Supervised Fine-tuning Trainer

Supervised fine-tuning (or SFT for short) is a crucial step in RLHF. In TRL we provide an easy-to-use API to create your SFT models and train them with few lines of code on your dataset.

Check out a complete flexible example at `examples/scripts/sft.py`. Experimental support for Vision Language Models is also included in the example `examples/scripts/vsft_llava.py`.

Quickstart

If you have a dataset hosted on the 🤗 Hub, you can easily fine-tune your SFT model using SFTTrainer from TRL. Let us assume your dataset is `imdb`, the text you want to predict is inside the `text` field of the dataset, and you want to fine-tune the `facebook/opt-350m` model. The following code-snippet takes care of all the data pre-processing and training for you:

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

dataset = load_dataset("stanfordnlp/imdb", split="train")

sft_config = SFTConfig(
    dataset_text_field="text",
    max_seq_length=512,
    output_dir="/tmp",
)
trainer = SFTTrainer(
    "facebook/opt-350m",
    train_dataset=dataset,
    args=sft_config,
)
trainer.train()
```

Make sure to pass the correct value for `max_seq_length` as the default value will be set to `min(tokenizer.model_max_length, 1024)`.

You can also construct a model outside of the trainer and pass it as follows:

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

dataset = load_dataset("stanfordnlp/imdb", split="train")

model = AutoModelForCausalLM.from_pretrained("facebook/opt-350m")

sft_config = SFTConfig(output_dir="/tmp")

trainer = SFTTrainer(
    model,
    train_dataset=dataset,
    args=sft_config,
)

trainer.train()
```

The above snippets will use the default training arguments from the SFTConfig class. If you want to modify the defaults pass in your modification to the `SFTConfig` constructor and pass them to the trainer via the `args` argument.

Advanced usage

Train on completions only

You can use the `DataCollatorForCompletionOnlyLM` to train your model on the generated prompts only. Note that this works only in the case when `packing=False`. To instantiate that collator for instruction data, pass a response template and the tokenizer. Here is an example of how it would work to fine-tune `opt-350m` on completions only on the CodeAlpaca dataset:

```
from transformers import AutoModelForCausalLM, AutoTokenizer
from datasets import load_dataset
from trl import SFTConfig, SFTTrainer, DataCollatorForCompletionOnlyLM

dataset = load_dataset("lucasmccabe-lmi/CodeAlpaca-20k", split="train")

model = AutoModelForCausalLM.from_pretrained("facebook/opt-350m")
tokenizer = AutoTokenizer.from_pretrained("facebook/opt-350m")

def formatting_prompts_func(example):
    output_texts = []
    for i in range(len(example['instruction'])):
        text = f"### Question: {example['instruction'][i]}\n ### Answer: {example['output'][i]}"
        output_texts.append(text)
    return output_texts
```

```
response_template = " ### Answer:"
collator = DataCollatorForCompletionOnlyLM(response_template, tokenizer=tokenizer)

trainer = SFTTrainer(
    model,
    train_dataset=dataset,
    args=SFTConfig(output_dir="/tmp"),
    formatting_func=formatting_prompts_func,
    data_collator=collator,
)

trainer.train()
```

To instantiate that collator for assistant style conversation data, pass a response template, an instruction template and the tokenizer. Here is an example of how it would work to fine-tune `opt-350m` on assistant completions only on the Open Assistant Guanaco dataset:

```
from transformers import AutoModelForCausalLM, AutoTokenizer
from datasets import load_dataset
from trl import SFTConfig, SFTTrainer, DataCollatorForCompletionOnlyLM

dataset = load_dataset("timdettmers/openassistant-guanaco", split="train")

model = AutoModelForCausalLM.from_pretrained("facebook/opt-350m")
tokenizer = AutoTokenizer.from_pretrained("facebook/opt-350m")

instruction_template = "### Human:"
response_template = "### Assistant:"
collator = DataCollatorForCompletionOnlyLM(instruction_template=instruction_template, response_t

trainer = SFTTrainer(
    model,
    args=SFTConfig(
        output_dir="/tmp",
        dataset_text_field = "text",
    ),
    train_dataset=dataset,
    data_collator=collator,
)

trainer.train()
```

Make sure to have a `pad_token_id` which is different from `eos_token_id` which can result in the model not properly predicting EOS (End of Sentence) tokens during generation.

Using token_ids directly for response_template

Some tokenizers like Llama 2 (`meta-llama/Llama-2-XXb-hf`) tokenize sequences differently depending on whether they have context or not. For example:

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("meta-llama/Llama-2-7b-hf")

def print_tokens_with_ids(txt):
    tokens = tokenizer.tokenize(txt, add_special_tokens=False)
    token_ids = tokenizer.encode(txt, add_special_tokens=False)
    print(list(zip(tokens, token_ids)))

prompt = """### User: Hello\n\n### Assistant: Hi, how can I help you?"""
print_tokens_with_ids(prompt)  # [..., ('_Hello', 15043), ('<0x0A>', 13), ('<0x0A>', 13), ('##',

response_template = "### Assistant:"
print_tokens_with_ids(response_template)  # [('_###', 835), ('_Ass', 4007), ('istant', 22137),
```

In this case, and due to lack of context in `response_template`, the same string ("### Assistant:") is tokenized differently:

- Text (with context): `[2277, 29937, 4007, 22137, 29901]`
- `response_template` (without context): `[835, 4007, 22137, 29901]`

This will lead to an error when the `DataCollatorForCompletionOnlyLM` does not find the `response_template` in the dataset example text:

```
RuntimeError: Could not find response key [835, 4007, 22137, 29901] in token IDs tensor([    1,
```

To solve this, you can tokenize the `response_template` with the same context as in the dataset, truncate it as needed and pass the `token_ids` directly to the `response_template` argument of the `DataCollatorForCompletionOnlyLM` class. For example:

```
response_template_with_context = "\n### Assistant:"  # We added context here: "\n". This is enou
response_template_ids = tokenizer.encode(response_template_with_context, add_special_tokens=Fals

data_collator = DataCollatorForCompletionOnlyLM(response_template_ids, tokenizer=tokenizer)
```

Add Special Tokens for Chat Format

Adding special tokens to a language model is crucial for training chat models. These tokens are added between the different roles in a conversation, such as the user, assistant, and system and help the model recognize the structure and flow of a conversation. This setup is essential for enabling the model to generate coherent and contextually appropriate responses in a chat environment. The `setup_chat_format()` function in `trl` easily sets up a model and tokenizer for conversational AI tasks. This function:

- Adds special tokens to the tokenizer, e.g. `<|im_start|>` and `<|im_end|>`, to indicate the start and end of a conversation.
- Resizes the model's embedding layer to accommodate the new tokens.
- Sets the `chat_template` of the tokenizer, which is used to format the input data into a chat-like format. The default is `chatml` from OpenAI.
- *optionally* you can pass `resize_to_multiple_of` to resize the embedding layer to a multiple of the `resize_to_multiple_of` argument, e.g. 64. If you want to see more formats being supported in the future, please open a GitHub issue on trl

```
from transformers import AutoModelForCausalLM, AutoTokenizer
from trl import setup_chat_format

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained("facebook/opt-350m")
tokenizer = AutoTokenizer.from_pretrained("facebook/opt-350m")

# Set up the chat format with default 'chatml' format
model, tokenizer = setup_chat_format(model, tokenizer)
```

With our model and tokenizer set up, we can now fine-tune our model on a conversational dataset. Below is an example of how a dataset can be formatted for fine-tuning.

Dataset format support

The SFTTrainer supports popular dataset formats. This allows you to pass the dataset to the trainer without any pre-processing directly. The following formats are supported:

- conversational format

```
{"messages": [{"role": "system", "content": "You are helpful"}, {"role": "user", "content": "Wha
{"messages": [{"role": "system", "content": "You are helpful"}, {"role": "user", "content": "Who
{"messages": [{"role": "system", "content": "You are helpful"}, {"role": "user", "content": "How
```

- instruction format

```
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}
```

If your dataset uses one of the above formats, you can directly pass it to the trainer without pre-processing. The SFTTrainer will then format the dataset for you using the defined format from the model's tokenizer with the apply_chat_template method.

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

...

# load jsonl dataset
dataset = load_dataset("json", data_files="path/to/dataset.jsonl", split="train")
# load dataset from the HuggingFace Hub
dataset = load_dataset("philschmid/dolly-15k-oai-style", split="train")

...

sft_config = SFTConfig(packing=True)
trainer = SFTTrainer(
    "facebook/opt-350m",
    args=sft_config,
    train_dataset=dataset,
)
```

If the dataset is not in one of those format you can either preprocess the dataset to match the formatting or pass a formatting function to the SFTTrainer to do it for you. Let's have a look.

Format your input prompts

For instruction fine-tuning, it is quite common to have two columns inside the dataset: one for the prompt & the other for the response. This allows people to format examples like Stanford-Alpaca did as follows:

```
Below is an instruction ...

### Instruction
{prompt}

### Response:
{completion}
```

Let us assume your dataset has two fields, `question` and `answer`. Therefore you can just run:

```
...
def formatting_prompts_func(example):
    output_texts = []
    for i in range(len(example['question'])):
        text = f"### Question: {example['question'][i]}\n ### Answer: {example['answer'][i]}"
        output_texts.append(text)
    return output_texts

trainer = SFTTrainer(
    model,
    args=sft_config,
    train_dataset=dataset,
    formatting_func=formatting_prompts_func,
)

trainer.train()
```

To properly format your input make sure to process all the examples by looping over them and returning a list of processed text. Check out a full example of how to use SFTTrainer on alpaca dataset here

Packing dataset (ConstantLengthDataset)

SFTTrainer supports *example packing*, where multiple short examples are packed in the same input sequence to increase training efficiency. This is done with the `ConstantLengthDataset` utility class that returns constant length chunks of tokens from a stream of examples. To enable the usage of this dataset class, simply pass `packing=True` to the SFTConfig constructor.

```
...
sft_config = SFTConfig(packing=True, dataset_text_field="text",)

trainer = SFTTrainer(
    "facebook/opt-350m",
    train_dataset=dataset,
    args=sft_config
)

trainer.train()
```

Note that if you use a packed dataset and if you pass `max_steps` in the training arguments you will probably train your models for more than few epochs, depending on the way you have configured the packed dataset and the training protocol. Double check that you know and understand what you are doing. If you don't want to pack your `eval_dataset`, you can pass `eval_packing=False` to the `SFTConfig` init method.

Customize your prompts using packed dataset

If your dataset has several fields that you want to combine, for example if the dataset has `question` and `answer` fields and you want to combine them, you can pass a formatting function to the trainer that will take care of that. For example:

```
def formatting_func(example):
    text = f"### Question: {example['question']}\n ### Answer: {example['answer']}"
    return text

sft_config = SFTConfig(packing=True)
trainer = SFTTrainer(
    "facebook/opt-350m",
    train_dataset=dataset,
    args=sft_config,
    formatting_func=formatting_func
)

trainer.train()
```

You can also customize the `ConstantLengthDataset` much more by directly passing the arguments to the SFTConfig constructor. Please refer to that class' signature for more information.

Control over the pretrained model

You can directly pass the kwargs of the `from_pretrained()` method to the SFTConfig. For example, if you want to load a model in a different precision, analogous to

```
model = AutoModelForCausalLM.from_pretrained("facebook/opt-350m", torch_dtype=torch.bfloat16)

...

sft_config = SFTConfig(
    model_init_kwargs={
        "torch_dtype": "bfloat16",
    },
    output_dir="/tmp",
)
trainer = SFTTrainer(
    "facebook/opt-350m",
    train_dataset=dataset,
    args=sft_config,
)

trainer.train()
```

Note that all keyword arguments of `from_pretrained()` are supported.

Training adapters

We also support tight integration with 🤗 PEFT library so that any user can conveniently train adapters and share them on the Hub instead of training the entire model

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

dataset = load_dataset("stanfordnlp/imdb", split="train")

peft_config = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)

trainer = SFTTrainer(
    "EleutherAI/gpt-neo-125m",
    train_dataset=dataset,
    args=SFTConfig(output_dir="/tmp"),
    peft_config=peft_config
)

trainer.train()
```

You can also continue training your `PeftModel`. For that, first load a `PeftModel` outside `SFTTrainer` and pass it directly to the trainer without the `peft_config` argument being passed.

Training adapters with base 8 bit models

For that, you need to first load your 8 bit model outside the Trainer and pass a `PeftConfig` to the trainer. For example:

```
...

peft_config = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)

model = AutoModelForCausalLM.from_pretrained(
```

```
    "EleutherAI/gpt-neo-125m",
    load_in_8bit=True,
    device_map="auto",
)

trainer = SFTTrainer(
    model,
    train_dataset=dataset,
    args=SFTConfig(),
    peft_config=peft_config,
)

trainer.train()
```

Using Flash Attention and Flash Attention 2

You can benefit from Flash Attention 1 & 2 using SFTTrainer out of the box with minimal changes of code. First, to make sure you have all the latest features from transformers, install transformers from source

```
pip install -U git+https://github.com/huggingface/transformers.git
```

Note that Flash Attention only works on GPU now and under half-precision regime (when using adapters, base model loaded in half-precision) Note also both features are perfectly compatible with other tools such as quantization.

Using Flash-Attention 1

For Flash Attention 1 you can use the `BetterTransformer` API and force-dispatch the API to use Flash Attention kernel. First, install the latest optimum package:

```
pip install -U optimum
```

Once you have loaded your model, wrap the `trainer.train()` call under the `with torch.backends.cuda.sdp_kernel(enable_flash=True, enable_math=False, enable_mem_efficient=False):` context manager:

```
...

+ with torch.backends.cuda.sdp_kernel(enable_flash=True, enable_math=False, enable_mem_efficient
    trainer.train()
```

Note that you cannot train your model using Flash Attention 1 on an arbitrary dataset as `torch.scaled_dot_product_attention` does not support training with padding tokens if you use Flash Attention kernels. Therefore you can only use that feature with `packing=True`. If your dataset contains padding tokens, consider switching to Flash Attention 2 integration.

Below are some numbers you can get in terms of speedup and memory efficiency, using Flash Attention 1, on a single NVIDIA-T4 16GB.

use_flash_attn_1	model_name	max_seq_len	batch_size	time per training step
x	facebook/opt-350m	2048	8	~59.1s
	facebook/opt-350m	2048	8	**OOM**
x	facebook/opt-350m	2048	4	~30.3s
	facebook/opt-350m	2048	4	~148.9s

Using Flash Attention-2

To use Flash Attention 2, first install the latest `flash-attn` package:

```
pip install -U flash-attn
```

And add `attn_implementation="flash_attention_2"` when calling `from_pretrained`:

```
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    load_in_4bit=True,
    attn_implementation="flash_attention_2"
)
```

If you don't use quantization, make sure your model is loaded in half-precision and dispatch your model on a supported GPU device. After loading your model, you can either train it as it is, or attach adapters and train adapters on it in case your model is quantized.

In contrast to Flash Attention 1, the integration makes it possible to train your model on an arbitrary dataset that also includes padding tokens.

Using model creation utility

We included a utility function to create your model.

class trl.ModelConfig

<>

```
( model_name_or_path: Optional = None, model_revision: str = 'main', torch_dtype: Optional = None,
trust_remote_code: bool = False, attn_implementation: Optional = None, use_peft: bool = False,
lora_r: int = 16, lora_alpha: int = 32, lora_dropout: float = 0.05, lora_target_modules: Optional =
None, lora_modules_to_save: Optional = None, lora_task_type: str = 'CAUSAL_LM', use_rslora: bool =
False, load_in_8bit: bool = False, load_in_4bit: bool = False, bnb_4bit_quant_type: Literal = 'nf4',
use_bnb_nested_quant: bool = False )
```

Expand 17 parameters

Configuration class for the models.

Using HfArgumentParser we can turn this class into argparse arguments that can be specified on the command line.

```
from trl import ModelConfig, SFTTrainer, get_kbit_device_map, get_peft_config, get_quantization_
model_config = ModelConfig(
    model_name_or_path="facebook/opt-350m"
    attn_implementation=None, # or "flash_attention_2"
)
torch_dtype = (
    model_config.torch_dtype
    if model_config.torch_dtype in ["auto", None]
    else getattr(torch, model_config.torch_dtype)
)
quantization_config = get_quantization_config(model_config)
model_kwargs = dict(
    revision=model_config.model_revision,
    trust_remote_code=model_config.trust_remote_code,
    attn_implementation=model_config.attn_implementation,
    torch_dtype=torch_dtype,
    use_cache=False if training_args.gradient_checkpointing else True,
```

```
    device_map=get_kbit_device_map() if quantization_config is not None else None,
    quantization_config=quantization_config,
)
model = AutoModelForCausalLM.from_pretrained(model_config.model_name_or_path, **model_kwargs)
trainer = SFTTrainer(
    ...,
    model=model_config.model_name_or_path,
    peft_config=get_peft_config(model_config),
)
```

Enhance the model's performances using NEFTune

NEFTune is a technique to boost the performance of chat models and was introduced by the paper "NEFTune: Noisy Embeddings Improve Instruction Finetuning" from Jain et al. it consists of adding noise to the embedding vectors during training. According to the abstract of the paper:

> *"Standard finetuning of LLaMA-2-7B using Alpaca achieves 29.79% on AlpacaEval, which rises to 64.69% using noisy embeddings. NEFTune also improves over strong baselines on modern instruction datasets. Models trained with Evol-Instruct see a 10% improvement, with ShareGPT an 8% improvement, and with OpenPlatypus an 8% improvement. Even powerful models further refined with RLHF such as LLaMA-2-Chat benefit from additional training with NEFTune."*

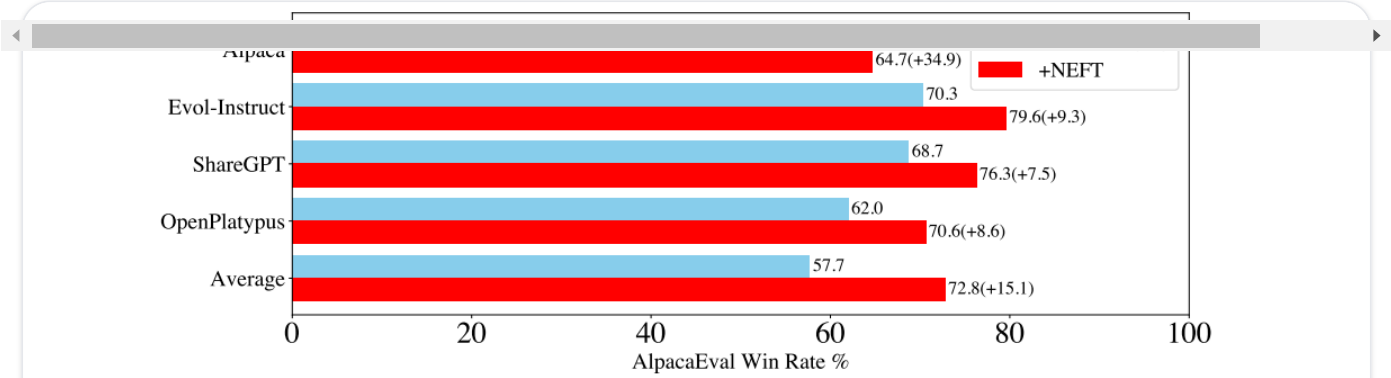


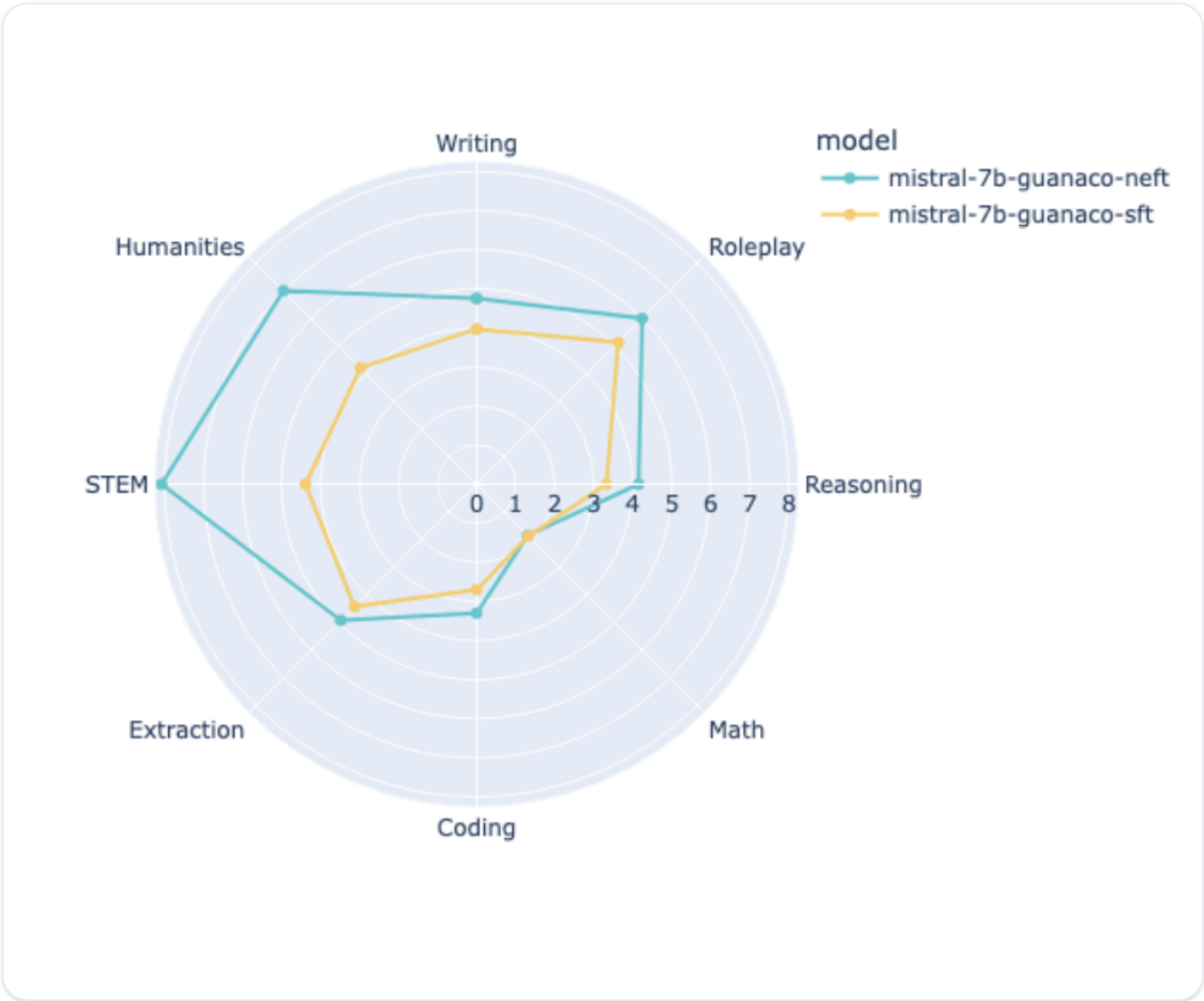
Figure 1: `AlpacaEval` Win Rate percentage for LLaMA-2-7B models finetuned on various datasets with and without `NEFTune`. `NEFTune` leads to massive performance boosts across all of these datasets, showcasing the increased conversational quality of the generated answers.

To use it in `SFTTrainer` simply pass `neftune_noise_alpha` when creating your `SFTConfig` instance. Note that to avoid any surprising behaviour, NEFTune is disabled after training to retrieve back the original behaviour of the embedding layer.

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

dataset = load_dataset("stanfordnlp/imdb", split="train")
```

```
sft_config = SFTConfig(
    neftune_noise_alpha=5,
)
trainer = SFTTrainer(
    "facebook/opt-350m",
    train_dataset=dataset,
    args=sft_config,
)
trainer.train()
```

We have tested NEFTune by training `mistralai/Mistral-7B-v0.1` on the OpenAssistant dataset and validated that using NEFTune led to a performance boost of ~25% on MT Bench.



Note however, that the amount of performance gain is *dataset dependent* and in particular, applying NEFTune on synthetic datasets like UltraChat typically produces smaller gains.

Accelerate fine-tuning 2x using unsloth

You can further accelerate QLoRA / LoRA (2x faster, 60% less memory) using the `unsloth` library that is fully compatible with `SFTTrainer`. Currently `unsloth` supports only Llama (Yi, TinyLlama, Qwen, Deepseek etc) and Mistral architectures. Some benchmarks on 1x A100 listed below:

1 A100 40GB	Dataset	🤗	🤗 + Flash Attention 2	🦥 Unsloth	🦥 VRAM saved
Code Llama 34b	Slim Orca	1x	1.01x	**1.94x**	-22.7%
Llama-2 7b	Slim Orca	1x	0.96x	**1.87x**	-39.3%
Mistral 7b	Slim Orca	1x	1.17x	**1.88x**	-65.9%
Tiny Llama 1.1b	Alpaca	1x	1.55x	**2.74x**	-57.8%

First install `unsloth` according to the official documentation. Once installed, you can incorporate unsloth into your workflow in a very simple manner; instead of loading `AutoModelForCausalLM`, you just need to load a `FastLanguageModel` as follows:

```
import torch
from trl import SFTConfig, SFTTrainer
from unsloth import FastLanguageModel

max_seq_length = 2048 # Supports automatic RoPE Scaling, so choose any number

# Load model
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name="unsloth/mistral-7b",
    max_seq_length=max_seq_length,
    dtype=None,  # None for auto detection. Float16 for Tesla T4, V100, Bfloat16 for Ampere+
    load_in_4bit=True,  # Use 4bit quantization to reduce memory usage. Can be False
    # token = "hf_...", # use one if using gated models like meta-llama/Llama-2-7b-hf
)

# Do model patching and add fast LoRA weights
model = FastLanguageModel.get_peft_model(
    model,
    r=16,
    target_modules=[
        "q_proj",
        "k_proj",
        "v_proj",
        "o_proj",
        "gate_proj",
        "up_proj",
        "down_proj",
    ],
    lora_alpha=16,
    lora_dropout=0,  # Dropout = 0 is currently optimized
```

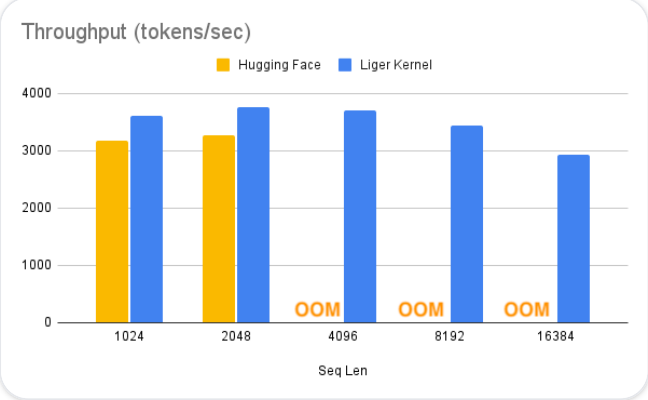
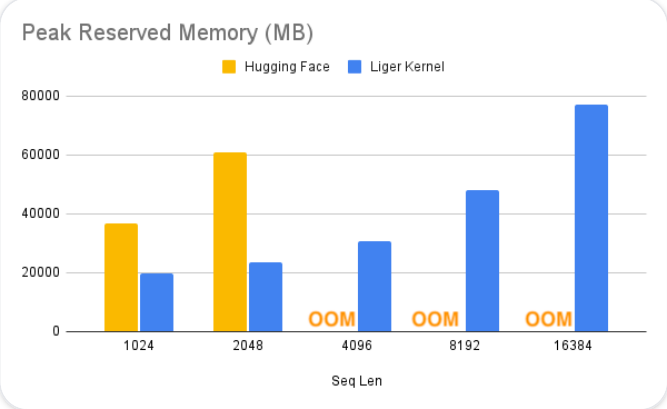
```
    bias="none",  # Bias = "none" is currently optimized
    use_gradient_checkpointing=True,
    random_state=3407,
)

args = SFTConfig(
    output_dir="./output",
    max_seq_length=max_seq_length,
    dataset_text_field="text",
)

trainer = SFTTrainer(
    model=model,
    args=args,
    train_dataset=dataset,
)
trainer.train()
```

The saved model is fully compatible with Hugging Face's transformers library. Learn more about unsloth in their official repository.

Liger-Kernel: Increase 20% throughput and reduces 60% memory for multi-GPU training

Liger Kernel is a collection of Triton kernels designed specifically for LLM training. It can effectively increase multi-GPU training throughput by 20% and reduces memory usage by 60%. That way, we can **4x** our context length, as described in the benchmark below. They have implemented Hugging Face Compatible `RMSNorm`, `RoPE`, `SwiGLU`, `CrossEntropy`, `FusedLinearCrossEntropy`, and more to come. The kernel works out of the box with Flash Attention, PyTorch FSDP, and Microsoft DeepSpeed.

With great memory reduction, you can potentially turn off cpu_offloading or gradient checkpointing to further boost the performance.

Speed Up	Memory Reduction
	

1. To use Liger-Kernel in `SFTTrainer`, first install by

```
pip install liger-kernel
```

2. Once installed, set `use_liger` in SFTConfig. No other changes are needed!

```
config = SFTConfig(
  use_liger=True
)
```

To learn more about Liger-Kernel, visit their official repository.

Best practices

Pay attention to the following best practices when training a model with that trainer:

- SFTTrainer always pads by default the sequences to the `max_seq_length` argument of the SFTTrainer. If none is passed, the trainer will retrieve that value from the tokenizer. Some tokenizers do not provide a default value, so there is a check to retrieve the minimum between 2048 and that value. Make sure to check it before training.
- For training adapters in 8bit, you might need to tweak the arguments of the `prepare_model_for_kbit_training` method from PEFT, hence we advise users to use `prepare_in_int8_kwargs` field, or create the `PeftModel` outside the SFTTrainer and pass it.
- For a more memory-efficient training using adapters, you can load the base model in 8bit, for that simply add `load_in_8bit` argument when creating the SFTTrainer, or create a base model in 8bit outside the trainer and pass it.
- If you create a model outside the trainer, make sure to not pass to the trainer any additional keyword arguments that are relative to `from_pretrained()` method.

Multi-GPU Training

Trainer (and thus SFTTrainer) supports multi-GPU training. If you run your script with `python script.py` it will default to using DP as the strategy, which may be slower than expected. To use DDP (which is generally recommended, see here for more info) you must launch the script with `python -m torch.distributed.launch script.py` or `accelerate launch script.py`. For DDP to work you must also check the following:

- If you're using gradient_checkpointing, add the following to the TrainingArguments: `gradient_checkpointing_kwargs={'use_reentrant':False}` (more info here

- Ensure that the model is placed on the correct device:

```
from accelerate import PartialState
device_string = PartialState().process_index
model = AutoModelForCausalLM.from_pretrained(
     ...
    device_map={'':device_string}
)
```

GPTQ Conversion

You may experience some issues with GPTQ Quantization after completing training. Lowering `gradient_accumulation_steps` to `4` will resolve most issues during the quantization process to GPTQ format.

Extending SFTTrainer for Vision Language Models

`SFTTrainer` does not inherently support vision-language data. However, we provide a guide on how to tweak the trainer to support vision-language data. Specifically, you need to use a custom data collator that is compatible with vision-language data. This guide outlines the steps to make these adjustments. For a concrete example, refer to the script `examples/scripts/vsft_llava.py` which demonstrates how to fine-tune the LLaVA 1.5 model on the HuggingFaceH4/llava-instruct-mix-vsft dataset.

Preparing the Data

The data format is flexible, provided it is compatible with the custom collator that we will define later. A common approach is to use conversational data. Given that the data includes both text and images, the format needs to be adjusted accordingly. Below is an example of a conversational data format involving both text and images:

```
images = ["obama.png"]
messages = [
    {
        "role": "user",
        "content": [
            {"type": "text", "text": "Who is this?"},
            {"type": "image"}
        ]
    },
    {
        "role": "assistant",
        "content": [
            {"type": "text", "text": "Barack Obama"}
```

```
        ]
    },
    {
        "role": "user",
        "content": [
            {"type": "text", "text": "What is he famous for?"}
        ]
    },
    {
        "role": "assistant",
        "content": [
            {"type": "text", "text": "He is the 44th President of the United States."}
        ]
    }
]
```

To illustrate how this data format will be processed using the LLaVA model, you can use the following code:

```
from transformers import AutoProcessor

processor = AutoProcessor.from_pretrained("llava-hf/llava-1.5-7b-hf")
print(processor.apply_chat_template(messages, tokenize=False))
```

The output will be formatted as follows:

```
Who is this? ASSISTANT: Barack Obama USER: What is he famous for? ASSISTANT: He is the 44th Pres
```

HuggingFaceH4 / **llava-instruct-mix-vsft**

Split (2)
train · 259k rows

Search this dataset | SQL Console

messages list · *lengths* 2 — 14	images images list · *lengths* 1 — 1
[{ "content": [{ "index": null, "text": "Who wrote this book?\n", "type": "text" }, { "index": 0, "text": null,…	
[{ "content": [{ "index": null, "text": "Who wrote this book?\n", "type": "text" }, { "index": 0, "text": null,…	
[{ "content": [{ "index": 0, "text": null, "type": "image" }, { "index": null, "text": "\nWhat potential activities migh…	
[{ "content": [{ "index": null, "text": "Who wrote this book?\n", "type": "text" }, { "index": 0, "text": null,…	
[{ "content": [{ "index": null, "text": "Who wrote this	

‹ Previous 1 2 3 ... 2,592 Next ›

A custom collator for processing multi-modal data

Unlike the default behavior of `SFTTrainer`, processing multi-modal data is done on the fly during the data collation process. To do this, you need to define a custom collator that processes both the text and images. This collator must take a list of examples as input (see the previous section for an example of the data format) and return a batch of processed data. Below is an example of such a collator:

```
def collate_fn(examples):
    # Get the texts and images, and apply the chat template
    texts = [processor.apply_chat_template(example["messages"], tokenize=False) for example in e
    images = [example["images"][0] for example in examples]

    # Tokenize the texts and process the images
    batch = processor(texts, images, return_tensors="pt", padding=True)

    # The labels are the input_ids, and we mask the padding tokens in the loss computation
    labels = batch["input_ids"].clone()
    labels[labels == processor.tokenizer.pad_token_id] = -100
    batch["labels"] = labels

    return batch
```

We can verify that the collator works as expected by running the following code:

```
from datasets import load_dataset

dataset = load_dataset("HuggingFaceH4/llava-instruct-mix-vsft", split="train")
examples = [dataset[0], dataset[1]]  # Just two examples for the sake of the example
collated_data = collate_fn(examples)
print(collated_data.keys())  # dict_keys(['input_ids', 'attention_mask', 'pixel_values', 'labels
```

Training the vision-language model

Now that we have prepared the data and defined the collator, we can proceed with training the model. To ensure that the data is not processed as text-only, we need to set a couple of arguments in the `SFTConfig`, specifically `dataset_text_field` and `remove_unused_columns`. We also need to set `skip_prepare_dataset` to `True` to avoid the default processing of the dataset. Below is an example of how to set up the `SFTTrainer`.

```
args.dataset_text_field = ""  # needs a dummy field
args.remove_unused_columns = False
args.dataset_kwargs = {"skip_prepare_dataset": True}

trainer = SFTTrainer(
    model=model,
    args=args,
    data_collator=collate_fn,
    train_dataset=train_dataset,
    tokenizer=processor.tokenizer,
)
```

A full example of training LLaVa 1.5 on the HuggingFaceH4/llava-instruct-mix-vsft dataset can be found in the script `examples/scripts/vsft_llava.py`.

- Experiment tracking
- Trained model

SFTTrainer

class trl.**SFTTrainer** <>

```
( model: Union = None, args: Optional = None, data_collator: Optional = None, train_dataset: Optional
= None, eval_dataset: Union = None, tokenizer: Optional = None, model_init: Optional = None,
compute_metrics: Optional = None, callbacks: Optional = None, optimizers: Tuple = (None, None),
preprocess_logits_for_metrics: Optional = None, peft_config: Optional = None, dataset_text_field:
```