Depending on the dataset and your use case, your test dataset may contain labels. In that case, this method will also return metrics, like in `evaluate()`.

> If your predictions or labels have different sequence lengths (for instance because you're doing dynamic padding in a token classification task) the predictions will be padded (on the right) to allow for concatenation into one array. The padding index is -100.

Returns: *NamedTuple* A namedtuple with the following keys:

- predictions (`np.ndarray`): The predictions on `test_dataset`.
- label_ids (`np.ndarray`, *optional*): The labels (if the dataset contained some).
- metrics (`Dict[str, float]`, *optional*): The potential dictionary of metrics (if the dataset contained labels).

TrainingArguments

class transformers.**TrainingArguments** <>

```
( output_dir: str, overwrite_output_dir: bool = False, do_train: bool = False, do_eval: bool =
False, do_predict: bool = False, eval_strategy: Union = '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: Optional = None, per_gpu_eval_batch_size: Optional = None,
gradient_accumulation_steps: int = 1, eval_accumulation_steps: Optional = None, eval_delay:
Optional = 0, 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: Union = 'linear',
lr_scheduler_kwargs: Union = <factory>, warmup_ratio: float = 0.0, warmup_steps: int = 0,
log_level: Optional = 'passive', log_level_replica: Optional = 'warning', log_on_each_node: bool =
True, logging_dir: Optional = None, logging_strategy: Union = 'steps', logging_first_step: bool =
False, logging_steps: float = 500, logging_nan_inf_filter: bool = True, save_strategy: Union =
'steps', save_steps: float = 500, save_total_limit: Optional = None, save_safetensors: Optional =
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: Optional = None, jit_mode_eval:
bool = False, use_ipex: bool = False, bf16: bool = False, fp16: bool = False, fp16_opt_level: str
= '01', half_precision_backend: str = 'auto', bf16_full_eval: bool = False, fp16_full_eval: bool =
False, tf32: Optional = None, local_rank: int = -1, ddp_backend: Optional = None, tpu_num_cores:
Optional = None, tpu_metrics_debug: bool = False, debug: Union = '', dataloader_drop_last: bool =
False, eval_steps: Optional = None, dataloader_num_workers: int = 0, dataloader_prefetch_factor:
Optional = None, past_index: int = -1, run_name: Optional = None, disable_tqdm: Optional = None,
remove_unused_columns: Optional = True, label_names: Optional = None, load_best_model_at_end:
Optional = False, metric_for_best_model: Optional = None, greater_is_better: Optional = None,
ignore_data_skip: bool = False, fsdp: Union = '', fsdp_min_num_params: int = 0, fsdp_config: Union
= None, fsdp_transformer_layer_cls_to_wrap: Optional = None, accelerator_config: Union = None,
deepspeed: Union = None, label_smoothing_factor: float = 0.0, optim: Union = 'adamw_torch',
optim_args: Optional = None, adafactor: bool = False, group_by_length: bool = False,
length_column_name: Optional = 'length', report_to: Union = None, ddp_find_unused_parameters:
```

```
Optional = None, ddp_bucket_cap_mb: Optional = None, ddp_broadcast_buffers: Optional = 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: Optional = None, hub_model_id: Optional = None, hub_strategy: Union
= 'every_save', hub_token: Optional = None, hub_private_repo: bool = False, hub_always_push: bool
= False, gradient_checkpointing: bool = False, gradient_checkpointing_kwargs: Union = None,
include_inputs_for_metrics: bool = False, eval_do_concat_batches: bool = True, fp16_backend: str =
'auto', evaluation_strategy: Union = None, push_to_hub_model_id: Optional = None,
push_to_hub_organization: Optional = None, push_to_hub_token: Optional = None, mp_parameters: str
= '', auto_find_batch_size: bool = False, full_determinism: bool = False, torchdynamo: Optional =
None, ray_scope: Optional = 'last', ddp_timeout: Optional = 1800, torch_compile: bool = False,
torch_compile_backend: Optional = None, torch_compile_mode: Optional = None, dispatch_batches:
Optional = None, split_batches: Optional = None, include_tokens_per_second: Optional = False,
include_num_input_tokens_seen: Optional = False, neftune_noise_alpha: Optional = None,
optim_target_modules: Union = None, batch_eval_metrics: bool = False )
```

Parameters

- **output_dir** (`str`) — The output directory where the model predictions and checkpoints will be written.
- **overwrite_output_dir** (`bool`, *optional*, defaults to `False`) — If `True`, overwrite the content of the output directory. Use this to continue training if `output_dir` points to a checkpoint directory.
- **do_train** (`bool`, *optional*, defaults to `False`) — Whether to run training or not. This argument is not directly used by Trainer, it's intended to be used by your training/evaluation scripts instead. See the example scripts for more details.
- **do_eval** (`bool`, *optional*) — Whether to run evaluation on the validation set or not. Will be set to `True` if `eval_strategy` is different from `"no"`. This argument is not directly used by Trainer, it's intended to be used by your training/evaluation scripts instead. See the example scripts for more details.
- **do_predict** (`bool`, *optional*, defaults to `False`) — Whether to run predictions on the test set or not. This argument is not directly used by Trainer, it's intended to be used by your training/evaluation scripts instead. See the example scripts for more details.
- **eval_strategy** (`str` or IntervalStrategy, *optional*, defaults to `"no"`) — The evaluation strategy to adopt during training. Possible values are:
    - `"no"`: No evaluation is done during training.
    - `"steps"`: Evaluation is done (and logged) every `eval_steps`.
    - `"epoch"`: Evaluation is done at the end of each epoch.
- **prediction_loss_only** (`bool`, *optional*, defaults to `False`) — When performing evaluation and generating predictions, only returns the loss.
- **per_device_train_batch_size** (`int`, *optional*, defaults to 8) — The batch size per GPU/XPU/TPU/MPS/NPU core/CPU for training.
- **per_device_eval_batch_size** (`int`, *optional*, defaults to 8) — The batch size per GPU/XPU/TPU/MPS/NPU core/CPU for evaluation.

- **gradient_accumulation_steps** (`int`, *optional*, defaults to 1) — Number of updates steps to accumulate the gradients for, before performing a backward/update pass.

  > 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.

- **eval_accumulation_steps** (`int`, *optional*) — 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/NPU/TPU before being moved to the CPU (faster but requires more memory).
- **eval_delay** (`float`, *optional*) — Number of epochs or steps to wait for before the first evaluation can be performed, depending on the eval_strategy.
- **learning_rate** (`float`, *optional*, defaults to 5e-5) — The initial learning rate for AdamW optimizer.
- **weight_decay** (`float`, *optional*, defaults to 0) — The weight decay to apply (if not zero) to all layers except all bias and LayerNorm weights in AdamW optimizer.
- **adam_beta1** (`float`, *optional*, defaults to 0.9) — The beta1 hyperparameter for the AdamW optimizer.
- **adam_beta2** (`float`, *optional*, defaults to 0.999) — The beta2 hyperparameter for the AdamW optimizer.
- **adam_epsilon** (`float`, *optional*, defaults to 1e-8) — The epsilon hyperparameter for the AdamW optimizer.
- **max_grad_norm** (`float`, *optional*, defaults to 1.0) — Maximum gradient norm (for gradient clipping).
- **num_train_epochs(`float`,** *optional*, defaults to 3.0) — 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) — 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.
- **lr_scheduler_type** (`str` or SchedulerType, *optional*, defaults to `"linear"`) — The scheduler type to use. See the documentation of SchedulerType for all possible values.
- **lr_scheduler_kwargs** ('dict', *optional*, defaults to {}) — The extra arguments for the lr_scheduler. See the documentation of each scheduler for possible values.
- **warmup_ratio** (`float`, *optional*, defaults to 0.0) — Ratio of total training steps used for a linear warmup from 0 to `learning_rate`.
- **warmup_steps** (`int`, *optional*, defaults to 0) — Number of steps used for a linear warmup from 0 to `learning_rate`. Overrides any effect of `warmup_ratio`.
- **log_level** (`str`, *optional*, defaults to `passive`) — Logger log level to use on the main process. Possible choices are the log levels as strings: 'debug', 'info', 'warning', 'error' and 'critical', plus a 'passive' level which doesn't set anything and keeps the current log level for the Transformers library (which will be `"warning"` by default).

- **log_level_replica** (`str`, *optional*, defaults to `"warning"`) — Logger log level to use on replicas. Same choices as `log_level`"
- **log_on_each_node** (`bool`, *optional*, defaults to `True`) — In multinode distributed training, whether to log using `log_level` once per node, or only on the main node.
- **logging_dir** (`str`, *optional*) — TensorBoard log directory. Will default to *output_dir/runs/**CURRENT_DATETIME_HOSTNAME***.
- **logging_strategy** (`str` or IntervalStrategy, *optional*, defaults to `"steps"`) — The logging strategy to adopt during training. Possible values are:
    - `"no"`: No logging is done during training.
    - `"epoch"`: Logging is done at the end of each epoch.
    - `"steps"`: Logging is done every `logging_steps`.
- **logging_first_step** (`bool`, *optional*, defaults to `False`) — Whether to log the first `global_step` or not.
- **logging_steps** (`int` or `float`, *optional*, defaults to 500) — Number of update steps between two logs if `logging_strategy="steps"`. Should be an integer or a float in range `[0,1)`. If smaller than 1, will be interpreted as ratio of total training steps.
- **logging_nan_inf_filter** (`bool`, *optional*, defaults to `True`) — 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.

    > `logging_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.

- **save_strategy** (`str` or IntervalStrategy, *optional*, defaults to `"steps"`) — The checkpoint save strategy to adopt during training. Possible values are:
    - `"no"`: No save is done during training.
    - `"epoch"`: Save is done at the end of each epoch.
    - `"steps"`: Save is done every `save_steps`.
- **save_steps** (`int` or `float`, *optional*, defaults to 500) — Number of updates steps before two checkpoint saves if `save_strategy="steps"`. Should be an integer or a float in range `[0,1)`. If smaller than 1, will be interpreted as ratio of total training steps.
- **save_total_limit** (`int`, *optional*) — If a value is passed, will limit the total amount of checkpoints. Deletes the older checkpoints in `output_dir`. When `load_best_model_at_end` is enabled, the "best" checkpoint according to `metric_for_best_model` will always be retained in addition to the most recent ones. For example, for `save_total_limit=5` and `load_best_model_at_end`, the four last checkpoints will always be retained alongside the best model. When `save_total_limit=1` and `load_best_model_at_end`, it is possible that two checkpoints are saved: the last one and the best one (if they are different).

- **save_safetensors** (`bool`, *optional*, defaults to `True`) — Use safetensors saving and loading for state dicts instead of default `torch.load` and `torch.save`.
- **save_on_each_node** (`bool`, *optional*, defaults to `False`) — 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.
- **save_only_model** (`bool`, *optional*, defaults to `False`) — When checkpointing, whether to only save the model, or also the optimizer, scheduler & rng state. Note that when this is true, you won't be able to resume training from checkpoint. This enables you to save storage by not storing the optimizer, scheduler & rng state. You can only load the model using `from_pretrained` with this option set to `True`.
- **restore_callback_states_from_checkpoint** (`bool`, *optional*, defaults to `False`) — Whether to restore the callback states from the checkpoint. If `True`, will override callbacks passed to the `Trainer` if they exist in the checkpoint."
- **use_cpu** (`bool`, *optional*, defaults to `False`) — Whether or not to use cpu. If set to False, we will use cuda or mps device if available.
- **seed** (`int`, *optional*, defaults to 42) — 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.
- **data_seed** (`int`, *optional*) — Random seed to be used with data samplers. If not set, random generators for data sampling will use the same seed as `seed`. This can be used to ensure reproducibility of data sampling, independent of the model seed.
- **jit_mode_eval** (`bool`, *optional*, defaults to `False`) — Whether or not to use PyTorch jit trace for inference.
- **use_ipex** (`bool`, *optional*, defaults to `False`) — Use Intel extension for PyTorch when it is available. IPEX installation.
- **bf16** (`bool`, *optional*, defaults to `False`) — Whether to use bf16 16-bit (mixed) precision training instead of 32-bit training. Requires Ampere or higher NVIDIA architecture or using CPU (use_cpu) or Ascend NPU. This is an experimental API and it may change.
- **fp16** (`bool`, *optional*, defaults to `False`) — Whether to use fp16 16-bit (mixed) precision training instead of 32-bit training.
- **fp16_opt_level** (`str`, *optional*, defaults to 'O1') — For `fp16` training, Apex AMP optimization level selected in ['O0', 'O1', 'O2', and 'O3']. See details on the Apex documentation.
- **fp16_backend** (`str`, *optional*, defaults to `"auto"`) — This argument is deprecated. Use `half_precision_backend` instead.
- **half_precision_backend** (`str`, *optional*, defaults to `"auto"`) — The backend to use for mixed precision training. Must be one of `"auto", "apex", "cpu_amp"`. `"auto"` will use CPU/CUDA AMP or APEX depending on the PyTorch version detected, while the other choices will force the requested backend.

- **bf16_full_eval** (`bool`, *optional*, defaults to `False`) — Whether to use full bfloat16 evaluation instead of 32-bit. This will be faster and save memory but can harm metric values. This is an experimental API and it may change.
- **fp16_full_eval** (`bool`, *optional*, defaults to `False`) — Whether to use full float16 evaluation instead of 32-bit. This will be faster and save memory but can harm metric values.
- **tf32** (`bool`, *optional*) — Whether to enable the TF32 mode, available in Ampere and newer GPU architectures. The default value depends on PyTorch's version default of `torch.backends.cuda.matmul.allow_tf32`. For more details please refer to the TF32 documentation. This is an experimental API and it may change.
- **local_rank** (`int`, *optional*, defaults to -1) — Rank of the process during distributed training.
- **ddp_backend** (`str`, *optional*) — The backend to use for distributed training. Must be one of `"nccl"`, `"mpi"`, `"ccl"`, `"gloo"`, `"hccl"`.
- **tpu_num_cores** (`int`, *optional*) — When training on TPU, the number of TPU cores (automatically passed by launcher script).
- **dataloader_drop_last** (`bool`, *optional*, defaults to `False`) — Whether to drop the last incomplete batch (if the length of the dataset is not divisible by the batch size) or not.
- **eval_steps** (`int` or `float`, *optional*) — Number of update steps between two evaluations if `eval_strategy="steps"`. Will default to the same value as `logging_steps` if not set. Should be an integer or a float in range `[0,1)`. If smaller than 1, will be interpreted as ratio of total training steps.
- **dataloader_num_workers** (`int`, *optional*, defaults to 0) — Number of subprocesses to use for data loading (PyTorch only). 0 means that the data will be loaded in the main process.
- **past_index** (`int`, *optional*, defaults to -1) — Some models like TransformerXL or XLNet can make use of the past hidden states for their predictions. If this argument is set to a positive int, the `Trainer` will use the corresponding output (usually index 2) as the past state and feed it to the model at the next training step under the keyword argument `mems`.
- **run_name** (`str`, *optional*, defaults to `output_dir`) — A descriptor for the run. Typically used for wandb and mlflow logging. If not specified, will be the same as `output_dir`.
- **disable_tqdm** (`bool`, *optional*) — Whether or not to disable the tqdm progress bars and table of metrics produced by ~`notebook.NotebookTrainingTracker` in Jupyter Notebooks. Will default to `True` if the logging level is set to warn or lower (default), `False` otherwise.
- **remove_unused_columns** (`bool`, *optional*, defaults to `True`) — Whether or not to automatically remove the columns unused by the model forward method.
- **label_names** (`List[str]`, *optional*) — The list of keys in your dictionary of inputs that correspond to the labels.

  Will eventually default to the list of argument names accepted by the model that contain the word "label", except if the model used is one of the `XxxForQuestionAnswering` in which case it will also include the `["start_positions", "end_positions"]` keys.
- **load_best_model_at_end** (`bool`, *optional*, defaults to `False`) — Whether or not to load the best model found during training at the end of training. When this option is enabled, the best checkpoint

will always be saved. See `save_total_limit` for more.

> When set to `True`, the parameters `save_strategy` needs to be the same as `eval_strategy`, and in the case it is "steps", `save_steps` must be a round multiple of `eval_steps`.

- **metric_for_best_model** (`str`, *optional*) — Use in conjunction with `load_best_model_at_end` to specify the metric to use to compare two different models. Must be the name of a metric returned by the evaluation with or without the prefix `"eval_"`. Will default to `"loss"` if unspecified and `load_best_model_at_end=True` (to use the evaluation loss).

  If you set this value, `greater_is_better` will default to `True`. Don't forget to set it to `False` if your metric is better when lower.

- **greater_is_better** (`bool`, *optional*) — Use in conjunction with `load_best_model_at_end` and `metric_for_best_model` to specify if better models should have a greater metric or not. Will default to:

  - `True` if `metric_for_best_model` is set to a value that isn't `"loss"` or `"eval_loss"`.
  - `False` if `metric_for_best_model` is not set, or set to `"loss"` or `"eval_loss"`.

- **ignore_data_skip** (`bool`, *optional*, defaults to `False`) — 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.

- **fsdp** (`bool`, `str` or list of `FSDPOption`, *optional*, defaults to `''`) — Use PyTorch Distributed Parallel Training (in distributed training only).

  A list of options along the following:

  - `"full_shard"`: Shard parameters, gradients and optimizer states.
  - `"shard_grad_op"`: Shard optimizer states and gradients.
  - `"hybrid_shard"`: Apply `FULL_SHARD` within a node, and replicate parameters across nodes.
  - `"hybrid_shard_zero2"`: Apply `SHARD_GRAD_OP` within a node, and replicate parameters across nodes.
  - `"offload"`: Offload parameters and gradients to CPUs (only compatible with `"full_shard"` and `"shard_grad_op"`).
  - `"auto_wrap"`: Automatically recursively wrap layers with FSDP using `default_auto_wrap_policy`.

- **fsdp_config** (`str` or `dict`, *optional*) — Config to be used with fsdp (Pytorch Distributed Parallel Training). The value is either a location of fsdp json config file (e.g., `fsdp_config.json`) or an already loaded json file as `dict`.

  A List of config and its options:

- min_num_params (`int`, *optional*, defaults to `0`): FSDP's minimum number of parameters for Default Auto Wrapping. (useful only when `fsdp` field is passed).

- transformer_layer_cls_to_wrap (`List[str]`, *optional*): List of transformer layer class names (case-sensitive) to wrap, e.g, `BertLayer`, `GPTJBlock`, `T5Block` … (useful only when `fsdp` flag is passed).

- backward_prefetch (`str`, *optional*) FSDP's backward prefetch mode. Controls when to prefetch next set of parameters (useful only when `fsdp` field is passed).

  A list of options along the following:

  - `"backward_pre"` : Prefetches the next set of parameters before the current set of parameter's gradient computation.
  - `"backward_post"` : This prefetches the next set of parameters after the current set of parameter's gradient computation.

- forward_prefetch (`bool`, *optional*, defaults to `False`) FSDP's forward prefetch mode (useful only when `fsdp` field is passed). If `"True"`, then FSDP explicitly prefetches the next upcoming all-gather while executing in the forward pass.

- limit_all_gathers (`bool`, *optional*, defaults to `False`) FSDP's limit_all_gathers (useful only when `fsdp` field is passed). If `"True"`, FSDP explicitly synchronizes the CPU thread to prevent too many in-flight all-gathers.

- use_orig_params (`bool`, *optional*, defaults to `True`) If `"True"`, allows non-uniform `requires_grad` during init, which means support for interspersed frozen and trainable paramteres. Useful in cases such as parameter-efficient fine-tuning. Please refer this [blog] (https://dev-discuss.pytorch.org/t/rethinking-pytorch-fully-sharded-data-parallel-fsdp-from-first-principles/1019

- sync_module_states (`bool`, *optional*, defaults to `True`) If `"True"`, each individually wrapped FSDP unit will broadcast module parameters from rank 0 to ensure they are the same across all ranks after initialization

- cpu_ram_efficient_loading (`bool`, *optional*, defaults to `False`) If `"True"`, only the first process loads the pretrained model checkpoint while all other processes have empty weights. When this setting as `"True"`, `sync_module_states` also must to be `"True"`, otherwise all the processes except the main process would have random weights leading to unexpected behaviour during training.

- activation_checkpointing (`bool`, *optional*, defaults to `False`): If `"True"`, activation checkpointing is a technique to reduce memory usage by clearing activations of certain layers and recomputing them during a backward pass. Effectively, this trades extra computation time for reduced memory usage.

- xla (`bool`, *optional*, defaults to `False`): Whether to use PyTorch/XLA Fully Sharded Data Parallel Training. This is an experimental feature and its API may evolve in the future.

- xla_fsdp_settings (`dict`, *optional*) The value is a dictionary which stores the XLA FSDP wrapping parameters.

  For a complete list of options, please see here.

- xla_fsdp_grad_ckpt (`bool`, *optional*, defaults to `False`): Will use gradient checkpointing over each nested XLA FSDP wrapped layer. This setting can only be used when the xla flag is set to true, and an auto wrapping policy is specified through fsdp_min_num_params or fsdp_transformer_layer_cls_to_wrap.

- **deepspeed** (`str` or `dict`, *optional*) — Use Deepspeed. This is an experimental feature and its API may evolve in the future. The value is either the location of DeepSpeed json config file (e.g., `ds_config.json`) or an already loaded json file as a `dict`"

  > If enabling any Zero-init, make sure that your model is not initialized until *after* initializing the `TrainingArguments`, else it will not be applied.

- **accelerator_config** (`str`, `dict`, or `AcceleratorConfig`, *optional*) — Config to be used with the internal `Accelerator` implementation. The value is either a location of accelerator json config file (e.g., `accelerator_config.json`), an already loaded json file as `dict`, or an instance of `AcceleratorConfig`.

  A list of config and its options:

  - split_batches (`bool`, *optional*, defaults to `False`): Whether or not the accelerator should split the batches yielded by the dataloaders across the devices. If `True` the actual batch size used will be the same on any kind of distributed processes, but it must be a round multiple of the `num_processes` you are using. If `False`, actual batch size used will be the one set in your script multiplied by the number of processes.
  - dispatch_batches (`bool`, *optional*): If set to `True`, the dataloader prepared by the Accelerator is only iterated through on the main process and then the batches are split and broadcast to each process. Will default to `True` for `DataLoader` whose underlying dataset is an `IterableDataset`, `False` otherwise.
  - even_batches (`bool`, *optional*, defaults to `True`): If set to `True`, in cases where the total batch size across all processes does not exactly divide the dataset, samples at the start of the dataset will be duplicated so the batch can be divided equally among all workers.
  - use_seedable_sampler (`bool`, *optional*, defaults to `True`): Whether or not use a fully seedable random sampler (`accelerate.data_loader.SeedableRandomSampler`). Ensures training results are fully reproducable using a different sampling technique. While seed-to-seed results may differ, on average the differences are neglible when using multiple different seeds to compare. Should also be ran with ~`utils.set_seed` for the best results.

- **label_smoothing_factor** (`float`, *optional*, defaults to 0.0) — The label smoothing factor to use. Zero means no label smoothing, otherwise the underlying onehot-encoded labels are changed from 0s and 1s to `label_smoothing_factor/num_labels` and `1 - label_smoothing_factor + label_smoothing_factor/num_labels` respectively.

- **debug** (`str` or list of `DebugOption`, *optional*, defaults to `""`) — Enable one or more debug features. This is an experimental feature.

  Possible options are:

  - `"underflow_overflow"`: detects overflow in model's input/outputs and reports the last frames that led to the event
  - `"tpu_metrics_debug"`: print debug metrics on TPU

  The options should be separated by whitespaces.

- **optim** (`str` or `training_args.OptimizerNames`, *optional*, defaults to `"adamw_torch"`) — The optimizer to use: adamw_hf, adamw_torch, adamw_torch_fused, adamw_apex_fused, adamw_anyprecision or adafactor.
- **optim_args** (`str`, *optional*) — Optional arguments that are supplied to AnyPrecisionAdamW.
- **group_by_length** (`bool`, *optional*, defaults to `False`) — Whether or not to group together samples of roughly the same length in the training dataset (to minimize padding applied and be more efficient). Only useful if applying dynamic padding.
- **length_column_name** (`str`, *optional*, defaults to `"length"`) — Column name for precomputed lengths. If the column exists, grouping by length will use these values rather than computing them on train startup. Ignored unless `group_by_length` is `True` and the dataset is an instance of `Dataset`.
- **report_to** (`str` or `List[str]`, *optional*, defaults to `"all"`) — The list of integrations to report the results and logs to. Supported platforms are `"azure_ml"`, `"clearml"`, `"codecarbon"`, `"comet_ml"`, `"dagshub"`, `"dvclive"`, `"flyte"`, `"mlflow"`, `"neptune"`, `"tensorboard"`, and `"wandb"`. Use `"all"` to report to all integrations installed, `"none"` for no integrations.
- **ddp_find_unused_parameters** (`bool`, *optional*) — When using distributed training, the value of the flag `find_unused_parameters` passed to `DistributedDataParallel`. Will default to `False` if gradient checkpointing is used, `True` otherwise.
- **ddp_bucket_cap_mb** (`int`, *optional*) — When using distributed training, the value of the flag `bucket_cap_mb` passed to `DistributedDataParallel`.
- **ddp_broadcast_buffers** (`bool`, *optional*) — When using distributed training, the value of the flag `broadcast_buffers` passed to `DistributedDataParallel`. Will default to `False` if gradient checkpointing is used, `True` otherwise.
- **dataloader_pin_memory** (`bool`, *optional*, defaults to `True`) — Whether you want to pin memory in data loaders or not. Will default to `True`.
- **dataloader_persistent_workers** (`bool`, *optional*, defaults to `False`) — 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`.
- **dataloader_prefetch_factor** (`int`, *optional*) — 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.

- **skip_memory_metrics** (`bool`, *optional*, defaults to `True`) — Whether to skip adding of memory profiler reports to metrics. This is skipped by default because it slows down the training and evaluation speed.
- **push_to_hub** (`bool`, *optional*, defaults to `False`) — Whether or not to push the model to the Hub every time the model is saved. If this is activated, `output_dir` will begin a git directory synced with the repo (determined by `hub_model_id`) and the content will be pushed each time a save is triggered (depending on your `save_strategy`). Calling save_model() will also trigger a push.

  > If `output_dir` exists, it needs to be a local clone of the repository to which the Trainer will be pushed.

- **resume_from_checkpoint** (`str`, *optional*) — The path to a folder with a valid checkpoint for your model. This argument is not directly used by Trainer, it's intended to be used by your training/evaluation scripts instead. See the example scripts for more details.
- **hub_model_id** (`str`, *optional*) — 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 `"user_name/model"`, which allows you to push to an organization you are a member of with `"organization_name/model"`. Will default to `user_name/output_dir_name` with *output_dir_name* being the name of `output_dir`.

  Will default to the name of `output_dir`.
- **hub_strategy** (`str` or `HubStrategy`, *optional*, defaults to `"every_save"`) — Defines the scope of what is pushed to the Hub and when. Possible values are:
  - `"end"`: push the model, its configuration, the tokenizer (if passed along to the Trainer) and a draft of a model card when the save_model() method is called.
  - `"every_save"`: 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.
  - `"checkpoint"`: like `"every_save"` 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="last-checkpoint")`.
  - `"all_checkpoints"`: like `"checkpoint"` 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)
- **hub_token** (`str`, *optional*) — 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`.
- **hub_private_repo** (`bool`, *optional*, defaults to `False`) — If True, the Hub repo will be set to private.
- **hub_always_push** (`bool`, *optional*, defaults to `False`) — Unless this is `True`, the `Trainer` will skip pushing a checkpoint when the previous push is not finished.
- **gradient_checkpointing** (`bool`, *optional*, defaults to `False`) — If True, use gradient checkpointing to save memory at the expense of slower backward pass.

- **gradient_checkpointing_kwargs** (`dict`, *optional*, defaults to `None`) — Key word arguments to be passed to the `gradient_checkpointing_enable` method.
- **include_inputs_for_metrics** (`bool`, *optional*, defaults to `False`) — Whether or not the inputs will be passed to the `compute_metrics` function. This is intended for metrics that need inputs, predictions and references for scoring calculation in Metric class.
- **eval_do_concat_batches** (`bool`, *optional*, defaults to `True`) — Whether to recursively concat inputs/losses/labels/predictions across batches. If `False`, will instead store them as lists, with each batch kept separate.
- **auto_find_batch_size** (`bool`, *optional*, defaults to `False`) — 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`)
- **full_determinism** (`bool`, *optional*, defaults to `False`) — If `True`, enable_full_determinism() is called instead of set_seed() to ensure reproducible results in distributed training. Important: this will negatively impact the performance, so only use it for debugging.
- **torchdynamo** (`str`, *optional*) — If set, the backend compiler for TorchDynamo. Possible choices are `"eager"`, `"aot_eager"`, `"inductor"`, `"nvfuser"`, `"aot_nvfuser"`, `"aot_cudagraphs"`, `"ofi"`, `"fx2trt"`, `"onnxrt"` and `"ipex"`.
- **ray_scope** (`str`, *optional*, defaults to `"last"`) — The scope to use when doing hyperparameter search with Ray. By default, `"last"` will be used. Ray will then use the last checkpoint of all trials, compare those, and select the best one. However, other options are also available. See the Ray documentation for more options.
- **ddp_timeout** (`int`, *optional*, defaults to 1800) — The timeout for `torch.distributed.init_process_group` calls, used to avoid GPU socket timeouts when performing slow operations in distributed runnings. Please refer the [PyTorch documentation] (https://pytorch.org/docs/stable/distributed.html#torch.distributed.init_process_group) for more information.
- **use_mps_device** (`bool`, *optional*, defaults to `False`) — This argument is deprecated.`mps` device will be used if it is available similar to `cuda` device.
- **torch_compile** (`bool`, *optional*, defaults to `False`) — Whether or not to compile the model using PyTorch 2.0 `torch.compile`.

  This will use the best defaults for the `torch.compile` API. You can customize the defaults with the argument `torch_compile_backend` and `torch_compile_mode` but we don't guarantee any of them will work as the support is progressively rolled in in PyTorch.

  This flag and the whole compile API is experimental and subject to change in future releases.
- **torch_compile_backend** (`str`, *optional*) — The backend to use in `torch.compile`. If set to any value, `torch_compile` will be set to `True`.

  Refer to the PyTorch doc for possible values and note that they may change across PyTorch versions.

  This flag is experimental and subject to change in future releases.

- **torch_compile_mode** (`str`, *optional*) — The mode to use in `torch.compile`. If set to any value, `torch_compile` will be set to `True`.

  Refer to the PyTorch doc for possible values and note that they may change across PyTorch versions.

  This flag is experimental and subject to change in future releases.

- **split_batches** (`bool`, *optional*) — Whether or not the accelerator should split the batches yielded by the dataloaders across the devices during distributed training. If

  set to `True`, the actual batch size used will be the same on any kind of distributed processes, but it must be a

  round multiple of the number of processes you are using (such as GPUs).

- **include_tokens_per_second** (`bool`, *optional*) — Whether or not to compute the number of tokens per second per device for training speed metrics.

  This will iterate over the entire training dataloader once beforehand,

  and will slow down the entire process.

- **include_num_input_tokens_seen** (`bool`, *optional*) — Whether or not to track the number of input tokens seen throughout training.

  May be slower in distributed training as gather operations must be called.

- **neftune_noise_alpha** (`Optional[float]`) — If not `None`, this will activate NEFTune noise embeddings. This can drastically improve model performance for instruction fine-tuning. Check out the original paper and the original code. Support transformers `PreTrainedModel` and also `PeftModel` from peft.

- **optim_target_modules** (`Union[str, List[str]]`, *optional*) — The target modules to optimize, i.e. the module names that you would like to train, right now this is used only for GaLore algorithm https://arxiv.org/abs/2403.03507 See: https://github.com/jiaweizzhao/GaLore for more details. You need to make sure to pass a valid GaloRe optimizer, e.g. one of: “galore_adamw”, “galore_adamw_8bit”, “galore_adafactor” and make sure that the target modules are `nn.Linear` modules only.

- **batch_eval_metrics** (`Optional[bool]`, defaults to `False`) — If set to `True`, evaluation will call compute_metrics at the end of each batch to accumulate statistics rather than saving all eval logits in memory. When set to `True`, you must pass a compute_metrics function that takes a boolean argument `compute_result`, which when passed `True`, will trigger the final global summary statistics from the batch-level summary statistics you’ve accumulated over the evaluation set.

TrainingArguments is the subset of the arguments we use in our example scripts **which relate to the training loop itself**.

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