# Adversarial methods for the alignment of CodeLLMs

**Ilyas Oulkadda**
(supervisor: Julien Perez)

Technical Report $n^o$output, June 2024
revision

As Large Language Models (LLMs) for code generation continue to gain widespread adoption, exemplified by GitHub Copilot's[1] user base surpassing a million, it is imperative that these models are aligned with functional standards to ensure their reliability and security. Code-LLMs are revolutionizing developer productivity by automating code generation, but they also raise significant concerns about the quality and security of the generated code. This research addresses these issues by employing innovative adversarial techniques to enhance the reasoning capabilities and security of Code-LLMs. Our study focuses on two primary approaches: Direct Preference Optimization (DPO), which leverages an adversarial game between two models, Alice and Bob; and Proximal Policy Optimization (PPO), which utilizes unit testing as a reward model. By combining these methods, we aim to systematically improve the robustness and reliability of Code-LLMs, ultimately advancing the development of a framework that not only mitigates risks associated with automated code generation but also reinforces the overall integrity and dependability of Code-LLMs.

**Keywords**
LLMs, Alignment, CodeLLMs, Adversariality, Reinforcement Learning


Laboratoire de Recherche de l'EPITA
14-16, rue Voltaire – FR-94276 Le Kremlin-Bicêtre CEDEX – France
Tél. +33 1 53 14 59 22 – Fax. +33 1 53 14 59 13
ilyas.oulkadda@lre.epita.fr – http://www.lre.epita.fr/

[1]Coding extension powered by a CodeLLMs to assist in code completion tasks

## Copying this document

# Contents

# Chapter 1

# Introduction

The rapid advancement of Large Language Models (LLMs), particularly those specialized in code generation such as GitHub Copilot, has significantly accelerated the automation of coding tasks, enhancing productivity in software development. Despite these benefits, this technological evolution introduces critical challenges concerning the quality, security, and ethical alignment of the generated code. Traditionally, aligning LLMs has relied on human-annotated data, a method becoming impractical due to its lack of scalability in the face of rapidly evolving model capabilities and application contexts.

In response to these challenges, this research aims to innovate upon existing methods by integrating adversarial components into established reinforcement learning techniques, specifically Proximal Policy Optimization (PPO) [18] and Direct Preference Optimization (DPO) [16]. This approach seeks to replace the traditional human-centric data annotation process with a dynamic, model-driven alignment mechanism, wherein models engage in adversarial interactions to refine and optimize the target model's performance continuously.

Our approach utilizes Proximal Policy Optimization (PPO), where a more extensive model generates coding exercises accompanied by descriptions and a critical test suite. This suite serves as a reward model, guiding the target model's policy optimization through real-time feedback. However, initial attempts using this method revealed instabilities in test outputs, which occasionally led to erroneous model penalization. Despite these challenges, we observed some intriguing behaviors during testing. Notably, the model often resorted to auto-completing with "pass" to strategically maximize the reward—successfully compiling and running while only failing the assertion tests. This behavior highlights a potential area for further refinement in the reward structure to ensure comprehensive assessment and learning.

Alternatively, the research explores an adversarial enhancement of DPO. In this setup, an iterative interaction between an 'oracle' model and a 'student' model is used. The oracle, being larger and more knowledgeable, is tasked with guiding the smaller student model, thereby aligning it more closely with ethical and functional standards. Conducted over multiple cycles and leveraging high-performance computing resources, this method has shown promise in preliminary tests, including slight improvements in benchmarks like HumanEval [5]. Additionally, our training logs indicate that the reward margins are clearly increasing during the training sessions, demonstrating that the student model is effectively learning to converge toward the distribution of the oracle. This observation underscores the potential of this method to facilitate significant advancements in model alignment.

This introduction sets the stage for a comprehensive discussion on how adversarial techniques can enhance traditional reinforcement learning methods for model alignment. The fol-

lowing sections will delve into related work, the specific adversarial adaptations applied to PPO and DPO, and strategies for managing computational resources effectively. This discussion aims to elucidate the potential impacts and future applications of these advanced adversarial techniques in the ongoing development and refinement of Code-LLMs.

To summarize, we first introduce an innovative use of adversarial techniques within established reinforcement learning frameworks like Proximal Policy Optimization (PPO) and Direct Preference Optimization (DPO) to enhance model training and performance without relying on human-annotated data. Second, we demonstrate the practical application of these techniques through a dynamic, adversarial interaction between 'oracle' and 'student' models, showcasing measurable improvements in model alignment and performance, as evidenced by increased reward margins and enhanced outcomes on coding benchmarks. Collectively, these advancements provide a scalable, efficient solution to the challenge of quality, security, and ethical alignment in code generation by LLMs, offering a substantial foundation for future explorations and developments in the domain.

# Chapter 2

# Context

This section provides a contextual backdrop for the current research by reviewing relevant literature across several key areas that inform our approach to refining and enhancing Code-LLMs through adversarial techniques.

**Data Annotation**: Data annotation serves as the foundation for training and refining LLMs. Traditional approaches heavily rely on manual annotation, which, while effective, scale poorly with the increasing size and complexity of models and datasets. An illustrative example from the realm of code generation LLMs is highlighted by [13], where 1,399 crowd-workers from 35 different countries were engaged to annotate a dataset for Personally Identifiable Information (PII) in source code. This example underscores the extensive human effort involved and the challenges in scalability and consistency across such a large and diverse group of annotators.

**Self-instruct**: The use of larger models to autonomously generate high-quality, well-structured data represents a shift towards more scalable methods of data creation and annotation. Notable examples include the phi 1.5 model [7], where a model was exclusively trained on high-quality generated coding textbooks, and the creation of the *Cosmopedia* dataset [3]. Both instances showcase the effectiveness of leveraging advanced models for data generation. Furthermore, as discussed by [6], the incorporation of topic sampling techniques is crucial for enhancing the diversity and coverage of the generated datasets, ensuring a broader range of scenarios and cases for the model to learn from.

**Adversarial and Curriculum Learning**: Adversarial learning techniques, particularly those coupled with curriculum strategies, have demonstrated substantial benefits in fields such as robotics. For example, "Asymmetric self-play" methodologies have shown potential in discovering optimal policies through adversarial setups [14]. This is further elaborated in the study on "intrinsic motivation and automatic curricula via asymmetric self-play," which illustrates how adversarial interactions can naturally induce a curriculum of tasks that progressively increase in complexity [20]. Such methods provide a framework for our approach to adversarial enhancements in training Code-LLMs, suggesting a pathway to more dynamic and effective model training regimes.

**Unit Tests in CodeLLMs**: Unique to the domain of code generation, unit tests offer a valuable feedback mechanism for both training and inference phases. The concept of an agentic loop as explored in the codeRL framework [12], highlights the utility of unit tests not just as a means of validation but as integral components of the learning process. By incorporating feedback from unit tests into the training loop, models can continuously refine their output, align more closely with functional requirements, and reduce the occurrence of bugs or undesired behaviors.

**Proximal Policy Optimization**: In the reinforcement learning with human feedback (RLHF)

step, the primary method utilized is Proximal Policy Optimization (PPO) [18]. This technique involves implementing a stochastic gradient ascent method, where gradient steps are taken after processing a mini-batch of sample data. Within the context of Large Language Models (LLMs), PPO entails training a reward model designed to replicate human preferences. The process involves the model generating textual responses to inputs, with the reward model evaluating these responses and assigning scores based on their alignment with desired outcomes. The objective is to maximize the average reward, thereby steering the model's outputs towards higher quality and greater alignment with human expectations. This method enhances the model's performance by iteratively adjusting based on feedback, aligning more closely with the intricacies of human language and preferences.

**Direct Preference Optimization**: While Proximal Policy Optimization (PPO) offers significant benefits, its implementation can be complex and potentially unstable due to its reliance on a reward model that approximates human preferences. This approximation may lead the model to drift too far from its original parameters, and the inherent nature of reinforcement learning can contribute to this instability. To address these challenges, alternatives such as Direct Preference Optimization (DPO) [16] have been developed. DPO simplifies the learning process by integrating an internal reward system, effectively transforming the optimization challenge into a classification loss problem. This approach eliminates the need for sampling the model during fine-tuning or hyperparameter adjustments. By directly comparing preferred and non-preferred model outputs, DPO streamlines the training process, reduces complexity, and increases stability, making it a more straightforward and robust alternative to PPO for aligning model outputs with human preferences.

**Adapters**: Training Large Language Models (LLMs) can be notably resource-intensive, as these models typically encompass billions of parameters. Fine-tuning such extensive models demands considerable computational resources and can be highly inefficient. An effective solution to this challenge is the use of adapters for fine-tuning [9]. Research has demonstrated that training only a small proportion of new parameters, which are strategically injected into existing model layers, can achieve performance comparable to fine-tuning the entire model. By freezing the original parameters and training only these newly added adapter parameters, it is possible to dramatically reduce the number of trainable parameters—often to less than 3% of the total. This method maintains the original model architecture and performance, offering a more resource-efficient approach to updating and refining LLMs.

Each of these areas contributes to the foundation upon which this research builds. By integrating insights from these related works, this study aims to advance the methodology of using adversarial techniques to enhance the scalability, efficiency, and ethical alignment of Code-LLMs. The subsequent sections will delve into the specifics of how adversarial methodologies, informed by these related works, are applied to improve the design and functionality of our model-training approaches.

# Chapter 3

# Memory Efficient Training

During my research, the availability of computational resources was limited, which is a common challenge when working with Code-LLMs that often consist of billions of parameters. Given this limitation, it is crucial to implement strategies that optimize memory usage, allowing these extensive models to be trained on relatively smaller GPUs. Beyond resource constraints, a key motivation for these strategies is to reduce the environmental impact of training large models by minimizing energy consumption. To address this challenge, I focused on three main methods: low-rank adaptation, quantization, and gradient accumulation. These techniques are pivotal in managing the substantial memory requirements of Code-LLMs, enabling efficient and eco-friendly training within the constraints of available hardware. This chapter will explore how these methods contribute to effective and sustainable training processes under resource limitations.

One effective approach to address the memory constraints encountered in training Code-LLMs is low rank adaptation [10]. This method leverages adapters to fine-tune the model, employing a strategic modification in parameter handling. Specifically, these adapters consist of two lower rank matrices involved in a matrix multiplication process. One matrix is initialized with a zero-centered normal distribution, while the other starts from zero.

| Model | Trainable Parameters | All Params | Trainable% |
|---|---|---|---|
| StarCoderBase | 835,584 | 1,14B | 0.073 |
| Phi-2 | 9,175,040 | 2,7B | 0.329 |
| Phi-1.5 | 5,505,024 | 1,4B | 0.386 |

Table 3.1: Trainable parameters of different models

By using low rank matrices instead of full parameter matrices, the number of trainable parameters is drastically reduced to just 0.3% of the total parameters of the model. This significant reduction not only decreases the memory required to store and update parameters during training but also simplifies the computational demands, enabling the training of large-scale models on hardware with more limited capabilities. The implementation of low rank adaptation thus represents a crucial advancement in making the training of sophisticated LLMs more accessible and manageable.

Quantization is another method employed to manage memory and computational efficiency in the training of Code-LLMs. This technique involves representing the model's parameters in lower precision formats. By reducing the number of bits required to store each parameter,

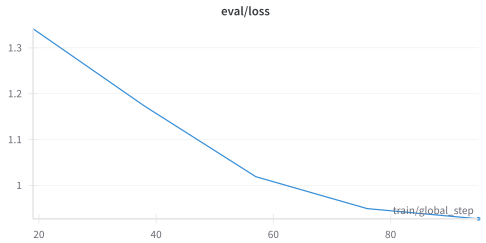


Figure 3.1: Phi 1.5 validation loss on HumanEval [5] using LoRA

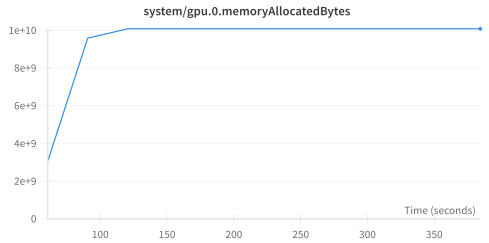


Figure 3.2: Memory allocation on GPU during training

quantization decreases the overall memory footprint of the model. This reduction not only saves storage space but also typically lowers energy consumption and speeds up operations such as matrix multiplication through the use of integer arithmetic, which is faster than floating-point arithmetic.

While quantization is often used during the inference phase on CPUs to enable large models to run more efficiently, it can also be adapted for use during the training phase. In training, certain parameters and matrix multiplication operations are converted to lower precision to accelerate the process. It's important to note, however, that this method depends on hardware compatibility; only some GPUs support lower precision data types, which can limit the applicability of quantization in certain training environments.

Table 3.2: Different quantization for Phi 1.5 [7], 1.3B parameters

| dtype | Model | Grad. Calc. | Backward Pass | Opt. Step |
|---|---|---|---|---|
| Float32 | 4.9 GB | 4.9 GB | 9.81 GB | 19.62 GB |
| Float16/BF16 | 4.9 GB | 7.36 GB | 9.81 GB | 9.81 GB |

Gradient accumulation is a crucial technique utilized to manage memory more efficiently while enabling the training of large-scale models with larger batch sizes. This method involves performing multiple backward passes and accumulating the gradients over these passes. The model's weights are only updated after accumulating the desired amount of gradients, equivalent to the gradient effect of a much larger batch.

By implementing gradient accumulation, it is possible to mimic the effects of training with larger batches without the corresponding increase in memory usage. This approach allows for more flexible management of memory resources, making it feasible to train more substantial models on hardware with limited memory capacity.

# Chapter 4

# Proximal Policy Optimization with Unit Test Signal

This chapter explores a novel implementation of Proximal Policy Optimization (PPO) involving two models, Alice and Bob, which interact to enhance the training process of code generation LLMs. In this setup, Alice is tasked with generating coding exercises, while Bob's objective is to solve these exercises correctly. The efficacy of Bob's solutions is assessed using unit tests, which serve as the basis for rewards in this training framework.

The interaction between Alice and Bob is designed to simulate a dynamic and realistic coding environment, where the generation of challenges and the formulation of solutions co-evolve. This method not only aims to improve the problem-solving capabilities of Bob but also refines Alice's ability to create relevant and challenging coding tasks. This chapter will delve into the specifics of how these models collaborate and the impact of this approach on enhancing the robustness and reliability of Code-LLMs.

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}, \tag{4.1}$$

$$L^{CLIP}(\theta) = \hat{E}_t \left[ \min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{4.2}$$

We use Proximal Policy Optimization (PPO) to align our model using test signals. PPO is a reinforcement learning method particularly well-suited for large language models (LLMs). In this setup, the environment is represented by the prompts the model receives. An action corresponds to generating the next token relative to the previous tokens, and each new token generated represents a new state. In our case, the training is configured such that the model has a limited number of actions, defined by the max token parameter. We evaluate the reward at the end of each episode, meaning after generating all tokens, by using unit test signals. We then optimize the policy (the LLM) using PPO.

PPO optimizes the policy by minimizing the objective function. The ratio component represents the momentum of the update, which helps stabilize learning. The loss function is based on the Trust Region Policy Optimization (TRPO) method [19], with PPO adding a clipping mechanism to control the updates when the advantage is positive. The advantage is a metric that indicates how much the policy underestimated or overestimated the reward of the current policy. When the advantage is positive, the policy should be updated by at most $1 + \epsilon$; when the advantage is negative, the update should be penalized by at least $1 - \epsilon$.

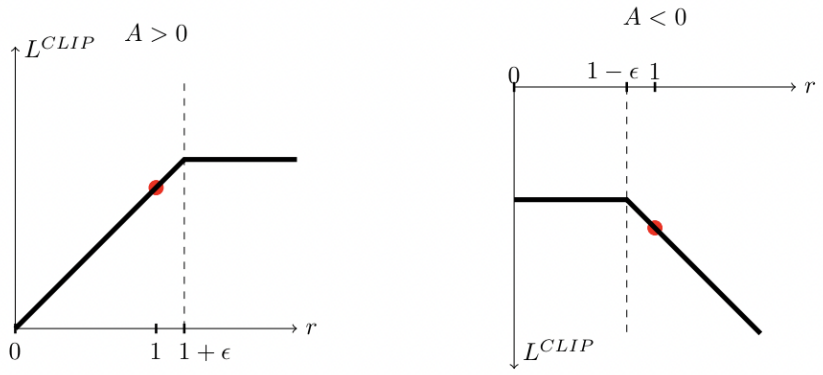


Figure 4.1: Objective function for both negative advantages and positive advantages. [18]

## 4.1 Dataset Generation

Generating a diverse synthetic dataset for training Code-LLMs poses significant challenges, particularly in balancing diversity with relevance and accuracy. Traditional methods such as increasing the generation temperature often lead to a more uniform distribution of tokens, which can result in the generation of irrelevant data. To counter this, we employed topic sampling, a technique that involves maintaining a list of topics and subtopics from which we randomly select for each prompt. This method ensures controlled diversity and high-quality output specific to our target domain.

In this project, we utilized the Mistral 7B instruct model installed at the LRE, which allowed us to interact with the model through VLLM's REST endpoints for data generation. Initially, we prompted the model to produce a broad range of topics related to coding exercises. However, some categories proved challenging to evaluate using simple unit tests due to their requirements for specific environments, such as I/O operations, databases, or front-end coding tasks.

To streamline our approach, we concentrated on Data Structures and Algorithms—a domain commonly featured in coding interviews and suitable for evaluation through straightforward unit tests that do not require specialized environments. We meticulously selected and categorized a number of main topics within this area, then generated multiple subcategories under each to enrich our dataset's diversity.

Given the constraints of using a relatively smaller 7B model, we adopted a multi-prompt strategy. This approach involves incrementally building each exercise through multiple stages, significantly increasing the number of prompts but ensuring a high-quality dataset. This contrasts with approaches using larger models, such as the Mixtral-8x7B-Instruct-v0.1 for Cosmopedia [3], GPT-4 and GPT-3.5 for phi 1.5 [7], and Llama 2 70B for CodeLlama [17], which typically require fewer prompts to generate comprehensive outputs.

Our dataset generation process incorporated three main attributes for each exercise: a "docstring" detailing the function signature with arguments and a doc-string describing the task, a "description" providing detailed information about the exercise, and "unit tests" comprising a test suite to assess the proposed solutions. The generation process is divided into three steps: starting with the creation of a detailed exercise description, followed by prompting the model to format this into a completion exercise resembling the HumanEval dataset structure, and concluding with the generation of unit tests.

Ensuring consistent output formats from smaller models can be challenging. To address this,

we instructed the model to deliver outputs in a JSON markdown box, which aids in maintaining a uniform format. Additionally, we employed specific sampling strategies, such as a top p of 0.9 and a temperature of 0.6, to consistently produce high-quality outputs. This methodological approach allowed us to generate 90 exercises across three categories, with each category containing ten subcategories and three exercises per subcategory, requiring over 210 prompts in total.

## 4.2 Reward Model

In typical Reinforcement Learning from Human Feedback (RLHF) setups for model alignment, the process often involves training a reward model on a dataset annotated according to human preferences. This trained model then serves as a basis for aligning the targeted model through proximal policy optimization on a larger scale, mimicking human evaluative patterns to guide the model's behavior.

However, in our specific application with code generation models, we diverge from this conventional method by leveraging a unique aspect of coding—unit tests—as a direct signal for performance evaluation. This approach eliminates the need for a traditional reward model. Instead, we define a straightforward reward function that quantitatively assesses the performance of the generated code based on several criteria, each reflecting common types of errors and successes encountered in coding tasks. This method is tailored to the intrinsic properties of coding, where outputs are inherently more structured and measurable.

Our reward function operates on the following scale:

- **Timeout**: A penalty of -2.
- **Compile-time syntax errors**: A penalty of -1, penalizing code that is not syntactically correct.
- **Runtime errors**: A penalty of -0.6, for code that compiles but fails during execution.
- **Assertion errors (test failures)**: A penalty of -0.3, applied when the code does not pass all provided unit tests.
- **Passing all unit tests**: A reward of +4, awarded when the code meets all specified criteria and successfully passes every test.

By directly integrating these specific metrics into our reward function, we can precisely guide the model's learning process towards producing functionally correct and syntactically accurate code, effectively utilizing the natural feedback mechanism inherent in programming. This method not only simplifies the training process but also enhances its relevance and effectiveness by directly addressing the unique challenges of code generation.

## 4.3 Training

For this method, the training dataset was generated using the Mistral 7B instruct model, with a focus on data structures and algorithms, as these are extremely well-suited for test suite evaluation. We opted for the DeepSeeker Coder Base 1B model [8], a 1.3 billion parameter model specifically tailored for coding completion tasks. This allows us to directly pass the function

Table 4.1: Deepseeker-Coder-1b-base [8] performances [2]

| Win Rate | humaneval-python | java | javascript | cpp |
|---|---|---|---|---|
| 16 | 32.13 | 27.16 | 28.46 | 27.96 |

and doc-string as a prompt, and the model smoothly completes it to solve the exercise. Additionally, this eases the process of parsing the answer since the model is extremely consistent in its outputs.

To enhance efficiency, we applied Low Rank Adaptation (LoRA) [10], reducing the model's trainable parameters to just 2.7 million. The training was conducted with a learning rate of 1e-5 and a batch size of 8. Although the training duration encompassed only a few epochs, it was sufficient to reveal several noteworthy issues.

A significant initial challenge was the absence of common libraries such as math, typing, datetime, and collections, which led to undefined errors when running the reward function. To resolve this, we prefixed each output from the model with these libraries without incorporating them into the completion exercises, to prevent the model from defaulting to solutions that overly rely on these libraries.

During the training, we encountered frequent indentation errors. Since correct indentation is crucial for Python coding, these errors were particularly relevant. We also observed initial parsing errors, where the model failed to format Python code correctly within markdown blocks. These errors diminished when we started working on a "base" model that was trained to complete code which was extremely suited to coding completion exercise as we can find in HumanEval. Small instruct model are still inconsistent when trying to have a desired output format.

Another issue we encountered was timeouts. In some coding exercises, the model occasionally generates infinite loops, which are undesirable for both functionality and security. Another source of infinite loops occurs when the model generates a solution using the input function in Python, causing the training to get stuck while waiting for input from stdin. To bypass this problem, we used multiprocessing to simulate a timer for the running thread. Additionally, we try to reduce the occurrence of this error by applying a heavy penalty whenever we encounter such issues.

However, the most persistent problem we faced was related to the quality of the generated unit tests. These tests were often misleading or incorrect, sometimes referencing undefined functions. For example, in an exercise involving Breadth-First Search (BFS), the model assumed the existence of a tree creation method within our context, which led to runtime failures and unjust penalties for the model. This highlighted a potential limitation of the current model's capacity to handle complex logical structures and the need for a larger model with more resources for more effective execution.

For this method, we prohibited the model from generating exercises, solutions, or tests that would require external libraries, as this would necessitate an adapted environment. However, this restriction led to some issues, particularly in generating test suites. For example, one exercise involved finding the eigenvalues of a matrix. Since the model was not allowed to use external libraries such as NumPy, it ended up hard-coding the answers, which we then ran assertions on. Some of these hard-coded answers were actually incorrect. In this type of mathematical exercise, allowing the use of NumPy would significantly enhance the quality of the test suite.

Additionally, we noticed an unintended learning behavior where the model began to frequently implement the completion exercise with only a "pass" statement. This tactic resulted

in only an assertion error during unit testing, leading to a lesser penalty of -0.3. This strategy minimized the negative reward impact, indicating a need to adjust the reward structure to discourage such minimalist output strategies and promote more comprehensive coding solutions.

## 4.4 Results

The training results did not show signs of learning. As seen in the average reward plot, the reward is not increasing but instead stabilizing around -0.3, which corresponds to the penalty we assigned for assertion errors. This indicates that the model is generating syntactically correct code without triggering runtime errors, but it is only encountering assertion errors during unit tests.

One potential solution to this issue would be to implement a more complex reward scheme. In this experiment, we treated the unit test score as a binary outcome: either the model passes all tests and receives the highest reward, or it receives a -0.3 penalty. We might need to introduce intermediate rewards based on the number of tests passed.

We also observed some high variation spikes in the rewards. We believe this is due to the high complexity variation in the coding exercises. When an exercise is easy, the unit tests are also relatively easy to pass, which is significant since we noticed that some exercises contained incorrect unit tests. Addressing these issues could lead to more consistent and meaningful training results.

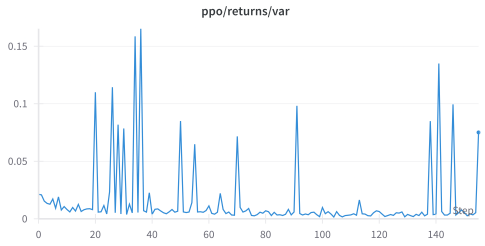


Figure 4.2: The observed high variance in rewards may be attributed to the varying difficulty levels of coding exercises.

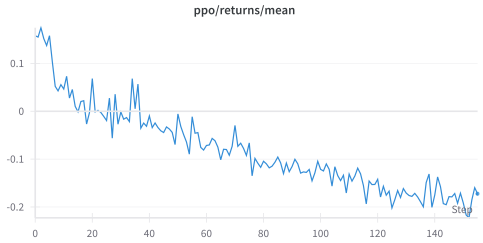


Figure 4.3: The mean reward does not show a consistent increase, indicating that the current PPO setup may not be maximizing expected rewards efficiently.

## 4.5 Limitations

Throughout the training process, we encountered several challenges and unexpected behaviors that underscored the limitations of our current approach. One significant hurdle is the necessity for highly precise and accurate synthetic data. Without this level of precision, the utility of the data for effective training is greatly compromised.

The main issue we faced was the quality of unit tests. Incorrect unit tests can completely halt the model's training, as it would be directly penalized by assertion errors. To improve this method, we need to enhance the quality of the tests. One approach could be to use larger models, such as OpenAI's API, to generate high-quality test suites. Another solution would be to fine-tune smaller, capable models such as the DeepSeeker Code 7B model [8] specifically for

the task of test suite generation. This would allow us to generate these tests locally in a scalable manner.

Another issue pertains to the use of unit tests for model alignment. While unit tests can effectively ensure the functionality of the implemented functions, they primarily align the model with performance objectives. However, they offer little to no insight into crucial aspects such as security and ethics. Previous studies have shown that Code-LLMs can generate functionally correct code that contains vulnerabilities even from benign prompts [21], highlighting a gap in the alignment process.

Moreover, unit tests are limited in their scope of evaluation, typically suited only for specific coding domains. Many important areas, such as web development, low-level hardware coding, and machine learning, present unique challenges that are difficult to assess through standard unit testing alone. Even simple I/O exercise would require to have an environment with pre-existing files and generating test suites while using the environment context for accurate testing.

Lastly, the use of reinforcement learning itself introduces instability, which can lead to significant deviations from the initial model characteristics. This instability can undermine the consistency and reliability of the training outcomes. However, alternatives such as Direct Preference Optimization [16] have been suggested as potential solutions to address these issues, offering a more stable approach to model alignment that may better accommodate the broader complexities of code generation.

# Chapter 5

# Iterative Adversarial Direct Preference Optimization

As highlighted in the previous chapter, while Proximal Policy Optimization (PPO) offers a dynamic approach to model training, it is not without complexities and instabilities. An effective alternative to address these challenges is Direct Preference Optimization (DPO), which significantly simplifies the process by reducing the reliance on a traditional reward model.

Unlike PPO, which necessitates a separate reward model to drive reinforcement learning, DPO integrates the concept of preference directly into the training process. The model itself acts as both the evaluator and the entity being optimized, streamlining the alignment mechanism. DPO operates by using a dataset that includes prompts alongside paired responses: a preferred response and a rejected one. This setup converts the optimization challenge into a straightforward binary classification task, where the model learns to distinguish between desirable and undesirable outputs based on direct comparisons. Over time, this process guides the model to produce outputs that closely align with the preferred responses, refining its output distribution effectively.

$$\mathcal{L}_{DPO}(\pi_\theta; \pi_{ref}) = -E_{(x,y_u,y_l)\sim\mathcal{D}}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_u \mid x)}{\pi_{ref}(y_u \mid x)} - \beta\log\frac{\pi_\theta(y_l \mid x)}{\pi_{ref}(y_l \mid x)}\right)\right] \tag{5.1}$$

As mentioned previously, our reinforcement learning problem is transformed into a supervised learning scenario with binary classes. In equation 5.1, we have two terms inside our sigmoid function. The left term represents the probability of the model generating the preferred output, while the right term corresponds to the probability of the model generating the rejected output. When optimizing our model to minimize this function, we aim to maximize the left term while simultaneously minimizing the right term. This approach drives our model to converge towards the preferred distribution while diverging from the initial distribution, particularly in the context of IADPO.

This chapter will delve into the intricacies of implementing Iterative Adversarial Direct Preference Optimization, exploring how this method not only addresses the limitations of PPO but also enhances the overall efficiency and stability of training Code-LLMs. We will discuss the specific adaptations made to employ DPO in an adversarial framework, which further refines the model's ability to generate high-quality, aligned code outputs.

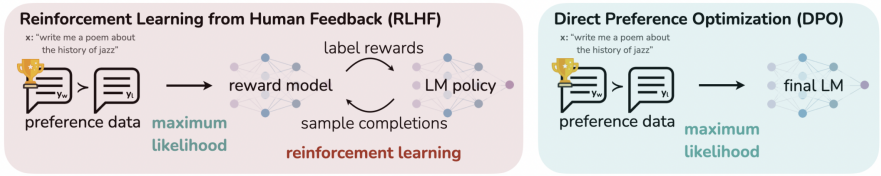


Figure 5.1: PPO RLHF VS DPO [16]

## 5.1 Adversarial Game

The objective of Iterative Adversarial Direct Preference Optimization (IADPO) is to infuse adversarial dynamics into the established Direct Preference Optimization (DPO) algorithm. This is achieved by orchestrating an adversarial game between two models: an Oracle and a Student. The Oracle, being a larger and more proficient model in the domain of coding, is tasked with generating coding exercises (prompts) and providing solutions that are assumed to align with expert human standards. These solutions are labeled as "chosen" answers. Conversely, the Student, which is a smaller model, attempts to solve the same exercises, and its solutions are labeled as "rejected" due to their presumed lesser alignment with expert preferences.

Our DPO setup involves a dataset with three key columns: the prompt (coding exercise), chosen (the Oracle's solution), and rejected (the Student's solution). This structured format facilitates the execution of DPO training iterations. To introduce adversariality, we conduct multiple rounds of these training iterations. At the end of each iteration, we calculate the loss for each training sample, which enables us to determine the rewards for each. A critical metric in this setup is the margin reward, which quantifies the difference between the rewards of the chosen and rejected answers. A higher margin indicates that the model is effectively converging towards the chosen solutions and diverging from the rejected ones.

Subsequently, we apply the softmax function to convert these margin rewards into a probability distribution, where exercises with lower margins are assigned higher probabilities. This probabilistic approach allows us to selectively sample a diverse array of exercises that the Student model found challenging. To ensure the retention of topic diversity across the exercises, this sampling process is applied to the entire batch as a whole rather than to individual samples. This batch-level operation helps maintain a diversified range of topics and challenges across the dataset.

The final step involves using these selectively sampled exercises to generate new prompts that target the identified weaknesses of the Student model. This cyclical process not only enhances the Student's performance over iterations but also ensures that the training dynamically adapts to the evolving capabilities of the model, maintaining a robust and diverse set of challenges.

## 5.2 Dataset Generation

The dataset generation process for our Iterative Adversarial Direct Preference Optimization (IADPO) was significantly enhanced by the resources available through our participation in the Mistral AI hackathon. We had access to a Groq API key, enabling us to utilize extremely fast inference endpoints for large models such as Mixtral 8x7B [11] and Llama 3 70B [1].

### 5.2.1 Choosing the Oracle Model

After evaluating both models on coding exercise generation tasks, we selected the Llama 3 70B as our Oracle model. This decision was based on its superior performance on standard coding benchmarks like HumanEval, indicating a higher quality of coding exercises and solutions generated by this model.

### 5.2.2 Implementing the Data Generation Pipeline

We utilized LangChain to implement a straightforward data generation pipeline. The high proficiency of the Llama 3 70B allowed us to generate coding exercises and the Oracle's solutions using just a single prompt. We ensured that the model output respected a JSON format with specific attributes by using function calling. This streamlined the process, making it more efficient and reducing the complexity typically associated with multi-layer prompt generation.

### 5.2.3 Enhancing Diversity through Profession-Based Prompts

To further enhance the diversity of the coding exercises, we introduced an innovative element to our data generation—profession-based prompts. By incorporating professions into the prompts, such as generating coding tasks specifically tailored for an astronaut with an array-focused problem, we were able to not only maintain the functionality and algorithmic rigor of the exercises but also vary the context and application of the coding tasks. This was achieved by sampling from a handcrafted list of professions, adding a unique and practical twist to each exercise.

### 5.2.4 Generating Student Responses

For the student model, we selected the Mistral 7B instruct. Using a HuggingFace text generation pipeline, we passed the Oracle-generated prompts to this model and extracted solutions, which were then labeled as 'rejected' in our dataset. This setup completed the initial dataset required for the first iteration of IADPO.

### 5.2.5 Iterative Enhancement and Adversarial Exercises

Following each iteration, and unless it was the final one, we enriched the dataset with new, adversarially generated exercises. This was done by identifying exercises with low margin rewards from the previous rounds and using them as references for the Oracle to generate additional content. By continuously integrating these new samples into our dataset, we effectively increased its robustness and relevance for subsequent training iterations, ensuring that the Student model progressively improved and adapted to more challenging coding scenarios thus inducing an implicit curriculum learning [20].

This iterative and dynamic approach to dataset generation not only optimized the training process but also ensured that our models were exposed to a wide array of coding problems, reflecting real-world diversity and complexity.

## 5.3 Training Setup

Our project benefited significantly from access to robust computational resources, including an NVIDIA H100 GPU, which provided considerable flexibility in our training setup. We utilized

the HuggingFace Transformers Reinforcement Learning (TRL) library to implement our adaptation of Direct Preference Optimization (DPO), ensuring an efficient integration of the latest AI training methodologies.

As discussed earlier in the report, we employed strategies such as Low Rank Adaptation (LoRA), gradient accumulation, and quantization to optimize memory usage and accelerate the training process. These methods were crucial in managing the computational demands, particularly given the iterative nature of our DPO implementation.

In our DPO setup, we faced a decision regarding the use of adapters across multiple training iterations: creating a new adapter for each iteration and merging it into the base model at the end of that iteration, or using a single adapter throughout all iterations and merging it only after the final iteration. We opted for the latter approach as it presented a simpler and more manageable method under our current constraints. This choice was supported by the capabilities of the H100, which allowed us to train with a batch size of 4 using the Adam optimizer with cosine decay. The learning rate was set at 5e-5, and a bf16 quantization to further enhance training efficiency. A beta parameter of 0.1 was used to stabilize the training dynamics. We trained our model on a maximum of 15 iterations.

## 5.4 Results

Following the training phase, we conducted performance evaluations using the BigCode code generation LM harness [2] as our benchmark framework, which facilitated efficient testing on the HumanEval benchmark. After two iterations of training with the Mistral 7B model, we observed a modest but promising 2% improvement in performance on HumanEval.

However, during the more extended training sessions with the Student model, we encountered an issue with the tokenizer, which resulted in the generation of numerous random characters. The cause of this problem remains unclear.

Despite this setback, the training logs indicate a clear increase in the reward margin, demonstrating that our model is effectively learning and converging towards the desired distribution. This suggests that the core training objectives are being met, even as we continue to address the tokenizer issue.


Figure 5.2: Chosen rewards during our training, the higher the reward the more our model converge to the chosen distribution

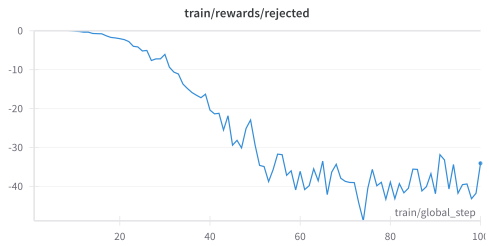


Figure 5.3: Rejected rewards during our training, the lower the reward the more our model is diverging from the rejected distribution

The interesting metric is actually the difference between these two rewards. It's called the margin reward, which represents how much our model is actually aligning with our preference data.

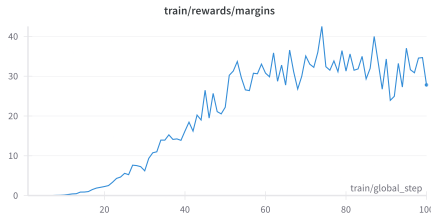


Figure 5.4: The margin being the difference between the chosen and rejected rewards, an increasing margin means we are converging toward our chosen distribution while diverging from the rejected one

## 5.5 Limitations

The effectiveness of our Iterative Adversarial Direct Preference Optimization (IADPO) approach faces several inherent limitations. One key limitation is that the alignment of the Student model is heavily reliant on the Oracle's alignment. While we can tweak this to some extent through prompt engineering, the effectiveness of these adjustments diminishes as the Oracle model size decreases.

Additionally, the current setup treats the Oracle as a static monolith that does not evolve or improve through the adversarial game. This static nature contrasts with our initial vision of creating an asymmetrical game where both policies—the Oracle and the Student—continuously improve with each iteration.

Another significant challenge encountered was the tokenization issue observed during extended training sessions, where the model began generating random characters. The root cause of this problem remains unidentified, adding an element of unpredictability to the training process.

Furthermore, our dataset management practices have room for improvement. Currently, we do not perform decontamination to ensure that our training data does not include benchmark samples. This oversight could potentially skew our model's learning in unintended ways. Additionally, the absence of a deduplication step means that despite efforts to enhance diversity, our dataset might still contain duplicate samples, which could limit the effectiveness of the training process and the model's ability to generalize from the training data.

## 5.6 Mistral Hackathon

I have had the privilege of participating in the Mistral Hackathon, where I was given extensive resources to develop my research idea. Within a span of just 24 hours, my team and I successfully developed an Iterative Adversarial DPO pipeline. This innovative system not only generates datasets and trains models but also iterates this process autonomously. Our approach demonstrated promising training outcomes, propelling us into the top four finalists among over 50 competing teams.

# Chapter 6

# Discussion

In this report, we explored two innovative methods aimed at enhancing the alignment of code-LLMs through the use of synthetic datasets and asymmetrical games, concepts inspired by advancements in the field of robotics. Both methods—Proximal Policy Optimization (PPO) and Iterative Adversarial Direct Preference Optimization (IADPO)—offer unique advantages and present distinct limitations.

While we did not fully achieve the anticipated outcomes, the groundwork laid by these initial prototypes provides a promising basis for further development. The availability of advanced GPUs at Jean Zay will facilitate more extensive testing and experimentation, allowing us to refine these approaches.

For the PPO method, there is potential to continue development with larger models, which could help ensure a more stable and reliable dataset. A similar opportunity exists for the IADPO method, where further iterations and enhancements could significantly improve its effectiveness.

A critical factor that could enhance both methods is the improvement of our dataset generation techniques. By increasing the quality and quantity of data, and incorporating steps we initially overlooked—such as deduplication—we can enhance the training outcomes across both methodologies. Furthermore, our exploration can extend to various training setups, adjustments in hyperparameters, and the employment of different model architectures.

Currently, our research has primarily focused on the functionality of the code generated by these models. However, true alignment with human preferences encompasses not only functionality but also performance and security. These dimensions have not yet been fully explored and represent vital areas for future research, potentially leading to more holistic and robust model alignment.

# Chapter 7

# Conclusion

In this report, we investigated scalable methods to align code-generating large language models (Code-LLMs) with human preferences, addressing the challenge through innovative memory-efficient training techniques such as low rank adaptation, quantization, and gradient accumulation.

We implemented and assessed two primary methods. The first method utilized unit tests as a reward function in a proximal policy optimization framework. Despite its potential, we encountered several challenges, such as issues with incorrect indentation leading to syntax errors and the model strategically generating "pass" statements to minimize penalties associated with assertion errors. These findings suggest that refining the reward system to more effectively penalize non-substantive responses could enhance model training. Additionally, inconsistencies and inaccuracies in the unit tests themselves sometimes led to unfair penalties, pointing to a need for better test validation processes.

The second method we explored was direct preference optimization, which showed that the quality of the dataset has a profound impact on performance. Challenges such as unresolved tokenizer errors highlighted areas requiring further investigation.

Looking forward, improving the dataset quality appears to be paramount for both methods. Enhancements in data generation and preprocessing could dramatically increase the effectiveness of the training process. As we refine these models, several unexplored avenues remain. For instance, the integration of performance and security metrics into the alignment process could provide a more holistic approach to model training. Additionally, the potential to apply these techniques in other domains of AI, beyond code generation, suggests a broad applicability of our findings.

These explorations also raise new questions about the scalability of such methods in different contexts and the adaptability of the models to other complex tasks. As we advance, it will be crucial to address these broader implications, paving the way for more robust and versatile AI systems.

# Bibliography


[1] AI@Meta. "Llama 3 Model Card". In: (2024). URL: https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

[2] Loubna Ben Allal et al. *A framework for the evaluation of code generation models*. https://github.com/bigcode-project/bigcode-evaluation-harness. 2022.

[3] Loubna Ben Allal et al. *Cosmopedia*. 2024. URL: https://huggingface.co/datasets/HuggingFaceTB/cosmopedia.

[4] Samuel R. Bowman et al. *Measuring Progress on Scalable Oversight for Large Language Models*. 2022. arXiv: 2211.03540 [cs.CL].

[5] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG].

[6] Ronen Eldan and Yuanzhi Li. *TinyStories: How Small Can Language Models Be and Still Speak Coherent English?* 2023. arXiv: 2305.07759 [cs.CL].

[7] Suriya Gunasekar et al. *Textbooks Are All You Need*. 2023. arXiv: 2306.11644 [cs.CL].

[8] Daya Guo et al. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. 2024. arXiv: 2401.14196 [cs.SE]. URL: https://arxiv.org/abs/2401.14196.

[9] Neil Houlsby et al. *Parameter-Efficient Transfer Learning for NLP*. 2019. arXiv: 1902.00751 [cs.CL].

[10] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL].

[11] Albert Q. Jiang et al. *Mixtral of Experts*. 2024. arXiv: 2401.04088 [cs.CL].

[12] Hung Le et al. *CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning*. Advances in Neural Information Processing Systems 35. 2022. arXiv: 2207.01780 [cs.LG].

[13] Raymond Li et al. *StarCoder: may the source be with you!* 2023.

[14] OpenAI OpenAI et al. *Asymmetric self-play for automatic goal discovery in robotic manipulation*. 2021. arXiv: 2101.04882 [cs.LG].

[15] Long Ouyang et al. *Training language models to follow instructions with human feedback*. 2022. arXiv: 2203.02155 [cs.CL].

[16] Rafael Rafailov et al. *Direct Preference Optimization: Your Language Model is Secretly a Reward Model*. 2023. arXiv: 2305.18290 [cs.CL].

[17] Baptiste Rozière et al. *Code Llama: Open Foundation Models for Code*. 2024. arXiv: 2308.12950 [cs.CL].

[18] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.CL].

[19] John Schulman et al. *Trust Region Policy Optimization*. 2017. arXiv: 1502.05477 [cs.LG]. URL: https://arxiv.org/abs/1502.05477.

[20] Sainbayar Sukhbaatar et al. *Intrinsic Motivation and Automatic Curricula via Asymmetric Self-Play*. ICLR 2018. 2018. arXiv: 1703.05407 [cs.LG].

[21] Fangzhou Wu, Xiaogeng Liu, and Chaowei Xiao. *DeceptPrompt: Exploiting LLM-driven Code Generation via Adversarial Natural Language Instructions*. 2023. arXiv: 2312.04730 [cs.CR].