Back to blog

Fine-tuning 20B LLMs with RLHF on a 24GB consumer GPU

Published March 9, 2023

Update on GitHub

 edbeeching
Edward Beeching

 ybelkada
Younes Belkada

 lvwerra
Leandro von Werra

 smangrul
Sourab Mangrulkar

 lewtun
Lewis Tunstall

 kashif
Kashif Rasul

We are excited to officially release the integration of `trl` with `peft` to make Large Language Model (LLM) fine-tuning with Reinforcement Learning more accessible to anyone! In this post, we explain why this is a competitive alternative to existing fine-tuning approaches.

Note `peft` is a general tool that can be applied to many ML use-cases but it's particularly interesting for RLHF as this method is especially memory-hungry!

If you want to directly deep dive into the code, check out the example scripts directly on the documentation page of TRL.

Introduction

LLMs & RLHF

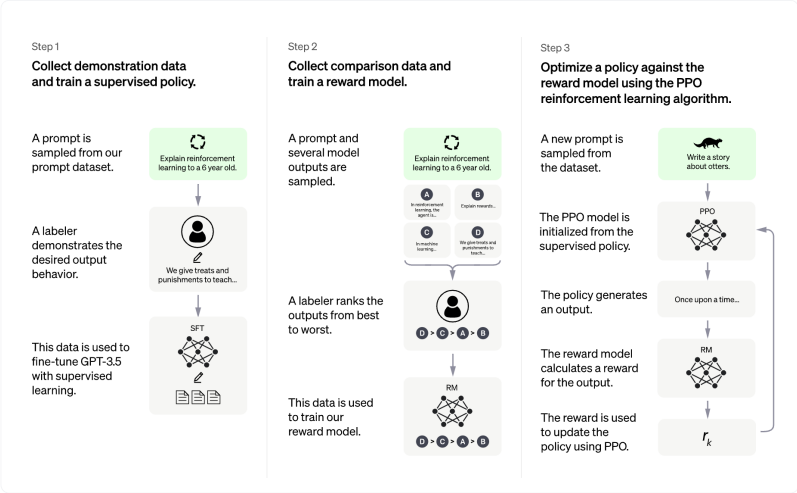
LLMs combined with RLHF (Reinforcement Learning with Human Feedback) seems to be the next go-to approach for building very powerful AI systems such as ChatGPT.

Training a language model with RLHF typically involves the following three steps:

1- Fine-tune a pretrained LLM on a specific domain or corpus of instructions and human demonstrations

2- Collect a human annotated dataset and train a reward model

3- Further fine-tune the LLM from step 1 with the reward model and this dataset using RL (e.g. PPO)



Overview of ChatGPT's training protocol, from the data collection to the RL part. Source: OpenAI's ChatGPT blogpost

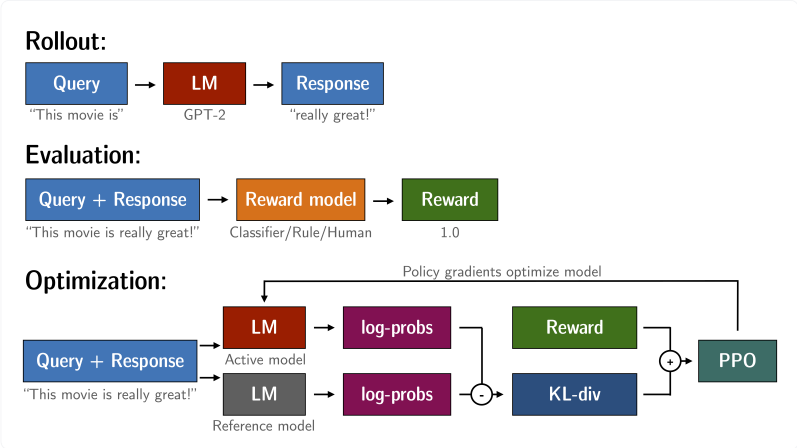
The choice of the base LLM is quite crucial here. At this time of writing, the "best" open-source LLM that can be used "out-of-the-box" for many tasks are instruction finetuned LLMs. Notable models being: BLOOMZ, Flan-T5, Flan-UL2, and OPT-IML. The downside of these models is their size. To get a decent model, you need at least to play with 10B+ scale models which would require up to 40GB GPU memory in full precision, just to fit the model on a single GPU device without doing any training at all!

What is TRL?

The `trl` library aims at making the RL step much easier and more flexible so that anyone can fine-tune their LM using RL on their custom dataset and training setup. Among many other applications, you can use this algorithm to fine-tune a model to generate positive movie reviews, do controlled generation or make the model less toxic.

Using `trl` you can run one of the most popular Deep RL algorithms, PPO, in a distributed manner or on a single device! We leverage `accelerate` from the Hugging Face ecosystem to make this possible, so that any user can scale up the experiments up to an interesting scale.

Fine-tuning a language model with RL follows roughly the protocol detailed below. This requires having 2 copies of the original model; to avoid the active model deviating too much from its original behavior / distribution you need to compute the logits of the reference model at each optimization step. This adds a hard constraint on the optimization process as you need always at least two copies of the model per GPU device. If the model grows in size, it becomes more and more tricky to fit the setup on a single GPU.



Overview of the PPO training setup in TRL.

In `trl` you can also use shared layers between reference and active models to avoid entire copies. A concrete example of this feature is showcased in the detoxification example.

Training at scale

Training at scale can be challenging. The first challenge is fitting the model and its optimizer states on the available GPU devices. The amount of GPU memory a single parameter takes depends on its "precision" (or more specifically `dtype`). The most common `dtype` being `float32` (32-bit), `float16`, and `bfloat16` (16-bit). More recently "exotic" precisions are supported out-of-the-box for training and inference (with certain conditions and constraints) such as `int8` (8-bit). In a nutshell, to load a model on a GPU device each billion parameters costs 4GB in float32 precision, 2GB in float16, and 1GB in int8. If you would like to learn more about this topic, have a look at this blogpost which dives deeper: https://huggingface.co/blog/hf-bitsandbytes-integration.

If you use an AdamW optimizer each parameter needs 8 bytes (e.g. if your model has 1B parameters, the full AdamW optimizer of the model would require 8GB GPU memory - source).

Many techniques have been adopted to tackle these challenges at scale. The most familiar paradigms are Pipeline Parallelism, Tensor Parallelism, and Data Parallelism.

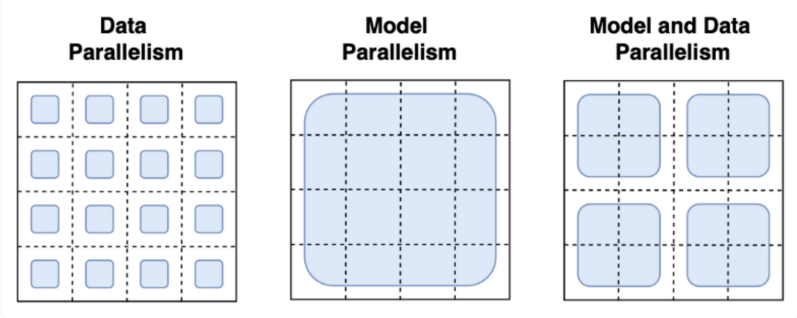


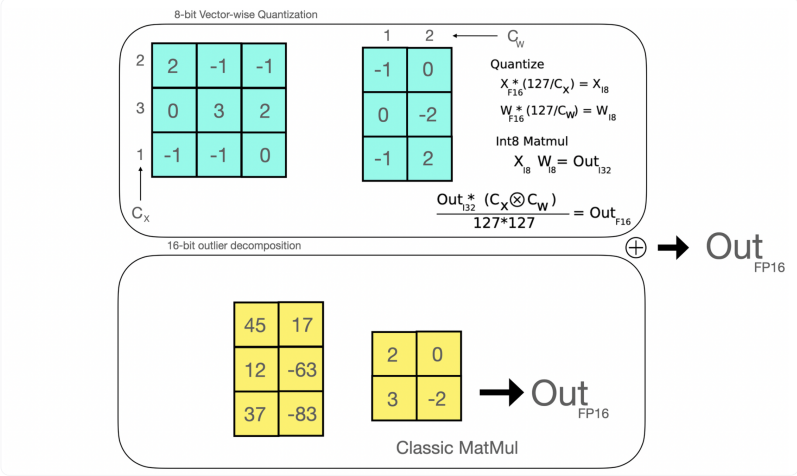
Image Credits to this blogpost

With data parallelism the same model is hosted in parallel on several machines and each instance is fed a different data batch. This is the most straight forward parallelism strategy essentially replicating the single-GPU case and is already supported by `trl`. With Pipeline and Tensor Parallelism the model itself is distributed across machines: in Pipeline Parallelism the model is split layer-wise, whereas Tensor Parallelism splits tensor operations across GPUs (e.g. matrix multiplications). With these Model Parallelism strategies, you need to shard the model weights across many devices which requires you to define a communication protocol of the activations and gradients across processes. This is not trivial to implement and might need the adoption of some frameworks such as `Megatron-DeepSpeed` or `Nemo`. It is also important to highlight other tools that are essential for scaling LLM training such as Adaptive activation checkpointing and fused kernels. Further reading about parallelism paradigms can be found here.

Therefore, we asked ourselves the following question: how far can we go with just data parallelism? Can we use existing tools to fit super-large training processes (including active model, reference model and optimizer states) in a single device? The answer appears to be yes. The main ingredients are: adapters and 8bit matrix multiplication! Let us cover these topics in the following sections:

8-bit matrix multiplication

Efficient 8-bit matrix multiplication is a method that has been first introduced in the paper LLM.int8() and aims to solve the performance degradation issue when quantizing large-scale models. The proposed method breaks down the matrix multiplications that are applied under the hood in Linear layers in two stages: the outlier hidden states part that is going to be performed in float16 & the “non-outlier” part that is performed in int8.

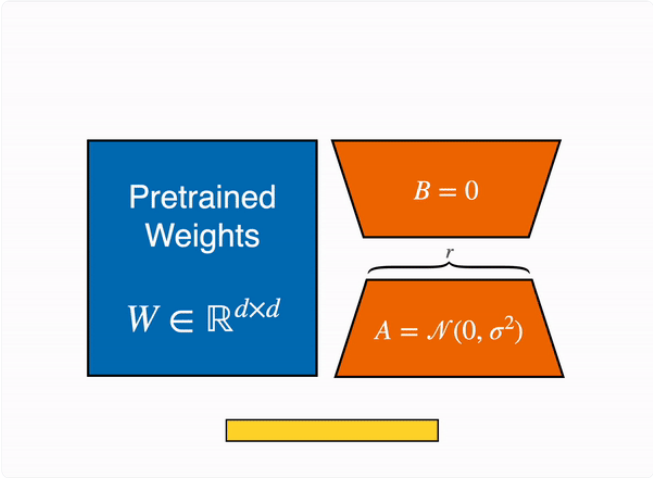


Efficient 8-bit matrix multiplication is a method that has been first introduced in the paper LLM.int8() and aims to solve the performance degradation issue when quantizing large-scale models. The proposed method breaks down the matrix multiplications that are applied under the hood in Linear layers in two stages: the outlier hidden states part that is going to be performed in float16 & the “non-outlier” part that is performed in int8.

In a nutshell, you can reduce the size of a full-precision model by 4 (thus, by 2 for half-precision models) if you use 8-bit matrix multiplication.

Low rank adaptation and PEFT

In 2021, a paper called LoRA: Low-Rank Adaption of Large Language Models demonstrated that fine tuning of large language models can be performed by freezing the pretrained weights and creating low rank versions of the query and value layers attention matrices. These low rank matrices have far fewer parameters than the original model, enabling fine-tuning with far less GPU memory. The authors demonstrate that fine-tuning of low-rank adapters achieved comparable results to fine-tuning the full pretrained model.



The output activations original (frozen) pretrained weights (left) are augmented by a low rank adapter comprised of weight matrics A and B (right).

This technique allows the fine tuning of LLMs using a fraction of the memory requirements. There are, however, some downsides. The forward and backward pass is approximately twice as slow, due to the additional matrix multiplications in the adapter layers.

What is PEFT?

Parameter-Efficient Fine-Tuning (PEFT), is a Hugging Face library, created to support the creation and fine tuning of adapter layers on LLMs.`peft` is seamlessly integrated with 🤗 Accelerate for large scale models leveraging DeepSpeed and Big Model Inference.

The library supports many state of the art models and has an extensive set of examples, including:

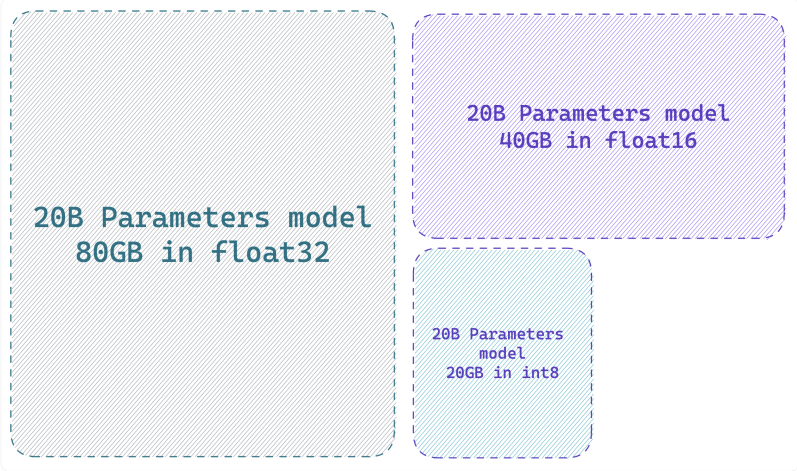
- Causal language modeling
- Conditional generation
- Image classification
- 8-bit int8 training
- Low Rank adaption of Dreambooth models
- Semantic segmentation
- Sequence classification
- Token classification

The library is still under extensive and active development, with many upcoming features to be announced in the coming months.

Fine-tuning 20B parameter models with Low Rank Adapters

Now that the prerequisites are out of the way, let us go through the entire pipeline step by step, and explain with figures how you can fine-tune a 20B parameter LLM with RL using the tools mentioned above on a single 24GB GPU!

Step 1: Load your active model in 8-bit precision



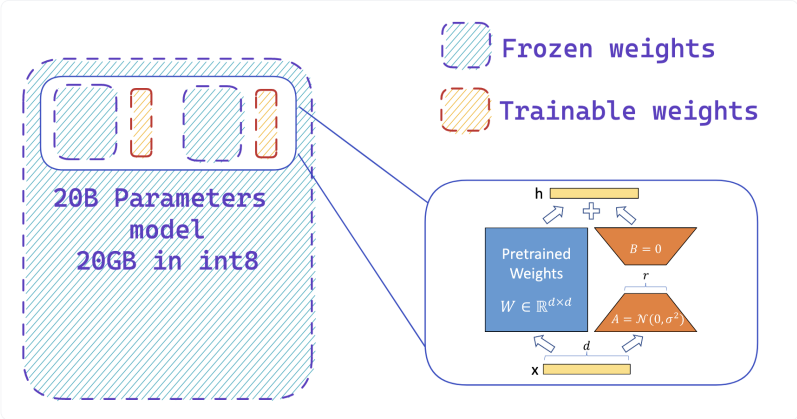
Loading a model in 8-bit precision can save up to 4x memory compared to full precision model

A “free-lunch” memory reduction of a LLM using `transformers` is to load your model in 8-bit precision using the method described in LLM.int8. This can be performed by simply adding the flag `load_in_8bit=True` when calling the `from_pretrained` method (you can read more about that here).

As stated in the previous section, a “hack” to compute the amount of GPU memory you should need to load your model is to think in terms of “billions of parameters”. As one byte needs 8 bits, you need 4GB per billion parameters for a full-precision model (32bit = 4bytes), 2GB per billion parameters for a half-precision model, and 1GB per billion parameters for an int8 model.

So in the first place, let’s just load the active model in 8-bit. Let’s see what we need to do for the second step!

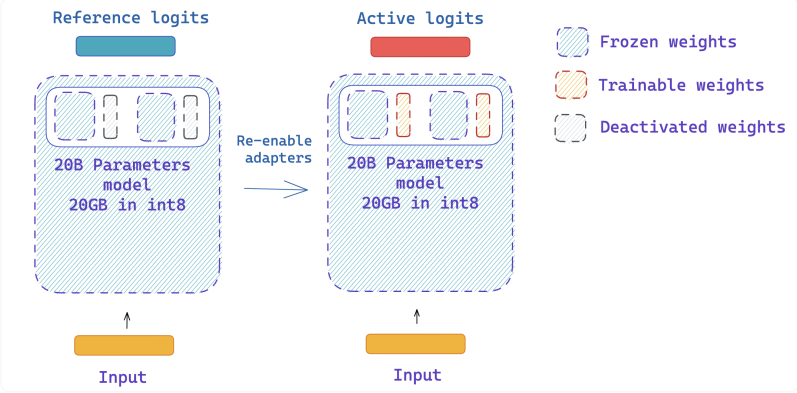
Step 2: Add extra trainable adapters using `peft`



You easily add adapters on a frozen 8-bit model thus reducing the memory requirements of the optimizer states, by training a small fraction of parameters

The second step is to load adapters inside the model and make these adapters trainable. This enables a drastic reduction of the number of trainable weights that are needed for the active model. This step leverages `peft` library and can be performed with a few lines of code. Note that once the adapters are trained, you can easily push them to the Hub to use them later.

Step 3: Use the same model to get the reference and active logits



You can easily disable and enable adapters using the `peft` API.

Since adapters can be deactivated, we can use the same model to get the reference and active logits for PPO, without having to create two copies of the same model! This leverages a feature in `peft` library, which is the `disable_adapters` context manager.

Overview of the training scripts:

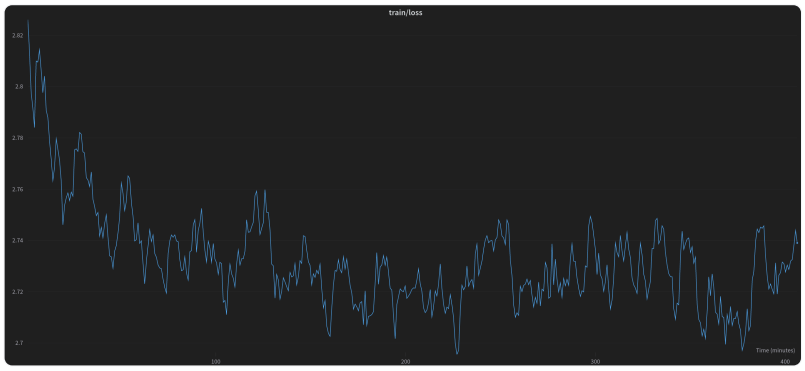
We will now describe how we trained a 20B parameter gpt-neox model using `transformers`, `peft` and `trl`. The end goal of this example was to fine-tune a LLM to generate positive movie reviews in a memory constrained settting. Similar steps could be applied for other tasks, such as dialogue models.

Overall there were three key steps and training scripts:

1. Script - Fine tuning a Low Rank Adapter on a frozen 8-bit model for text generation on the imdb dataset.
2. Script - Merging of the adapter layers into the base model's weights and storing these on the hub.
3. Script - Sentiment fine-tuning of a Low Rank Adapter to create positive reviews.

We tested these steps on a 24GB NVIDIA 4090 GPU. While it is possible to perform the entire training run on a 24 GB GPU, the full training runs were untaken on a single A100 on the 🤗 reseach cluster.

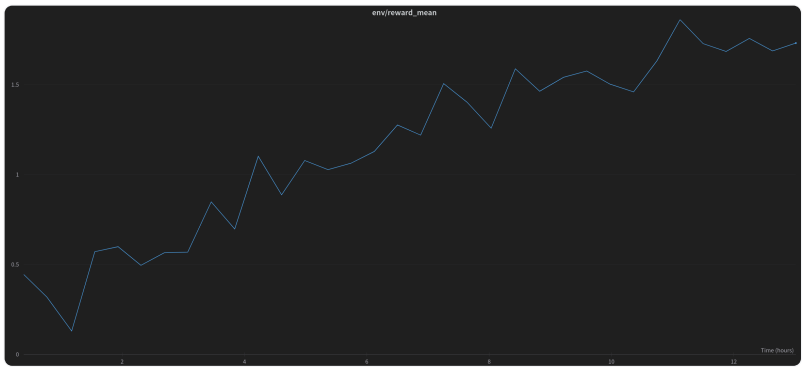
The first step in the training process was fine-tuning on the pretrained model. Typically this would require several high-end 80GB A100 GPUs, so we chose to train a low rank adapter. We treated this as a Causal Language modeling setting and trained for one epoch of examples from the imdb dataset, which features movie reviews and labels indicating whether they are of positive or negative sentiment.



Training loss during one epoch of training of a gpt-neox-20b model for one epoch on the imdb dataset

In order to take the adapted model and perform further finetuning with RL, we first needed to combine the adapted weights, this was achieved by loading the pretrained model and adapter in 16-bit floating point and summary with weight matrices (with the appropriate scaling applied).

Finally, we could then fine-tune another low-rank adapter, on top of the frozen imdb-finetuned model. We use an imdb sentiment classifier to provide the rewards for the RL algorithm.



Mean of rewards when RL fine-tuning of a peft adapted 20B parameter model to generate positive movie reviews.

The full Weights and Biases report is available for this experiment here, if you want to check out more plots and text generations.

Conclusion

We have implemented a new functionality in `trl` that allows users to fine-tune large language models using RLHF at a reasonable cost by leveraging the `peft` and `bitsandbytes` libraries. We demonstrated that fine-tuning `gpt-neo-x` (40GB in `bfloat16`!) on a 24GB consumer GPU is possible, and we expect that this integration will be widely used by the community to fine-tune larger models utilizing RLHF and share great artifacts.

We have identified some interesting directions for the next steps to push the limits of this integration

- *How this will scale in the multi-GPU setting?* We'll mainly explore how this integration will scale with respect to the number of GPUs, whether it is possible to apply Data Parallelism out-of-the-box or if it'll require some new feature adoption on any of the involved libraries.
- *What tools can we leverage to increase training speed?* We have observed that the main downside of this integration is the overall training speed. In the future we would be keen to explore the possible directions to make the training much faster.

References

- parallelism paradigms: https://huggingface.co/docs/transformers/v4.17.0/en/parallelism
- 8-bit integration in `transformers`: https://huggingface.co/blog/hf-bitsandbytes-integration
- LLM.int8 paper: https://arxiv.org/abs/2208.07339
- Gradient checkpoiting explained: https://docs.aws.amazon.com/sagemaker/latest/dg/model-parallel-extended-features-pytorch-activation-checkpointing.html

More articles from our Blog

Fine-tune Llama 2 with DPO

By kashif · August 8, 2023

StackLLaMA: A hands-on guide to train LLaMA with RLHF

By edbeeching · April 5, 2023