**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 inside [examples/scripts](https://github.com/huggingface/trl/tree/main/examples/scripts/sft.py) folder.

**Quickstart**

If you have a dataset hosted on the 🤗 Hub, you can easily fine-tune your SFT model using [SFTTrainer](https://huggingface.co/docs/trl/v0.7.4/en/sft_trainer#trl.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:

Copied

from datasets import load\_dataset

from trl import SFTTrainer

dataset = load\_dataset("imdb", split="train")

trainer = SFTTrainer(

"facebook/opt-350m",

train\_dataset=dataset,

dataset\_text\_field="text",

max\_seq\_length=512,

)

trainer.train()

Make sure to pass a 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:

Copied

from transformers import AutoModelForCausalLM

from datasets import load\_dataset

from trl import SFTTrainer

dataset = load\_dataset("imdb", split="train")

model = AutoModelForCausalLM.from\_pretrained("facebook/opt-350m")

trainer = SFTTrainer(

model,

train\_dataset=dataset,

dataset\_text\_field="text",

max\_seq\_length=512,

)

trainer.train()

The above snippets will use the default training arguments from the [transformers.TrainingArguments](https://huggingface.co/docs/transformers/main_classes/trainer#transformers.TrainingArguments) class. If you want to modify that, make sure to create your own TrainingArguments object and pass it to the [SFTTrainer](https://huggingface.co/docs/trl/v0.7.4/en/sft_trainer#trl.SFTTrainer) constructor as it is done on the [supervised\_finetuning.py script](https://github.com/huggingface/trl/blob/main/examples/stack_llama/scripts/supervised_finetuning.py) on the stack-llama example.

**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:

Copied

from transformers import AutoModelForCausalLM, AutoTokenizer

from datasets import load\_dataset

from trl import 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,

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:

Copied

from transformers import AutoModelForCausalLM, AutoTokenizer

from datasets import load\_dataset

from trl import 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\_template=response\_template, tokenizer=tokenizer, mlm=False)

trainer = SFTTrainer(

model,

train\_dataset=dataset,

dataset\_text\_field="text",

data\_collator=collator,

)

trainer.train()

**Using token\_ids directly for response\_template**

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

Copied

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), ('##', 2277), ('#', 29937), ('▁Ass', 4007), ('istant', 22137), (':', 29901), ...]*

response\_template = "### Assistant:"

print\_tokens\_with\_ids(response\_template) *# [('▁###', 835), ('▁Ass', 4007), ('istant', 22137), (':', 29901)]*

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:

Copied

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

To solve this, you can tokenize the response\_template with the same context than in the dataset, truncate it as needed and pass the token\_ids directly to the response\_template argument of the DataCollatorForCompletionOnlyLM class. For example:

Copied

response\_template\_with\_context = "\n### Assistant:" *# We added context here: "\n". This is enough for this tokenizer*

response\_template\_ids = tokenizer.encode(response\_template\_with\_context, add\_special\_tokens=False)[2:] *# Now we have it like in the dataset texts: `[2277, 29937, 4007, 22137, 29901]`*

data\_collator = DataCollatorForCompletionOnlyLM(response\_template\_ids, tokenizer=tokenizer)

**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](https://github.com/tatsu-lab/stanford_alpaca) did as follows:

Copied

Below is an instruction ...

*### Instruction*

{prompt}

*### Response:*

{completion}

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

Copied

...

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,

train\_dataset=dataset,

formatting\_func=formatting\_prompts\_func,

)

trainer.train()

To preperly 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 on how to use SFTTrainer on alpaca dataset [here](https://github.com/huggingface/trl/pull/444#issue-1760952763)

**Packing dataset ( ConstantLengthDataset )**

[SFTTrainer](https://huggingface.co/docs/trl/v0.7.4/en/sft_trainer#trl.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 [SFTTrainer](https://huggingface.co/docs/trl/v0.7.4/en/sft_trainer#trl.SFTTrainer) constructor.

Copied

...

trainer = SFTTrainer(

"facebook/opt-350m",

train\_dataset=dataset,

dataset\_text\_field="text",

packing=True

)

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.

**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:

Copied

def formatting\_func(example):

text = f"### Question: {example['question']}\n ### Answer: {example['answer']}"

return text

trainer = SFTTrainer(

"facebook/opt-350m",

train\_dataset=dataset,

packing=True,

formatting\_func=formatting\_func

)

trainer.train()

You can also customize the ConstantLengthDataset much more by directly passing the arguments to the [SFTTrainer](https://huggingface.co/docs/trl/v0.7.4/en/sft_trainer#trl.SFTTrainer) 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 [SFTTrainer](https://huggingface.co/docs/trl/v0.7.4/en/sft_trainer#trl.SFTTrainer). For example, if you want to load a model in a different precision, analogous to

Copied

model = AutoModelForCausalLM.from\_pretrained("facebook/opt-350m", torch\_dtype=torch.bfloat16)

```python

...

trainer = SFTTrainer(

"facebook/opt-350m",

train\_dataset=dataset,

dataset\_text\_field="text",

)

trainer.train()

Note that all keyword arguments of from\_pretrained() are supported.

**Training adapters**

We also support a 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

Copied

from datasets import load\_dataset

from trl import SFTTrainer

from peft import LoraConfig

dataset = load\_dataset("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,

dataset\_text\_field="text",

peft\_config=peft\_config

)

trainer.train()

Note that in case of training adapters, we manually add a saving callback to automatically save the adapters only:

Copied

class PeftSavingCallback(TrainerCallback):

def on\_save(self, args, state, control, \*\*kwargs):

checkpoint\_path = os.path.join(args.output\_dir, f"checkpoint-{state.global\_step}")

kwargs["model"].save\_pretrained(checkpoint\_path)

if "pytorch\_model.bin" in os.listdir(checkpoint\_path):

os.remove(os.path.join(checkpoint\_path, "pytorch\_model.bin"))

If you want to add more callbacks, make sure to add this one as well to properly save the adapters only during training.

Copied

...

callbacks = [YourCustomCallback(), PeftSavingCallback()]

trainer = SFTTrainer(

"EleutherAI/gpt-neo-125m",

train\_dataset=dataset,

dataset\_text\_field="text",

peft\_config=peft\_config,

callbacks=callbacks

)

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 8bit model outside the Trainer and pass a PeftConfig to the trainer. For example:

Copied

...

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,

dataset\_text\_field="text",

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

Copied

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:

Copied

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:

Copied

...

+ with torch.backends.cuda.sdp\_kernel(enable\_flash=True, enable\_math=False, enable\_mem\_efficient=False):

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:

Copied

pip install -U flash-attn

And add use\_flash\_attention\_2=True when calling from\_pretrained:

Copied

model = AutoModelForCausalLM.from\_pretrained(

model\_id,

load\_in\_4bit=True,

use\_flash\_attention\_2=True

)

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

**Enhance 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”](https://arxiv.org/abs/2310.05914) 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.*

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

Copied

from datasets import load\_dataset

from trl import SFTTrainer

dataset = load\_dataset("imdb", split="train")

trainer = SFTTrainer(

"facebook/opt-350m",

train\_dataset=dataset,

dataset\_text\_field="text",

max\_seq\_length=512,

neftune\_noise\_alpha=5,

)

trainer.train()

We have tested NEFTune by training mistralai/Mistral-7B-v0.1 on the [OpenAssistant dataset](https://huggingface.co/datasets/timdettmers/openassistant-guanaco) 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](https://huggingface.co/datasets/stingning/ultrachat) typically produces smaller gains.

**Best practices**

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

* [SFTTrainer](https://huggingface.co/docs/trl/v0.7.4/en/sft_trainer#trl.SFTTrainer) always pads by default the sequences to the max\_seq\_length argument of the [SFTTrainer](https://huggingface.co/docs/trl/v0.7.4/en/sft_trainer#trl.SFTTrainer). If none is passed, the trainer will retrieve that value from the tokenizer. Some tokenizers do not provide 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](https://huggingface.co/docs/trl/v0.7.4/en/sft_trainer#trl.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](https://huggingface.co/docs/trl/v0.7.4/en/sft_trainer#trl.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.

**SFTTrainer**

classtrl.**SFTTrainer**

[<source>](https://github.com/huggingface/trl/blob/v0.7.4/trl/trainer/sft_trainer.py#L50)

( model: typing.Union[transformers.modeling\_utils.PreTrainedModel, torch.nn.modules.module.Module, str] = Noneargs: TrainingArguments = Nonedata\_collator: typing.Optional[DataCollator] = Nonetrain\_dataset: typing.Optional[datasets.arrow\_dataset.Dataset] = Noneeval\_dataset: typing.Union[datasets.arrow\_dataset.Dataset, typing.Dict[str, datasets.arrow\_dataset.Dataset], NoneType] = Nonetokenizer: typing.Optional[transformers.tokenization\_utils\_base.PreTrainedTokenizerBase] = Nonemodel\_init: typing.Union[typing.Callable[[], transformers.modeling\_utils.PreTrainedModel], NoneType] = Nonecompute\_metrics: typing.Union[typing.Callable[[transformers.trainer\_utils.EvalPrediction], typing.Dict], NoneType] = Nonecallbacks: typing.Optional[typing.List[transformers.trainer\_callback.TrainerCallback]] = Noneoptimizers: typing.Tuple[torch.optim.optimizer.Optimizer, torch.optim.lr\_scheduler.LambdaLR] = (None, None)preprocess\_logits\_for\_metrics: typing.Union[typing.Callable[[torch.Tensor, torch.Tensor], torch.Tensor], NoneType] = Nonepeft\_config: typing.Optional[ForwardRef('PeftConfig')] = Nonedataset\_text\_field: typing.Optional[str] = Nonepacking: typing.Optional[bool] = Falseformatting\_func: typing.Optional[typing.Callable] = Nonemax\_seq\_length: typing.Optional[int] = Noneinfinite: typing.Optional[bool] = Falsenum\_of\_sequences: typing.Optional[int] = 1024chars\_per\_token: typing.Optional[float] = 3.6dataset\_num\_proc: typing.Optional[int] = Nonedataset\_batch\_size: int = 1000neftune\_noise\_alpha: typing.Optional[float] = Nonemodel\_init\_kwargs: typing.Optional[typing.Dict] = None )

Expand 22 parameters

**Parameters**

* **model** (Union[transformers.PreTrainedModel, nn.Module, str]) — The model to train, can be a PreTrainedModel, a torch.nn.Module or a string with the model name to load from cache or download. The model can be also converted to a PeftModel if a PeftConfig object is passed to the peft\_config argument.
* **args** (Optional[transformers.TrainingArguments](https://huggingface.co/docs/transformers/v4.35.0/en/main_classes/trainer#transformers.TrainingArguments)) — The arguments to tweak for training. Please refer to the official documentation of transformers.TrainingArguments for more information.
* **data\_collator** (Optionaltransformers.DataCollator) — The data collator to use for training.
* **train\_dataset** (Optionaldatasets.Dataset) — The dataset to use for training. We recommend users to use trl.trainer.ConstantLengthDataset to create their dataset.
* **eval\_dataset** (Optional[Union[datasets.Dataset, Dict[str, datasets.Dataset]]]) — The dataset to use for evaluation. We recommend users to use trl.trainer.ConstantLengthDataset to create their dataset.
* **tokenizer** (Optional[transformers.PreTrainedTokenizer](https://huggingface.co/docs/transformers/v4.35.0/en/main_classes/tokenizer#transformers.PreTrainedTokenizer)) — The tokenizer to use for training. If not specified, the tokenizer associated to the model will be used.
* **model\_init** (Callable[[], transformers.PreTrainedModel]) — The model initializer to use for training. If None is specified, the default model initializer will be used.
* **compute\_metrics** (Callable[[transformers.EvalPrediction], Dict], *optional* defaults to compute\_accuracy) — The metrics to use for evaluation. If no metrics are specified, the default metric (compute\_accuracy) will be used.
* **callbacks** (List[transformers.TrainerCallback]) — The callbacks to use for training.
* **optimizers** (Tuple[torch.optim.Optimizer, torch.optim.lr\_scheduler.LambdaLR]) — The optimizer and scheduler to use for training.
* **preprocess\_logits\_for\_metrics** (Callable[[torch.Tensor, torch.Tensor], torch.Tensor]) — The function to use to preprocess the logits before computing the metrics.
* **peft\_config** (Optional[PeftConfig]) — The PeftConfig object to use to initialize the PeftModel.
* **dataset\_text\_field** (Optional[str]) — The name of the text field of the dataset, in case this is passed by a user, the trainer will automatically create a ConstantLengthDataset based on the dataset\_text\_field argument.
* **formatting\_func** (Optional[Callable]) — The formatting function to be used for creating the ConstantLengthDataset.
* **max\_seq\_length** (Optional[int]) — The maximum sequence length to use for the ConstantLengthDataset and for automaticallty creating the Dataset. Defaults to 512.
* **infinite** (Optional[bool]) — Whether to use an infinite dataset or not. Defaults to False.
* **num\_of\_sequences** (Optional[int]) — The number of sequences to use for the ConstantLengthDataset. Defaults to 1024.
* **chars\_per\_token** (Optional[float]) — The number of characters per token to use for the ConstantLengthDataset. Defaults to 3.6. You can check how this is computed in the stack-llama example: <https://github.com/huggingface/trl/blob/08f550674c553c36c51d1027613c29f14f3676a5/examples/stack_llama/scripts/supervised_finetuning.py#L53>.
* **packing** (Optional[bool]) — Used only in case dataset\_text\_field is passed. This argument is used by the ConstantLengthDataset to pack the sequences of the dataset.
* **dataset\_num\_proc** (Optional[int]) — The number of workers to use to tokenize the data. Only used when packing=False. Defaults to None.
* **dataset\_batch\_size** (int) — The number of examples to tokenize per batch. If batch\_size <= 0 or batch\_size == None, tokenize the full dataset as a single batch. Defaults to 1000.
* **neftune\_noise\_alpha** (Optional[float]) — If not None, this will activate NEFTune noise embeddings. This has been proven to drastically improve model performances for instrcution fine-tuning. Check out the original paper here: <https://arxiv.org/abs/2310.05914> and the original code here: <https://github.com/neelsjain/NEFTune> model\_init\_kwargs — (Optional[Dict], *optional*): Dict of Optional kwargs to pass when instantiating the model from a string

Class definition of the Supervised Finetuning Trainer (SFT Trainer). This class is a wrapper around the transformers.Trainer class and inherits all of its attributes and methods. The trainer takes care of properly initializing the PeftModel in case a user passes a PeftConfig object.

**ConstantLengthDataset**

classtrl.trainer.**ConstantLengthDataset**

[<source>](https://github.com/huggingface/trl/blob/v0.7.4/trl/trainer/utils.py#L482)

( \*args\*\*kwds )

**Parameters**

* **tokenizer** (transformers.PreTrainedTokenizer) — The processor used for processing the data.
* **dataset** (dataset.Dataset) — Dataset with text files.
* **dataset\_text\_field** (str, **optional**) — Name of the field in the dataset that contains the text. Used only if formatting\_func is None.
* **formatting\_func** (Callable, **optional**) — Function that formats the text before tokenization. Usually it is recommended to have follows a certain pattern such as `”### Question: {question}

Iterable dataset that returns constant length chunks of tokens from stream of text files. The dataset also formats the text before tokenization with a specific format that is provided by the user.

**Answer: {answer}**

”infinite (bool, \*optional\*, defaults to False): If True the iterator is reset after dataset reaches end else stops. seq\_length (int, \*optional\*, defaults to 1024): Length of token sequences to return. num\_of\_sequences (int, \*optional\*, defaults to 1024): Number of token sequences to keep in buffer. chars\_per\_token (int, \*optional\*, defaults to 3.6): Number of characters per token used to estimate number of tokens in text buffer. eos\_token\_id (int, \*optional\*, defaults to 0`): Id of the end of sequence token if the passed tokenizer does not have an EOS token. shuffle (‘bool’, *optional*, defaults to True) Shuffle the examples before they are returned