[![Logo](../../../_static/logo.png)](../../../index.html)

Getting Started

Sentence Transformer

Package Reference

  * [CachedGISTEmbedLoss](losses.html#cachedgistembedloss)
  * [MSELoss](losses.html#mseloss)
  * [MarginMSELoss](losses.html#marginmseloss)
  * [MatryoshkaLoss](losses.html#matryoshkaloss)
  * [Matryoshka2dLoss](losses.html#matryoshka2dloss)
  * [AdaptiveLayerLoss](losses.html#adaptivelayerloss)
  * [MegaBatchMarginLoss](losses.html#megabatchmarginloss)
  * [MultipleNegativesRankingLoss](losses.html#multiplenegativesrankingloss)
  * [CachedMultipleNegativesRankingLoss](losses.html#cachedmultiplenegativesrankingloss)
  * [MultipleNegativesSymmetricRankingLoss](losses.html#multiplenegativessymmetricrankingloss)
  * [CachedMultipleNegativesSymmetricRankingLoss](losses.html#cachedmultiplenegativessymmetricrankingloss)
  * [SoftmaxLoss](losses.html#softmaxloss)
  * [TripletLoss](losses.html#tripletloss)
* [Samplers](sampler.html)
  * [BatchSamplers](sampler.html#batchsamplers)
  * [MultiDatasetBatchSamplers](sampler.html#multidatasetbatchsamplers)
* [Evaluation](evaluation.html)
  * [BinaryClassificationEvaluator](evaluation.html#binaryclassificationevaluator)
  * [EmbeddingSimilarityEvaluator](evaluation.html#embeddingsimilarityevaluator)
  * [InformationRetrievalEvaluator](evaluation.html#informationretrievalevaluator)
  * [NanoBEIREvaluator](evaluation.html#nanobeirevaluator)
  * [MSEEvaluator](evaluation.html#mseevaluator)
  * [ParaphraseMiningEvaluator](evaluation.html#paraphraseminingevaluator)
  * [RerankingEvaluator](evaluation.html#rerankingevaluator)

atic_quantized_openvino_model)

    * [Similarity Metrics](../util.html#module-sentence_transformers.util)

      * [`cos_sim()`](../util.html#sentence_transformers.util.cos_sim)

      * [`dot_score()`](../util.html#sentence_transformers.util.dot_score)

      * [`euclidean_sim()`](../util.html#sentence_transformers.util.euclidean_sim)

      * [`manhattan_sim()`](../util.html#sentence_transformers.util.manhattan_sim)

      * [`pairwise_cos_sim()`](../util.html#sentence_transformers.util.pairwise_cos_sim)

      * [`pairwise_dot_score()`](../util.html#sentence_transformers.util.pairwise_dot_score)

      * [`pairwise_euclidean_sim()`](../util.html#sentence_transformers.util.pairwise_euclidean_sim)

      * [`pairwise_manhattan_sim()`](../util.html#sentence_transformers.util.pairwise_manhattan_sim)

__[Sentence Transformers](../../../index.html)

  * [](../../../index.html)

  * [Sentence Transformer](index.html)

  * Training Arguments

  * [Edit on GitHub](https://github.com/UKPLab/sentence-transformers/blob/master/docs/package_reference/sentence_transformer/training_args.md)

* * *

# Training Argumentsïƒ•

## SentenceTransformerTrainingArgumentsïƒ•

_class _sentence_transformers.training_args.SentenceTransformerTrainingArguments(_output_dir:

str_, _overwrite_output_dir: bool = False_, _do_train: bool = False_, _do_eval: bool = False_, _do_predict: bool = False_, _eval_strategy: ~transformers.trainer_utils.IntervalStrategy | str = 'no'_, _prediction_loss_only: bool = False_, _per_device_train_batch_size: int = 8_, _per_device_eval_batch_size: int = 8_, _per_gpu_train_batch_size: int | None = None_, _per_gpu_eval_batch_size: int | None = None_, _gradient_accumulation_steps: int = 1_, _eval_accumulation_steps: int | None = None_, _eval_delay: float | None = 0_, _torch_empty_cache_steps: int | None = None_, _learning_rate: float = 5e-05_, _weight_decay: float = 0.0_, _adam_beta1: float = 0.9_, _adam_beta2: float = 0.999_, _adam_epsilon: float = 1e-08_, _max_grad_norm: float = 1.0_, _num_train_epochs: float = 3.0_, _max_steps: int = -1_, _lr_scheduler_type: ~transformers.trainer_utils.SchedulerType | str = 'linear'_, _lr_scheduler_kwargs: dict | str | None = <factory>_, _warmup_ratio: float = 0.0_, _warmup_steps: int = 0_, _log_level: str | None = 'passive'_, _log_level_replica: str | None = 'warning'_, _log_on_each_node: bool = True_, _logging_dir: str | None = None_, _logging_strategy: ~transformers.trainer_utils.IntervalStrategy | str = 'steps'_, _logging_first_step: bool = False_, _logging_steps: float = 500_, _logging_nan_inf_filter: bool = True_, _save_strategy: ~transformers.trainer_utils.IntervalStrategy | str = 'steps'_, _save_steps: float = 500_, _save_total_limit: int | None = None_, _save_safetensors: bool | None = True_, _save_on_each_node: bool = False_, _save_only_model: bool = False_, _restore_callback_states_from_checkpoint: bool = False_, _no_cuda: bool = False_, _use_cpu: bool = False_, _use_mps_device: bool = False_, _seed: int = 42_, _data_seed: int | None = None_, _jit_mode_eval: bool = False_, _use_ipex: bool = False_, _bf16: bool = False_, _fp16: bool = False_, _fp16_opt_level: str = 'O1'_, _half_precision_backend: str = 'auto'_, _bf16_full_eval: bool = False_, _fp16_full_eval: bool = False_, _tf32: bool | None = None_, _local_rank: int = -1_, _ddp_backend: str | None = None_, _tpu_num_cores: int | None = None_, _tpu_metrics_debug: bool = False_, _debug: str | ~typing.List[~transformers.debug_utils.DebugOption] = ''_, _dataloader_drop_last: bool = False_, _eval_steps: float | None = None_, _dataloader_num_workers: int = 0_, _dataloader_prefetch_factor: int | None = None_, _past_index:

int = -1_, _run_name: str | None = None_, _disable_tqdm: bool | None = None_, _remove_unused_columns: bool | None = True_, _label_names: ~typing.List[str] | None = None_, _load_best_model_at_end: bool | None = False_, _metric_for_best_model: str | None = None_, _greater_is_better: bool | None = None_, _ignore_data_skip: bool = False_, _fsdp: ~typing.List[~transformers.trainer_utils.FSDPOption] | str | None = ''_, _fsdp_min_num_params: int = 0_, _fsdp_config: dict | str | None = None_, _fsdp_transformer_layer_cls_to_wrap: str | None = None_, _accelerator_config: dict | str | None = None_, _deepspeed: dict | str | None = None_, _label_smoothing_factor: float = 0.0_, _optim: ~transformers.training_args.OptimizerNames | str = 'adamw_torch'_, _optim_args: str | None = None_, _adafactor: bool = False_, _group_by_length: bool = False_, _length_column_name: str | None = 'length'_, _report_to: None | str | ~typing.List[str] = None_, _ddp_find_unused_parameters: bool | None = None_, _ddp_bucket_cap_mb: int | None = None_, _ddp_broadcast_buffers: bool | None = None_, _dataloader_pin_memory: bool = True_, _dataloader_persistent_workers: bool = False_, _skip_memory_metrics: bool = True_, _use_legacy_prediction_loop: bool = False_, _push_to_hub: bool = False_, _resume_from_checkpoint: str | None = None_, _hub_model_id: str | None = None_, _hub_strategy: ~transformers.trainer_utils.HubStrategy | str = 'every_save'_, _hub_token: str | None = None_, _hub_private_repo: bool = False_, _hub_always_push: bool = False_, _gradient_checkpointing: bool = False_, _gradient_checkpointing_kwargs: dict | str | None = None_, _include_inputs_for_metrics: bool = False_, _eval_do_concat_batches: bool = True_, _fp16_backend: str = 'auto'_, _evaluation_strategy: ~transformers.trainer_utils.IntervalStrategy | str | None = None_, _push_to_hub_model_id: str | None = None_, _push_to_hub_organization: str | None = None_, _push_to_hub_token: str | None = None_, _mp_parameters: str = ''_, _auto_find_batch_size: bool = False_, _full_determinism: bool = False_, _torchdynamo: str | None = None_, _ray_scope: str | None = 'last'_, _ddp_timeout: int | None = 1800_, _torch_compile: bool = False_, _torch_compile_backend: str | None = None_, _torch_compile_mode: str | None = None_, _dispatch_batches: bool | None = None_, _split_batches: bool | None = None_, _include_tokens_per_second: bool | None = False_, _include_num_input_tokens_seen: bool | None

= False_, _neftune_noise_alpha: float | None = None_, _optim_target_modules: None | str | ~typing.List[str] = None_, _batch_eval_metrics: bool = False_, _eval_on_start: bool = False_, _eval_use_gather_object: bool | None = False_, _prompts: str | None = None_, _batch_sampler: ~sentence_transformers.training_args.BatchSamplers | str = BatchSamplers.BATCH_SAMPLER_, _multi_dataset_batch_sampler: ~sentence_transformers.training_args.MultiDatasetBatchSamplers | str = MultiDatasetBatchSamplers.PROPORTIONAL_)[[source]](https://github.com/UKPLab/sentence-transformers/blob/master/sentence_transformers\\training_args.py#L143-L220)ïƒ•

SentenceTransformerTrainingArguments extends
[`TrainingArguments`](https://huggingface.co/docs/transformers/main/en/main_classes/trainer#transformers.TrainingArguments
"\(in transformers vmain\)") with additional arguments specific to Sentence
Transformers. See
[`TrainingArguments`](https://huggingface.co/docs/transformers/main/en/main_classes/trainer#transformers.TrainingArguments
"\(in transformers vmain\)") for the complete list of available arguments.

Parameters:

* **output_dir** (str) â€“ The output directory where the model checkpoints will be written.

* **prompts** (Union[Dict[str, Dict[str, str]], Dict[str, str], str, _optional_) â€“

The prompts to use for each column in the training, evaluation and test
datasets. Four formats are accepted:

1. str: A single prompt to use for all columns in the datasets, regardless of whether the training/evaluation/test datasets are [`datasets.Dataset`](https://huggingface.co/docs/datasets/main/en/package_reference/main_classes#datasets.Dataset "\(in datasets vmain\)") or a [`datasets.DatasetDict`](https://huggingface.co/docs/datasets/main/en/package_reference/main_classes#datasets.DatasetDict "\(in datasets vmain\)").

2. Dict[str, str]: A dictionary mapping column names to prompts, regardless of whether the training/evaluation/test datasets are [`datasets.Dataset`](https://huggingface.co/docs/datasets/main/en/package_reference/main_classes#datasets.Dataset "\(in datasets vmain\)") or a [`datasets.DatasetDict`](https://huggingface.co/docs/datasets/main/en/package_reference/main_classes#datasets.DatasetDict "\(in datasets vmain\)").

3. Dict[str, str]: A dictionary mapping dataset names to prompts. This should only be used if your training/evaluation/test datasets are a [`datasets.DatasetDict`](https://huggingface.co/docs/datasets/main/en/package_reference/main_classes#datasets.DatasetDict "\(in datasets vmain\)") or a dictionary of [`datasets.Dataset`](https://huggingface.co/docs/datasets/main/en/package_reference/main_classes#datasets.Dataset "\(in datasets vmain\)").

4. Dict[str, Dict[str, str]]: A dictionary mapping dataset names to dictionaries mapping column names to prompts. This should only be used if your training/evaluation/test datasets are a [`datasets.DatasetDict`](https://huggingface.co/docs/datasets/main/en/package_reference/main_cla

sses#datasets.DatasetDict "\(in datasets vmain\)") or a dictionary of [`datasets.Dataset`](https://huggingface.co/docs/datasets/main/en/package_reference/main_classes#datasets.Dataset "\(in datasets vmain\)").

* **batch_sampler** (Union[[`BatchSamplers`](sampler.html#sentence_transformers.training_args.BatchSamplers "sentence_transformers.training_args.BatchSamplers"), str], _optional_) â€“ The batch sampler to use. See [`BatchSamplers`](sampler.html#sentence_transformers.training_args.BatchSamplers "sentence_transformers.training_args.BatchSamplers") for valid options. Defaults to `BatchSamplers.BATCH_SAMPLER`.

* **multi_dataset_batch_sampler** (Union[[`MultiDatasetBatchSamplers`](sampler.html#sentence_transformers.training_args.MultiDatasetBatchSamplers "sentence_transformers.training_args.MultiDatasetBatchSamplers"), str], _optional_) â€“ The multi-dataset batch sampler to use. See [`MultiDatasetBatchSamplers`](sampler.html#sentence_transformers.training_args.MultiDatasetBatchSamplers "sentence_transformers.training_args.MultiDatasetBatchSamplers") for valid options. Defaults to `MultiDatasetBatchSamplers.PROPORTIONAL`.

_property _ddp_timeout_delta _: timedelta_ïƒ•

The actual timeout for torch.distributed.init_process_group since it expects a

timedelta variable.

_property _device _:

[device](https://pytorch.org/docs/stable/tensor_attributes.html#torch.device "\(in PyTorch v2.5\)")_ïƒ•

The device used by this process.

_property _eval_batch_size _: int_ïƒ•

The actual batch size for evaluation (may differ from per_gpu_eval_batch_size in distributed training).

get_process_log_level()ïƒ•

Returns the log level to be used depending on whether this process is the main process of node 0, main process of node non-0, or a non-main process.

For the main process the log level defaults to the logging level set (logging.WARNING if you didnâ€™t do anything) unless overridden by log_level argument.

For the replica processes the log level defaults to logging.WARNING unless overridden by log_level_replica argument.

The choice between the main and replica process settings is made according to the return value of should_log.

get_warmup_steps(_num_training_steps : int_)ïƒ•

Get number of steps used for a linear warmup.

_property _local_process_indexïƒ•

The index of the local process used.

main_process_first(_local =True_, _desc ='work'_)ïƒ•

A context manager for torch distributed environment where on needs to do something on the main process, while blocking replicas, and when itâ€™s finished releasing the replicas.

One such use is for datasetsâ€™s map feature which to be efficient should be run once on the main process, which upon completion saves a cached version of results and which then automatically gets loaded by the replicas.

Parameters:

* **local** (bool, _optional_ , defaults to True) â€“ if True first means process of rank 0 of each node if False first means process of rank 0 of node rank 0 In multi-node environment with a shared filesystem you most likely will want to use local=False so that only the main process of the first node will do the processing. If however, the filesystem is not shared, then the main process of each node will need to do the processing, which is the default behavior.

* **desc** (str, _optional_ , defaults to â€œworkâ€•) â€“ a work description to be used in debug logs

_property _n_gpuïƒ•

The number of GPUs used by this process.

Note

This will only be greater than one when you have multiple GPUs available but
are not using distributed training. For distributed training, it will always
be 1.

_property _parallel_modeïƒ•

The current mode used for parallelism if multiple GPUs/TPU cores are

available. One of:

* ParallelMode.NOT_PARALLEL: no parallelism (CPU or one GPU).

* ParallelMode.NOT_DISTRIBUTED: several GPUs in one single process (uses torch.nn.DataParallel).

* ParallelMode.DISTRIBUTED: several GPUs, each having its own process (uses torch.nn.DistributedDataParallel).

* ParallelMode.TPU: several TPU cores.

_property _place_model_on_deviceïƒ•

Can be subclassed and overridden for some specific integrations.

_property _process_indexïƒ•

The index of the current process used.

set_dataloader(_train_batch_size : int = 8_, _eval_batch_size : int = 8_, _drop_last : bool = False_, _num_workers : int = 0_, _pin_memory : bool = True_, _persistent_workers : bool = False_, _prefetch_factor : int | None = None_, _auto_find_batch_size : bool = False_, _ignore_data_skip : bool = False_, _sampler_seed : int | None = None_)ïƒ•

A method that regroups all arguments linked to the dataloaders creation.

Parameters:

* **drop_last** (bool, _optional_ , defaults to False) â€“ Whether to drop the last incomplete batch (if the length of the dataset is not divisible by the batch size) or not.

* **num_workers** (int, _optional_ , defaults to 0) â€“ Number of subprocesses to use for data loading (PyTorch only). 0 means that the data will be loaded in the main process.

* **pin_memory** (bool, _optional_ , defaults to True) â€“ Whether you want to pin memory in data loaders or not. Will default to True.

* **persistent_workers** (bool, _optional_ , defaults to False) â€“ If True, the data loader will not shut down the worker processes after a dataset has been consumed once. This allows to maintain the workers Dataset instances alive. Can potentially speed up training, but will increase RAM usage. Will default to False.

* **prefetch_factor** (int, _optional_) â€“ Number of batches loaded in advance by each worker. 2 means there will be a total of 2 * num_workers batches prefetched across all workers.

* **auto_find_batch_size** (bool, _optional_ , defaults to False) â€“ Whether to find a batch size that will fit into memory automatically through exponential decay, avoiding CUDA Out-of-Memory errors. Requires accelerate to be installed (pip install accelerate)

* **ignore_data_skip** (bool, _optional_ , defaults to False) â€“ When resuming training, whether or not to skip the epochs and batches to get the data loading at the same stage as in the previous training. If set to True, the training will begin faster (as that skipping step can take a long time) but will not yield the same results as the interrupted training would have.

* **sampler_seed** (int, _optional_) â€“ Random seed to be used with data samplers. If not set, random generators for data sampling will use the same seed as self.seed. This can be used to ensure reproducibility of data sampling, independent of the model seed.

Example:

```py >>> from transformers import TrainingArguments

    >>> args = TrainingArguments("working_dir")
    >>> args = args.set_dataloader(train_batch_size=16, eval_batch_size=64)
    >>> args.per_device_train_batch_size
    16
    ```

set_evaluate(_strategy : str | IntervalStrategy = 'no'_, _steps : int = 500_, _batch_size : int = 8_, _accumulation_steps : int | None = None_, _delay : float | None = None_, _loss_only : bool = False_, _jit_mode : bool = False_)ïƒ•

A method that regroups all arguments linked to evaluation.

Parameters:

* **strategy** (str or [~trainer_utils.IntervalStrategy], _optional_ , defaults to â€œnoâ€•) â€“

The evaluation strategy to adopt during training. Possible values are:

> * â€•noâ€•: No evaluation is done during training.
>
> * â€•stepsâ€•: Evaluation is done (and logged) every steps.
>
> * â€•epochâ€•: Evaluation is done at the end of each epoch.

Setting a strategy different from â€œnoâ€• will set self.do_eval to True.

* **steps** (int, _optional_ , defaults to 500) â€“ Number of update steps between two evaluations if strategy=â€•stepsâ€•.

* **batch_size** (int _optional_ , defaults to 8) â€“ The batch size per device (GPU/TPU core/CPUâ€¦) used for evaluation.

* **accumulation_steps** (int, _optional_) â€“ Number of predictions steps to accumulate the output tensors for, before moving the results to the CPU. If left unset, the whole predictions are accumulated on GPU/TPU before being moved to the CPU (faster but requires more memory).

* **delay** (float, _optional_) â€“ Number of epochs or steps to wait for before the first evaluation can be performed, depending on the eval_strategy.

* **loss_only** (bool, _optional_ , defaults to False) â€“ Ignores all outputs except the loss.

* **jit_mode** (bool, _optional_) â€“ Whether or not to use PyTorch jit trace for inference.

Example:

```py >>> from transformers import TrainingArguments

    >>> args = TrainingArguments("working_dir")
    >>> args = args.set_evaluate(strategy="steps", steps=100)
    >>> args.eval_steps
    100
    ```

set_logging(_strategy : str | IntervalStrategy = 'steps'_, _steps : int = 500_, _report_to : str | List[str] = 'none'_, _level : str = 'passive'_, _first_step : bool = False_, _nan_inf_filter : bool = False_, _on_each_node : bool = False_, _replica_level : str = 'passive'_)ïƒ•

A method that regroups all arguments linked to logging.

Parameters:

  * **strategy** (str or [~trainer_utils.IntervalStrategy], _optional_ , defaults to â€œstepsâ€•) â€“

The logging strategy to adopt during training. Possible values are:

>   * â€•noâ€•: No logging is done during training.
>
>   * â€•epochâ€•: Logging is done at the end of each epoch.
>
>   * â€•stepsâ€•: Logging is done every logging_steps.

  * **steps** (int, _optional_ , defaults to 500) â€“ Number of update steps between two logs if strategy=â€•stepsâ€•.

  * **level** (str, _optional_ , defaults to â€œpassiveâ€•) â€“ Logger log level to use on the main process. Possible choices are the log levels as strings: â€œdebugâ€•, â€œinfoâ€•, â€œwarningâ€•,

â€œerrorâ€• and â€œcriticalâ€•, plus a â€œpassiveâ€• level which doesnâ€™t set anything and lets the application set the level.

* **report_to** (str or List[str], _optional_ , defaults to â€œallâ€•) â€“ The list of integrations to report the results and logs to. Supported platforms are â€œazure_mlâ€•, â€œclearmlâ€•, â€œcodecarbonâ€•, â€œcomet_mlâ€•, â€œdagshubâ€•, â€œdvcliveâ€•, â€œflyteâ€•, â€œmlflowâ€•, â€œneptuneâ€•, â€œtensorboardâ€•, and â€œwandbâ€•. Use â€œallâ€• to report to all integrations installed, â€œnoneâ€• for no integrations.

* **first_step** (bool, _optional_ , defaults to False) â€“ Whether to log and evaluate the first global_step or not.

* **nan_inf_filter** (bool, _optional_ , defaults to True) â€“

Whether to filter nan and inf losses for logging. If set to True the loss of
every step that is nan or inf is filtered and the average loss of the current
logging window is taken instead.

<Tip>

nan_inf_filter only influences the logging of loss values, it does not change
the behavior the gradient is computed or applied to the model.

</Tip>

* **on_each_node** (bool, _optional_ , defaults to True) â€“ In multinode distributed training, whether to log using log_level once per node, or only on the main node.

   * **replica_level** (str, _optional_ , defaults to â€œpassiveâ€•) â€“ Logger log level to use on replicas. Same choices as log_level

Example:

```py >>> from transformers import TrainingArguments

    >>> args = TrainingArguments("working_dir")
    >>> args = args.set_logging(strategy="steps", steps=100)
    >>> args.logging_steps
    100
    ```

set_lr_scheduler(_name : str | SchedulerType = 'linear'_, _num_epochs : float = 3.0_, _max_steps : int = -1_, _warmup_ratio : float = 0_, _warmup_steps : int = 0_)ïƒ•

A method that regroups all arguments linked to the learning rate scheduler and

its hyperparameters.

Parameters:

* **name** (str or [SchedulerType], _optional_ , defaults to â€œlinearâ€•) â€“ The scheduler type to use. See the documentation of [SchedulerType] for all possible values.

* **num_epochs** (float, _optional_ , defaults to 3.0) â€“ Total number of training epochs to perform (if not an integer, will perform the decimal part percents of the last epoch before stopping training).

* **max_steps** (int, _optional_ , defaults to -1) â€“ If set to a positive number, the total number of training steps to perform. Overrides num_train_epochs. For a finite dataset, training is reiterated through the dataset (if all data is exhausted) until max_steps is reached.

* **warmup_ratio** (float, _optional_ , defaults to 0.0) â€“ Ratio of total training steps used for a linear warmup from 0 to learning_rate.

* **warmup_steps** (int, _optional_ , defaults to 0) â€“ Number of steps used for a linear warmup from 0 to learning_rate. Overrides any effect of warmup_ratio.

Example:

```py >>> from transformers import TrainingArguments

```
    >>> args = TrainingArguments("working_dir")
    >>> args = args.set_lr_scheduler(name="cosine", warmup_ratio=0.05)
    >>> args.warmup_ratio
```
```

0.05

```

set_optimizer(_name : str | OptimizerNames = 'adamw_torch'_, _learning_rate : float = 5e-05_, _weight_decay : float = 0_, _beta1 : float = 0.9_, _beta2 : float = 0.999_, _epsilon : float = 1e-08_, _args : str | None = None_)ïƒ•

A method that regroups all arguments linked to the optimizer and its

hyperparameters.

Parameters:

* **name** (str or [training_args.OptimizerNames], _optional_ , defaults to â€œadamw_torchâ€•) â€“ The optimizer to use: â€œadamw_hfâ€•, â€œadamw_torchâ€•, â€œadamw_torch_fusedâ€•, â€œadamw_apex_fusedâ€•, â€œadamw_anyprecisionâ€• or â€œadafactorâ€•.

* **learning_rate** (float, _optional_ , defaults to 5e-5) â€“ The initial learning rate.

* **weight_decay** (float, _optional_ , defaults to 0) â€“ The weight decay to apply (if not zero) to all layers except all bias and LayerNorm weights.

* **beta1** (float, _optional_ , defaults to 0.9) â€“ The beta1 hyperparameter for the adam optimizer or its variants.
```

* **beta2** (float, _optional_ , defaults to 0.999) â€“ The beta2 hyperparameter for the adam optimizer or its variants.

* **epsilon** (float, _optional_ , defaults to 1e-8) â€“ The epsilon hyperparameter for the adam optimizer or its variants.

* **args** (str, _optional_) â€“ Optional arguments that are supplied to AnyPrecisionAdamW (only useful when optim=â€•adamw_anyprecisionâ€•).

Example:

```py >>> from transformers import TrainingArguments

    >>> args = TrainingArguments("working_dir")
    >>> args = args.set_optimizer(name="adamw_torch", beta1=0.8)
    >>> args.optim
    'adamw_torch'
    ```

set_push_to_hub(_model_id : str_, _strategy : str | HubStrategy = 'every_save'_, _token : str | None = None_, _private_repo : bool = False_, _always_push : bool = False_)ïƒ•

A method that regroups all arguments linked to synchronizing checkpoints with

the Hub.

<Tip>

Calling this method will set self.push_to_hub to True, which means the

output_dir will begin a git directory synced with the repo (determined by

model_id) and the content will be pushed each time a save is triggered

(depending on your self.save_strategy). Calling [~Trainer.save_model] will

also trigger a push.

</Tip>

Parameters:

* **model_id** (str) â€“ The name of the repository to keep in sync with the local _output_dir_. It can be a simple model ID in which case the model will be pushed in your namespace. Otherwise it should be the whole repository name, for instance â€œuser_name/modelâ€•, which allows you to push to an organization you are a member of with â€œorganization_name/modelâ€•.

* **strategy** (str or [~trainer_utils.HubStrategy], _optional_ , defaults to â€œevery_saveâ€•) â€“

Defines the scope of what is pushed to the Hub and when. Possible values are:

* â€•endâ€•: push the model, its configuration, the tokenizer (if passed along to the [Trainer]) and

a

draft of a model card when the [~Trainer.save_model] method is called. \-
â€œevery_saveâ€•: push the model, its configuration, the tokenizer (if passed
along to the [Trainer])

> and

a draft of a model card each time there is a model save. The pushes are
asynchronous to not block training, and in case the save are very frequent, a
new push is only attempted if the previous one is finished. A last push is
made with the final model at the end of training. \- â€œcheckpointâ€•: like
â€œevery_saveâ€• but the latest checkpoint is also pushed in a subfolder named
last-checkpoint, allowing you to resume training easily with
trainer.train(resume_from_checkpoint=â€•last-checkpointâ€•). \-
â€œall_checkpointsâ€•: like â€œcheckpointâ€• but all checkpoints are pushed
like they appear in the

> output

folder (so you will get one checkpoint folder per folder in your final
repository)

  * **token** (str, _optional_) â€“ The token to use to push the model to the Hub. Will default to the token in the cache folder obtained with huggingface-cli login.

  * **private_repo** (bool, _optional_ , defaults to False) â€“ If True, the Hub repo will be set to

private.

  * **always_push** (bool, _optional_ , defaults to False) â€“ Unless this is True, the Trainer will skip pushing a checkpoint when the previous push is not finished.

Example:

```py >>> from transformers import TrainingArguments

    >>> args = TrainingArguments("working_dir")
    >>> args = args.set_push_to_hub("me/awesome-model")
    >>> args.hub_model_id
    'me/awesome-model'
    ```

set_save(_strategy : str | IntervalStrategy = 'steps'_, _steps : int = 500_, _total_limit : int | None = None_, _on_each_node : bool = False_)ïƒ•

A method that regroups all arguments linked to checkpoint saving.

Parameters:

* **strategy** (str or [~trainer_utils.IntervalStrategy], _optional_ , defaults to â€œstepsâ€•) â€“

The checkpoint save strategy to adopt during training. Possible values are:

> * â€•noâ€•: No save is done during training.
>
> * â€•epochâ€•: Save is done at the end of each epoch.
>
> * â€•stepsâ€•: Save is done every save_steps.

* **steps** (int, _optional_ , defaults to 500) â€“ Number of updates steps before two checkpoint saves if strategy=â€•stepsâ€•.

* **total_limit** (int, _optional_) â€“ If a value is passed, will limit the total amount of checkpoints. Deletes the older checkpoints in output_dir.

* **on_each_node** (bool, _optional_ , defaults to False) â€“

When doing multi-node distributed training, whether to save models and
checkpoints on each node, or only on the main one.

This should not be activated when the different nodes use the same storage as
the files will be saved with the same names for each node.

Example:

```py >>> from transformers import TrainingArguments

    >>> args = TrainingArguments("working_dir")
    >>> args = args.set_save(strategy="steps", steps=100)
    >>> args.save_steps
    100
    ```

set_testing(_batch_size : int = 8_, _loss_only : bool = False_, _jit_mode : bool = False_)ïƒ•

A method that regroups all basic arguments linked to testing on a held-out dataset.

<Tip>

Calling this method will automatically set self.do_predict to True.

</Tip>

Parameters:

* **batch_size** (int _optional_ , defaults to 8) â€“ The batch size per device (GPU/TPU core/CPUâ€¦) used for testing.

* **loss_only** (bool, _optional_ , defaults to False) â€“ Ignores all outputs except the loss.

* **jit_mode** (bool, _optional_) â€“ Whether or not to use PyTorch jit trace for inference.

Example:

```py >>> from transformers import TrainingArguments

    >>> args = TrainingArguments("working_dir")
    >>> args = args.set_testing(batch_size=32)
    >>> args.per_device_eval_batch_size
    32
    ```

set_training(_learning_rate : float = 5e-05_, _batch_size : int = 8_,

_weight_decay : float = 0_, _num_epochs : float = 3_, _max_steps : int = -1_,

_gradient_accumulation_steps : int = 1_, _seed : int = 42_,

_gradient_checkpointing : bool = False_)ïƒ•

A method that regroups all basic arguments linked to the training.

<Tip>

Calling this method will automatically set self.do_train to True.

</Tip>

Parameters:

* **learning_rate** (float, _optional_ , defaults to 5e-5) â€“ The initial learning rate for the optimizer.

* **batch_size** (int _optional_ , defaults to 8) â€“ The batch size per device (GPU/TPU core/CPUâ€¦) used for training.

* **weight_decay** (float, _optional_ , defaults to 0) â€“ The weight decay to apply (if not zero) to all layers except all bias and LayerNorm weights in the optimizer.

* **num_train_epochs** (float, _optional_ , defaults to 3.0) â€“ Total number of training epochs to perform (if not an integer, will perform the decimal part percents of the last epoch before stopping training).

* **max_steps** (int, _optional_ , defaults to -1) â€“ If set to a positive number, the total number of

training steps to perform. Overrides num_train_epochs. For a finite dataset, training is reiterated through the dataset (if all data is exhausted) until max_steps is reached.

* **gradient_accumulation_steps** (int, _optional_ , defaults to 1) â€“

Number of updates steps to accumulate the gradients for, before performing a
backward/update pass.

<Tip warning={true}>

When using gradient accumulation, one step is counted as one step with
backward pass. Therefore, logging, evaluation, save will be conducted every
gradient_accumulation_steps * xxx_step training examples.

</Tip>

* **seed** (int, _optional_ , defaults to 42) â€“ Random seed that will be set at the beginning of training. To ensure reproducibility across runs, use the [~Trainer.model_init] function to instantiate the model if it has some randomly initialized parameters.

* **gradient_checkpointing** (bool, _optional_ , defaults to False) â€“ If True, use gradient checkpointing to save memory at the expense of slower backward pass.

Example:

```py >>> from transformers import TrainingArguments
```

```
>>> args = TrainingArguments("working_dir")
>>> args = args.set_training(learning_rate=1e-4, batch_size=32)
>>> args.learning_rate
1e-4
```

_property _should_logïƒ•

Whether or not the current process should produce log.

_property _should_saveïƒ•

Whether or not the current process should write to disk, e.g., to save models
and checkpoints.

to_dict()ïƒ•

Serializes this instance while replace Enum by their values (for JSON serialization support). It obfuscates the token values by removing their

value.

to_json_string()ïƒ•

Serializes this instance to a JSON string.

to_sanitized_dict() -> Dict[str, Any]ïƒ•

Sanitized serialization to use with TensorBoardâ€™s hparams

_property _train_batch_size _: int_ïƒ•

The actual batch size for training (may differ from per_gpu_train_batch_size

in distributed training).

_property _world_sizeïƒ•

The number of processes used in parallel.

[Previous](trainer.html "Trainer") [Next](losses.html "Losses")

* * *