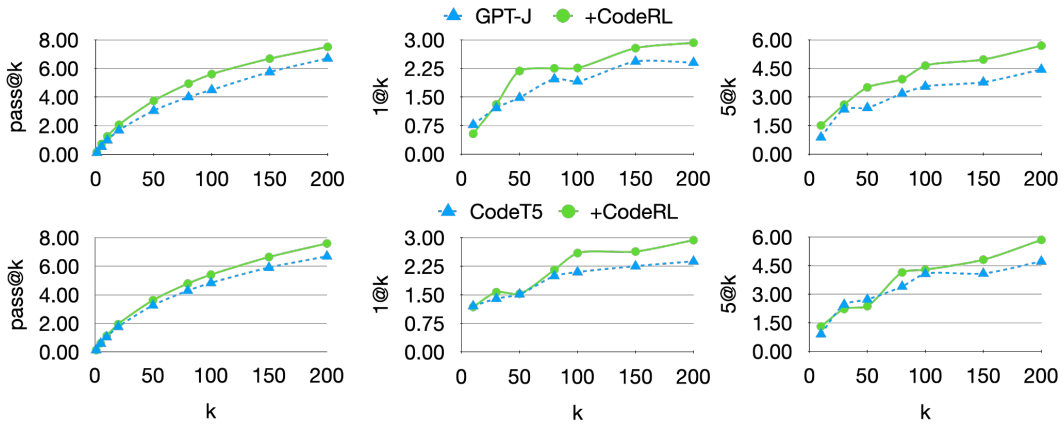

Figure 6: **Results on APPS competition-level test samples:** We investigate the most challenging programming problem tasks, i.e. competition level, in the APPS benchmark. Integrating CodeRL with both CodeT5 and GPT-J, we observe good performance improvement across $pass@k$ and $n@k$ metrics, with increasing performance gains as k increases.

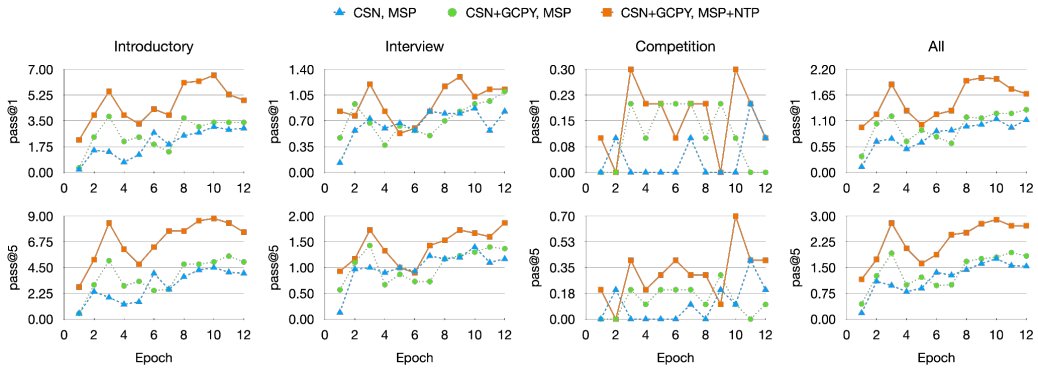

Figure 7: **Ablation results by finetuning epochs**: We report the finetuning progress of CodeT5-770M models that are pretrained on different configurations by pretraining data and pretraining tasks. CSN: CodeSearchNet, GCPY: Github Code Python, MSP: Masked Span Prediction, NTP: Next Token Prediction. All models are finetuned only with $\mathcal{L}_{ce}$ on APPS.

epoch 10 in our case. Hence, to further finetune large-scale LMs, we recommend adopting our RL objective $\mathcal{L}_{rl}$ to utilize synthetic training samples and avoid overfitting models.

D.5 Impacts of CodeRL on Program Quality by Unit Test Signals

Figure 8 demonstrates the average percentages of generated programs by their test signals. Specifically, we use CodeT5 or CodeRL+CodeT5 to generate programs and randomly select 200 generated programs per test sample in the APPS test split. We pass programs to either example unit tests or hidden unit tests corresponding to the problem and group the output programs by their output signals, including CompileError, RuntimeError, FailedTest, and PassedTest. We observe that integrating CodeRL can increase the likelihood that a program can pass unit tests, and reduces the likelihood that it fails one or more unit tests, or whether it contains compiling or runtime errors. However, we note that there are significant gaps in performance by test signals between example unit tests and hidden unit tests. This observation suggests that example tests are not as comprehensive as hidden tests and hence, applying our CS procedure might lead to false positive samples for regeneration. We recommend additional methods to improve example unit tests, such as through data augmentation by mutating input/output pairs [Austin et al., 2021].

Figure 8: **Qualitative results of CodeT5 and CodeT5+CodeRL:** We generate 200 programs per test sample and report the % programs per sample by their unit test signals, including CompileError, RuntimeError, FailedTest, and PassedTest.

D.6 Example Generated Programs and Qualitative Analysis

Figure 9 to 11 show examples of programming problems from the APPS benchmark and corresponding programs generated by CodeT5 variants. Specifically, based on the same foundation pretrained CodeT5 (pretrained with GCPY data and NTP objective), we compare the CodeT5 model that is finetuned by $\mathcal{L}_{ce}$ only and another model that follows our CodeRL framework. In CodeRL+CodeT5, we further show programs before and after applying the CS procedure. The example programs show that applying CodeRL can improve the quality of generated programs and the CS procedure further improves the functional correctness of the programs. For instance, in Figure 11, CodeT5 model misunderstands the problem and focuses on finding the greatest common divisor between a and b only. Instead, the CodeRL model avoids this mistake and tackles the problem based on the *factorials* of a and b. In Figure 10, we note that the CS procedure improves the program by reordering the `elif` code blocks. The resulting program is more correct and is able to pass all hidden unit tests.

We also found that CodeRL can improve the efficiency of the programs, an important quality in complex programming problems. For instance, in the interview-level programs in Figure 11, we note that without applying CS, the generated program is functionally correct but fails during execution due to a timeout error. Applying the CS procedure can condition models on parts of the prior program and (re)generates new tokens to produce a more efficient program. Hence, the resulting final program is able to pass all hidden unit tests (including tests with extremely large values) without timeout errors.

Input Problem

QUESTION:
There is a right triangle ABC with ∠ABC=90°.
Given the lengths of the three sides, |AB|,|BC| and |CA|, find the area of the right triangle ABC.
It is guaranteed that the area of the triangle ABC is an integer.

-----Constraints-----
- 1 \leq |AB|,|BC|,|CA| \leq 100
- All values in input are integers.
- The area of the triangle ABC is an integer.

-----Input-----
Input is given from Standard Input in the following format:
|AB| |BC| |CA|

-----Output-----
Print the area of the triangle ABC.

-----Sample Input-----
3 4 5

-----Sample Output-----
6

This triangle has an area of 6.
Use Standard Input format

ANSWER:

Example Unit Tests

```
Input:   3 4 5
Output:  6
```

Hidden Unit Tests

```
Input:   5 12 13
Output:  30

Input:   24 7 25
Output:  84
...
```

Ground-truth Program

```
L = list(map(int, input().split()))
sortL = sorted(L)
ans = L[0] * L[1] /2
print(int(ans))
```

CodeT5 Output Program

```
ab, bc, ca = map(int, input().split())
print((ab*bc*ca))
```

Example tests: Failed
Hidden tests: Failed

CodeRL+CodeT5 Output Program (before CS)

```
a,b,c = map(int,input().split())
x = (a+b+c)/2
print(x)
```

Example tests: Passed
Hidden tests: Failed

CodeRL+CodeT5 Output Program (after CS)

```
a,b,c = map(int,input().split())
x = (a+b+c)/2
y = (a+b-c)/2
print((x*y))
```

Example tests: Passed
Hidden tests: Passed

Figure 9: **An example synthesis task from the APPS benchmark and corresponding programs generated by CodeT5 variants:** Without the CS generation procedure, CodeRL+CodeT5 model can generate programs that pass all example tests but fail hidden tests. With the CS generation procedure, the model can condition on prior programs and generate a better program that passes all hidden tests.

Input Problem

QUESTION:
Allen has a LOT of money. He has n dollars in the bank. For security reasons, he wants to withdraw it in cash (we will not disclose the reasons here). The denominations for dollar bills are 1, 5, 10, 20, 100. What is the minimum number of bills Allen could receive after withdrawing his entire balance?

-----Input-----

The first and only line of input contains a single integer n ($1 \le n \le 10^9$).

-----Output-----

Output the minimum number of bills that Allen could receive.

-----Examples-----
Input
125

Output
3

Input
43

Output
5

Input
1000000000

Output
10000000

-----Note-----

In the first sample case, Allen can withdraw this with a 100 dollar bill, a 20 dollar bill, and a 5 dollar bill. There is no way for Allen to receive 125 dollars in one or two bills.

In the second sample case, Allen can withdraw two 20 dollar bills and three 1 dollar bills.

In the third sample case, Allen can withdraw 100000000 (ten million!) 100 dollar bills.
Use Standard Input format

ANSWER:

Example Unit Tests

```
Input:   125
Output:  3

Input:   43
Output:  5
...
```

Hidden Unit Tests

```
Input:   74
Output:  8

Input:   82655
Output:  830
...
```

Ground-truth Program

```
1  n = int(input())
2
3  ans = 0
4  ans += n//100
5  n %= 100
6  ans += (n // 20)
7  n %= 20
8  ans += n //10
9  n %= 10
10 ans += n//5
11 n%= 5
12 ans += n//1
13
14 print(ans)
```

Example tests: Passed
Hidden tests: Passed

CodeT5 Output Program

```
1  n = int(input())
2
3  a = defaultdict(int)
4
5  for i in range(1, 101):
6      a[i] = n // i
7      n = n % i
8
9  ans = 0
10 for i in range(1, 101):
11     if n < a[i]:
12         ans += a[i]
13         n -= a[i]
14
15 print(ans)
```

Example tests: Failed
Hidden tests: Failed

CodeRL+CodeT5 Output Program (before CS)

```
1  def main():
2      n = int(input())
3      cnt = 0
4      while n > 0:
5          if n % 100 == 0:
6              n -= 100
7              cnt += 1
8          elif n % 20 == 0:
9              n -= 20
10             cnt += 1
11         elif n % 5 == 0:
12             n -= 5
13             cnt += 1
14         elif n % 10 == 0:
15             n -= 10
16             cnt += 1
17         elif n > 0:
18             n -= 1
19             cnt += 1
20     print(cnt)
21
22
23 def __starting_point():
24     main()
25
26 __starting_point()
```

Example tests: Passed
Hidden tests: Failed

CodeRL+CodeT5 Output Program (after CS)

```
1  def main():
2      n = int(input())
3      cnt = 0
4      while n > 0:
5          if n % 100 == 0:
6              n -= 100
7              cnt += 1
8          elif n % 20 == 0:
9              n -= 20
10             cnt += 1
11         elif n % 10 == 0:
12             n -= 10
13             cnt += 1
14         elif n % 5 == 0:
15             n -= 5
16             cnt += 1
17         else:
18             n -= 1
19             cnt += 1
20     print(cnt)
21
22
23 def __starting_point():
24     main()
25
26 __starting_point()
```

Example tests: Passed
Hidden tests: Passed

Figure 10: **An example synthesis task from the APPS benchmark and corresponding programs generated by CodeT5 variants:** Without the CS generation procedure, CodeRL+CodeT5 model can generate programs that pass all example tests but fail hidden tests, especially those of corner cases. With the CS generation procedure, the model can condition on prior programs and refine the code. Specifically, we observe the model can simply reorder the `elif` blocks between line 11 and 15 to fix the error. The resulting program is functionally correct and passes all hidden tests.

Input Problem

QUESTION:
Holidays have finished. Thanks to the help of the hacker Leha, Noora managed to enter the university of her dreams which is located in a town Pavlopolis. It's well known that universities provide students with dormitory for the period of university studies. Consequently Noora had to leave Vičkopolis and move to Pavlopolis. Thus Leha was left completely alone in a quiet town Vičkopolis. He almost even fell into a depression from boredom!

Leha came up with a task for himself to relax a little. He chooses two integers A and B and then calculates the greatest common divisor of integers "A factorial" and "B factorial". Formally the hacker wants to find out GCD(A!, B!). It's well known that the factorial of an integer x is a product of all positive integers less than or equal to x. Thus x! = 1·2·3·...·(x - 1)·x. For example 4! = 1·2·3·4 = 24. Recall that GCD(x, y) is the largest positive integer q that divides (without a remainder) both x and y.

Leha has learned how to solve this task very effective. You are able to cope with it not worse, aren't you?

-----Input-----

The first and single line contains two integers A and B (1 ≤ A, B ≤ 10^9, min(A, B) ≤ 12).

-----Output-----

Print a single integer denoting the greatest common divisor of integers A! and B!.

-----Example-----
Input
4 3

Output
6

-----Note-----

Consider the sample.

4! = 1·2·3·4 = 24. 3! = 1·2·3 = 6. The greatest common divisor of integers 24 and 6 is exactly 6.
Use Standard Input format

ANSWER:

Example Unit Tests

```
Input:   4 3
Output:  6
```

Hidden Unit Tests

```
Input:  10 399603090
Output: 3628800

Input:   5 5
Output:  120
...
```

Ground-truth Program

```python
a, b = map(int, input().split())
a = min(a, b)
ans = 1
for i in range(1, a + 1):
    ans *= i
print(ans)
```

CodeT5 Output Program

```python
from math import gcd
a, b = map(int, input().split())
c = math.gcd(a,b)
print(c)
```

Example tests: Failed
Hidden tests: Failed

CodeRL+CodeT5 Output Program (before CS)

```python
from math import gcd
a, b = list(map(int, input().split()))
gcd = lambda x, y: x if y == 0 else gcd(y, x % y)

a_prod = 1
b_prod = 1
for i in range(1, a + 1):
    a_prod *= i
for i in range(1, b + 1):
    b_prod *= i

print(gcd(a_prod, b_prod))
```

Example tests: Passed
Hidden tests: RuntimeError (TimeOut)

CodeRL+CodeT5 Output Program (after CS)

```python
from math import gcd
a, b = list(map(int, input().split()))
gcd = lambda x, y: x if y == 0 else gcd(y, x % y)

a_prod = 1
b_prod = 1
for i in range(1, min(a, b) + 1):
    a_prod *= i
    b_prod *= i

print(gcd(a_prod, b_prod))
```

Example tests: Passed
Hidden tests: Passed

Figure 11: **An example interview-level APPS programming task and programs generated by CodeT5 variants:** The program generated by CodeT5 model fails all unit tests while CodeRL+CodeT5 (without CS generation) can generate a functionally correct program. However, this program leads to runtime errors in extreme test cases. After applying the CS generation procedure, the program is improved and able to pass all hidden unit tests.