direct-preference-optimization Public

eric-mitchell 16 Commits

co...	Rename ...	2 months ago
LI...	Initial co...	7 months ago
R...	Updated...	last month
pr...	Fixed tok...	7 months ago
re...	Initial co...	7 months ago
tr...	Added c...	2 months ago
tr...	Rename ...	2 months ago
uti...	Updated...	6 months ago

About

Reference implementation for DPO (Direct Preference Optimization)

Readme
Apache-2.0 license
Activity
1.2k stars
13 watching
86 forks
Report repository

Releases

No releases published

Packages

README Apache-2.0 license

Languages

Python 100.0%

DPO: Direct Preference Optimization

New: in addition to the original DPO algorithm, this repo now supports 'conservative' DPO and IPO.

For conservative DPO, you just need to additionally pass the parameter `loss.label_smoothing=X` for some `X` between 0 and 0.5 when performing DPO training (0 gives the original DPO loss). This parameter is essentially the conservativeness parameter, i.e., the fraction of the training preference data that is incorrect (flipped preference direction). Starting with something like 0.1 might be reasonable, but I haven't tested this yet (and it will depend on the preference dataset).

For IPO, just pass `loss=ipo` and `loss.beta=X` for some non-negative `X` (same as with DPO/conservative DPO).

What is this repo?

This repo includes a reference implementation of the DPO algorithm for training language models from preference data, as described in the paper Direct Preference Optimization: Your Language Model is Secretly a Reward Model.

The code here supports any causal HuggingFace model- look at our examples in `config/model` to add your own. Adding your own datasets is also easy. See the README section on adding datasets.

The DPO pipeline has two stages:

1. Run supervised fine-tuning (SFT) on the dataset(s) of interest.
2. Run preference learning on the model from step 1, using preference data (ideally from the same distribution as the SFT examples).

The files in this repo are:

- `train.py` : the main entry point for training (either SFT or DPO preference-based training)
- `trainers.py` : the trainer classes (e.g., implementing the loop of learning as well as multi-GPU logic)
- `utils.py` : some convenience functions used by multiple other files
- `preference_datasets.py` : dataset processing logic for both SFT and DPO preference-based training; **this is where you'll need to make some additions to train on your own data**

Running SFT

For DPO, the SFT stage essentially ensures that the preference data we train on is in-distribution for our policy before we actually do the learning from preferences part.

Run SFT for Pythia 6.9B on Anthropic-HH data with batch size 64:

```
python -u train.py model=pythia69 datasets=[hh] loss=sft
exp_name=anthropic_dpo_pythia69
gradient_accumulation_steps=2 batch_size=64
eval_batch_size=32 trainer=FSDPTrainer
sample_during_eval=false
```

Run SFT for a custom model (for example, Llama at a local path) on Anthropic-HH + Stanford Human Preference data with batch size 64:

```
python -u train.py model=blank_model
model.name_or_path=/PATH/TO/LLAMA/WEIGHTS
model.block_name=LlamaDecoderLayer datasets=[hh,shp]
loss=sft exp_name=anthropic_shp_sft_llama_7b
gradient_accumulation_steps=2 batch_size=64
eval_batch_size=32 trainer=FSDPTrainer
sample_during_eval=false
```

> Note: Since we're not using one of our predefined model configs, we also need to pass `model.block_name` to tell FSDP what modules to wrap.

By default, evaluation will run every 20k **examples**. You can change this arg with `eval_every` arg. If you don't pass `sample_during_eval=false`, sampling will happen during each eval as well.

To run a different model, either add a new model config to `config/model`, or use the `blank_model` option for `model` and pass `model.name_or_path` (and `model.block_name` if training with FSDP trainer) explicitly. For example, for GPT-2, this would look like:

```
python -u train.py ... model=blank_model
model.name_or_path=gpt2-xl model.block=GPT2Block
```

Running DPO

To run DPO, use the same command as SFT, but pass `loss=dpo`, `loss.beta=DESIRED_BETA` (0.1-0.5 is a good starting point), and `model.archive=/path/to/checkpoint/from/sft/step-XXXX/policy.pt`. If SFT completed successfully, you should also have a `/.../LATEST/policy.pt` from the end of training.

Run DPO on Pythia 6.9B with effective batch size 64:

```
python -u train.py model=pythia69 datasets=[hh] loss=dpo
loss.beta=0.1
model.archive=/path/to/checkpoint/from/sft/step-
XXXX/policy.pt exp_name=anthropic_dpo_pythia69
gradient_accumulation_steps=2 batch_size=32
eval_batch_size=32 trainer=FSDPTrainer
sample_during_eval=false
```

> Note: `eval_every` is measured in **examples**.

A complete example

Let's work through a complete example training pythia 2.8B on the Anthropic-HH dataset.

See sample wandb outputs for this example here (tagged `readme-example`).

Step 1: Set up environment

First, create a virtualenv and install the dependencies. Python 3.8+ is recommended.

```
python3 -m venv env
source env/bin/activate
pip install -r requirements.txt
```

Step 2: Run SFT

We'll take advantage of FSDP's mixed precision in bfloat16 to speed up training; we usually see about a 50% speedup. By default, SFT will run for a single epoch over a mixture of the selected datasets. Datasets will be downloaded on the fly and cached locally.

```
python -u train.py model=pythia28 datasets=[hh] loss=sft
exp_name=anthropic_dpo_pythia28
gradient_accumulation_steps=2 batch_size=64
eval_batch_size=32 trainer=FSDPTrainer
sample_during_eval=false model.fsdp_policy_mp=bfloat16
```

> Note: this command is run on a machine with 4 80GB A100s; on this hardware, SFT takes about 1hr 30min. If you have less compute available, you might need to increase the number of gradient accumulation steps, and SFT will take longer.

See sample wandb outputs for the SFT step here.

Step 3: Run DPO

Check either wandb (if enabled, it is by default) or your output log to find the local run directory. To run DPO, you'll need the path to the final weights, which will look something like `/some/cache/dir/YOUR_USERNAME/pythia28_hh_sft_bf16_2023-06-21_16-58-17_973996/LATEST/policy.pt`. The `LATEST` directory contains the final set of weights from the end of training.

```
python -u train.py model=pythia28 datasets=[hh] loss=dpo
loss.beta=0.1 exp_name=anthropic_dpo_pythia28
gradient_accumulation_steps=2 batch_size=64
eval_batch_size=32 trainer=FSDPTrainer
sample_during_eval=false model.fsdp_policy_mp=bfloat16
model.archive=/path/to/archive/from/sft/LATEST/policy.pt
```

On 4 80GB A100s, DPO training took about 2hrs 45min.

See sample wandb outputs for the DPO step here.

Customizing training

The options for training are in `config/config.yaml`, `config/model/blank_model.yaml`, and `config/loss/dpo.yaml`. See the comments in these files for more information on what they do.

You can use one of the pre-configured models by passing `model=some_model`, where `config/model/some_model.yaml` exists. We have a few examples already given.

If you want to use another model, just create a new config for that model (following our examples; it must be a `.yaml` file!), or use `model=blank_model` with `model.name_or_path=NAME_OR_PATH`, optionally `model.tokenizer_name_or_path=TOKENIZER_NAME_OR_PATH` if it is different than the model's name/path, and `model.block_name=NAME_OF_TRANSFORMER_BLOCK` (if you are using FSDP). The only other options you might want to change are the dpo loss options, which are `loss.beta` and `loss.reference_free` (see `config/loss/dpo.yaml`).

Trainer classes

We implement three different trainer classes in `trainers.py`:

- `BasicTrainer`: For multiple GPUs, naively partition the model among them. e.g., for two GPUs, the first half of the model layers will be on GPU 0, the second half will be on GPU 1. This trainer effectively increases your available GPU memory without using multiple GPUs are

once for compute (so you get no speedup).

- `FSDPTrainer`: Use PyTorch's Fully Sharded Data Parallel (FSDP) implementation to shard each transformer block amongst available GPUs. Should give a significant speedup over `BasicTrainer` with batch size per GPU >1. The batch size per gpu is equal to `batch_size / (gradient_accumulation_steps * num_gpus)`. **You may need to run `ulimit -n 64000` in your launch script before calling `train.py` with this trainer; e.g., `ulimit -n 64000; python train.py ...`.**
- `TensorParallelTrainer`: Use PyTorch tensor parallelism (with this wrapper) to shard each linear layer amongst available GPUs. This trainer is experimental, but should work.

Warning: Sampling may be very slow for `FSDPTrainer` and especially `TensorParallelTrainer` (see this issue and this issue, respectively for `FSDPTrainer` and `TensorParallelTrainer`). Passing `sample_during_eval=false` is recommended for these trainers.

Which trainer do I use?

For single GPU training, use `BasicTrainer`. For many-GPU setups, `FSDPTrainer` will most likely be the best choice, though these haven't been benchmarked yet.

Adding new datasets

Adding new/custom datasets is easy, and shouldn't take more than 10 minutes or so. Add your dataset to `preference_datasets.py` (we've implemented Anthropic-HH, Stanford Human Preferences, and StackExchange as references). Follow our reference datasets (in the functions `get_se()`, `get_shp()`, `get_hh()`); you essentially need to return a dict mapping each prompt to another dict containing three values:

- `responses: List[str]`: the list of responses on which preferences are given
- `pairs: List[Tuple[int]]`: the preference pairs, where the first value in each tuple is the preferred response and the second value is the

dispreferred response

- `sft_target: str` : the response to use for this prompt during SFT (this response may or may not be one of the values in `responses`)

Once you've added your dataset, for example `xyz` , you can train on it by passing it to `datasets=[xyz]` to an SFT or DPO train command.

Make sure you've updated `preference_datasets:get_dataset()` to return your new dataset when its name is passed in!

Tips for faster training on multiple GPUs

FSDP is recommended for faster training when multiple GPUs are available. In general, you should try to use a batch size of at least 2 on each GPU (i.e., `batch_size // (grad_accumulation_steps * N_GPUS)` is at least 2) to see a speedup from FSDP compared to the `BasicTrainer` . One way to do this is to use mixed precision. This repo implements mixed precision through FSDP. Enable mixed precision (only supported for `FSDPTrainer` , currently) by passing `model.fsdp_policy_mp=bfloat16` or `model.fsdp_policy_mp=float16` (only `bfloat16` has been tested). Another way to reduce memory usage is activation checkpointing (or *gradient checkpointing*), which can be enabled with `activation_checkpointing=true` (also implemented only for `FSDPTrainer`). Activation checkpointing doesn't always increase throughput, but if you're stuck at batch size per GPU of 1, it's worth a try.

See this article for more information about optimizing FSDP.

Citing DPO

If DPO or this repository is useful in your own research, you can use the following BibTeX entry:

```
@inproceedings{
    rafailov2023direct,
    title={Direct Preference Optimization: Your Language
Model is Secretly a Reward Model},
    author={Rafael Rafailov and Archit Sharma and Eric
Mitchell and Christopher D Manning and Stefano Ermon and
Chelsea Finn},
    booktitle={Thirty-seventh Conference on Neural
Information Processing Systems},
    year={2023},
    url={https://arxiv.org/abs/2305.18290}
}
```