Extended Abstract

Motivation Deep RL methods for improving LLMs are emerging, with AI labs releasing RL-finetuned models excelling at coding, math, and instruction-following through RLHF techniques using preference datasets. This project emulated these successes by applying SFT, DPO, and GenRM to a base LLM, primarily aiming to enhance instruction-following capabilities where models answer general chat questions using pre-training knowledge.

Method We implemented three methodologies
Supervised Fine-Tuning (SFT): Uses standard next-token prediction but masks user prompts, applying loss only to model responses. Training data comes from `HuggingFaceTB/smol-smoltalk` dataset.
Direct Preference Optimization (DPO): Refines the SFT model using human preference data from `HuggingFaceH4/ultrafeedback_binarized`. Employs the Bradley-Terry model to learn from preferred vs. dispreferred response pairs, with KL-divergence regularization to prevent overfitting.
Generative Reward Modeling (GenRM): Combines reward modeling with generation training in a unified framework (used `psr-ai/genrm-cot` dataset). The model learns to both generate solutions and verify their correctness using Chain-of-Thought (CoT) reasoning.
Key Innovation: An adaptive inference mechanism that dynamically samples between 8-32 verification rationales based on prediction uncertainty. If variance in "Yes" token probabilities drops below 0.05, sampling stops early. This allocates more compute to ambiguous cases while efficiently handling clear examples, potentially improving both accuracy and inference speed compared to fixed sampling approaches. The overall goal is creating a more accurate, efficient instruction-following model through this progressive refinement process.

Implementation Implementation can be categoirized into three steps
Dataset Preparation: Used three datasets across stages: Smoltalk (multi-turn chat) for SFT with user turn masking, Ultrafeedback-binarized (preference pairs) for DPO, and GenRM-CoT dataset with added verification prompts ("Is this answer right?") and chain-of-thought reasoning ("Let's verify step by step").
Training and inference on validation: Built with PyTorch Lightning and HuggingFace tokenizers using 98% train and 98% validation. Implemented mixed precision training (bf16 initially, then fp8 with Nvidia's transformer-engine on H100s) and Distributed Data Parallel via Ray.io. Tested various beta values for DPO optimization and lambda for GenRM CoT. Our main metrics while inference on validation sets were edit distance, loss, winrate and verifier score.
Batch Inference Pipeline: Used VLLM for batch inference and Llama 3.1 Nemotron 70B Reward Model via OpenAI API for win-rate evaluation against leaderboard prompts.

Results We calculated winrate against the `Qwen/Qwen2.5-0.5B-Instruct`, our SFT model got 38% winrate, DPO got 45% and GenRM CoT got 39%. We attributed lesser winrate of our extension to generation of verifications even on prompts that did not require verifications. We concluded that finetuning on chain of thought prior to GenRM CoT can improve the results.

Discussion Distributed training on H100s with DDP reduced training time by 10-fold, making iterations feasible across different methods. The team used sanity-train and train-percentile parameters to control dataset size and enable model overfitting for testing. After DPO training, models lacked chain-of-thought reasoning by default since they weren't trained on CoT datasets, which may have caused a reduced winrate after GenRM CoT. Ray.io enabled centralized resource sharing across teammates and cloud providers, though required local patches for Azure private images and disk space issues. The team contributed a successful pull request to Ray.io that was merged within 8 days.

Conclusion Starting with a base model, SFT improved response quality by reducing cross-entropy loss through imitation learning, aligning model weights to expert policies. Adding DPO further enhanced performance with a 7-8% winrate increase over the Instruct model within few epochs, effectively directing the model toward chosen over rejected outputs. However, the GenRM CoT extension slightly decreased winrate, likely because the SFT+DPO model lacked inherent chain-of-thought reasoning, causing repetitive outputs during verification training.

RL Fine-Tuning of Language Models with GenRM-CoT

Prabhjot Singh Rai
CGOE
Stanford University
prabhjot@stanford.edu

Anirban Chatterjee
CGOE
Stanford University
ani1991@stanford.edu

Abstract

In this paper we have applied three different finetuning methods - SFT, DPO and GenRM on an LLM to improve the accuracy for instruction tuning task. For GenRM based extension we applied a CoT based augmentation to our training data (with a modified loss function) and adaptive computation during inference time to improve CoT efficiency. Based on our current experiments we observed that DPO based method seems to perform better than any other techniques. We were able to achieve a performance of 45% against Qwen 2.5 0.5B Instruct model using SFT+DPO. Our model scored 15.7% accuracy in leaderboard using SFT+DPO and 12.7% using GenRM + CoT based method.

1 Introduction

Application of Deep RL methods to improve the capabilities of Large Language Models (LLMs) is a newly emerging field and we have seen frontier AI labs releasing multiple versions of models finetuned using RL methods which can successfully solve coding and math challenges, follow user instruction and perform many other kinds of tasks. Majority of the successes can be categorized broadly under RLHF (Reinforcement Learning under Human Feedback) where the model is trained using a preference dataset to teach it what kind of output is expected against which kind of question. In our project we tried to emulate the success by taking a base LLM model and applying multiple different techniques including SFT, DPO and GenRM. Our primary objective was to improve the capability of the model for instruction following where user asks a general question in chat format and model tries to answer based on it's existing knowledge from pre-training.

The main research question that our approach answers is whether, along with SFT and DPO, can the inference metrics for an LLM be improved, specifically using the model as a verifier and using the prediction probability of "Yes" and "No" tokens by this verifier.

2 Related Work

One of the most pertinent contributions in this area is Generative Verifiers: Reward Modeling as Next Token Prediction [(11)]. This work introduces Generative Reward Modeling (GenRM), a paradigm shift from conventional discriminative verifiers that assign numerical scores to evaluate reward. Instead, GenRM reframes the task as a next-token prediction problem. This approach can operate both with and without chain-of-thought (CoT) rationales and leverages the prediction probability of "Yes" and "No" for correct and incorrect solutions respectively.

By leveraging the generative nature of large language models (LLMs), the authors show that GenRM not only accommodates interpretable CoT rationales but also benefits from scalable inference-time

improvements, such as majority voting across multiple sampled rationales. This marks a significant contrast with traditional discriminative reward models, which rely on fixed scoring functions and do not improve with additional inference effort.

Empirically, this work increases the performance of the model on Algorithmic Reasoning (Δ 5% to 45%), Grade School Math (Δ 73% to 93.4%), and Transfer to MATH (Δ 28% to 44.6%), surpassing discriminative models, LLM-as-a-Judge, and DPO, achieving significant gains in Best-of-N accuracy. The paper demonstrates that synthetic rationales can be sufficient for training, reducing dependence on human-labeled data. A notable strength is the unification of generation and verification within a single training objective, which enhances both capabilities via positive transfer.

A key advantage of this method lies in its precision: GenRM effectively identifies nuanced reasoning errors in mathematical solutions that other verification models often overlook. In its basic form, GenRM generates a "Yes" or "No" token to judge correctness, while the GenRM-CoT variant first produces a detailed rationale before issuing the verdict—enhancing both interpretability and accuracy of verification.

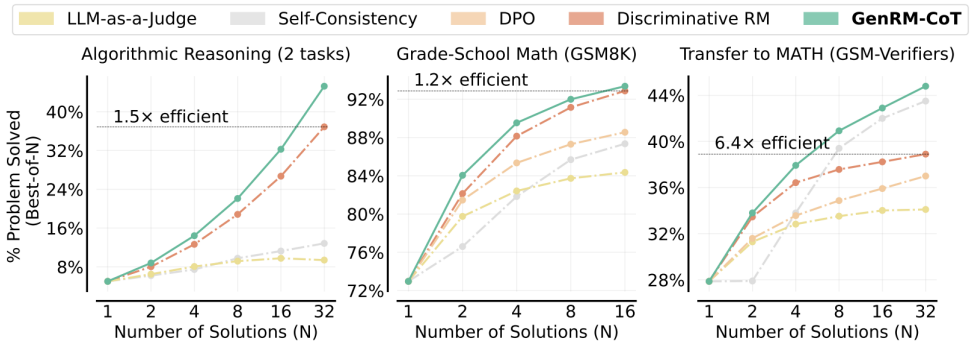

Figure 1: Sample-Efficient Scaling with Generative Verifiers

In the figure, we can see that the GenRM-CoT is sample efficient, specifically in transfer to MATH using Best-of-N strategy. It demonstrates superior sample efficiency, solving more problems with fewer solution candidates - 1.5x, 1.2x, and 6.4x more efficient for algorithmic reasoning, grade school math and transfer to MATH, respectively. These gains highlight GenRM-CoT's strength in reasoning, generalization, and inference-time compute utilization.

2.1 Difference from related work

From what we understood, one of the key aspects of this paper is that there is a fixed number of rationale that are sampled and averaged. We refined the inference time compute for GenRM-CoT as a modification to address this limitation. Instead of using a fixed number of rationales, we compute the rationales only when the variance is higher than a given number - thereby reducing the need to sample more when the model is already performing well during sampling. The details of it have been outlined in the method section under section 3.3.

3 Method

Our primary aim was to improve the accuracy of our model on the instruction following task. We used Qwen 2.5 0.5B as our base LLM model and focused on three distinct methods for our finetuning work.

3.1 Supervised Fine-tuning(SFT)

Supervised Fine-Tuning optimizes using same next-token prediction objective that is used in pre-training. However, no loss is applied to the query tokens (by masking out user prompts). This supervised learning objective is optimized over queries x and completions y that are drawn from an

expert distribution. The objective can be written as follows:

$$\mathbf{max}_\theta \mathbf{E}_{\mathbf{x},\mathbf{y}\in\mathbf{D}} \sum_{t=1}^{|y|} \mathbf{log}\pi_\theta \left(\mathbf{y_t} \mid \mathbf{x}, \mathbf{y_{<t}}\right), \tag{1}$$

In this stage we used `HuggingFaceTB/smol-smoltalk` dataset to finetune our model.

3.2 Preference optimization using DPO

After obtaining a π_{ref} model using SFT in the first stage we moved on to improving the model quality by using preference dataset from `HuggingFaceH4/ultrafeedback_binarized`. To fine-tune π_{ref} with human preferences, usually a preference dataset $\mathcal{D}_{\text{pref}} = \{x^{(i)}, y_w^{(i)}, y_l^{(i)}\}$ is collected, where $x^{(i)}$ denotes a prompt and $y_w^{(i)}, y_l^{(i)}$ denote preferred and dispreferred responses, often obtained by sampling from π_{ref}. Given a preference dataset, most fine-tuning pipelines assume the existence of an underlying reward function $r^*(x, \cdot)$. One popular framework for this is the Bradley-Terry (BT) model (8), assuming that human preferences can be written as:

$$p^*(y_1 \succ y_2|x) = \frac{e^{r^*(x,y_1)}}{e^{r^*(x,y_1)} + e^{r^*(x,y_2)}} \tag{2}$$

Given this reward function r^*, preference fine-tuning aims to find the optimum of the reward r^*. While the ultimate goal of preference fine-tuning is to find the unconstrained optimum of the reward function, in practice, we often replace the reward function with a reward model. Since the reward model is erroneous, we apply a KL-divergence constraint on the policy to prevent exploitation in the reward model. To align our results with typical preference fine-tuning procedures, we will consider such a KL-constrained reward optimization as our fine-tuning goal:

$$\max_{\pi_\theta} \mathbb{E}_{x\sim\mathcal{D}_{\text{pref}}, y\sim\pi_\theta(\cdot|x)}[r^*(x, y)] - \beta D_{KL}[\pi_\theta(\cdot|x)||\pi_{\text{ref}}(\cdot|x)] \tag{3}$$

The regularizer, weighted by β, controls the deviation of π from π_{ref} under the reverse KL divergence.

Reward model training: In order to fine-tune an LLM policy $\pi_\theta(y|x)$, Eq. (1) provides a convenient way to learn a reward model either explicitly (i.e., by fitting a parametric reward model $r_\varphi(x, y)$) or implicitly (i.e., via direct preference optimization (DPO) (10) or IPO (7), that re-purposes the log-likelihood $\log \pi_\theta(y|x)$ of the policy to represent the reward $r_\theta(x, y)$). Explicit reward models are trained using the following classification objective:

$$\max_{\varphi} \mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}_{\text{pref}}}[\log \sigma(r_\varphi(x, y_w) - r_\varphi(x, y_l))] \tag{4}$$

where σ is the logistic function. Contrastive learning objectives on the other hand repurposed $\log \pi_\theta(y|x)$ as the implicit reward $r_\theta(x, y)$:

$$r_\theta(x, y) = \beta[\log \pi_\theta(y|x) - \log \pi_{\text{ref}}(y|x)] \tag{5}$$

In (10), the constrained RL problem has been reformulated as a supervised preference classification problem on human preference data. More formally, the DPO loss is:

$$\mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}} \left[\log \sigma \left(\beta \log \frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \log \frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right)\right] \tag{6}$$

where x is the prompt, y_w is the preferred responses, y_l is the dispreferred responses, π_θ is the policy that is being optimized and π_{ref} is the reference policy.

We used the loss in equation (6) on this stage to further finetune the obtained model from SFT.

3.3 Extension of performance using GenRM (Generative Reward Modeling)

The paper[(11)] shows that GenRM can seamlessly integrates reward modeling, which distinguishes between correct and incorrect solutions, with SFT for generating correct solutions. This can be done by simply changing the data mixture in the SFT loss (1) to include both verification and generation tasks. Given a verification dataset $\mathcal{D}_{\text{verify}}$, which can be $\mathcal{D}_{\text{Direct}}$ or $\mathcal{D}_{\text{CoT}}$ (discussed below) of problems-solution pairs with correctness tokens (optionally with CoT rationales), GenRM minimizes the loss:

$$\mathcal{L}_{\text{GenRM}}(\theta, \mathcal{D}_{\text{verify}}) = \mathcal{L}_{\text{SFT}}(\theta, \mathcal{D}_{\text{verify}}) + \lambda\mathcal{L}_{\text{SFT}}(\theta, \mathcal{D}_{\text{correct}}) \tag{7}$$

where $\lambda > 0$ is a hyperparameter that controls the mixture ratio between verification ($\mathcal{D}_{\text{verify}}$) and generating correct solutions ($\mathcal{D}_{\text{correct}}$). This unified training can improve verifier and generation performance via positive transfer between these two related tasks: how to generate a correct solution, and whether a solution is correct. By default, we train GenRM verifiers using the unified loss in (7).

Authors show that we can generate intermediate reasoning steps or critique (CoT) before making a decision about the solution correctness, which may identify subtle reasoning errors missed by direct verifiers. To train CoT verifiers, we can minimize the SFT loss $\mathcal{L}_{\text{GenRM}}$ on the dataset $\mathcal{D}_{\text{CoT}}$ containing problem-solution pairs as inputs, and corresponding verification rationales v_{CoT} appended with a final question I and 'Yes' or 'No' token as targets:

$$\mathcal{D}_{\text{CoT}} = \{(x, y^+, I_{\text{CoT}}), (v_{\text{CoT}}, I, \text{'Yes'})\} \cup \{(x, y^-, I_{\text{CoT}}), (v_{\text{CoT}}, I, \text{'No'})\} \tag{8}$$

where $I_{\text{CoT}} =$ 'Let's verify step by step.'. Notably, these rationales can either be human or LLM-generated, both of which we explore in this work. During inference, we first generate a CoT rationale v_{CoT} from GenRM-CoT and then use the probability of 'Yes' for assigning the correctness score:

$$r_{\text{CoT}}(x, y) = p_\theta(\text{Yes}|x, y, I_{\text{CoT}}, v_{\text{CoT}}, I), \quad \text{where } v_{\text{CoT}} \sim p_\theta(\cdot|x, y, I_{\text{CoT}}) \tag{9}$$

Compared to (4) that only uses the instruction I to produce a score, the above CoT reward additionally conditions on I_{CoT} and self-generated v_{CoT} before getting a score via instruction I.

Inference-time compute for CoT verifier. When sampling verification CoTs, the generative verifier can use different reasoning paths and yield different correctness probabilities for the same problem-solution pair. As such, we would like to marginalize out these reasoning paths to select the most consistent correctness answer (Wang et al., 2022). To do so, we use majority voting where we first generate K verification CoT rationales, and average the CoT-verifier score for these rationales:

$$r_{\text{MajV@K}}(x, y) = \frac{1}{K} \sum_{i=1}^{K} p_\theta \left(\text{Yes}|x, y, I_{\text{CoT}}, v_{\text{CoT}}^{(i)}, I\right), \quad \text{where } v_{\text{CoT}}^{(i)} \sim p_\theta(\cdot|x, y, I_{\text{CoT}}) \tag{10}$$

Modifications on top of original work: We refined the inference time compute for GenRM-CoT as a modification to address the limitation of using fixed number of rationales even when the variance of the score was low. The current paper uses a set number of rationales for sampling and averaging, whereas we capped it within a set number of K_{min} and K_{max} based on the variability in the token prediction probability. Specifically, setting K_{min} to 8 and K_{max} to 32 and increasing from K_{min} to K_{max} only if the variance in the predicted probability of the 'Yes' token remains above a predefined threshold. Moreover, instead of drawing one sample at a time, we drew 8 samples since in FP8 training, the requirement for generation was a matrix with a height of 8.

This above adaptive computation mechanism would allow the model to spend more compute on ambiguous cases while avoiding unnecessary sampling on confidently classifiable examples, thus reducing average inference latency.

We define the threshold τ as an empirical value indicating uncertainty, and in our experiments we consider $\tau = 0.05$. That is, if

$$\text{Var}(p_{\text{Yes}}^{(1)}, \ldots, p_{\text{Yes}}^{(K)}) < \tau,$$

we stop early and use the current estimate. This adaptive compute mechanism aims to reduce average inference cost while maintaining accuracy, particularly by allocating more compute to borderline or ambiguous cases.

4 Experimental Setup

Our experimental set-up can be split into four steps

4.1 Dataset preparation

- SFT Dataset Preparation: In this step, we used the Huggingface Smoltalk dataset (`HuggingFaceTB/smol-smoltalk`). This is a multi-turn chat response dataset between users and assistant. We implemented a masking strategy which can be extended for both single and multi-turn chats to mask out labels for user turns. We also applied sampling strategies to help us quickly debug the training and validation process (example, using a subset of dataset for training).

- DPO data preparation: We used `HuggingFaceH4/ultrafeedback_binarized` dataset which included a prompt and two responses (one preferred over another). We applied similar masking and sampling strategies here.
- Extension using GenRM: In this stage we used the dataset provided by the original authors(1) and transformed that to a HF repo here (`psr-ai/genrm-cot`). We used the new loss function for training on this dataset.

4.2 Training and inference on validation

We used Pytorch lighting and huggingface's in-built tokenizer build our training pipeline. We generally used 98% of the dataset on both Smoltalk and Ultrafeedback to train our SFT and DPO stages. We implemented a mixed precision training method where we initially started with bf16 in order to save memory and be able to accommodate larger set of tokens. But later on moved on to Fp8 based training using nvidia's transformer-engine library on H100. We also used a Distributed Data Parallel (DDP) strategy by using Ray.io for all 3 methods. For DPO we trained with different beta values to see if we observe increased accuracy. For extension we used the DPO model as the reference model against which training process is optimizing.

For validation, we used multiple **metrics**, such as eval edit distance, evaluation loss, winrate and verifier scores. These were computed based on the experimental setup and the training type. Our baseline model was `Qwen/Qwen2.5-0.5B` and we expected our model to improve on the different mentioned metrics with time.

4.3 Uploading model to HuggingFace

During the training pipeline, the models are saved as checkpoints in AWS S3. We created the pipeline to move the models from a given checkpoint to HuggingFace repository as the next step.

4.4 Batch inference pipeline and win-rate calculation

In this stage we used VLLM to efficiently run inference for prompts to create output for the leaderboard submissions. Additionally we used Llama 3.1 Nemotron 70B Reward Model (https://huggingface.co/nvidia/Llama-3.1-Nemotron-70B-Reward) using OpenAI API for win-rate calculation.

5 Results

5.1 SFT - Supervised finetuning

5.1.1 Precision: bf16-true

Our first metrics were average train loss for all batches and validation edit distance for supervised finetuning on the `HuggingFaceTB/smol-smoltalk` but on a small subset of data (`source` being `smol-constraints`). This is only 7.5% of the dataset. We chose validation edit distance to measure the edit distance between the model's output and the saved output in the validation set. These measures are reported below and please find the log files here(6):

We quickly realized two things:

- If 1 epoch of 7.5% of dataset takes 1.5 hours, we would need to speed up the training using strategies like DDP (since our model was able to fit in one GPU with batch size of 1) and lower precision training like fp8
- Validation edit distance will not be a good measure as a validation metric because the model's output may not be exactly same as the test outputs, as they would be semantically similar but not exactly same

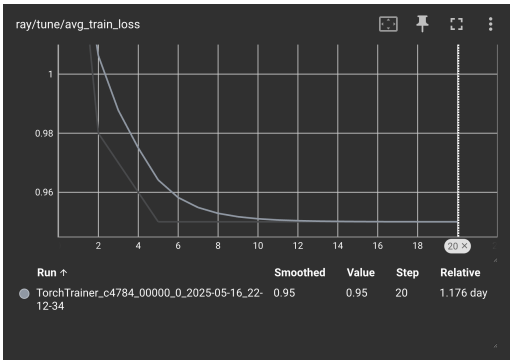

Figure 2: Train loss over time

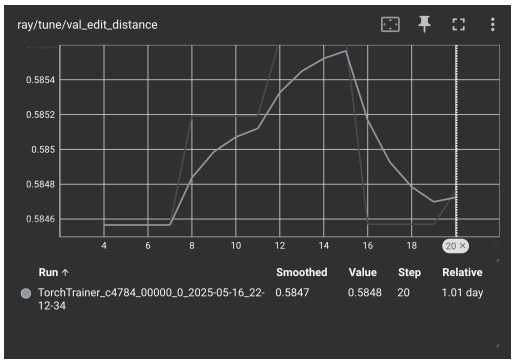

Figure 3: Validation edit distance over time

5.1.2 Precision: FP8 (transformer-engine) with DDP strategy

In order to address the first point of our findings in 5.1.1, the next step was to implement SFT with FP8 precision to decrease the training time along with performing distributed data parallel (DDP). Below are the training metrics for fp8 training and please find the log files on the drive link(3):

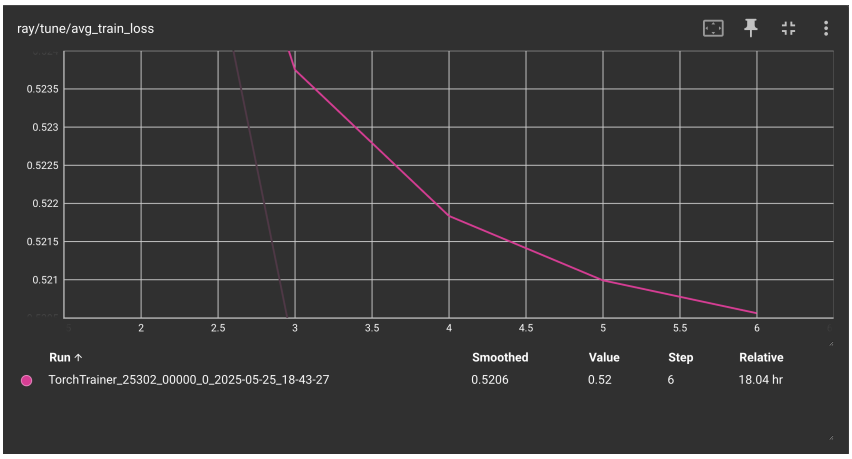

Figure 4: Train loss over time for FP8

Looking at the graph, we can see that using the strategies like DDP and FP8 training, we were able to perform a single epoch over 100% of the dataset in 4 hours. This minimum 7 times decrease in the training time (assuming that distribution of `smol-constraints` is same as the entire dataset). We used the checkpoints that we created in this training as the final SFT model that we uploaded to HuggingFace (using the script `upload_to_hub.py` in our repository).

Final uploaded model after this section: psr-ai/Qwen2.5-0.5B-SFT

5.2 DPO - Direct Preference Optimization

5.2.1 Beta: 1.0 vs Beta: 2.0, Precision FP8 and DDP strategy

We used two different values of Beta for our Direct Preferences Optimizations. We used `HuggingFaceH4/ultrafeedback_binarized` and trained the finetuned model from the previous section as the base model on 98 percentile of the dataset (based on string length). Moving ahead, we started using evaluation loss as a metric for comparison for more semantic comparisons of the outputs. Below are the comparisons of both on average train loss and average validation loss, and log file for beta 2 training can be found here (2).

Between the two betas, we went ahead with beta 1.0 because the average validation loss was lesser in this case.

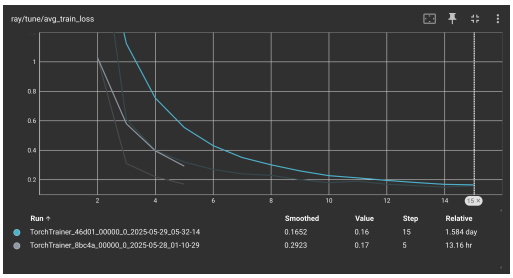

Figure 5: Train loss over time

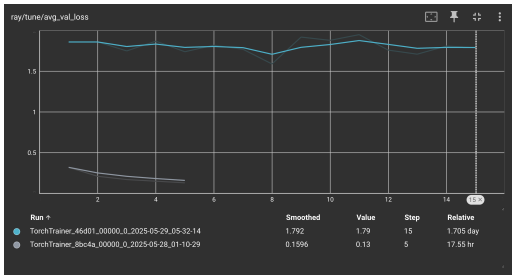

Figure 6: Validation loss over time

Final uploaded model after this section: psr-ai/Qwen2.5-0.5B-SFT-DPO

5.3 Extension - GenRM

5.3.1 Lambda 0.1 with precision FP8 and DDP strategy

Proceeding with the extension, we finetuned the model we obtained from the previous section on the 98 percentile of `psr-ai/genrm-cot` dataset based on the loss function described in the methods. Below are the training and evaluation metrics for the extension training, and please find the log files here(5):

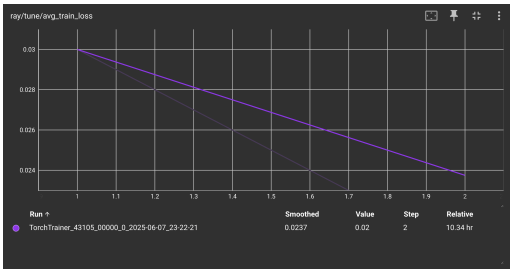

Figure 7: Train loss over time

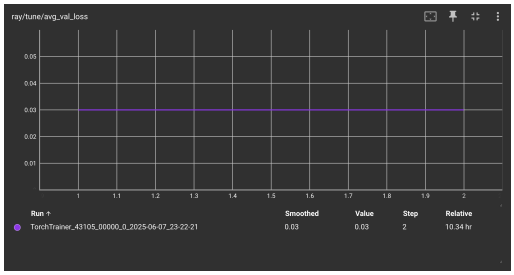

Figure 8: Validation loss over time

We are also computing verifier scores per epoch. The metrics for validation verifier score are given in Figure 8.

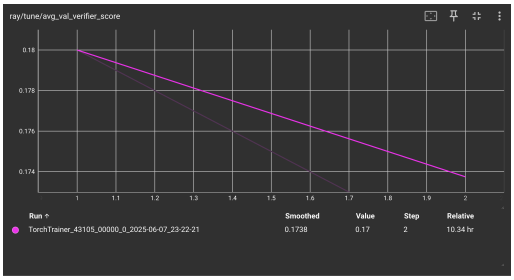

Figure 9: Validation verifier score

We observed here that the average validation verifier score decreased by 0.01 in the second step. This could be possible because since we are not training on the validation dataset so the verifier score decreased slightly. This could also be because we are also using `do_sample=True`, there is some stochasticity while calculating the validation verifier score. But for the train loss, we do observe that it is decreasing with time.

Table 1: Performance Comparison

Method	Winrate (`Instruct`)	Leaderboard (Milestone)	Leaderboard (RL + Extension)
SFT	38%	75%	N/A
DPO	45%	81.5%	15.75%
Extension (GenRM)	39%	N/A	12.75%

5.3.2 Lambda 0.01 with FP8 precision and DDP strategy

We also experimented on a smaller lambda (0.01) as a part of this extension to check if we could get better results. Below are the metrics for the same, and please find the log files here(4):

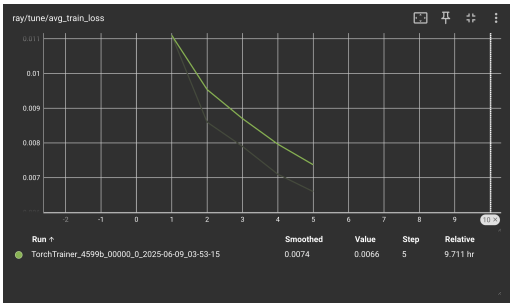

Figure 10: Train loss over time

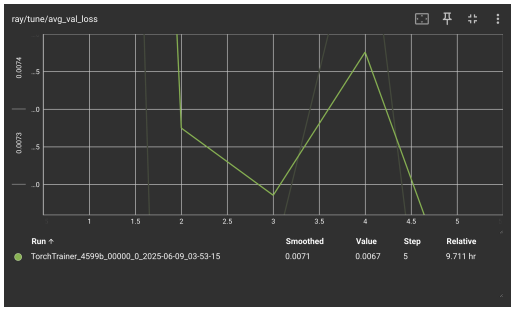

Figure 11: Validation loss over time

We were also computing verifier scores per epoch. Below are the metrics for the same:

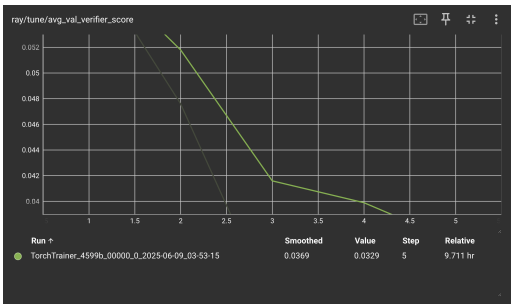

Figure 12: Validation verifier score

We observed here that the average validation verifier score was less in this lambda as compared to lambda 0.1, hence we moved ahead with finalizing the lambda 0.1 model.

Final uploaded model after this section: psr-ai/Qwen2.5-0.5B-SFT-DPO-Extension

5.4 Quantitative Evaluation

Along with the above quantitative evaluations on different experiments, we submitted the outputs from our model to the leaderboard on different training steps to compare the winrate for the generated models. We also wrote down implementation of winrate to compare a set of prompts with the `Qwen2.5-0.5B-Instruct` model and the given model.

Our observation was that the winrate increased after DPO against the `Qwen2.5-0.5B-Instruct` by 7-8 percent as described in Table 1. But the winrate decreased by similar number after running the extension model against the `Qwen2.5-0.5B-Instruct` model. This is because given our lambda, the model has become skewed towards outputting answer separately assuming that the prompt is asking for a validation output.

5.5 Qualitative Analysis

When performing SFT and DPO, the results against the `Qwen2.5-0.5B-Instruct` model are evident. We were able to get close to 40 to 45% winrate, asserting the fact that the training is going as expected and our metrics are improving. But moving onto the extension, our winrate suddenly dropped. We could attribute this to the model generating assertions on the final output by repeating the answers. For example, analyzing one of the prompt and responses which outputs a preferred answer:

Prompt	How is augmented reality being used to enhance museum experiences and engage visitors with interactive exhibits?
Response	Augmented Reality (AR) is being used to enhance museum experiences in several ways. For instance, virtual reality (VR) is being used to create immersive experiences that allow visitors to explore and interact with the exhibits in a way that is both engaging and interactive. AR overlays digital content onto the real world, creating a more lifelike and interactive experience. This can include interactive exhibits, virtual tours, and 3D models of objects.reality is also being used to create 3D models of objects and environments, allowing visitors to explore and interact with them in a more immersive way. This can include virtual reality experiences that allow visitors to explore and interact with objects in a way that is both realistic and engaging.addition, AR is being used to create interactive exhibits that allow visitors to explore and interact with the exhibits in a way that is both engaging and interactive. For instance, visitors can use AR to scan and interact with objects in the exhibits, or to explore and interact with digital content that is presented in the exhibits.is also being used to create interactive exhibits that allow visitors to explore and interact with the exhibits in a way that is both engaging and interactive. For instance, visitors can use AR to scan and interact with objects in the exhibits, or to explore and interact with digital content that is presented in the exhibits., AR is being used to create immersive and interactive experiences that can engage visitors in new and exciting ways, and can help to increase visitor engagement and enjoyment of the museum exhibits.

But for another prompt, the model outputs an imperfect answer:

Prompt	Tom has $1200 saved for a trip. He spends 1/3 of the money on flights and then 25% of the remaining money on accommodations. How much money does Tom have left?
Response	The amount of money spent on flights is 1200 * 1/3 = 400. The remaining money after flights is 1200 - 400 = 800. Tom spends 25% of the remaining money on accommodations, so he spends 800 * 25/100 = 200. The remaining money after accommodations is 800 - 200 = 600. Tom has 600 left. #### 600 The answer is: 600

We could see that the model has started generating assertions in the form of verifier dataset. This could be attributed to missing Chain of Thought ability inherently within the model, hence the model learning to output in the format of verifier datasets instead.

6 Discussion

- Finetuning on H100 with DDP (distributed data parallel) was our standard approach to decrease the finetuning time. In some cases, this led to a decrease in the training time by 10 folds. This made our training and iterations feasible with different methods
- Intitiating the training with `sanity-train` parameter combined with `train-percentile` to control the training dataset size and making evalutation dataset same as training to overfit the model on it
- The model after training using direct preference optimization did not seem to have a chain of thought by default. We checked this by prompting the model with text such as "Let's verify this step by step". This is because the model has not trained on chain of thought dataset hence we did not assume any such strategical outputs by the models on these prompts which were expected by our extension techniques.
- With ray.io, we were able to create one central cluster and the teammates were able to run parallel jobs with specific training type, precision, strategy and accelerator type - this helped us in resource sharing across accounts and cloud providers

- We saw that in ray.io, there was no support for private images in Azure along with specifying the disk os size. This was important because sometimes we were running out of disk space and we also wanted to decrease the time of spinning up the instances - in order to mediate it, we patched the ray library locally (`patches/azure_vm_template.json` in the codebase) and also opened a PR on the ray.io GitHub library which got merged in 8 days (9)

7 Conclusion

We can conclude that starting with a given model, SFT increases the quality of the responses from the model by decreasing the cross entropy loss. This is said to be an imitation learning problem, where we know the expected action (output tokens) by our expert policy and we want our model's weights to align to that strategy. Adding DPO (direct preference optimization) increased the response quality and winrate by a 7 to 8 percent against the Instruct model in a small number of epochs. Hence, DPO is effective in making the model direct towards chosen outputs against the rejected outputs.

Finally, our extension was based on GenRM CoT strategy. It asks the model to come up with a chain of thought to verify a correct solution as correct and an incorrect solution as incorrect. We observed that the winrate decreased slightly and we could attribute this to missing chain of thought strategy by the SFT+DPO model inherently hence the outputs weren't very good. The model started memorizing the format of the the responses using the verification dataset during this training.

8 Team Contributions

- **Anirban Chatterjee:** DPO, Winrate and eval calculations
- **Prabhjot Singh Rai:** SFT with FP8, DPO and Extension Implementation (dataset preparation and loss implementation)

Changes from Proposal In the proposal, specifically in the extension, we had mentioned that we would like to sample new inferences only if the variance was greater than 0.01, we increased this hyperparameter to 0.05 and also, since on FP8 the minimum height required was 8 on inference, we sampled 8 at one time instead of just one since we had the bandwidth to do so. Hence, minimum was 8 and then we would go on from 8 to 32 on a step size of 8 instead of 1 and calculate variance on each step. Also, due to time constraint, we were not able to perform winrate testing on the GSM8K dataset, but we were able to implement winrate on the given prompt dataset or using `-calculate-win-rate` on the validation dataset as well.

References

[1] [n. d.]. *GitHub repo for release of GenRM CoT dataset*. `https://github.com/genrm-star/genrm-critiques`

[2] [n. d.]. *Logfile for DPO with Beta 2*. `https://drive.google.com/file/d/14F6ObgIqfxbuo1GHTFE5-JVrtoF3EWU8/view?usp=drive_link`

[3] [n. d.]. *Logfile for SFT*. `https://drive.google.com/file/d/1TPzSLbRPc4WUH3KMOAX1-nY_pv6gX_kn/view?usp=drive_link`

[4] [n. d.]. *Logfile for sft verifier training with lambda 0.01*. `https://drive.google.com/file/d/1QPB9e7xMsYBA2gYPo7UkA4eL9kX-laO7/view?usp=drive_link`

[5] [n. d.]. *Logfile for sft verifier training with lambda 0.1*. `https://drive.google.com/file/d/1ABQ4QsJpqfOLTVXw57UZPWTJ3JSOfxZf/view?usp=drive_link`

[6] [n. d.]. *Logfile for SFT with smol-constraint dataset*. `https://drive.google.com/file/d/1ztY8HusyOl-B-xlK5b1epVEf8LD2Ydsu/view?usp=drive_link`

[7] Mohammad Gheshlaghi Azar, Mark Rowland, Bilal Piot, Daniel Guo, Daniele Calandriello, Michal Valko, and Rémi Munos. 2023. A General Theoretical Paradigm to Understand Learning from Human Preferences. arXiv:2310.12036 [cs.AI] `https://arxiv.org/abs/2310.12036`

[8] Ralph Allan Bradley and Milton E. Terry. 1952. Rank Analysis of Incomplete Block Designs: I. The Method of Paired Comparisons. *Biometrika* 39, 3/4 (1952), 324–345. http://www.jstor.org/stable/2334029

[9] Ray Project Contributors. 2024. Add Image ID and Size Parameters to Azure node provider. https://github.com/ray-project/ray/pull/53298. https://github.com/ray-project/ray/pull/53298 GitHub Pull Request #53298.

[10] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2024. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. arXiv:2305.18290 [cs.LG] https://arxiv.org/abs/2305.18290

[11] Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. 2025. Generative Verifiers: Reward Modeling as Next-Token Prediction. arXiv:2408.15240 [cs.LG] https://arxiv.org/abs/2408.15240

A Additional Experiments

We also performed some additional experiments, such as testing out our implementations using `-sanity-train` parameter. This would make the train and validation dataset as the same so that we are able to overfit on the training dataset and evaluate against the same dataset and check our implementations. Moreoever, once the sanity training was done, we were able to control the input dataset using `-train-percentile` and `-test-percentile` inputs. For example, for the extension part, we first tested out on 0.1% of the training data (`psr-ai/genrm-cot` having total 523K rows) and the training and validation losses are reported below:

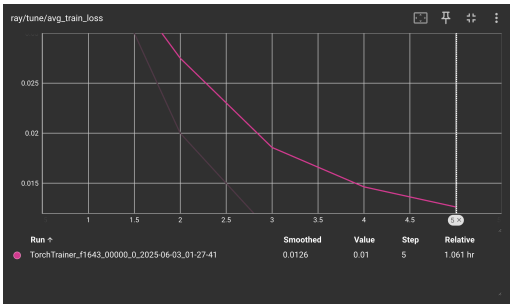

Figure 13: Train loss over time

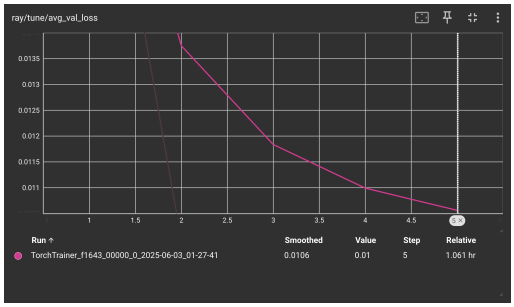

Figure 14: Validation loss over time

Once we confirmed that our training and validation losses are decreasing, we were ready to perform the training on the extensive dataset (98%).

Overall, for testing out the code, we ran many other experiments. Please follow the following steps on the terminal to checkout different experiments (for extension):

```
export AWS_ACCESS_KEY_ID=AKIA2GK3E4AHHON3EF3X
export AWS_SECRET_ACCESS_KEY=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
tensorboard --logdir="s3://rl-sf/SFT-verifier-final"
```

While testing out SFT-verfier implementation, please use the `rl-sf` bucket and `sft-verfier` folder

```
export AWS_ACCESS_KEY_ID=AKIA2GK3E4AHHON3EF3X
export AWS_SECRET_ACCESS_KEY=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
tensorboard --logdir="s3://rl-sf/sft-verifier"
```

If you want to checkout the DPO experiments, please use the bucket `rl-sf` and folder `DPO`:

```
export AWS_ACCESS_KEY_ID=AKIA2GK3E4AHHON3EF3X
export AWS_SECRET_ACCESS_KEY=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
tensorboard --logdir="s3://rl-sf/DPO"
```

and similarly for SFT:

```
export AWS_ACCESS_KEY_ID=AKIA2GK3E4AHHON3EF3X
export AWS_SECRET_ACCESS_KEY=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
tensorboard --logdir="s3://rl-sf/SFT"
```

B Implementation Details

Please use the `requirements.txt` to install the dependencies.

B.1 Autoscaling

As mentioned before, we have used ray.io for autoscaling compute. We created the cluster files for both AWS and Azure, both of them are located in `hippo` folder, example `hippo/aws.yaml` for AWS compute and `hippo/azure.yaml` for Azure compute. The commands `ray up hippo/azure.yaml` helped us bring up the instances with the autoscaler configuration - this file contains the different compute instances like A100, H100, T4 etc. The file `patches/azure-vm-template.json` was used as the patch since the PR we got merged into ray.io was not deployed yet, hence we manually patch it as specified in `hippo/azure.yaml`. We use the `ray dashboard` command to port forward from the cluster's driver instance to local and we were able to view the dashboard locally. Here's how the dashboard looks like:

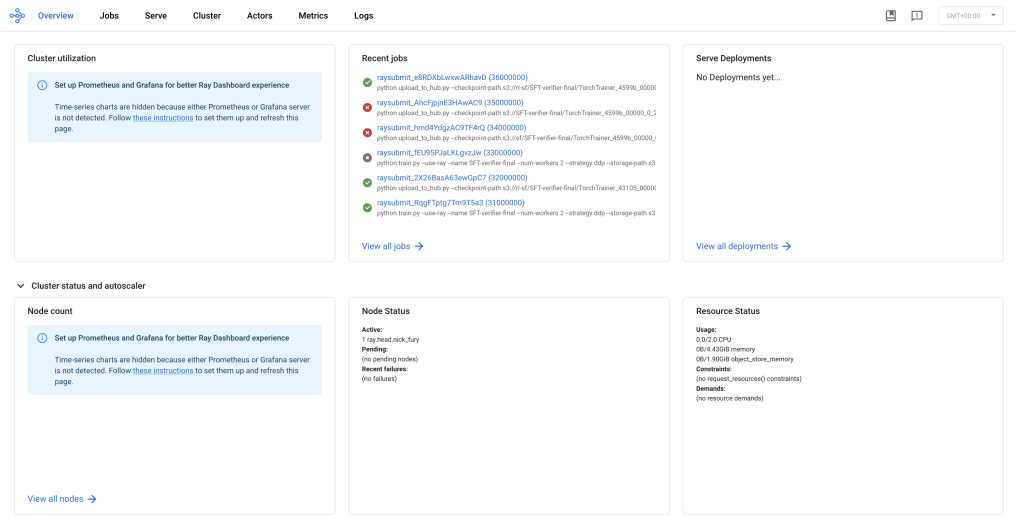

Figure 15: Ray dashboard to share experiments

B.2 Dataset preparation

B.2.1 Batch Dataset Preparation

Although for all the datasets we did not need to perform a separate batch operation of dataset preparation and storing them (we perform in-memory collation using `collate_functions.py` file), we had to do it for the extension (GenRM-CoT) since the chain of thought and verification datasets were part of the github repo. For this, please refer to `extension/data_preparation.py` file and the example run we did was:

```
HF_TOKEN=[masked] python extension/data_preparation.py --path
    /Users/psr/GitHub/genrm-star/genrm-critiques/critiques --repo
    psr-ai/genrm-cot
```

where the `-path` represents the local path to the GitHub repo.

B.2.2 In-memory Dataset Collation

We have used the `collate_functions.py` to map the dataset before sending it to the training or validation methods inside the module. Moreover, we also created a wrapper over the huggingface dataset called `custom_dataset.py` file and classname `CustomDataset` while filters and maps based on the given percentile of the data.

B.3 Training

B.3.1 Entrypoint

Once we have port forwarded, we can submit the jobs to this cluster and therefore we use `ray job submit` to submit jobs to this cluster. The training file is used to submit the jobs. Here is the extensive explanation of each of the arguments to the train.py script:

```
> python train.py --help
usage: train.py [-h] [--use-ray | --no-use-ray] [--model-name MODEL_NAME]
    [--dataset-name DATASET_NAME]
           [--training-type {sft,dpo,bt,sft_verifier}] [--storage-path
               STORAGE_PATH] [--name NAME]
           [--num-workers NUM_WORKERS] [--accelerator-type ACCELERATOR_TYPE]
           [--placement-strategy PLACEMENT_STRATEGY] [--seed SEED] [--max-epochs
               MAX_EPOCHS] [--use-gpu]
           [--batch-size BATCH_SIZE] [--lr LR] [--resume-from-checkpoint
               RESUME_FROM_CHECKPOINT]
           [--accumulate-grad-batches ACCUMULATE_GRAD_BATCHES] [--strategy
               STRATEGY] [--precision PRECISION]
           [--mode MODE] [--check-val-every-n-epoch CHECK_VAL_EVERY_N_EPOCH]
               [--ckpt-to-keep CKPT_TO_KEEP]
           [--checkpoint-frequency CHECKPOINT_FREQUENCY] [--state-dict-type
               STATE_DICT_TYPE]
           [--dataset-source DATASET_SOURCE] [--verbose | --no-verbose]
           [--train-dataset-length TRAIN_DATASET_LENGTH] [--test-dataset-length
               TEST_DATASET_LENGTH]
           [--num-sanity-val-steps NUM_SANITY_VAL_STEPS] [--sanity-train |
               --no-sanity-train]
           [--dataset-random-sample | --no-dataset-random-sample] [--beta BETA]
               [--roundoff ROUNDOFF]
           [--train-percentile TRAIN_PERCENTILE] [--test-percentile TEST_PERCENTILE]
           [--calculate-win-rate | --no-calculate-win-rate] [--lambda LAMBDA]

options:
-h, --help show this help message and exit
--use-ray, --no-use-ray
--model-name MODEL_NAME
                  Model name or path to the model
--dataset-name DATASET_NAME
                  Path to the dataset
--training-type {sft,dpo,bt,sft_verifier}
                  Type of training to perform: sft, dpo or bt (Bradley-Terry reward
                      training)
--storage-path STORAGE_PATH
                  Path where to store ray results and checkpoints
--name NAME Name of the experiment
--num-workers NUM_WORKERS
--accelerator-type ACCELERATOR_TYPE
                  Type of accelerator to use
--placement-strategy PLACEMENT_STRATEGY
                  Placement strategy for Ray
--seed SEED
--max-epochs MAX_EPOCHS
--use-gpu Use GPU for training
--batch-size BATCH_SIZE
                  Batch size
--lr LR learning rate
```

```
--resume-from-checkpoint RESUME_FROM_CHECKPOINT
                  Specify a checkpoint to resume from
--accumulate-grad-batches ACCUMULATE_GRAD_BATCHES
                  Accumulate gradient batches
--strategy STRATEGY Type of strategy
--precision PRECISION
                  Precision
--mode MODE train, test
--check-val-every-n-epoch CHECK_VAL_EVERY_N_EPOCH
--ckpt-to-keep CKPT_TO_KEEP
--checkpoint-frequency CHECKPOINT_FREQUENCY
--state-dict-type STATE_DICT_TYPE
--dataset-source DATASET_SOURCE
                  Source of the dataset to use
--verbose, --no-verbose
--train-dataset-length TRAIN_DATASET_LENGTH
                  Length of the dataset to use
--test-dataset-length TEST_DATASET_LENGTH
                  Length of the dataset to use
--num-sanity-val-steps NUM_SANITY_VAL_STEPS
                  Number of sanity val steps
--sanity-train, --no-sanity-train
                  Run a sanity check training
--dataset-random-sample, --no-dataset-random-sample
                  Randomly sample the dataset
--beta BETA Beta value for DPO training
--roundoff ROUNDOFF Roundoff value for the loss
--train-percentile TRAIN_PERCENTILE
                  Percentile of the dataset to use for training
--test-percentile TEST_PERCENTILE
                  Percentile of the dataset to use for testing
--calculate-win-rate, --no-calculate-win-rate
                  Calculate win rate against a reference model
--lambda LAMBDA Lambda value for SFT verifier training
```

B.3.2 Example runs

Some of the example jobs can be submitted as follows, for example, for a finetuning job which uses 2 H100 GPUs with DDP strategy and training method as sft-verifier, following command would be used:

```
ray job submit --no-wait --working-dir . --runtime-env-json '{"excludes": [".venv",
    ".git", "milestone-logs"], "env_vars": {"HF_TOKEN": "[masked]",
    "AWS_ACCESS_KEY_ID": "[masked]", "AWS_SECRET_ACCESS_KEY": "[masked]",
    "NVIDIA_API_KEY":"[masked]"}}' -- python train.py --use-ray --name
    "SFT-verifier-final" --num-workers 2 --strategy ddp --storage-path
    "s3://rl-sf" --precision transformer-engine --batch-size 1
    --num-sanity-val-steps 0 --checkpoint-frequency 1 --check-val-every-n-epoch 1
    --accumulate-grad-batches 1 --accelerator-type H100 --train-percentile 98
    --test-percentile 98 --training-type sft_verifier --model-name
    psr-ai/qwen2.5-0.5B-SFT-DPO --no-verbose --roundoff 4 --lambda 0.01
```

Another example, DPO with ddp on 4 T4 instances with bfloat-16 precision would be as follows

```
ray job submit --no-wait --working-dir . --runtime-env-json '{"excludes": [".venv",
    ".git", "milestone-logs"], "env_vars": {"HF_TOKEN": "[masked]",
    "AWS_ACCESS_KEY_ID": "[masked]", "AWS_SECRET_ACCESS_KEY": "[masked]",
    "NVIDIA_API_KEY":"[masked]"}}' -- python train.py --use-ray --name "DPO"
    --num-workers 4 --strategy ddp --storage-path "s3://rl-sf" --precision
    bf16-true --batch-size 1 --num-sanity-val-steps 0 --checkpoint-frequency 1
    --check-val-every-n-epoch 1 --accumulate-grad-batches 1 --accelerator-type
    H100 --train-percentile 98 --test-percentile 98 --training-type dpo
    --model-name psr-ai/qwen2.5-0.5B-SFT
```

B.3.3 Module

The `module.py` file contains the pytorch lightning module. It contains the steps for training, validation and creation of data loaders.

B.3.4 Loss computations

The `losses` directory consists of the implementations of loss computations for each of the training methods. Please find the corresponding filename for the loss method.

B.3.5 Metrics

The metrics are emitted to the `-storage-path` and `-name` specified in the submitted job. For example, a job named `JOB_A` and `-storage-path` named `s3://rl-sf`, the tensorboard can be viewed as:

```
export AWS_ACCESS_KEY_ID=AKIA2GK3E4AHHON3EF3X
export AWS_SECRET_ACCESS_KEY=3D60Nx4zgeJcwWTpdO+Jww/jpOlplfdxjzeYz3nH
tensorboard --logdir="s3://rl-sf/JOB_A"
```

If we also want the winrate metrics (for example while tranining over DPO), we also created a parameter `-calculate-win-rate` which calculates the winrate for the evaluation dataset against the original model.

When implementing the extension, we also used verifier score calculations as the output metric. This function can be found in `losses/sft.py` and the function is named `compute`$_{v}erifier_{s}cores$.

B.4 Uploading to Hub

We also created a script to upload the model from a checkpoint to huggingface repository. This script is upload_to_hub.py. An example command to run this would be:

```
ray job submit --no-wait --working-dir . --runtime-env-json '{"excludes":
    [".venv", ".git", "milestone-logs"], "env_vars": {"HF_TOKEN": "[masked]",
    "AWS_ACCESS_KEY_ID": "[masked]", "AWS_SECRET_ACCESS_KEY": "[masked]"}}' --
    python upload_to_hub.py --checkpoint-path "[path-to-checkpoint]" --repo-id
    "[huggingface-repo-id]" --training-type sft_verifier
```

Here the path to checkpoint can be found in the corresponding log file adn huggingface repo id is the path to the private repository on huggingface.