



Fine Tuning with TRL



Sergio Paniego Blanco (@sergiopaniego)
Machine Learning Engineer @ Hugging Face



From fine-tuning to TRL (fine-tuning+alignment)

“I fine-tuned the model and ...” 😄



- 💬 It can't chat naturally
- 🧠 It can't reason or explain
- 🌀 It gives incoherent answers
- 🔄 It gives inconsistent answers
- ⚠️ It's not aligned with human preferences (unhelpful or unsafe)
- 🚫 It doesn't follow instructions

...



From fine-tuning to TRL (fine-tuning+alignment)

Classic fine-tuning adapts the model to **your dataset**, not necessarily to **your intentions**.

- Dataset ($X \rightarrow Y$) $\rightarrow$ Pretrained model $\rightarrow$ Fine-tuned model
- Optimizes likelihood: predicts correctly based on the data
- **Doesn't guarantee alignment** with human preferences

We may need something stronger (**TRL**!), tools that can teach the model, not just fit it (**TRL**?!).

Data $\rightarrow$ Model $\rightarrow$ Fine-tuned model
💭 (Still says weird stuff)



Where does TRL fits in the model lifecycle? 🤔

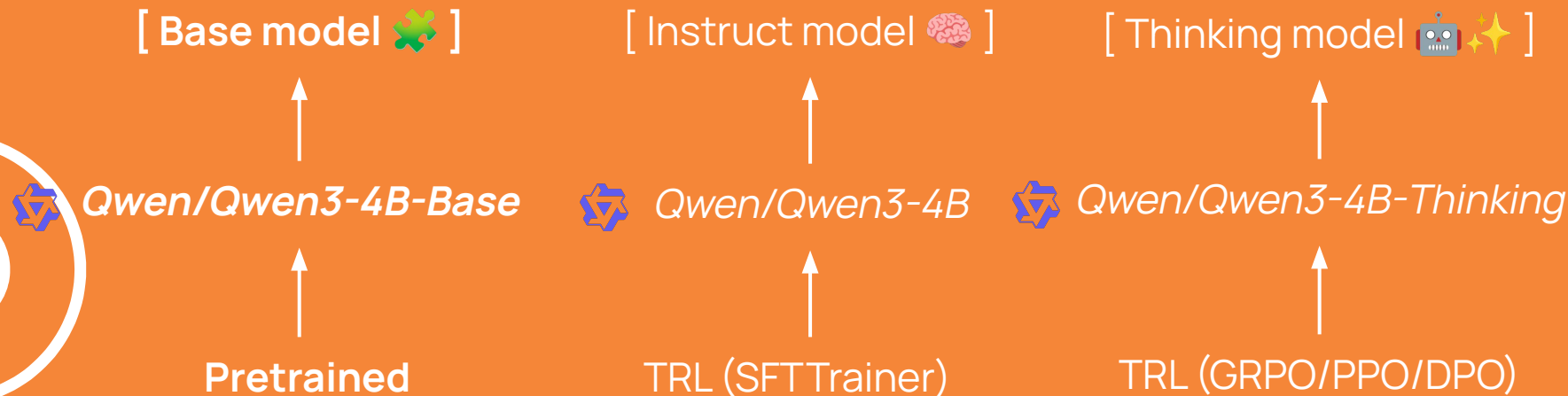
Pre-training: teaches general capacities such as language use, broad reasoning, and world knowledge

Mid-training: imparts domain knowledge (code, medical databases, internal docs...)

Post-training/Alignment (TRL!): learns to behave as we want (instruction following, reasoning...)

◇
~
◇ TRL is **not for pretraining** → it's for the final stages, where we make models useful, safe, and aligned

From token prediction to alignment



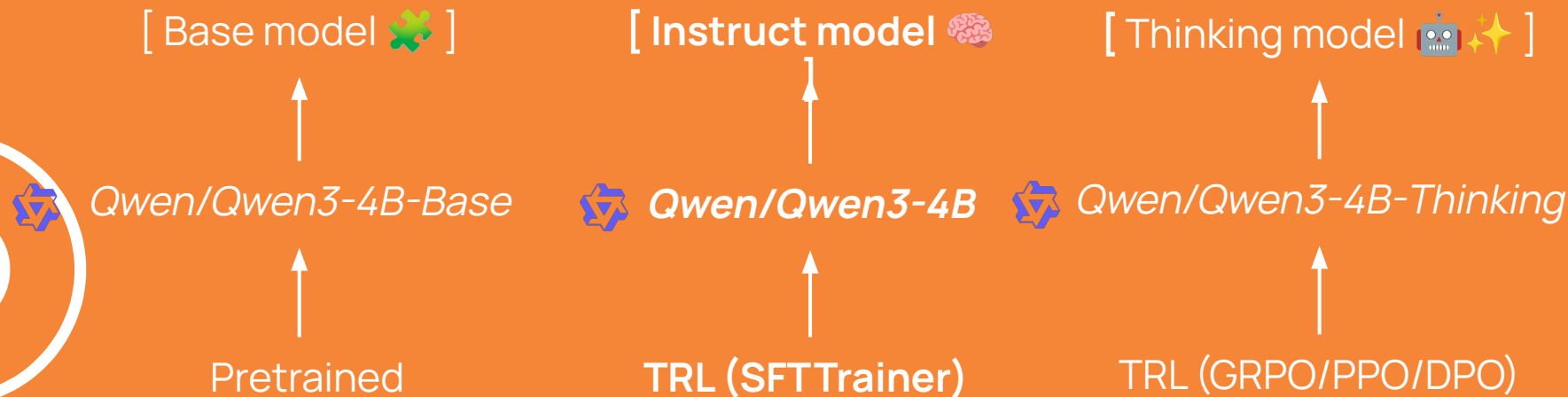
Base: predicts next token

Raw text, high quantity of data, masked language modelling, industrial compute

“Translate this sentence → Translate this sentence into...”



From token prediction to alignment



Instruct: follows instructions

Structured responses, high quality data, reduced compute

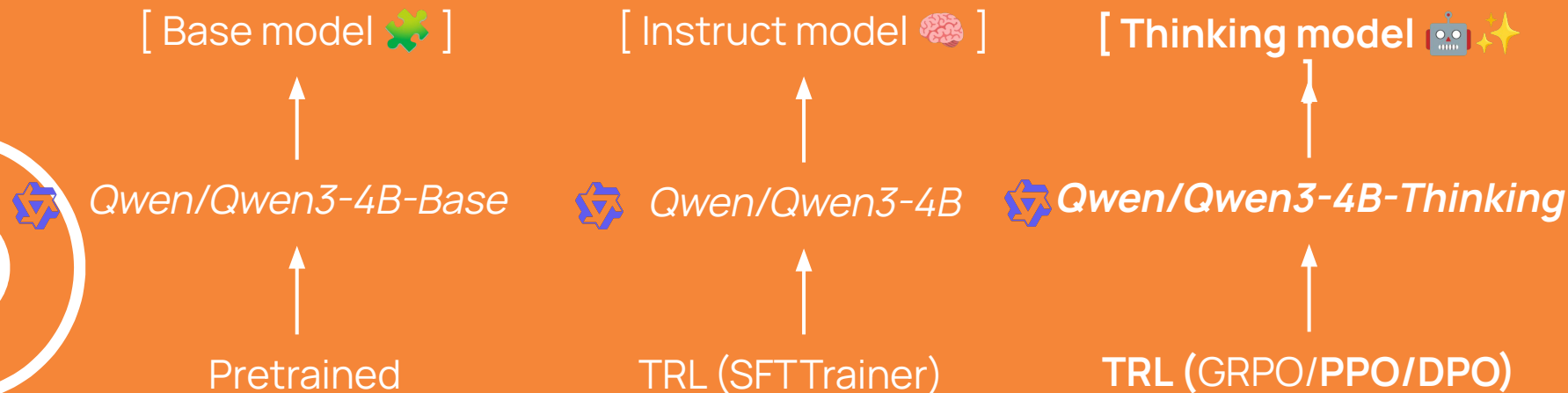
"<luser!> *Translate this sentence* </s>

<lassistant!> *Sure! Here's the translation* </s>"

Example dataset



From token prediction to alignment



Aligning model: helpfulness, truthfulness, and safety.

Preferences, high quality, preference optim, consumer compute

"< luser! > How many helicopters can a human eat? < /s >

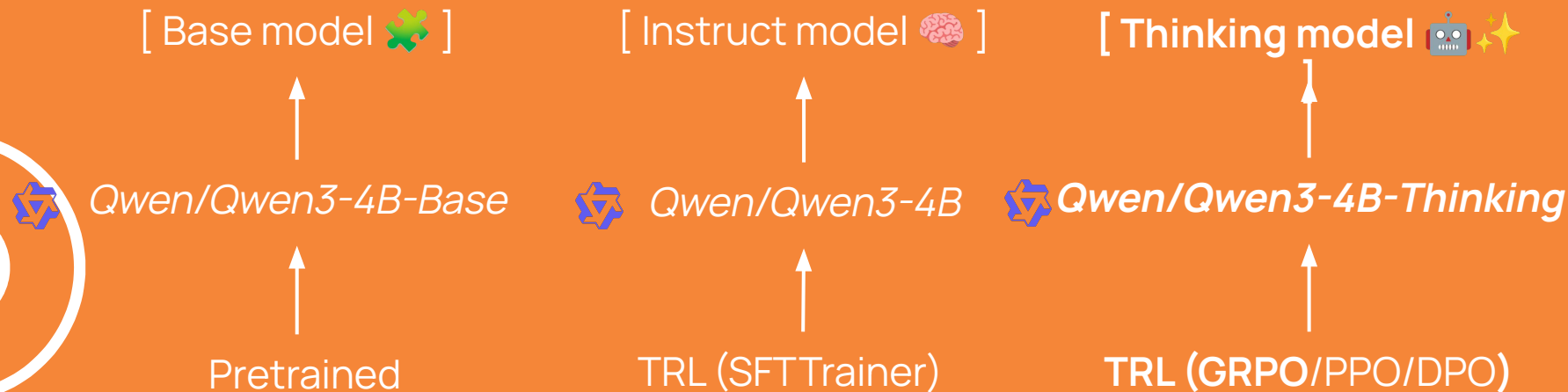
👎 < lassistant! > Humans cannot eat helicopters < /s >

👍 < lassistant! > That's not possible! < /s > "

Example dataset



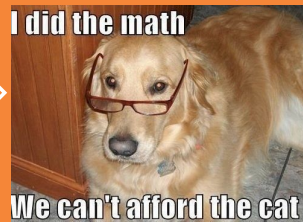
From token prediction to alignment



Thinking model: reasoning!

◇ “<user> How many helicopters can a human eat?</s>
<assistant>
<think> “Tricky question. Let’s break it down...</think>
That’s not possible! I found that...</s>”

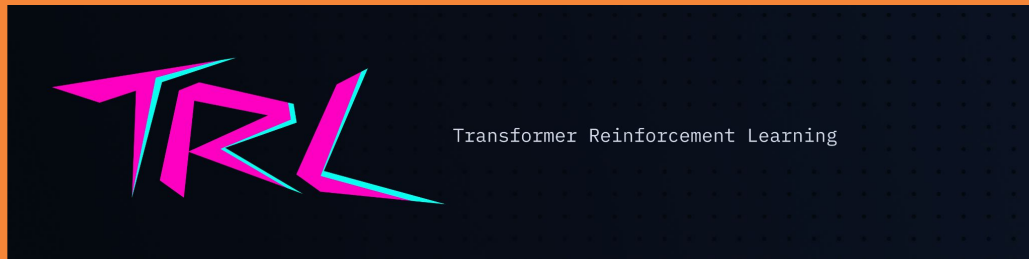
• Example dataset



TRL: Fine-tuning + alignment!

TRL: HF library that offers specialized trainers (SFT, GRPO, PPO, DPO...)

- TRL bridges the gap between **instruction following** and **full alignment pipelines**.
- Compatible with **accelerate**, **PEFT**, **kernels**, **trackio**, **vLLM**...
- **Multimodal** models support...
- **Post-training: SFT, alignment (DPO), reasoning (GRPO)**



Trainer vs TRL

TRL trainers are **built on top of transformers Trainer**, enhancing its functionality.

Each trainer **abstracts a full training method**, so you don't need to reimplement everything from scratch.

Gives ML engineers **ready-to-use recipes** for SFT, GRPO, PPO, DPO...

Focus: practicality, reproducibility, and reduced boilerplate

```
from trl import SFTTrainer
from datasets import load_dataset

trainer = SFTTrainer(
    model="Qwen/Qwen3-0.6B",
    train_dataset=load_dataset("trl-lib/Capybara", split="train"),
)
trainer.train()
```

SFT and GRPO

We'll focus here on two trainers. But there are more 🙄

Supervised Fine-Tuning (SFT)

- Teach the model to follow instructions
Uses human-labeled data (prompt → response)
- Improves helpfulness

Group Relative Policy Optimization (GRPO)

- ◊ • Train with online reinforcement learning
- Uses reward models or human preference signals
- Improves alignment and reasoning



TRL taxonomy

Online methods

(learn from feedback) Model generates responses and learns from reward signals during training

- **GRPO** ⚡
- **RLOO** ⚡
- **OnlineDPO** ⚡
- **NashMD** ⚡
- **XPO** ⚡
- **PPO**

Reward modeling

Training a model to judge responses

- **PRM**
- **Reward**

⚡ = **LLM** support

Offline methods

(data-driven) Model learns from human preference datasets or supervised signals.

- **SFT**
- **DPO**
- **ORPO**
- **BCO**
- **CPO**
- **KTO**

Knowledge distillation

Transferring knowledge from a stronger model to a smaller one

- **GKD**

Choose your fighter!



SFT and GRPO are all you need?

SFT and GRPO dominate the scene (apparently)

TRL includes +10 trainers → research moves fast!

Hard to maintain? There's even an open issue in the repo

New training methods appear!

[RFC] Moving Most TRL Trainers to the experimental Submodule to Streamline the Core #4223

Open

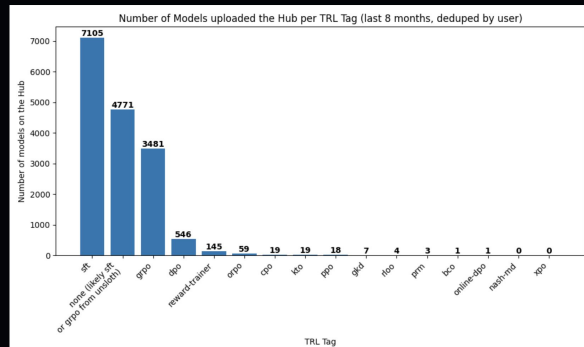


ggalouedec opened 3 weeks ago · edited by ggalouedec

Edits Member

Context

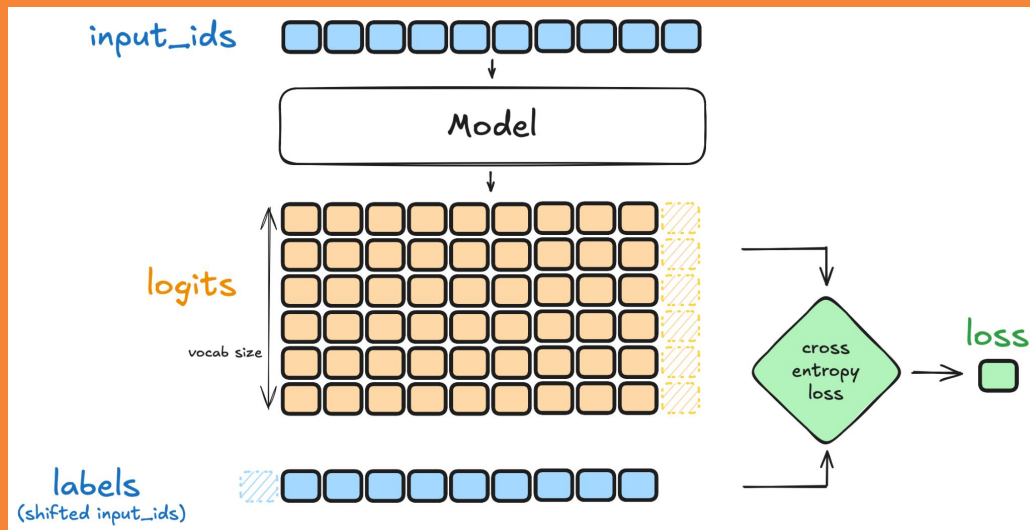
TRL currently includes 15 trainers, which vary significantly in usage and maintenance requirements.



Supervised Fine-Tuning

SFT is the simplest and most commonly used method to adapt a LM to a target dataset

- **Goal:** minimize negative log-likelihood of a sequence
- Simple, stable, and super effective.



Supervised Fine-Tuning

SFTTrainer supports both **standard** and **conversational** dataset formats.

```
# Standard
language_modeling_example = {"text": "The sky is blue."}

# Conversational
messages = [
    {"role": "user", "content": "Hello, how are you?"},
    {"role": "assistant", "content": "I'm doing great. How can I help you today?"},
    {"role": "user", "content": "I'd like to show off how chat templating works!"},
]
```

SFT is easy to instantiate and train

Only needs a model (even just the name!) and a dataset from the Hub to train

```
from trl import SFTTrainer
from datasets import load_dataset

trainer = SFTTrainer(
    model="Qwen/Qwen3-0.6B",
    train_dataset=load_dataset("trl-lib/Capybara", split="train"),
)
trainer.train()
```



SFTTrainer and SFTConfig

```
from trl import SFTTrainer, SFTConfig
from datasets import load_dataset

trainer = SFTTrainer(
    model="Qwen/Qwen3-0.6B",
    train_dataset=load_dataset("trl-lib/Capybara", split="train"),
    args = SFTConfig(
        # Training schedule / optimization
        per_device_train_batch_size = 2,      # Batch size per GPU
        gradient_accumulation_steps = 8,      # Gradients are accumulated over multiple steps → effective batch size = 2 * 8 = 16
        num_train_epochs = 1,                # Number of full dataset passes
        learning_rate = 2e-4,                # Learning rate for the optimizer
        optim = "paged_adamw_8bit",          # Optimizer

        # Logging / reporting
        logging_steps=1,                     # Log training metrics every N steps
        report_to="trackio",                 # Experiment tracking tool

        # Hub integration
        push_to_hub=True,                    # Automatically push the trained model to the Hugging Face Hub
        ...
    )
    peft_config=LoraConfig(),
    # ...
)
trainer.train()
```



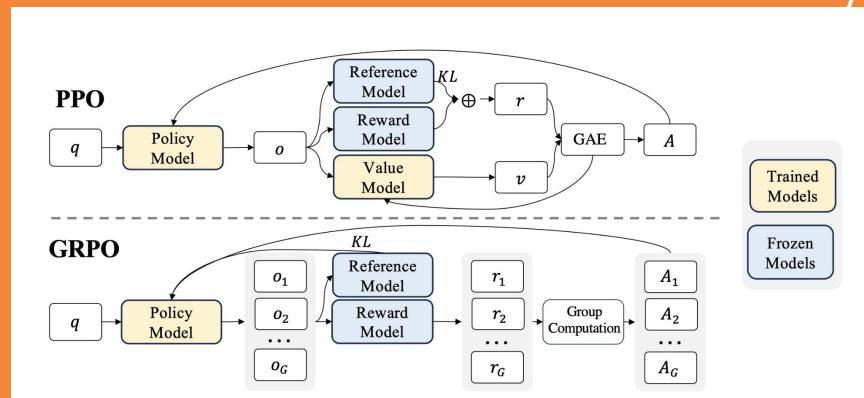
Group Relative Policy

Optimization

GRPO was described in the paper.

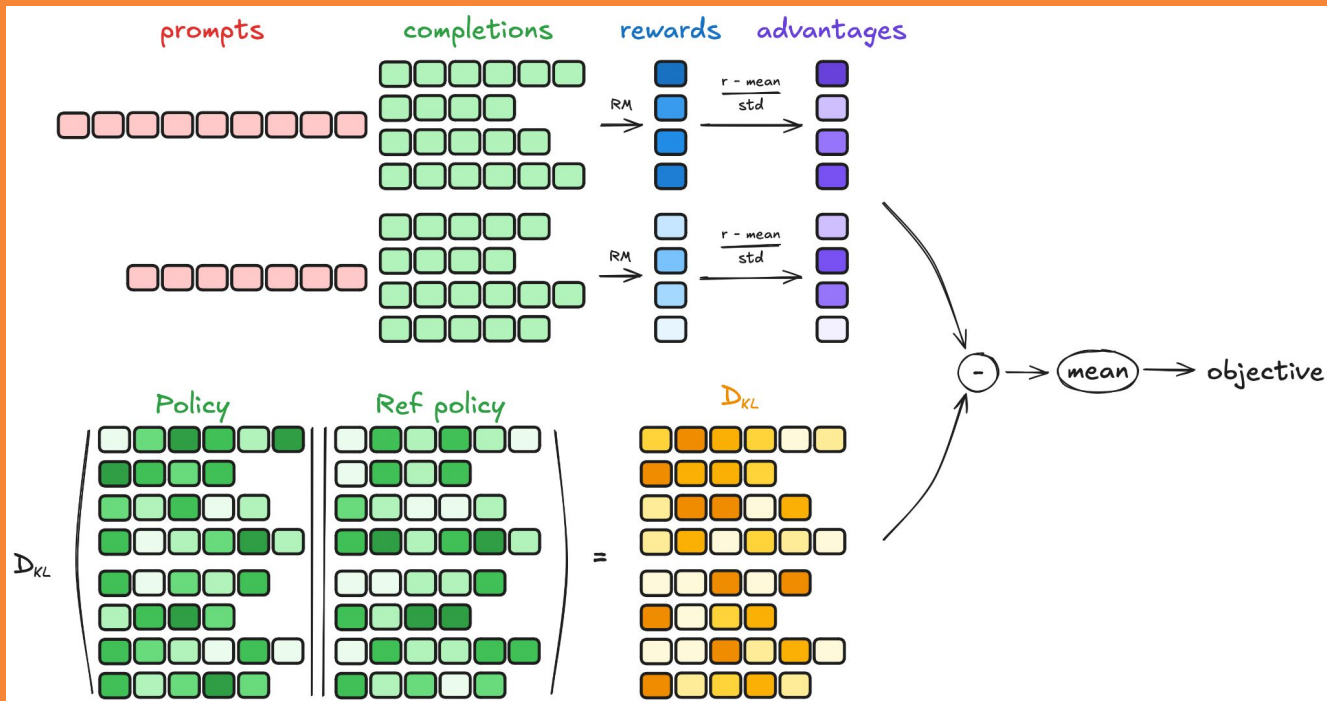
Variant of Proximal Policy Optimization (PPO), that enhances maths reasoning abilities while optimizing memory usage

- **Online** method ⚡
- Optimizes behavior using **rewards** from **preference models** or heuristics (**reward functions**)
- Successor of PPO: more stable and memory-efficient



GRPO

Steps: generating completions, computing advantage, ~~estimating the KL divergence~~, computing the loss



GRPOTrainer and GRPOConfig

```
from trl import GRPOTrainer, GRPOConfig
import re

def format_reward_func(completions, **kwargs):
    """Reward function that checks if the completion has a specific format."""
    pattern = r"^<think>.*?</think><answer>.*?</answer>$"
    completion_contents = [completion[0]["content"] for completion in completions]
    matches = [re.match(pattern, content) for content in completion_contents]
    return [1.0 if match else 0.0 for match in matches]

trainer = GRPOTrainer(
    model="Qwen/Qwen2-0.5B-Instruct",
    reward_funcs=[format_reward_func, ... ],
    args=GRPOConfig( ... ),
    train_dataset=load_dataset("trl-lib/tldr", split="train"),
    # ...
)
trainer.train()
```

GRPOConfig detail

```
from trl import GRPOConfig

# Configure training arguments using GRPOConfig
training_args = GRPOConfig(
    # ... Similar to SFTConfig

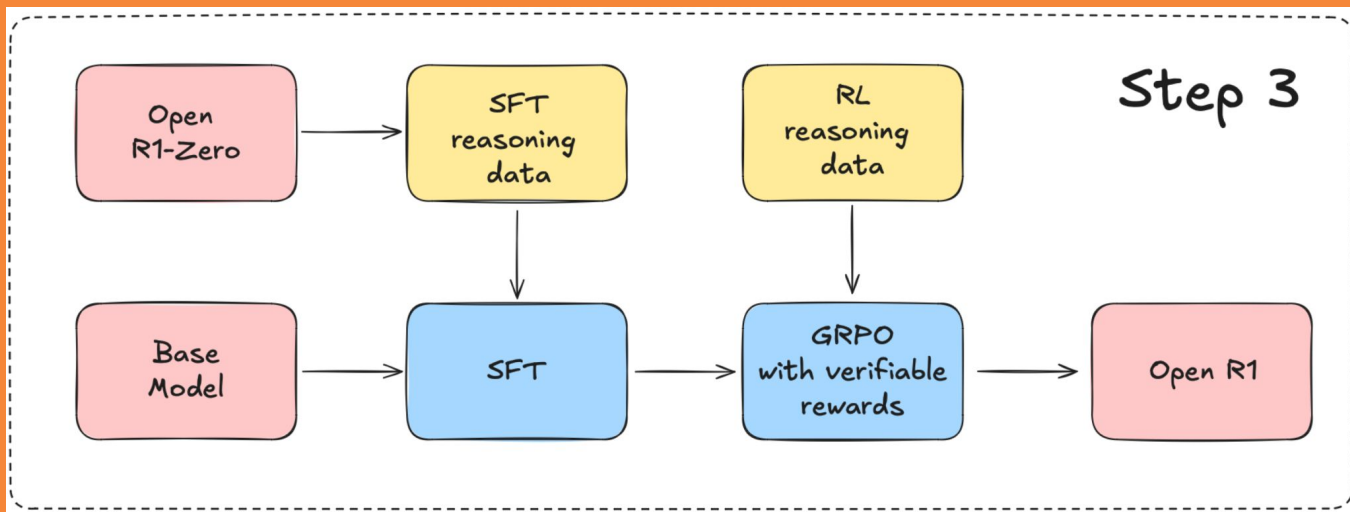
    # Parameters that control de data preprocessing
    max_completion_length=512, # default: 256
    num_generations=8, # default: 8
    max_prompt_length=512, # default: 512
)
```

OpenR1



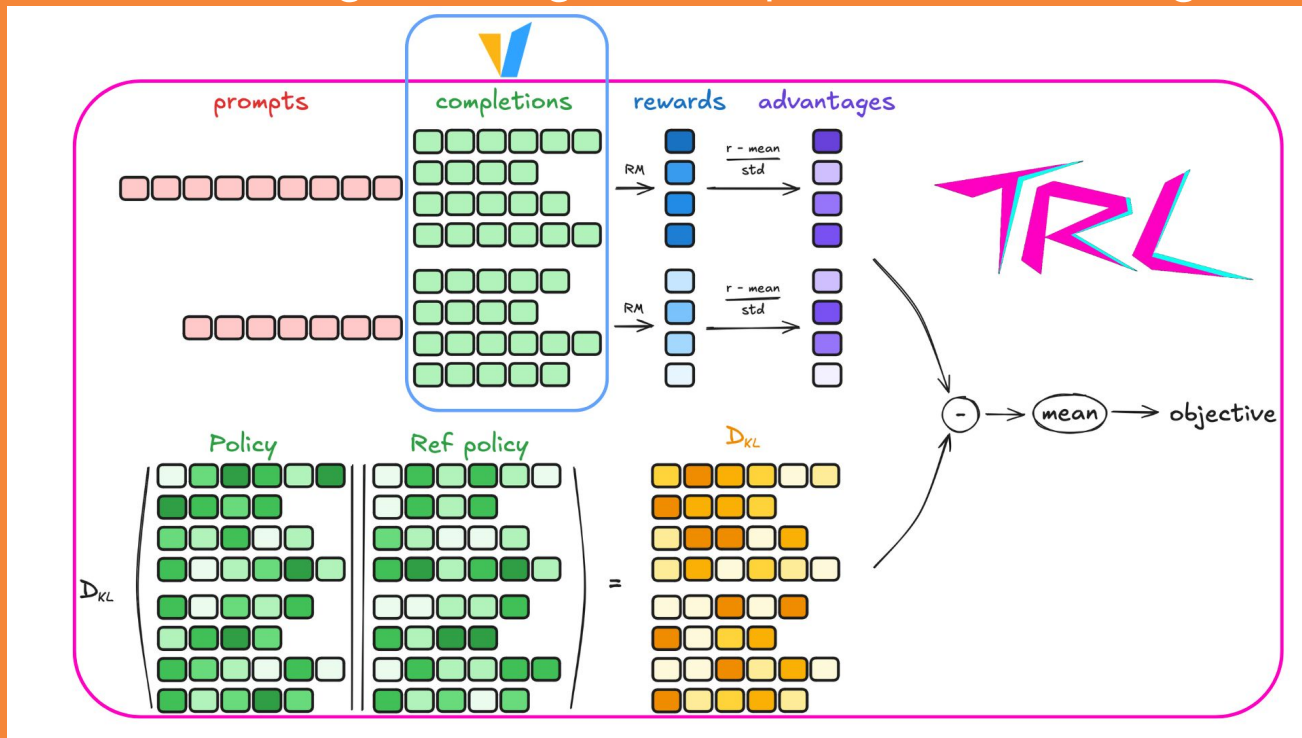
Initiative by HF to replicate and extend the techniques behind DeepSeek-R1 🐳

This model was built using SFT + GRPO!



Can we improve online training?

Some TRL trainers work online ⚡, the model generates completions during training to compute a reward signal.



Can we improve online training?

⚠️ **Bottleneck!** Generation is slow and inefficient
⚡ **vLLM** solves this problem and TRL supports it!
You can call it from the **CLI** (yes, TRL has a CLI)



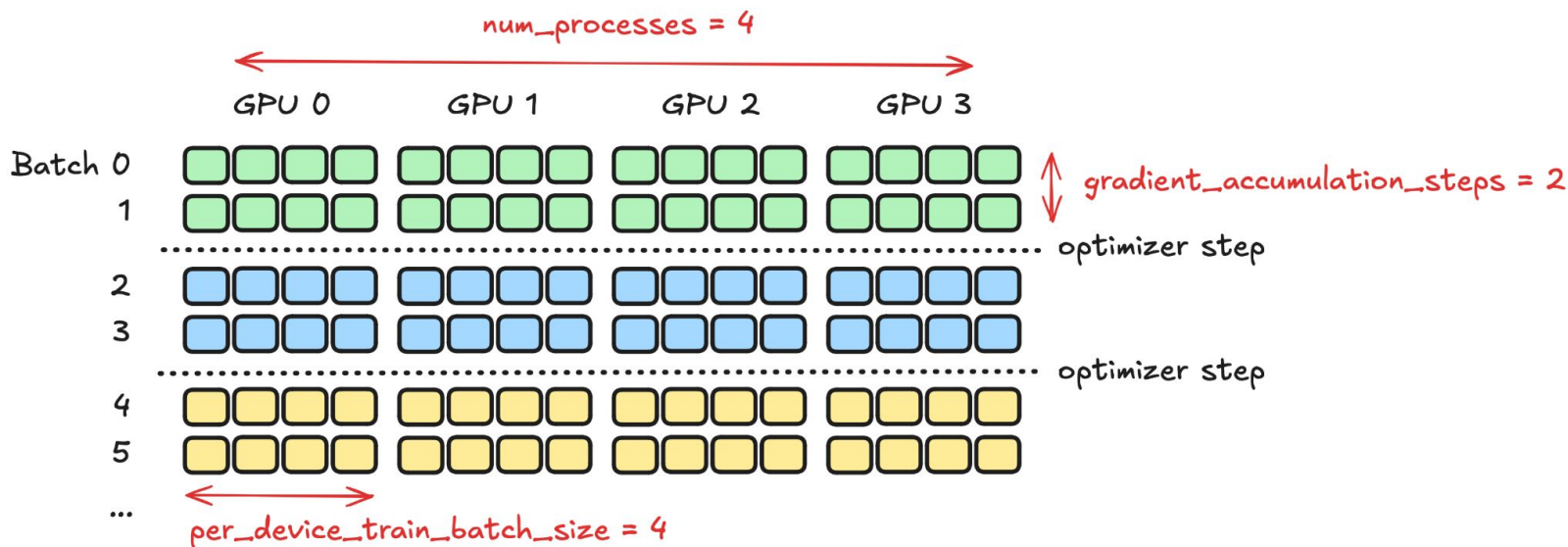
```
CUDA_VISIBLE_DEVICES=0,1,2,3 trl vllm-serve --model Qwen/Qwen2.5-7B --tensor-parallel-size 2 --data-parallel-size 2
```

```
from trl import GRPOConfig

training_args = GRPOConfig(
    ...,
    use_vllm=True,
    vllm_mode="server", # default value, can be omitted. Can also be "colocate"
)
```


But can we scale training?

TRL also integrates seamlessly with accelerate and DeepSpeed (model sharing, zero redundancy optimized, mixed precision training, offloading...)



But can we scale training?

In the config we specify num of machines, num of processes...

```
accelerate launch --config_file examples/accelerate_configs/deepspeed_zero2.yaml train.py
```



But can we scale training?

Train long contexts via **context parallelism** (*accelerate*): by splitting the sequence across multiple GPUs

Even **ND-parallelism** (N model replicas, N devices, tensor parallelism, context parallelism)

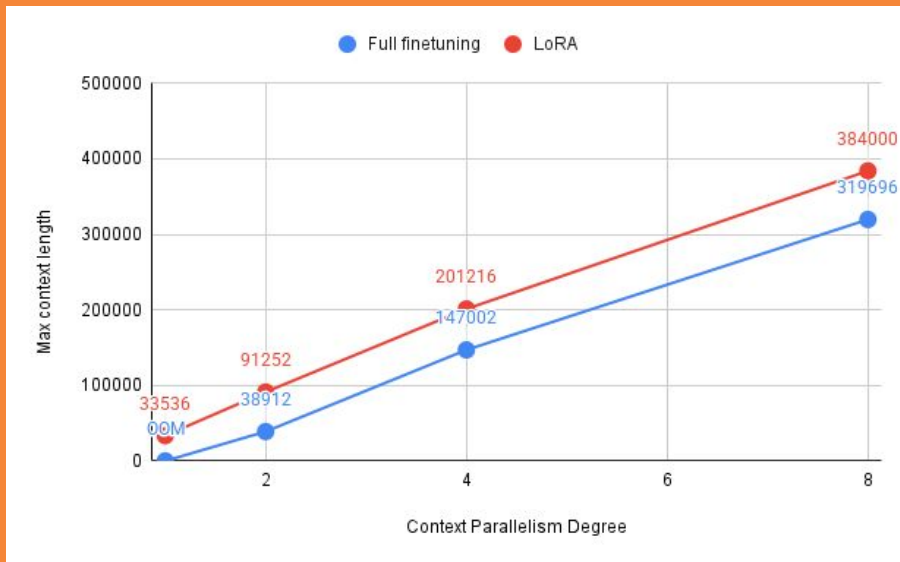
```
accelerate launch --config_file context_parallel_2gpu.yaml train.py
```

```
from trl import SFTConfig

training_args = SFTConfig(
    # required
    pad_to_multiple_of=4,          # ensures divisibility by cp_size * 2
    # to get the most out of CP
    max_length=400000,            # long sequence length
    packing=True,                 # use packing to reduce padding
    use_liger_kernel=True,        # compatible with CP
    gradient_checkpointing=False,  # The activation_checkpointing in FSDP config and
    # the gradient_checkpointing in training arg can't be set to True simultaneously
    per_device_train_batch_size=1,
    ...
)
```

But can we scale training?

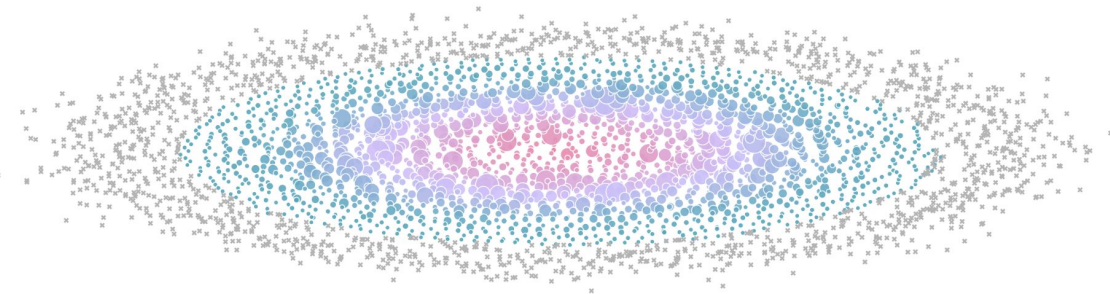
Context Parallelism(CP) with *Qwen/Qwen3-8B* scales to over 300k tokens in 8 GPUs 🙌



But can we scale training?

nanotron / ultrascale-playbook

The Ultra-Scale Playbook: Training LLMs on GPU Clusters



We ran over 4,000 scaling experiments on up to 512 GPUs and measured throughput (size of markers) and GPU utilization (color of markers). Note that both are normalized per model size in this visualization.

ORDER BOOK

GET PDF

AUTHORS

Nouamane Tazi, Ferdinand Mom, Haojun Zhao, Phuc Nguyen,
Mohamed Mekkouri, Leandro Werra, Thomas Wolf

AFFILIATION

Hugging Face

PUBLISHED

Feb 19, 2025

The Ultra-Scale
Playbook: Training LLMs
on GPU Clusters



Optimizing all the things!

Transformers already provides us with some optimizing strategies like gradient checkpointing and we also have LoRA/QLoRA. In addition, TRL has:

- Truncation
- Packing
- Padding-free
- Padding sequences to a multiple
- Liger kernels  LIGER KERNEL
- ◊ • Activation offloading

We developed a notebook for training a **14B** model using QLoRA in [free Colab](#)!



Optimizing is super easy

```
from trl import SFTConfig

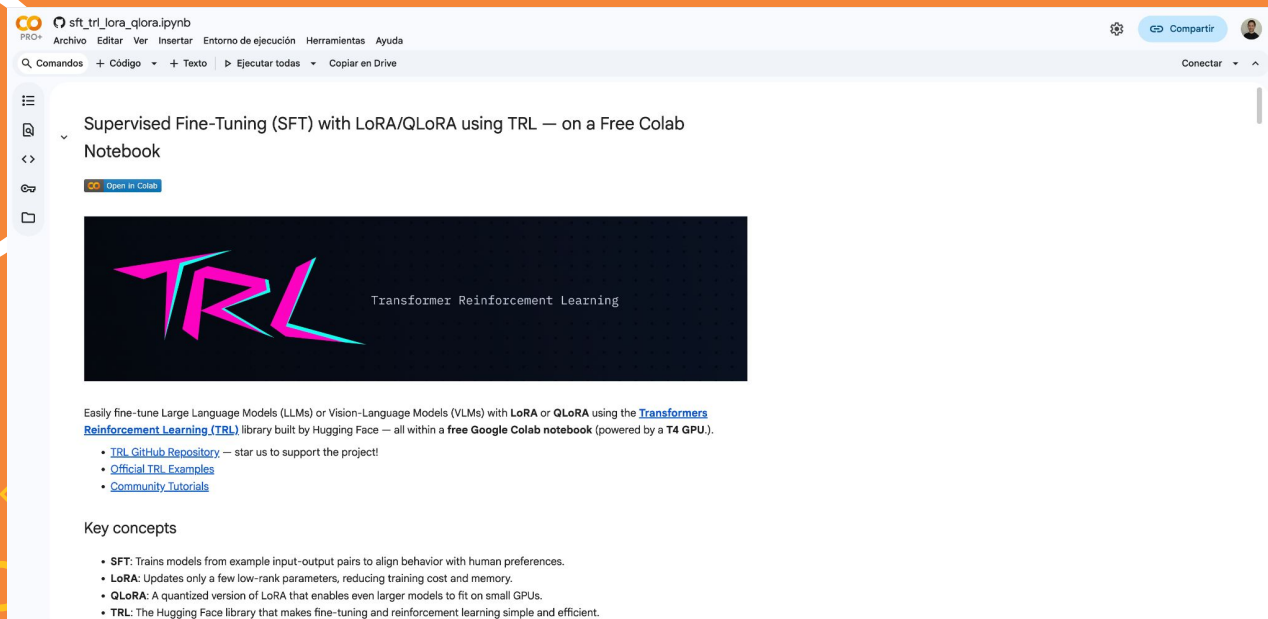
training_args = SFTConfig(
    max_length=1024,                # (Truncation) Maximum input sequence length
    packing=True                   # Packing
    padding_free=True, model_init_kwargs={"attn_implementation": "flash_attention_2"}, # Padding-free
    pad_to_multiple_of=2048        # Padding sequences to a multiple
    use_liger_kernel=True,         # Enable Liger kernel optimizations for faster training
    activation_offloading=True,    # Offload activations to CPU to reduce GPU memory usage

    gradient_checkpointing=True,   # Save memory by re-computing activations during backpropagation
)
```



Optimizing for GPU poors

We developed a notebook for training a **14B** model using QLoRA in free Colab!



The screenshot shows a Google Colab notebook interface. The title bar indicates the notebook is named 'sft_trl_lora_qlora.ipynb'. The main content area features a large graphic with the text 'TRL Transformer Reinforcement Learning'. Below the graphic, there is a paragraph explaining that the notebook is for easily fine-tuning Large Language Models (LLMs) or Vision-Language Models (VLMs) with LoRA or QLoRA using the Transformers Reinforcement Learning (TRL) library. It mentions that the library is built by Hugging Face and is available within a free Google Colab notebook (powered by a T4 GPU). There are three links provided: 'TRL GitHub Repository', 'Official TRL Examples', and 'Community Tutorials'. At the bottom, under the heading 'Key concepts', there are four bullet points: 'SFT: Trains models from example input-output pairs to align behavior with human preferences.', 'LoRA: Updates only a few low-rank parameters, reducing training cost and memory.', 'QLoRA: A quantized version of LoRA that enables even larger models to fit on small GPUs.', and 'TRL: The Hugging Face library that makes fine-tuning and reinforcement learning simple and efficient.'



HF jobs + TRL

Models can be trained using HF's infra via HF Jobs

- **trl-jobs** cli package
- **hf jobs** (cli or python)

```
# pip install trl-jobs
trl-jobs sft --model_name Qwen/Qwen3-0.6B --dataset_name trl-lib/Capybara
```

```
hf jobs uv run \
  --flavor a100-large \
  --with trl \
  --secrets HF_TOKEN \
  train.py
```

```
from huggingface_hub import run_uv_job

run_uv_job(
    "train.py",
    dependencies=["trl"],
    flavor="a100-large",
    secrets={"HF_TOKEN": "hf_..."},
)
```



Multimodality!



Vision Language Models (VLMs) are getting stronger, but **aligning** them to human preferences still matters.

TRL supports VLMs in many trainers, including examples and recipes.

- SFT
- DPO (Direct Preference Optimization)
- MPO (Mixed Preference Optimization)
- ◊ GRPO (Group Relative Policy Optimization)
- GSPO (Group Sequence Policy Optimization)
- RLOO (Reinforce Leave One Out)
- Online DPO



Multimodality!!



Native SFT support for VLMs
Just can still use your own collator

```
from trl import SFTConfig, SFTTrainer
from datasets import load_dataset

trainer = SFTTrainer(
    model="Qwen/Qwen2.5-VL-3B-Instruct",
    args=SFTConfig(max_length=None), # To avoid truncation that may remove image tokens during training
    train_dataset=load_dataset("trl-lib/llava-instruct-mix", split="train"),
)
trainer.train()
```

Example dataset



Multimodality!!!



Shared scripts and recipes!

examples/scripts/dpo_vlm.py	This script shows how to use the DPOTrainer to fine-tune a Vision Language Model to reduce hallucinations using the openbmb/RLAIF-V-dataset dataset.
examples/scripts/evals/judge_tldr.py	This script shows how to use HPairwiseJudge or OpenPairwiseJudge to judge model generations.
examples/scripts/gkd.py	This script shows how to use the GKDTTrainer to fine-tune a model.
trl/scripts/grpo.py	This script shows how to use the GRPOTrainer to fine-tune a model.
examples/scripts/grpo_vlm.py	This script shows how to use the GRPOTrainer to fine-tune a multimodal model for reasoning using the lmms-lab/multimodal-open-r1.8k-verified dataset.
examples/scripts/gspo.py	This script shows how to use GSPo via the GRPOTrainer to fine-tune model for reasoning using the AI-MO/NuminaMath-T1R dataset.
examples/scripts/gspo_vlm.py	This script shows how to use GSPo via the GRPOTrainer to fine-tune a multimodal model for reasoning using the lmms-lab/multimodal-open-r1.8k-verified dataset.
examples/scripts/kto.py	This script shows how to use the KTOTrainer to fine-tune a model.
examples/scripts/mpo_vlm.py	This script shows how to use MPO via the DPOTrainer to align a model based on preferences using the HuggingFaceH4/rlaif-v formatted dataset and a set of loss weights with weights.
examples/scripts/nash_md.py	This script shows how to use the NashMDTrainer to fine-tune a model.
examples/scripts/online_dpo.py	This script shows how to use the OnlineDPOTrainer to fine-tune a model.
examples/scripts/online_dpo_vlm.py	This script shows how to use the OnlineDPOTrainer to fine-tune a Vision Language Model.

examples/scripts/sft_vlm.py	This script shows how to use the SFTTrainer to fine-tune a Vision Language Model in a chat setting. The script has only been tested with LLaVA 1.5 , LLaVA 1.6 , and Llama-3.2-11B-Vision-Instruct models so users may see unexpected behaviour in other model architectures.
examples/scripts/sft_vlm_gemma3.py	This script shows how to use the SFTTrainer to fine-tune a Gemma 3 model on vision to text tasks.
examples/scripts/sft_vlm_smol_vlm.py	This script shows how to use the SFTTrainer to fine-tune a SmoIVLM model.

Post training a VLM for reasoning with GRPO using TRL

Authored by: [Sereno Pavesio](#)

WARNING: This notebook is resource-intensive and requires substantial computational power. If you're running this in Colab, it will utilize an A30 GPU.

In this recipe, we'll demonstrate how to post-train a [Vision Language Model \(VLM\)](#) using [GRPO](#) for adding reasoning capabilities to a VLM using the Hugging Face ecosystem, specifically with the [Transformer Reinforcement Learning library \(TRL\)](#).

We'll fine-tune [Qwen2.5-VL-72B-Instruct](#) using a subset of the [lmms-lab/multimodal-open-r1.8k-verified](#) dataset. This dataset includes images with problem descriptions along with their solution and thinking traces to reach that solution. We'll leverage this data format, along with the GRPO reward functions, to teach the model how to reason to reach the solution.

```
graph LR
    UserQuery[User query] --> VLMModel[VLM model]
    VLMModel --> ReasoningTrace[Reasoning trace]
    VLMModel --> VLMAnswer[VLM answer]
    ReasoningTrace --> RewardModel1[Reward model]
    VLMAnswer --> RewardModel2[Reward model]
    RewardModel1 --> Reward1[Reward]
    RewardModel2 --> Reward2[Reward]
    Reward1 --> VLMModel
    Reward2 --> VLMModel
```

Fine-Tuning a Vision Language Model (Qwen2-VL-7B) with the Hugging Face Ecosystem (TRL)

Authored by: [Sereno Pavesio](#)

WARNING: This notebook is resource-intensive and requires substantial computational power. If you're running this in Colab, it will utilize an A30 GPU.

In this recipe, we'll demonstrate how to fine-tune a [Vision Language Model \(VLM\)](#) using the Hugging Face ecosystem, specifically with the [Transformer Reinforcement Learning library \(TRL\)](#).

Model & Dataset Overview

We'll fine-tune the [Qwen2-VL-7B](#) model on the [ChatVQA](#) dataset. This dataset includes images of various chart types paired with question-answer pairs—ideal for enhancing the model's visual question-answering capabilities.

Additional Resources

If you're interested in more VLM applications, check out:

- [Multimodal Retrieval-Augmented Generation \(RAG\) Recipe](#): where I guide you through building a RAG system using Document Retrieval (CoPa) and Vision Language Models (VLMs).
- [Phi-Scribe's tutorial](#): an excellent deep dive into fine-tuning multimodal LLMs with TRL.
- [Merco Novati's smol-vision repository](#): a collection of engaging notebooks on cutting-edge vision and multimodal AI topics.

```
graph LR
    UserQuery[User query] --> VLMModel[VLM model]
    VLMModel --> VLMAnswer[VLM answer]
    VLMAnswer --> RewardModel[Reward model]
    RewardModel --> Reward[Reward]
    Reward --> VLMModel
```

Fine-Tuning a Vision Language Model with TRL using MPO

Authored by: [Sereno Pavesio](#)

In this recipe, we'll demonstrate how to fine-tune a [Vision Language Model \(VLM\)](#) using [Mixed Preference Optimization \(MPO\)](#) with the [Transformer Reinforcement Learning \(TRL\)](#) library.

MPO is a training approach that combines multiple optimization objectives and was introduced in the paper [Enhancing the Reasoning Ability of Multimodal Large Language Models via Mixed Preference Optimization](#). It is part of the [Direct Preference Optimization \(DPO\)](#) trainer and works by combining multiple loss functions with different weights, enabling more sophisticated optimization strategies.

We'll fine-tune [Qwen2.5-VL-72B-Instruct](#), a small VLM with strong performances, using a preference dataset to help the model align with desired outputs. Check out [this blog post](#) to learn more about preference optimization for vision-language models.

The dataset we'll use is [HuggingFaceH4/rlaif-v](#), a specially formatted version of the [RLAIF-V](#) dataset. This dataset contains pairs of `prompt + image` along with a `chosen` and `rejected` response for each sample. The final goal of the fine-tuning process is to train a model that consistently prefers the `chosen` answers over the `rejected` ones, thereby reducing hallucinations. To achieve this, multiple loss functions will be used in combination.

```
graph LR
    UserQuery[User query] --> VLMModel[VLM model]
    VLMModel --> ChosenAnswer[Chosen VLM answer]
    VLMModel --> RejectedAnswer[Rejected VLM answer]
    ChosenAnswer --> RewardModel1[Reward model]
    RejectedAnswer --> RewardModel2[Reward model]
    RewardModel1 --> Reward1[Reward]
    RewardModel2 --> Reward2[Reward]
    Reward1 --> VLMModel
    Reward2 --> VLMModel
```

Fine-Tuning SmoIVLM using direct preference optimization (DPO) with TRL on a consumer GPU

Authored by: [Sereno Pavesio](#)

In this recipe, we'll guide you through fine-tuning a [smol-vl](#) [Vision Language Model \(VLM\)](#) with [Direct Preference Optimization \(DPO\)](#) using the [Transformer Reinforcement Learning \(TRL\)](#) library to demonstrate how you can tailor VLMs to suit your specific needs, even when working with consumer-grade GPUs.

We'll fine-tune [SmolVLM](#) using a [preference dataset](#) to help the model align with desired outputs. SmoVLM is a highly performance and memory-efficient model, making it an ideal choice for this task. If you're new to [Preference Optimization](#) for language or vision-language models, check out [this blog](#) for an in-depth introduction.

The dataset we'll use is [HuggingFaceH4/rlaif-v](#), which contains pairs of `prompt + image` along with a `chosen` and `rejected` answer for each pair. The goal of this fine-tuning process is to make the model consistently prefer the `chosen` answers from the dataset, reducing hallucinations.

This notebook has been tested using an [NVIDIA L4 GPU](#).

```
graph LR
    UserQuery[User query] --> SmoVLM[smol VLM]
    SmoVLM --> ChosenAnswer[Chosen LLM Answer]
    SmoVLM --> RejectedAnswer[Rejected LLM Answer]
    ChosenAnswer --> RewardModel1[Reward model]
    RejectedAnswer --> RewardModel2[Reward model]
    RewardModel1 --> Reward1[Reward]
    RewardModel2 --> Reward2[Reward]
    Reward1 --> SmoVLM
    Reward2 --> SmoVLM
```

Agents 🕵️ via OpenEnv

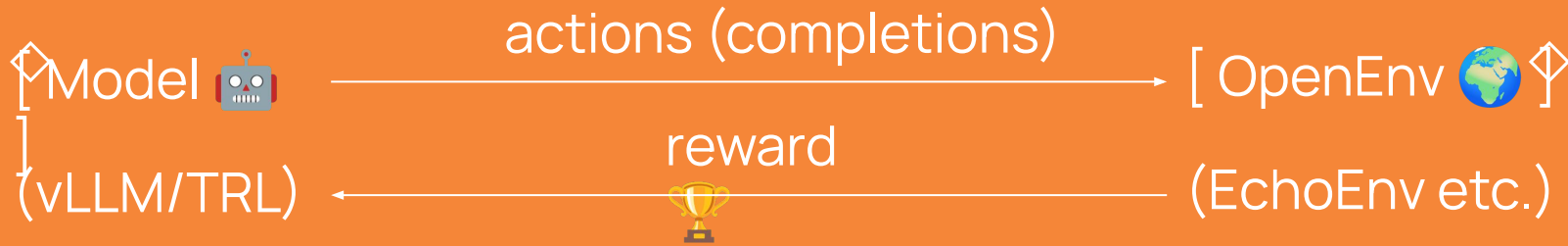
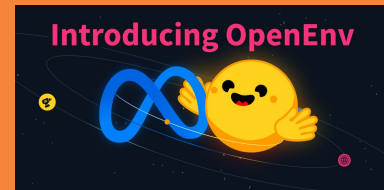
🧩 **OpenEnv** → framework for interactive TRL environments
(*Gymnasium* for LLMs)

💻 Runs envs locally or as backend servers

🌐 Find ready-to-use OpenEnv envs on the Hub

🔗 Integrates with TRL → use `'rollout_func'` in GRPO to replace text generation with environment interaction

🎯 Models can act, observe, and learn, not just predict text.



(rollout_func): bridges generation & env steps

Agents via OpenEnv

```
from envs.echo_env import EchoEnv, EchoAction
from trl import GRPOConfig, GRPOTrainer

# Create HTTP client for Echo Environment
client = EchoEnv.from_docker_image("echo-env:latest")

def rollout_func(prompts, args, processing_class):
    # 1. Generate completions via VLLM inference server (running on port 8000)
    ...
    response = requests.post("http://0.0.0.0:8000/generate/", json=payload)
    ...

    # 2. Step through the environment to get rewards
    client.reset()
    env_rewards = []
    for msg in completions_text:
        env_result = client.step(EchoAction(message=msg))
        env_rewards.append(env_result.reward)

    # 3. Add environment rewards as extra field
    result["env_reward"] = env_rewards
    return result

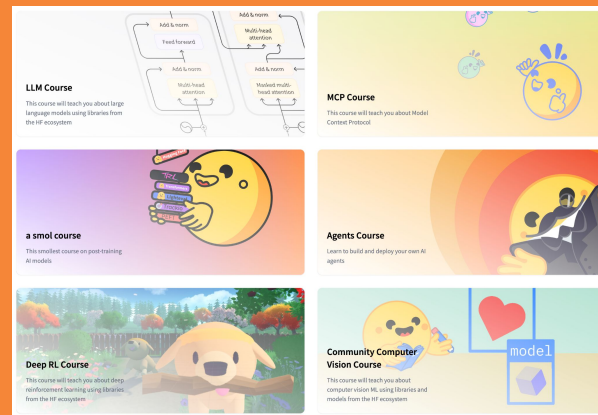
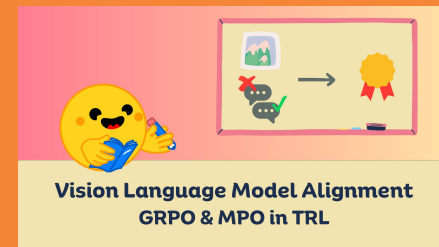
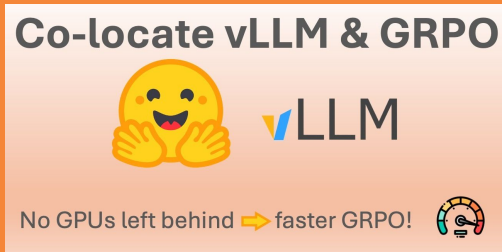
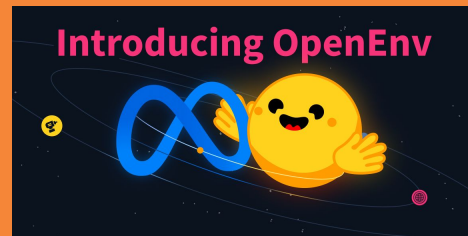
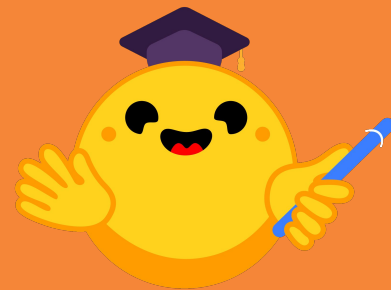
def reward_from_env(completions, **kwargs):
    """Extract environment rewards passed via rollout_func kwargs."""
    env_rewards = kwargs.get("env_reward", [])
    return [float(reward) for reward in env_rewards] if env_rewards else [0.0] * len(completions)

dataset = Dataset.from_dict({"prompt": ["You are an AI that interacts with an *Echo* environment. Word to echo:" ] * 64})

# Setup trainer with custom rollout
trainer = GRPOTrainer(
    model="Qwen/Qwen2.5-0.5B-Instruct",
    reward_funcs=reward_from_env,
    train_dataset=dataset,
    rollout_func=rollout_func, # Use custom rollout
    args=GRPOConfig(
        ...
    ),
)
trainer.train()
```

Resources

- Scripts
- Notebooks
- Cookbooks
- Example guides
- Blogs
- Courses



Thanks!

Sergio Paniego Blanco (@sergiopaniego)

Machine Learning Engineer @ Hugging Face

