Enhancing RTL Code Generation with Large Language Models via Reinforcement Learning from Human Feedback

Yang Hsueh Mu-Feng Chua Xing-Han Huang

National Tsing Hua University

Abstract

Recent advancements in large language models (LLMs) have led to significant progress in code generation. However, generating register transfer level (RTL) code from natural language descriptions remains a challenging task due to the strict syntactic and semantic requirements of hardware description languages.

This project investigates the use of reinforcement learning from human feedback (RLHF) to fine-tune LLMs for RTL code generation. We construct a preference dataset based on 27k examples from the training dataset of RTLCoder, where each example includes a problem description and RTL code generated by GPT-3.5. Positive samples are taken from these high-quality outputs, while negative samples are generated by the target LLM to be fine-tuned. Using this preference dataset, we conducted experiments with Direct Preference Optimization (DPO) to fine-tune the LLM for RTL code generation. While other RLHF algorithms were considered, we focused our evaluation on DPO in this study.

Research objectives

The present study investigates the following objectives:

- **Objective 1:** Construct a preference dataset for RTL code generatrion to support RLHF.
- **Objective 2:** Investigate the effectiveness of RLHF techniques (e.g., PPO, GRPO, DPO) in improving RTL code generation performance.

Problem Formulation

RTL Code Generation Task

- Given a vocabulary $\mathcal{V}$ and a problem description as an input sequence x, the objective is to generate an RTL program representing the flattened solution program: $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_T)$, where each token $\hat{y}_t \in \mathcal{V}$.
- An LLM parameterized by θ generates the RTL program autoregressively, sampling each token $\hat{y}_t$ from the conditional distribution $p_\theta(\cdot \mid \hat{y}_{1:t-1}, x)$.
- A generated program $\hat{y}$ is deemed correct if $\hat{y}(i_j) = o_j$ holds for all $j \in \{1, \ldots, J\}$, where $\{(i_j, o_j)\}_{j=1}^{J}$ is the set of unit test cases.

RLHF Objective

- The traditional RLHF objective is

$$J(\pi) = \mathbb{E}_\pi[r(x, y)] - \tau D_{\mathrm{KL}}(\pi \| \pi_{\mathrm{ref}})$$

that is, we want to find a model π that generates y maximizing the expected reward $r(x, y)$ for any given input x, while keeping π close to the original model π_{ref} through the KL divergence.

Methodology

Base Model Selection

We first conducted baseline experiments on several open-source models, including CodeLlama [1], DeepSeek-Coder [2], and CodeQwen [3], to evaluate their performance on RTL code generation. As shown in Table 1, DeepSeek-Coder achieves the best overall performance, and therefore, we selected it as our base model.

Dataset Construction

To perform RLHF, we constructed a preference dataset based on 27k examples from the training data of the RTLCoder [4]. Each example consists of a problem description paired with RTL code generated by GPT-3.5. Positive samples are taken from these high-quality GPT-3.5 outputs, while negative samples are generated by DeepSeek-Coder.

Finetuning with RLHF

We initially planned to fine-tune DeepSeek-Coder with several common RLHF algorithms, including DPO, PPO, and GRPO, using the constructed preference dataset. As a first step, we trained a reward model. However, due to time constraints, we were only able to complete fine-tuning with DPO in this study, using LoRA with a rank of 16.

Results and discussion

We use VerilogEval [5] as our benchmark, and adopt $pass@k$ as the evaluation metric. This metric estimates the probability that at least one out of k generated code samples passes the test. As shown in Table 1, our method improves the $pass@1$ performance of DeepSeek-Coder by about 2%, and achieves significant improvements on $pass@5$ and $pass@10$. However, the performance remains lower than that of the current state-of-the-art models, RTLCoder [4] and BetterV [6], both fine-tuned with supervised learning. This may be due to the lack of hyperparameter tuning during fine-tuning, or possibly because RLHF-based fine-tuning is currently less effective than supervised fine-tuning for this task.

Prompt

I would like you to implement a module named TopModule with the following interface. All input and output ports are one bit unless otherwise specified.
- input a (100 bits)
- input b (100 bits)
- input sel
- output out (100 bits)

The module should implement a 2-1 multiplexer. When sel=0, choose a. When sel=1, choose b.

Response

```
module TopModule
(
    input logic [99:0] a,
    input logic [99:0] b,
    input logic sel,
    output logic [99:0] out
);
    // Combinational logic
    assign out = sel ? b : a;
endmodule
```

Model	# Params	VerilogEval pass@1	pass@5	pass@10
GPT-3.5	N/A	26.7	45.8	51.7
GPT-4	N/A	43.5	**55.8**	**58.9**
CodeLlama	7B	18.2	22.7	24.3
DeepSeek-Coder	6.7B	30.2	33.9	34.9
CodeQwen	7B	22.5	26.1	28.0
RTLCoder	6.7B	41.6	50.1	53.4
BetterV	7B	**46.1**	53.7	58.2
DeepSeek-Coder w/DPO	6.7B	32.1	46.0	51.2

Table 1. Baseline performance comparison on the VerilogEval benchmark.

Conclusions

Our original goal was to fine-tune LLMs for RTL code generation using RL algorithms. However, applying RL requires a reliable benchmark to provide appropriate reward signals, and constructing such a benchmark is challenging. As a result, we shifted our focus to RLHF. Based on the training data of the RTLCoder model, we constructed the first preference dataset for RTL code generation, and fine-tuned DeepSeek-Coder with DPO. The fine-tuned model achieved improvements over the base model, but its performance remained lower than that of the current state-of-the-art models. For future work, we aim to further explore RL-based fine-tuning approaches for RTL code generation, tune the hyperparameters of DPO to enhance model performance, and conduct a comprehensive comparison of RLHF methods such as PPO and GRPO, to better understand their impact on RTL code generation performance.

References

[1] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[2] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[3] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang, *et al.*, "Qwen technical report," *arXiv preprint arXiv:2309.16609*, 2023.

[4] S. Liu, W. Fang, Y. Lu, J. Wang, Q. Zhang, H. Zhang, and Z. Xie, "Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[5] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Verilogeval: Evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–8, IEEE, 2023.

[6] Z. Pei, H.-L. Zhen, M. Yuan, Y. Huang, and B. Yu, "Betterv: Controlled verilog generation with discriminative guidance," *arXiv preprint arXiv:2402.03375*, 2024.