Adversarial RL for Hard-Negative Code Generation

Eric Li
Department of Computer Science
Stanford University
`ericlij@stanford.edu`

Ella Mao
Graduate School of Business
Stanford University
`ellamao@stanford.edu`

Komei Ryu
Department of Computer Science
Stanford University
`komeiryu@stanford.edu`

Abstract

Large language models for code are typically trained on datasets dominated by correct code, limiting their exposure to realistic buggy examples needed for robust training. In this paper, we propose an adversarial reinforcement learning framework for LLMs to generate hard negative examples - code that is deceptively similar to the correct implementation but reliably fails unit tests. Our approach combines PPO with a GAN-style discriminator, rewarding an actor/generator for failing unit tests while fooling the discriminator and maintaining minimal edit distance from reference solutions. Our PPO training dynamics demonstrate effective, iterative updates between the actor/generator and the discriminator. Evaluating our PPO-fine-tuned Qwen2.5-Coder-0.5B on the MBPP dataset, we show that our method can successfully generate hard negative samples through subtle mutations (single character changes, variable swaps) that achieve high similarity to correct code (low edit distance and high discriminator reward) while maintaining 83-89% test failure rates. Compared to rule-based mutations, our PPO-generated negative code snippets are also better suited as preference pairs for downstream DPO fine-tuning, achieving 0.420 pass@1 rate against unit tests for code synthesis tasks.

1 Introduction

While large language models (LLMs) are widely used for code synthesis tasks, these models are typically trained on internet-scraped repositories where almost every code snippet compiles and passes its unit tests. We refer to such examples as positive samples, as they satisfy certain oracles (e.g., compilation or unit tests). In contrast, negative samples - snippets that fail at least one oracle - are scarcely represented in public corpora. While machine learning and LLMs can generally benefit from hard negative samples via contrastive learning or Direct Preference Optimization (DPO) (Robinson et al., 2020; Shaikh et al., 2025), existing public code corpora provide very few negative samples, leaving LLMs unexposed to realistic buggy code during training. This gap could lead to poor performances or unexpected behaviors when dealing with buggy code during inference.

To address this limitation, we design a learning-based, automated procedure that, given a positive example, produces a companion negative example that (i) looks deceptively similar to the positive example, yet (ii) reliably breaks a chosen oracle. Our proposed procedure couples Proximal Policy Optimization (PPO) with a GAN-style approach. A PPO-driven actor/generator is rewarded whenever the code it generates fails the oracle and gains additional rewards from the normalized edit distance between the reference and generated code as well as from a discriminator that pushes the generator toward edits that remain hard to distinguish from genuine positives. Conditioning the generator on the original positive snippets guarantees tight, informative counter-examples that can be useful for downstream tasks - for example, these negative samples can be paired with positives for DPO or Reinforcement Learning from Human Feedback (RLHF) fine-tuning for LLM code synthesis tasks.

In this work, we implement and evaluate our approach on the Most Basic Python Problems dataset (Austin et al., 2021). By comparing the model-generated code with rule-base-mutated code, our experiments demonstrate that the PPO-trained generator successfully learns to produce high-reward negative code by making small mutations to numbers, variable names, function signatures, and syntax. These generated outputs are not trivially distinguishable from correct code through simple inspection, indicating the promise of our approach. Specifically, at inference time, the outputs of the PPO-trained generator fail unit tests on 88.57% of examples in the training set, and receive an average discriminator reward of 0.1722 and an average edit distance reward of 0.9512 among training examples that fail unit tests. These results demonstrate the ability of the PPO-trained model to output hard negative examples that are very similar to the corresponding positive examples whilst failing unit tests.

Moreover, we demonstrate that these generated outputs are useful for downstream tasks by fine-tuning a model using Direct Preference Optimization (DPO) on a preference dataset that consists of the generated negative examples. The performance of this DPO-fine-tuned model on a code synthesis task outperforms baselines, achieving a pass@1 rate of 0.420 against unit tests on our test set, surpassing the performance of all three of our baselines. This improvement in code synthesis task performance highlights the potential impact of our proposed procedure on downstream tasks.

Prior work touches only fragments of our project's main objective. Efforts that do generate negative code samples rely on hand-crafted mutation rules, limiting diversity and realism. By unifying PPO with an adversarial discriminator, our framework provides an end-to-end, learning-based pipeline for hard-negative code generation, supplying the missing data that modern algorithms can leverage to make LLMs more robust to subtle programming errors.

2 Related Work

Generating truly challenging negatives in the code domain has been shown to benefit downstream tasks such as model fine-tuning and code search (Jain et al., 2020; Shi et al., 2023). However, most prior approaches rely on either hand-crafted, rule-based mutations or retrieving snippets from embedding spaces (Naik et al., 2023; Shi et al., 2023). While these methods can yield reasonable negative examples - by, for instance, replacing tokens or borrowing similar AST structures - they may not be best suited to generate sufficiently difficult negative samples that closely resemble correct code while introducing subtle bugs.

Reinforcement learning (RL) offers a natural way to steer a code generator toward desired behaviors by using functional feedback as a reward signal. Le et al. (2022) pioneered this direction with CodeRL, an offline actor-critic framework where a critic predicts unit-test outcomes and provides dense, token-level reward signals. Similarly, Liu et al. (2023) proposed RLTF, an online fine-tuning setup that instruments unit tests to give granular error feedback. Dou et al. (2024) introduced StepCoder, which further stabilizes exploration through a curriculum of code-completion subtasks. Although these works demonstrate that RL can substantially improve the accuracy and reliability of generated code, they focus exclusively on pushing models toward correctness and do not investigate using RL to intentionally produce hard negatives.

Distinguishing from these existing approaches, our approach fills this gap by leveraging RL to generate seemingly plausible (thus hard) yet failure-inducing (thus negative) code samples. To drive the actor toward generating increasingly challenging bugs, we augment its reward to also fool a discriminator. Integrating discriminator rewards into RL rewards was pioneered by Ho and Ermon (2016), whose work demonstrated that using discriminator outputs as rewards can effectively drive agent behaviors toward a set of target trajectories. We draw insights from this line of adversarial RL frameworks (Fu et al., 2017) and design an adversarial signal to encourage the actor to craft negatives that not only fail oracles (e.g., compilers or unit tests) but also closely resemble correct solutions, creating a dynamic curriculum of hard negatives.

Once a pool of such negatives is generated, it can be reused indefinitely as a dataset in downstream fine-tuning via contrastive objectives or DPO to push a model toward correctness. In contrast, existing methods that directly use RL to achieve similar objectives (i.e., push models toward correctness) must include the compiler, test harness, or a learned reward model in the training loop for each new RL fine-tuning run, incurring repeated time, compute, and engineering overhead.

3 Method

In this section, we introduce our proposed method to fine-tune a LLM to generate *hard negative code* - incorrect solutions that closely resemble correct ones - using GAN-style RL. Our approach leverages Proximal Policy Optimization (Schulman et al., 2017) (PPO) to fine tune an LLM (also referred to as the *actor*, *agent*, or *generator* depending on context) guided by three reward signals: a discriminator reward, an edit distance reward, and a unit test reward. An overview of our method is shown in Figure 1.

Our training builds on a set of coding problems with reference solution code snippets and associated unit tests. Our particular experiment builds on the Most Basic Python Problems (MBPP) dataset (Austin et al., 2021), which supplies all the above components (Section 4.1).

3.1 Supervised Warm Start

Before PPO training, we warm-start the LLM actor/generator via supervised fine-tuning (SFT). We use the reference solution code snippets in the MBPP training set (denoted $\mathcal{D}_{correct}$) as inputs. We apply a set of rule-based mutations on each reference solution code to create the target outputs, making sure that the mutated code fails the associated unit tests. Section 4.2 contains details of the rule-based mutations. We fine-tune the actor/generator to output these buggy mutants given correct code snippets, warm-starting the actor towards generating plausible but faulty solutions.

3.2 PPO Fine-Tuning

We then run PPO fine-tuning on the supervised fine-tuned LLM actor/generator. The actor receives a batch of reference solution code snippets (each denoted as x) as inputs, and generate a batch of corresponding output code snippets (each denoted as y). Each output is evaluated using three reward components: edit distance reward, unit test reward, and discriminator reward.

Unit Test Reward If y fails compilation or any unit test, it receives a reward of 0. Otherwise, it is penalized with a reward of -3. This strongly encourages each generated code to be incorrect.

Edit Distance Reward ($r_{y_{\text{ed}}}$) This reward measures similarity to the reference solution, normalized by the lengths of the reference and generated code snippets. For a generated code y and its paired correct input code snippet x, we compute:

$$r_{y_{\text{ed}}} = \frac{\text{CharOverlap}(x, y)}{\max(\text{len}(x), \text{len}(y))}$$

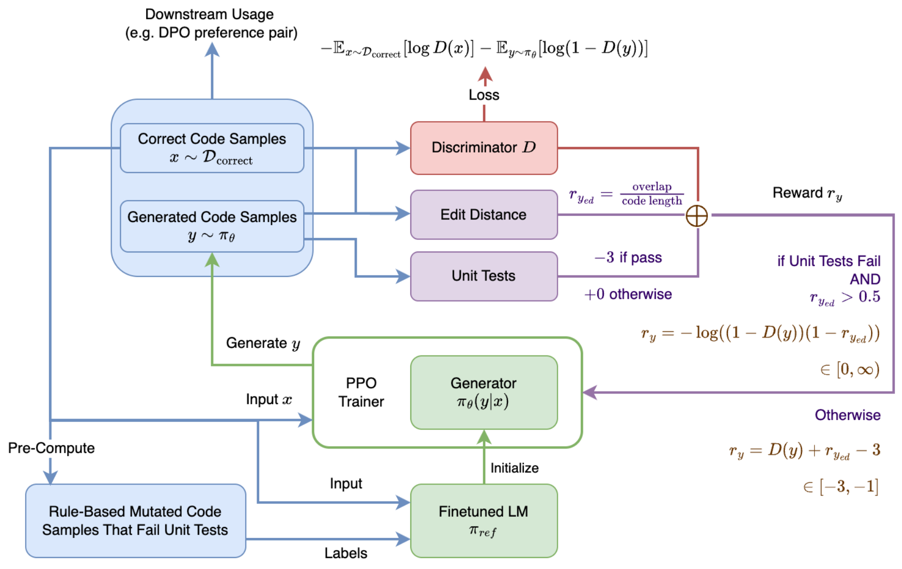
This encourages minimal perturbations from the reference, which is useful under the assumption that minimal perturbations create *hard* negative code snippets.

Discriminator Reward A discriminator D predicts the probability that a code snippet belongs to the set of correct reference solutions $\mathcal{D}_{correct}$. It is trained to minimize the standard GAN loss:

$$\mathcal{L}_D = -\mathbb{E}_{x \sim \mathcal{D}_{\text{correct}}}[\log D(x)] - \mathbb{E}_{y \sim \mathcal{D}_{\text{neg}}}[\log(1 - D(y))]$$

where $\mathcal{D}_{\text{neg}}$ is a buffer containing an incorrect solution for each coding problem. The buffer is initialized with the set of rule-based mutants of the reference solutions (Section 4.2), which we also used earlier in SFT (Section 3.1). During PPO training, we update this buffer by adding actor-generated code that fails unit tests into the buffer. Empirically, we found that sampling negative examples from this buffer produces increasingly challenging inputs for the discriminator and results in more stable discriminator scores across updates.

After each PPO update step, we update the discriminator using the current batch's reference solution code inputs as positive samples and randomly sampled buffer entries as negatives. To stabilize training, we pause discriminator updates if the generator fails to fool it (i.e., average $D(y) < 0.4$ over the past



Downstream Usage (e.g. DPO preference pair)
$-\mathbb{E}_{x\sim\mathcal{D}_{\text{correct}}}[\log D(x)] - \mathbb{E}_{y\sim\pi_\theta}[\log(1-D(y))]$
Loss
Correct Code Samples $x \sim \mathcal{D}_{\text{correct}}$
Generated Code Samples $y \sim \pi_\theta$
Discriminator D
Edit Distance
Unit Tests
$r_{y_{ed}} = \frac{\text{overlap}}{\text{code length}}$
-3 if pass
$+0$ otherwise
Reward r_y
if Unit Tests Fail AND $r_{y_{ed}} > 0.5$
$r_y = -\log((1-D(y))(1-r_{y_{ed}}))$
$\in [0,\infty)$
Otherwise
$r_y = D(y) + r_{y_{ed}} - 3$
$\in [-3,-1]$
Generate y
Input x
PPO Trainer
Generator $\pi_\theta(y|x)$
Initialize
Pre-Compute
Input
Labels
Rule-Based Mutated Code Samples That Fail Unit Tests
Finetuned LM π_{ref}

Figure 1: Method Overview.

10 steps), and resume updates if the generator becomes too successful (i.e., average $D(y) > 0.6$ over 10 steps). This pacing mitigates the imbalance between fast discriminator convergence (due to fast fully-supervised learning) and slower generator learning (due to slow RL learning), allowing the generator to receive richer reward signals from the discriminator.

Total reward r_y for each output y is a combination of the edit distance reward, the unit test reward, and the discriminator reward, based on whether it passes unit tests:

$$r_y = \begin{cases} -\log\left[(1-D(y))(1-r_{y_{\text{ed}}}) + \epsilon\right], & \text{if } y \text{ fails unit tests or compilation} \\ D(y) + r_{y_{\text{ed}}} - 3, & \text{otherwise} \end{cases}$$

where ϵ is a small constant for numerical stability. This conditional reward makes sure that we penalize the actor heavily if its output passes the test suite, where the highest reward it could obtain is -1, regardless of how well the output fools the discriminator or stays close to the reference solution in terms of edit distance. On the other hand, if the actor/generator output fails unit tests, it can obtain a very high reward if it effectively fools the discriminator and has high character overlap ratio with the reference solution. For any output that fails unit tests, we transform both the edit distance and discriminator rewards via $-\log(1-x)$ to amplify values near 1.

Within the PPO training pipeline, we apply token-level KL penalties to prevent the generator from drifting too far from the SFT-initialized model. A value head is added to the LLM to serve as the value function in PPO updates.

On a high level, both the edit distance and discriminator reward encourage the actor to generate code that is similar to the reference solution, while the unit test reward enforces functional failure. Together, these rewards encourage the actor to learn to generate hard, negative code outputs based on the reference solution input. The edit distance reward is used in conjunction with the discriminator reward to stabilize training (see more in Section 6).

We leverage `Qwen2.5-Coder-0.5B` (Hui et al., 2024) as the base model for both the actor and the discriminator.

4 Experimental Setup

4.1 Dataset

We use the MBPP dataset Austin et al. (2021)for this project. The MBPP dataset is a collection of approximately 1,000 crowd-sourced Python programming problems designed to be solvable by entry-level programmers. The dataset focuses on programming fundamentals and standard library functionality, making it suitable for evaluating the code generation capabilities of language models. In this project, we use the full version with 974 coding tasks (observations). Each observation contains a programming problem that includes a task description, reference solution code, and 3 unit test cases. There are four splits in the MBPP data: Train (374 observations), Validation (90 observations), Prompt (10 observations), and Test (500 observations). From now on, we refer to the combination of the Train, Validation, and Prompt sets as the training set and the Test set as the test set, since we have combined the first three splits together for training.

4.2 Rule-Based Mutations

For fine-tuning our actor/generator (Section 3.1) and evaluation baselines (Section 4.3), we create exactly one naive, rule-based mutant for every MBPP problem. A rule is sampled uniformly at random from Table 1 and applied to the reference solution code to generate the corresponding mutant. If the resulting program does not fail the unit test cases, another mutation is attempted until a version that fails the unit test is obtained. This procedure guarantees one failing mutant per reference solution while keeping the mutation policy simple and non-adaptive.

Heuristic Rule	**Explanation**
variable_swap	Randomly swaps two local variables.
off_by_one	Regex tweak of a single `range()` call (e.g., range(n) $\rightarrow$ range(n - 1)).
operator_flip	Replaces one arithmetic operator with a confusable alternative.
comparison_flip	Toggles one comparison (e.g. == $\rightarrow$!=).
premature_return	Inserts `return None` early in the first function body.
line_delete	Removes one random executable line.
arg_order_swap	Swaps the first two positional arguments in a function call.
constant_perturb	Increments the first integer literal by 1.
reverse_list	Reverses elements in a list literal. (e.g. [‘a’, ‘b’, ‘c’] $\rightarrow$ [‘c’, ‘b’, ‘a’])
comment_out	Comments out a random executable line.
infinite_loop	Insert a infinite loop at the top of the first function body.
input_block	Force an `input()` call, causing `RuntimeError`.
sleep_call	Add a `time.sleep` long enough to trigger timeout.

Table 1: Heuristic Rules

4.3 Evaluation Design

4.3.1 PPO Evaluation

We evaluate our PPO training procedure using two approaches:

Training Dynamics. We analyze training curves by tracking the actor/generator's reward progression over time. This reveals how the actor/generator adapts to updates from the discriminator and provides insights into training stability and reward dynamics.

Post-Training Inference Evaluation. We evaluate the quality of negative samples produced by the PPO-fine-tuned generator using the final saved discriminator. For each reference solution in the MBPP training set, the generator is allowed up to five attempts during inference to produce a mutant that fails the unit tests; the first such failure is selected. If all attempts pass, the fifth output is used. We then compute the unit test reward and discriminator score for each model-generated output using the saved discriminator - without applying the $-\log(1-x)$ transformation used during training.

As baselines, we also include (1) the SFT-fine-tuned generator (used to initialize PPO), which is allowed only one attempt at inference to produce a buggy output per reference solution, and (2) rule-based mutants of the same reference solutions. All mutants of reference solutions are evaluated with the same metrics: unit test reward and discriminator score.

4.3.2 DPO Evaluation

We recognize the difficulty in directly measuring the performance of the PPO-fine-tuned generator and the quality of the negative examples it generates. Therefore, we propose an evaluation procedure that involves using the PPO-fine-tuned generator to perform inference on the reference code solutions from the training set. We pair these generated negative code examples with their corresponding positive reference solution examples to form a preference dataset. We fine-tune a `Qwen2.5-Coder-0.5B-Instruct` model with DPO using this constructed preference dataset. During both training and inference, the input is a formatted message containing program descriptions and a list of unit tests from the MBPP dataset. Details of this input format are included in Appendix Section D.2.

After fine-tuning with DPO, we prompt the model with the program descriptions and unit tests in the test set, then measure the pass@1 rate, which is the percentage of problems for which the model generates a correct solution on its first attempt. We evaluate whether a generated output is correct by checking whether it passes all unit tests for that program.

Our baselines include (1) the `Qwen2.5-Coder-0.5B-Instruct` model without any SFT or DPO fine-tuning, (2) the model after SFT using reference code solutions as target outputs, and (3) the model after DPO fine-tuning but using rule-based mutated code (Section 4.2) instead of PPO-generated code as negative examples to construct the preference dataset. The first baseline illustrates the gain from fine-tuning the model with DPO using our preference dataset. The second baseline illustrates whether using DPO leads to better performance than using SFT. The third baseline illustrates whether our PPO-generated hard negative examples are more useful than rule-based-mutated negative examples when used to fine-tune an LLM on code synthesis tasks.

More details on our dataset preprocessing and hyperparameter tuning for DPO can be found in Appendix D.2.

5 Results

5.1 Quantitative Evaluation

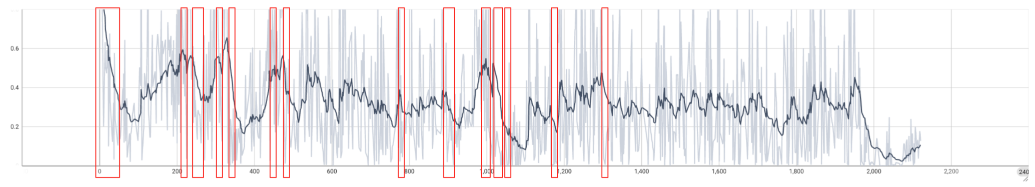
5.1.1 PPO Results

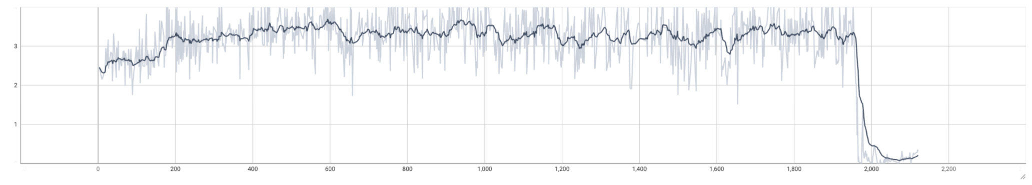
Training Dynamics. First, we examine the PPO training process. Figure 2 presents smoothed PPO training curves (smoothing factor 0.9) over 2,000 iterations: Panel (a) discriminator reward; (b) edit distance reward; (c) unit test reward; and (d) KL divergence relative to the SFT reference model. For any output that fails unit tests, both edit distance and discriminator scores are transformed via $-\log(1-x)$ to amplify values near 1. Throughout training, the edit distance reward remains above 2.0 and the unit test reward stays above -0.4, suggesting the agent is able to consistently generate output that is similar to the reference solution but also reliably fails unit tests. Red boxes on the discriminator curve denote discriminator update events. In early iterations, the generator can gradually exploit the discriminator between updates and steadily recovers higher discriminator rewards. As training progresses, however, the discriminator outpaces the generator - whose exploration diminish (see more discussions in Section 6) - making post-update recovery increasingly difficult. At around step 1950 (after roughly four hours of training), the generator irreversibly collapses - likely triggered by reward hacking (e.g., by producing non-code outputs; see qualitative evaluation in Section 5.2.2).

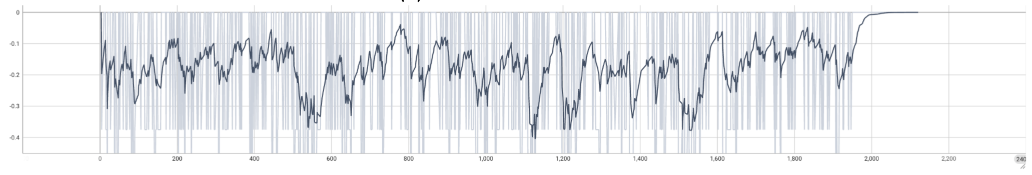
Post-Training Evaluation. Before the eventual model collapse, we save a checkpoint of both the generator and the discriminator. Using the saved generator, we then generate a negative code snippet for each reference solution. We compute the unit test reward and discriminator reward for each generated snippet using the saved discriminator. For comparison, we leverage the SFT-only fine-tuned generator (used to initialize PPO) and rule-based mutants of the reference solutions as baselines.

Table 2 summarizes the results: the PPO-fine-tuned generator achieves higher average discriminator and edit distance rewards than both the SFT generator and rule-based mutations. This suggests that PPO training improves the generator's ability to produce negative code that is possibly more deceptively similar to the reference solutions. Although discriminator reward may not perfectly reflect how hard a negative sample is - since it can be gamed by exploiting regions in the discriminator where it is underfit - it still provides a useful proxy for training success. Notably, the PPO-fine-tuned model consistently outperforms its SFT-only counterpart on both rewards.

We also acknowledge that failure rates on unit tests are not directly comparable across methods, as we allow PPO-generated samples multiple attempts while SFT-generated samples receive only one. Therefore, reward metrics provide a more meaningful basis for evaluating the PPO's effectiveness.


(a). Discriminator Reward


(b). Edit Distance Reward


(c). Unit Test Reward

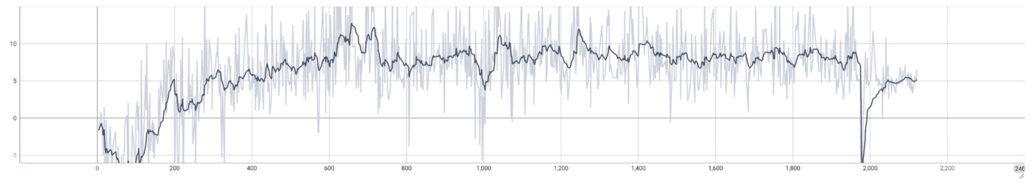

(d). KL divergence from Ref Model

Figure 2: Smoothed PPO training curves (smoothing 0.9): (a) discriminator reward; (b) edit distance reward; (c) unit test reward; (d) KL divergence from the SFT reference model. Discriminator reward and edit distance reward for outputs failing unit tests are transformed via $-\log(1-x)$. Red boxes on the discriminator reward curve mark discriminator update events.

5.1.2 DPO Results

Table 3 summarizes the pass@1 rate on our proposed DPO procedure as well as the three baselines, on the training set, validation set, and test set. We see that our proposed DPO procedure, which uses PPO-generated negative examples, achieves the highest pass@1 rate on the test set and validation set, suggesting that our hard negative example generation procedure can successfully generate negative examples that are useful in downstream code synthesis tasks. Through DPO fine-tuning, these PPO-generated negative examples help the model generalize to unseen code synthesis tasks.

Besides the superior performance of our proposed DPO procedure, we observe that fine-tuning with DPO using rule-based negative examples also achieves relatively high pass@1 rate on the test set, although a noticeably lower pass@1 rate on the validation set. The relative success of using rule-based negatives for DPO validates the hypothesis that using DPO with a preference dataset consisting of positive and negative code examples helps LLMs avoid hard-to-notice pitfalls when generating code. It is possible that, since the programming tasks in MBPP are relatively easy and the reference solutions are short, the advantages of model-generated mutations relative to rule-based mutations might not be evident in terms of the performance boost they bring via DPO. The relatively large difference between performance on the validation set and on the test set is possibly due to the small validation and test set sizes, which means they may not represent the general distribution of code synthesis tasks well.

SFT using reference code solutions as target outputs leads to a higher pass@1 rate on the training set and lower pass@1 rates on the validation set and test set. This suggests that only supplying the model with positive examples but not negative examples during training does not help the model learn to avoid subtle pitfalls and generalize to unseen inputs, especially when the training set size is small. As

Table 2: Comparison of negative code qualities from different generation methods. We compare (1) rule-based mutations, (2) the SFT-fine-tuned generator (used to initialize PPO), and (3) the PPO-fine-tuned generator. The PPO generator and discriminator are taken from the last checkpoint before model collapse. Results are reported on both training and test splits of MBPP.

Generation Model	Split	Unit Tests	Percentage of Data	Discriminator Reward	Edit Distance Reward
Rule-based mutations	Train	Fail	100 %	0.0477 ± 0.1634	0.9001 ± 0.0789
		Pass	0%	N/A	N/A
	Test	Fail	100 %	0.0566 ± 0.1809	0.9062 ± 0.0804
		Pass	0%	N/A	N/A
SFT fine-tuned	Train	Fail	85.23%	0.0325 ± 0.1322	0.9127 ± 0.0782
		Pass	14.77%	0.1867 ± 0.2393	0.9866 ± 0.0086
	Test	Fail	74.00%	0.0125 ± 0.0775	0.9189 ± 0.0680
		Pass	26.00%	0.1539 ± 0.2079	0.9864 ± 0.0096
PPO fine-tuned	Train	Fail	88.57%	$\mathbf{0.1722 \pm 0.2924}$	$\mathbf{0.9512 \pm 0.0818}$
		Pass	11.43%	0.6724 ± 0.2679	0.9987 ± 0.0032
	Test	Fail	83.46%	$\mathbf{0.1794 \pm 0.2871}$	$\mathbf{0.9690 \pm 0.0499}$
		Pass	16.54%	0.5625 ± 0.2909	0.9957 ± 0.0071

a caveat, we acknowledge that our hyperparameter tuning process might not be extensive enough due to computational constraints, possibly leading to suboptimal performance.

Table 3: pass@1 rate on code synthesis tasks.

	Training Set	Validation Set	Test Set
No SFT or DPO	0.430	**0.500**	0.365
SFT with Reference Code Solutions	**0.478**	0.316	0.353
DPO with Rule-Based Mutated Negative Examples	0.475	0.395	0.415
DPO with PPO-Generated Negative Examples	0.445	**0.500**	**0.420**

5.2 Qualitative Analysis

5.2.1 Mutation Categories in PPO-Generated Code

We investigate the code generated by the PPO-fine-tuned generator by looking into the distribution of code mutations produced by the generator, comparing against the rule-based mutations. Table 4 summarizes the distribution of mutations in model-generated negative code vs. that in rule-based mutations. Specifically, we deployed two methods - automatic recognition and manual inspection - to categorize the mutation types in model-generated code. Specifications regarding these methods can be found in Appendix B.

Overall, we observe that micro-edits - i.e., single-character insertions/deletions, variable renames, or adding a leading '#' - account for most of the changes in model-generated mutation. Comparing to the rule-based mutations, more apparent changes like adding input blocks or infinite loops are never seen. These micro-edits make sense because they preserve maximum overlap with the reference solutions, thereby gaining high edit distance rewards, on top of plausible high discriminator rewards. This preference for micro-edits is further confirmed by Figure 3 in Appendix C, which shows that the distribution of normalized edit distances between model-generated mutants and their corresponding reference solutions is heavily skewed toward zero. More discussions about edit distance distribution can be found in Appendix C.

5.2.2 Examples of PPO-Generated Code

Code 1 illustrates an example of negative code snippets generated by our PPO-fine-tuned model. Both snippets appear deceptively similar at first glance, except for the only difference in a single right parenthesis in the generated version `(zip(*lst)))`. This triggers a syntax error and cause all unit tests to fail, yet it is difficult to distinguish both by humans and the discriminator (high reward). In fact, we observe that the model often preserves nearly the entire reference implementation while inserting or deleting just one or two characters that critically affect correctness.

	Rule-Based	Model-Generated	
		Automatic Recognition	Manual Inspection
premature_return	57	1	1
input_block	54	0	0
comment_out	47	106	154
line_delete	46	17	16
infinite_loop	42	0	0
sleep_call	40	2	2
variable_swap	33	0	0
arg_order_swap	31	1	0
constant_perturb	22	12	62
operator_flip	4	56	6
reverse_list	4	0	0
comparison_flip	3	11	0
off_by_one	2	4	0
no_change	-	44	44
add_lines	-	-	7
add_letter	-	-	5
add_symbol	-	-	9
change_symbol	-	-	4
change_variable_name	-	-	103
delete_letter	-	-	1
delete_symbol	-	-	4
delete_variables	-	-	1
gibberish	-	-	9
Other	0	131	0
Sum	385	385	428

Table 4: Distribution of code mutations in model-generated vs. rule-based mutations.

```
# Reference Solution Code
def merge(lst):
    return [list(ele) for ele in list(zip(*lst))]

# Model-Generated Code
def merge(lst):
    return [list(ele) for ele in list(zip(*lst)))
```

Code 1: Model-generated Output Example.

Another set of interesting cases is exemplified in Code 2, which is an instance showing reward hacking behaviors that naturally occur during training. The model-generated code repeats the function header and then inserts a sequence of nonsensical Russian characters. Syntactically, this fragment is invalid Python, so the code cannot be compile. Yet, at the moment it was produced, the discriminator awarded the snippet a high reward because the discriminator is previously unexposed to non-code and is thus underfit. Such behavior is a sign of reward hacking: the policy discovers an underfit region in reward signal and maximizes it by emitting non-code that is cheap to generate but still scores favorably. In practice, we observe that reward hacking by generating non-code is a good indicator of imminent model collapse. See more discussion on possible future mitigation on reward hacking in Section 6.

```
# Reference Solution Code
def triangle_area(r) :
    if r < 0 :
        return -1
    return r * r

# Model-Generated Code
def triangle_area(r) : [Russian Characeters]
```

Code 2: Reward Hacking Example.

6 Discussion

Our GAN-style PPO fine-tuning approach shows strong potential for generating high-quality negative code samples. The PPO-fine-tuned generator learns to produce high-reward mutations by making subtle edits to reference solutions - such as modifying numeric constants, renaming variables, altering function signatures, or tweaking syntax. These minimal changes often result in code that is visually indistinguishable from correct implementations, yet strategically fails unit tests or fools the discriminator. Importantly, when these PPO-generated negatives are used as preference pairs alongside reference solutions, they lead to improved downstream performance in DPO fine-tuning, highlighting their utility for preference-based training. Together, these findings suggest that our approach is a promising direction for scalable and automated generation of hard negative examples - particularly valuable for robustifying code generation models by exposing them to hard, buggy programs.

Despite our GAN-style RL framework's empirical successes and promiess, it still exhibits several limitations. First, unlike classical GANs which admit a Nash equilibrium when the generator perfectly matches the target distribution (Goodfellow et al., 2014), our setup cannot converge to such an equilibrium: the generator is explicitly discouraged from producing correct solutions (to avoid passing unit tests), so the interaction reduces to a perpetual "cat-and-mouse" game with no theoretical guarantee of stability. In practice, we observe that the generator eventually "reward-hacks" by outputting non–code snippets (e.g., Russian text unseen by the discriminator), leading to rapid discriminator defeat and catastrophic collapse of the generator.

Second, our edit distance reward mitigates reward hacking by keeping outputs close to valid code, and it also biases the generator toward minimal edits. Moreover, because edit distance is a simple, low-variance signal, it is easier for the agent to learn and thus stabilizes early training more effectively than the higher-variance discriminator reward. However, this advantage comes at the cost of discouraging more complex but semantically adversarial mutations (e.g., statement reordering or subtle control-flow changes) that might better fool the discriminator. In future work, one could employ a discriminator pre-trained on both code and non-code examples (or with an auxiliary "code validity" head) to more robustly penalize out-of-domain outputs, and combine edit-distance constraints with higher-level structural or type-based similarity metrics.

Third, exploration decays over the course of PPO updates. As the policy becomes more certain, it is less likely to sample novel mutations after each discriminator update. Nevertheless, after each discriminator update during PPO training, the agent should ideally explore again to find ways to exploit the discriminator. Future works can likely benefit from introducing explicit "re-exploration" incentives (e.g., periodic entropy bonuses or count-based novelty rewards) following discriminator updates, ensuring the generator continues to search for new failure modes.

Additionally, like most adversarial and RL-based methods, our approach is hyperparameter-sensitive and inherently unstable. The combination of competing objectives - unit-test failure, edit-distance proximity, and discriminator fooling - amplifies sensitivities in learning rates, clipping thresholds, and reward scales. Scaling to larger models or datasets, or deploying automated hyperparameter search (e.g., population-based training), may alleviate some of these instabilities and improve robustness over longer training runs.

Finally, although our PPO-fine-tuned generator produces negative code mutants with higher discriminator and edit distance rewards - and using these as preference pairs improves DPO fine-tuning performance - such results remain indirect evidence of our method's effectiveness. Directly evaluating whether a generated negative code example is genuinely "hard" for a downstream LLM remains challenging. Future work could benefit significantly from developing more direct and principled evaluations of this hardness.

7 Conclusion

This work addresses a fundamental problem where LLMs are unexposed to buggy code during training and hence vulnerable to buggy code at inference time. To mitigate this, we introduce an adversarial RL framework for generating hard negative code examples. Through formulating a uniquely designed reward function that takes unit test outcomes, edit distance from reference solution, and a GAN-style discriminator into account, our PPO-fine-tuned generator successfully learned to generate hard negative code examples that are similar to their positive counterparts but reliably fail compilation or unit tests. We show that models can learn to produce subtle bugs that may be difficult to capture through hand-crafted rules, as evidenced by both quantitative and qualitative results. Moreover, we show that the hard negative code snippets that our PPO-fine-tuned generator produces can be used to construct a preference dataset to improve downstream performance on code synthesis tasks via DPO, validating the practical utility of our approach.

However, the challenges we encountered - including training instability, reward hacking, and eventual model collapse - highlight important research directions for the broader field. Future work should focus on developing more robust adversarial training procedures that can maintain stable exploration throughout the training process. Investigating alternative reward functions that better capture semantic correctness beyond simple edit distance, and developing more sophisticated discriminators that can resist reward hacking, will be crucial for scaling this approach to larger models and more complex programming tasks. A quantitative metric evaluating how “hard” a negative example might help address these limitations, hence an important future work direction.

The broader implications extend beyond code synthesis to any domain where subtle negative examples are scarce but critical for a balanced and robust model behavior. As LLMs become increasingly deployed, their ability to systematically generate hard negative examples through adversarial training represents an important tool for improving model reliability, and our work provides a starting point for this direction.

8 Team Contributions

- **Eric Li:** Designed, implemented, and conducted the adversarial PPO algorithm and associated experiments. Contributed to report writing.
- **Ella Mao:** Implemented rule-based mutations and DPO training. Contributed to running experiments and report writing.
- **Komei Ryu:** Implemented DPO training, hyperparameter tuning, and evaluation on code synthesis tasks. Contributed to the implementation of and experiments regarding the unit test and normalized edit distance reward components. Contributed to writing the report.

Changes from Proposal We did not implement the CodeRL reward component since we find that compiling code and running unit tests during training are sufficiently fast. We switched from our originally proposed APPS dataset (Hendrycks et al., 2021) to the current MBPP dataset because APPS dataset was shown to be too difficult for a 0.5B model. Ella and Komei worked on DPO training and evaluation together instead of separately implementing DPO with PPO-generated code and DPO with rule-base-mutated code due to their shared implementation.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

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

Justin Fu, Katie Luo, and Sergey Levine. 2017. Learning robust rewards with adversarial inverse reinforcement learning. *arXiv preprint arXiv:1710.11248* (2017).

Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. *Advances in neural information processing systems* 27 (2014).

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938* (2021).

Jonathan Ho and Stefano Ermon. 2016. Generative adversarial imitation learning. *Advances in neural information processing systems* 29 (2016).

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E Gonzalez, and Ion Stoica. 2020. Contrastive code representation learning. *arXiv preprint arXiv:2007.04973* (2020).

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

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023. Rltf: Reinforcement learning from unit test feedback. *arXiv preprint arXiv:2307.04349* (2023).

Atharva Naik, Soumitra Das, Jyothi Vedurada, and Somak Aditya. 2023. SYNC: A Structurally Guided Hard Negative Curricula for Generalizable Neural Code Search. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*. 83–103.

Joshua Robinson, Ching-Yao Chuang, Suvrit Sra, and Stefanie Jegelka. 2020. Contrastive learning with hard negative samples. *arXiv preprint arXiv:2010.04592* (2020).

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

Omar Shaikh, Michelle S Lam, Joey Hejna, Yijia Shao, Hyundong Justin Cho, Michael S Bernstein, and Diyi Yang. 2025. Aligning Language Models with Demonstrated Feedback. In *The Thirteenth International Conference on Learning Representations*.

Ensheng Shi, Yanlin Wang, Wenchao Gu, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Cocosoda: Effective contrastive learning for code search. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2198–2210.

A A Single Example from MBPP Dataset

```
  task_id: 11
  text   : ‘Write a python function to remove first and last occurrence of
           a given character from the string.’
  code   :
def remove_Occ(s,ch):
    for i in range(len(s)):
        if (s[i] == ch):
            s = s[0 : i] + s[i + 1:]
            break
    for i in range(len(s) - 1,-1,-1):
        if (s[i] == ch):
            s = s[0 : i] + s[i + 1:]
            break
    return s
  # of unit test: 3
```

B Qualitative Analysis Methods

We developed an automatic pipeline to recognize whether the model-generated code follows one of the rules we used in the random rule-based code mutation. For every code sample, we compare the original reference solution code to the model-generated variant. A suite of 13 deterministic detectors

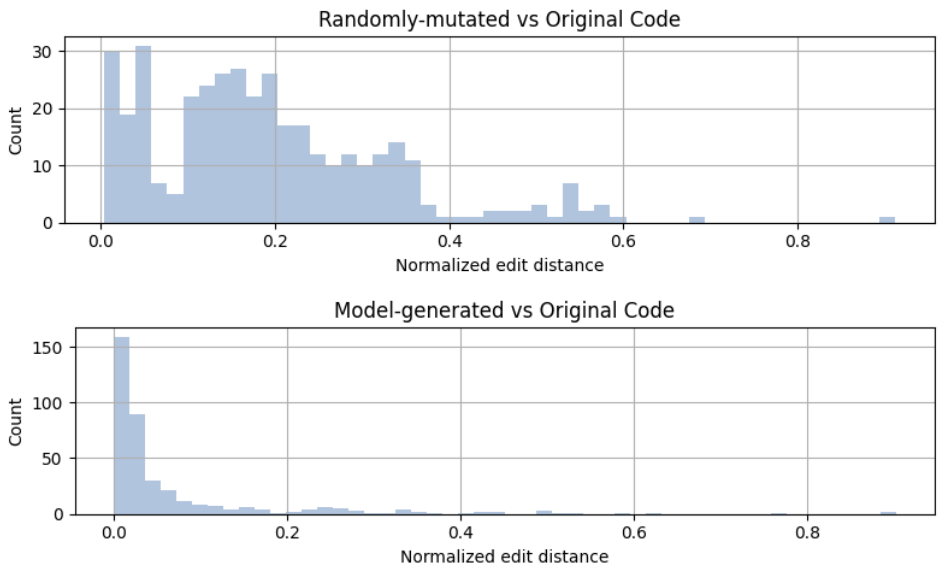


Figure 3: Distribution of normalized edit distance between rule-based mutants (top) / model-generated mutants (bottom) vs. their corresponding reference solutions. Normalized edit distance is calculated by scaling raw Levenshtein distance by the number of characters in the reference solution code.

searches for the syntactic "signature" left by each mutation rule. For instance, we look for insertion of `return None` for `premature_return`, a single-line deletion for `line_delete`, or a flipped comparison operator for `comparison_flip`. Each detector is a pure function that returns `True` if the mutation signature is present. The first matching detector assigns the corresponding rule label; if none is found, we label the sample `Other`. Finally, identical copies are marked `no_change`. Results are shown under the column "Automatic Recognition" in Table 4. After the automatic recognition, 131 samples remain uncategorized. Additionally, because the automatic recognition process captures only one mutation even in multi-category edit cases, we then conduct a manually inspection. We summarize the changes manually into 22 categories (including the 13 from rule-based mutation) and group the generated code into multi-label categories, in which a single generated snippet may receive several mutation categories.[1]

C Edit Distance Distribution

The normalized edit distance in Figure 3 scales the raw Levenshtein distance by the number of characters in the reference solution code, providing how much two snippets differ. Across all problems, the generated code clusters tightly around zero: roughly a third of the edits fall below 0.02 and the vast majority lie under 0.15. There are also generated code with edit distance equals to zero, which means it's the same as the original reference solution code. This sharp left-skew indicates that the model mostly keeps the original structure and alters only a handful of characters. By contrast, the rule-based mutants exhibit a far broader dispersion. Both distributions have noticeable long tail around 0.9. These higher scores reflect the coarse-grained edits such as line deletions. Taken together, the two distributions expose the generated code showing more subtle deviations, which align with our objective of producing visually similar code snippets but still break functional correctness. This result is also mechanical since we specifically regularize our model to have smaller edit distance.

D Implementation Details

The reference solutions in MBPP are relatively short: the mean length is 181.1 characters, with a median of 145.5 and a range of 30–1331. For efficiency, we keep only tasks whose reference solution

[1]Manual counts therefore sum to 428 labels even though only 385 model-generated snippets exist.

contains fewer than $\ell < 256$ characters. After this filtering, the training split contains 385 problems, while the test split comprises 405 problems.

D.1 SFT & PPO

Both SFT and PPO are run with one Nvidia A100, for 10 minutes and 4 hour and 30 minutes respectively. We leverage `Qwen2.5-Coder-0.5B` as the base model for both the agent and the discriminator. We add a special token `<ECHO>` after the end of the input code as a signal for model generation. SFT and PPO are run on the entire train set. For SFT, we use batch size 4, learning rate 5×10^{-5}, and epochs 3. We implement PPO using HuggingFace's TRL library with a generator instantiated as `AutoModelForCausalLMWithValueHead` and the binary discriminator as `AutoModelForSequenceClassification`. Based on the results of extensive hyperparameter tuning, we decide to configure PPO with learning rate 5×10^{-6}, batch size 8, mini-batch 4, PPO epochs 4, init_kl_coef 0.2, clip range 0.2, and max-grad-norm 1.0. We clip the total reward r_y at a max of 10 to avoid instabilities. When running unit tests, we set timeout at 1 second to classify extended executions (e.g. infinite loops) as failures.

D.2 DPO

We leverage `Qwen2.5-Coder-0.5B-Instruct` as the base model from which we fine-tune with DPO using the constructed preference dataset. Since we aim to improve the performance of the DPO-fine-tuned model on code synthesis tasks, we formulate the input prompt according to the conversational input structure that `Qwen2.5-Coder-0.5B-Instruct` used during its training. We rely on the `AutoTokenizer.apply_chat_template` function from HuggingFace's Transformers library to convert the messages shown in Listing 1 into the desired conversational structure and use the formatted messages as inputs during both training and inference. The messages template is based on the suggested template in Austin et al. (2021), with modifications that instruct the model to only generate code solutions for easier evaluation. During inference, the model's response is processed based on manually designed parsing rules to only preserve the code snippets in each response, before running their unit test cases.

Listing 1: Training and Inference Prompt Formulation for Code Synthesis Task. Here, "prompt" is a program description, and "test_list" is a list of unit tests. Both "prompt" and "test_list" are from the MBPP dataset.

```
messages = [
    {"role": "system", "content": "You are an expert Python
    programmer."},
    {"role": "user", "content": f"Here is your task: {prompt} Your
    code should pass these tests:\n\n{test_list}\nPlease only output
    your implementation. Please do not output any explanation.
    Please do not include my tests in your answer.\n"}
]
```

For our DPO evaluation, we filter the training set and only preserve examples whether the reference code, the PPO-generated code, and the mutated code all have lengths fewer than 256 characters, to be consistent with the inputs seen during PPO training. This results in 375 examples in the training set. Next, we randomly split the training set into a training and a validation set for our DPO fine-tuning approach, the second baseline and the third baseline. After splitting, the training set contains 337 examples in the training set and 38 examples in the validation set. The examples in the training set and validation set are kept the same for all three training methods to ensure consistency when comparing with baselines.

The validation set is used to conduct hyperparameter tuning. When running SFT with reference code solutions as target outputs, we use the standard validation loss to choose between hyperparameter values. When running DPO, we notice that a lower validation loss does not lead to better performance, since the model can decrease the probabilities of both outputting the positive and negative examples and still achieve a lower validation loss. Thus, we choose the hyperparameter values that lead to the highest pass@1 rate on the validation set.