Search models, datasets, users...

Back to Articles

Fine-tune Llama 2 with DPO

Published August 8, 2023

Update on GitHub

Upvote 16 +10

kashif
Kashif Rasul

ybelkada
Younes Belkada

lvwerra
Leandro von Werra

Introduction

Reinforcement Learning from Human Feedback (RLHF) has become the de facto last training step of LLMs such as GPT-4 or Claude to ensure that the language model's outputs are aligned with human expectations such as chattiness or safety features. However, it brings some of the complexity of RL into NLP: we need to build a good reward function, train the model to estimate the value of a state, and at the same time be careful not to strive too far from the original model and produce gibberish instead of sensible text. Such a process is quite involved requiring a number of complex moving parts where it is not always easy to get things right.

The recent paper Direct Preference Optimization by Rafailov, Sharma, Mitchell et al. proposes to cast the RL-based objective used by existing methods to an objective which can be directly optimized via a simple binary cross-entropy loss which simplifies this process of refining LLMs greatly.

This blog-post introduces the Direct Preference Optimization (DPO) method which is now available in the TRL library and shows how one can fine tune the recent Llama v2 7B-parameter model on the stack-exchange preference dataset which contains ranked answers to questions on the various stack-exchange portals.

DPO vs PPO

In the traditional model of optimising human derived preferences via RL, the goto method has been to use an auxiliary reward model and fine-tune the model of interest so that it maximizes this given reward via the machinery of RL. Intuitively we use the reward model to provide feedback to the model we are optimising so that it generates high-reward samples more often and low-reward samples less often. At the same time we use a frozen reference model to make sure that whatever is generated does not deviate too much and continues to maintain generation diversity. This is typically done by adding a KL penalty to the full reward maximisation objective via a reference model, which serves to prevent the model from learning to cheat or exploit the reward model.

The DPO formulation bypasses the reward modeling step and directly optimises the language model on preference data via a key insight: namely an analytical mapping from the reward function to the optimal RL policy that enables the authors to transform the RL loss over the reward and reference models to a loss over the reference model directly! This mapping intuitively measures how well a given reward function aligns with the given preference data. DPO thus starts with the optimal solution to the RLHF loss and via a change of variables derives a loss over *only* the reference model!

Thus this direct likelihood objective can be optimized without the need for a reward model or the need to perform the potentially fiddly RL based optimisation.

How to train with TRL

As mentioned, typically the RLHF pipeline consists of these distinct parts:

1. a supervised fine-tuning (SFT) step

2. the process of annotating data with preference labels
3. training a reward model on the preference data
4. and the RL optmization step

The TRL library comes with helpers for all these parts, however the DPO training does away with the task of reward modeling and RL (steps 3 and 4) and directly optimizes the DPO object on preference annotated data.

In this respect we would still need to do the step 1, but instead of steps 3 and 4 we need to provide the `DPOTrainer` in TRL with preference data from step 2 which has a very specific format, namely a dictionary with the following three keys:

- `prompt` this consists of the context prompt which is given to a model at inference time for text generation
- `chosen` contains the preferred generated response to the corresponding prompt
- `rejected` contains the response which is not preferred or should not be the sampled response with respect to the given prompt

As an example, for the stack-exchange preference pairs dataset, we can map the dataset entries to return the desired dictionary via the following helper and drop all the original columns:

```
def return_prompt_and_responses(samples) -> Dict[str, str, str]:
    return {
        "prompt": [
            "Question: " + question + "\n\nAnswer: "
            for question in samples["question"]
        ],
        "chosen": samples["response_j"],   # rated better than k
        "rejected": samples["response_k"], # rated worse than j
    }

dataset = load_dataset(
    "lvwerra/stack-exchange-paired",
    split="train",
    data_dir="data/rl"
)
original_columns = dataset.column_names

dataset.map(
    return_prompt_and_responses,
    batched=True,
    remove_columns=original_columns
)
```

Once we have the dataset sorted the DPO loss is essentially a supervised loss which obtains an implicit reward via a reference model and thus at a high-level the `DPOTrainer` requires the base model we wish to optimize as well as a reference model:

```
dpo_trainer = DPOTrainer(
    model,                 # base model from SFT pipeline
    model_ref,             # typically a copy of the SFT trained base 
    beta=0.1,              # temperature hyperparameter of DPO
    train_dataset=dataset, # dataset prepared above
    tokenizer=tokenizer,   # tokenizer
    args=training_args,    # training arguments e.g. batch size, lr, e
)
```

where the `beta` hyper-parameter is the temperature parameter for the DPO loss, typically in the range `0.1` to `0.5`. This controls how much we pay attention to the reference model in the sense that as `beta` gets smaller the more we ignore the reference model. Once we have our trainer initialised we can then train it on the dataset with the given `training_args` by simply calling:

```
dpo_trainer.train()
```

Experiment with Llama v2

The benefit of implementing the DPO trainer in TRL is that one can take advantage of all the extra bells and whistles of training large LLMs which come with TRL and its dependent libraries like Peft and Accelerate. With these libraries we are even able to train a Llama v2 model using the QLoRA technique provided by the bitsandbytes library.

Supervised Fine Tuning

The process as introduced above involves the supervised fine-tuning step using QLoRA on the 7B Llama v2 model on the SFT split of the data via TRL's `SFTTrainer`:

```
# load the base model in 4-bit quantization
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16,
)

base_model = AutoModelForCausalLM.from_pretrained(
    script_args.model_name,        # "meta-llama/Llama-2-7b-hf"
    quantization_config=bnb_config,
    device_map={"": 0},
    trust_remote_code=True,
    use_auth_token=True,
)
base_model.config.use_cache = False

# add LoRA layers on top of the quantized base model
peft_config = LoraConfig(
    r=script_args.lora_r,
    lora_alpha=script_args.lora_alpha,
    lora_dropout=script_args.lora_dropout,
    target_modules=["q_proj", "v_proj"],
    bias="none",
    task_type="CAUSAL_LM",
)
...
trainer = SFTTrainer(
    model=base_model,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    peft_config=peft_config,
    packing=True,
    max_seq_length=None,
    tokenizer=tokenizer,
    args=training_args,         # HF Trainer arguments
)
trainer.train()
```

DPO Training

Once the SFT has finished, we can save the resulting model and move onto the DPO training. As is typically done we will utilize the saved model from the previous SFT step for both the base model as well as reference model of DPO. Then we can use these to train the model with the DPO objective on the stack-exchange preference data shown above. Since the models were trained via LoRa adapters, we load the models via Peft's `AutoPeftModelForCausalLM` helpers:

```
model = AutoPeftModelForCausalLM.from_pretrained(
    script_args.model_name_or_path, # location of saved SFT model
    low_cpu_mem_usage=True,
    torch_dtype=torch.float16,
    load_in_4bit=True,
    is_trainable=True,
)
model_ref = AutoPeftModelForCausalLM.from_pretrained(
    script_args.model_name_or_path,  # same model as the main one
    low_cpu_mem_usage=True,
    torch_dtype=torch.float16,
    load_in_4bit=True,
)
...
dpo_trainer = DPOTrainer(
    model,
    model_ref,
    args=training_args,
    beta=script_args.beta,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    tokenizer=tokenizer,
    peft_config=peft_config,
)
dpo_trainer.train()
dpo_trainer.save_model()
```

So as can be seen we load the model in the 4-bit configuration and then train it via the QLora method via the `peft_config` arguments. The trainer will also evaluate the progress during training with respect to the evaluation dataset and report back a number of key metrics like the implicit reward which can be recorded and displayed via WandB for example. We can then push the final trained model to the HuggingFace Hub.

Conclusion

The full source code of the training scripts for the SFT and DPO are available in the following examples/stack_llama_2 directory and the trained model with the merged adapters can be found on the HF Hub here.

The WandB logs for the DPO training run can be found here where during training and evaluation the `DPOTrainer` records the following reward metrics:

- `rewards/chosen`: the mean difference between the log probabilities of the policy model and the reference model for the chosen responses scaled by `beta`
- `rewards/rejected`: the mean difference between the log probabilities of the policy model and the reference model for the rejected responses scaled by `beta`
- `rewards/accuracies`: mean of how often the chosen rewards are > than the corresponding rejected rewards
- `rewards/margins`: the mean difference between the chosen and corresponding rejected rewards.

Intuitively, during training we want the margins to increase and the accuracies to go to 1.0, or in other words the chosen reward to be higher than the rejected reward (or the margin bigger than zero). These metrics can then be calculated over some evaluation dataset.

We hope with the code release it lowers the barrier to entry for you the readers to try out this method of aligning large language models on your own datasets and we cannot wait to see what you build! And if you want to try out the model yourself you can do so here: trl-lib/stack-llama.

More Articles from our Blog

Putting RL back in RLHF

By vwxyzjn June 12, 2024 • △ 46

Jack of All Trades, Master of Some, a Multi-Purpose Transformer Agent

By qgallouedec April 22, 2024 • △ 75

🤗

Company

TOS

Privacy

About

Jobs

Website

Models

Datasets

Spaces

Pricing

Docs