Trainer

Trainer is a complete training and evaluation loop for Transformers' PyTorch models. Plug a model, preprocessor, dataset, and training arguments into Trainer and let it handle the rest to start training faster.

Trainer is also powered by Accelerate, a library for handling large models for distributed training.

This guide will show you how Trainer works and how to customize it for your use case with a callback.

```
!pip install accelerate --upgrade
```

Trainer contains all the necessary components of a training loop.

1. calculate the loss from a training step
2. calculate the gradients with the backward method
3. update the weights based on the gradients
4. repeat until the predetermined number of epochs is reached

Manually coding this training loop everytime can be inconvenient or a barrier if you're just getting started with machine learning. Trainer abstracts this process, allowing you to focus on the model, dataset, and training design choices.

Configure your training with hyperparameters and options from TrainingArguments which supports many features such as distributed training, torch.compile, mixed precision training, and saving the model to the Hub.

> The number of available parameters available in TrainingArguments may be intimidating at first. If there is a specific hyperparameter or feature you want to use, try searching for it directly.

> Otherwise, feel free to start with the default values and gradually customize them as you become more familiar with the training process.

The example below demonstrates an example of TrainingArguments that evaluates and saves the model at the end of each epoch. It also loads the best model found during training and pushes it to the Hub.

```
from transformers import TrainingArguments

training_args = TrainingArguments(
    output_dir="your-model",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=2,
    weight_decay=0.01,
    eval_strategy="epoch",
    save_strategy="epoch",
    load_best_model_at_end=True,
    push_to_hub=True,
)
```

Pass your model, dataset, preprocessor, and TrainingArguments to Trainer, and call train() to start training.

> Refer to the Fine-tuning guide for a more complete overview of the training process.

```
from transformers import Trainer

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=dataset["train"],
    eval_dataset=dataset["test"],
    processing_class=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)

trainer.train()
```

Checkpoints

Trainer saves checkpoints (the optimizer state is not saved by default) to the directory in `output_dir` in TrainingArguments to a subfolder named `checkpoint-000`. The number at the end is the training step at which the checkpoint was saved.

Saving checkpoints are useful for resuming training or recovering your training progress if you encounter an error. Set the `resume_from_checkpoint` parameter in train() to resume training from the last checkpoint or a specific checkpoint.

latest checkpoint | specific checkpoint

```
trainer.train(resume_from_checkpoint=True)
```

Checkpoints can be saved to the Hub by setting `push_to_hub=True` in TrainingArguments. The default method (`"every_save"`) saves a checkpoint to the Hub every time a model is saved, which is typically the final model at the end of training. Some other options for deciding how to save checkpoints to the Hub include the following.

- `hub_strategy="end"` only pushes a checkpoint when save_model() is called
- `hub_strategy="checkpoint"` pushes the latest checkpoint to a subfolder named *last-checkpoint* from which training can be resumed
- `hub_strategy="all_checkpoints"` pushes all checkpoints to the Hub with one checkpoint per subfolder in your model repository

Trainer attempts to maintain the same Python, NumPy, and PyTorch RNG states when you resume training from a checkpoint. But PyTorch has various non-deterministic settings which can't guarantee the RNG states are identical. To enable full determinism, refer to the Controlling sources of randomness guide to learn what settings to adjust to make training fully deterministic (some settings may result in slower training).

Logging

Trainer is set to `logging.INFO` by default to report errors, warnings, and other basic information. Use `log_level()` to change the logging level and log verbosity.

The example below sets the main code and modules to use the same log level.

```
logger = logging.getLogger(__name__)

logging.basicConfig(
    format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
    datefmt="%m/%d/%Y %H:%M:%S",
    handlers=[logging.StreamHandler(sys.stdout)],
)

log_level = training_args.get_process_log_level()
logger.setLevel(log_level)
datasets.utils.logging.set_verbosity(log_level)
transformers.utils.logging.set_verbosity(log_level)

trainer = Trainer(...)
```

In a distributed environment, Trainer replicas are set to `logging.WARNING` to only report errors and warnings. Use `log_level_replica()` to change the logging level and log verbosity. To configure the log level for each node, use `log_on_each_node()` to determine whether to use a specific log level on each node or only the main node.

Use different combinations of `log_level` and `log_level_replica` to configure what gets logged on each node.

single node | multi-node

```
my_app.py ... --log_level warning --log_level_replica error
```

> The log level is separately set for each node in the `__init__()` method. Consider setting this sooner if you're using other Transformers functionalities before creating the Trainer instance.

Customize

Tailor Trainer to your use case by subclassing or overriding its methods to support the functionality you want to add or use, without rewriting the entire training loop from scratch. The table below lists some of the methods that can be customized.

method	description
get_train_dataloader()	create a training DataLoader
get_eval_dataloader()	create an evaluation DataLoader
get_test_dataloader()	create a test DataLoader
log()	log information about the training process
create_optimizer_and_scheduler()	create an optimizer and learning rate scheduler (can also be separately customized with create_optimizer() and create_scheduler() if they weren't passed in `__init__`)
compute_loss()	compute the loss of a batch of training inputs
training_step()	perform the training step
prediction_step()	perform the prediction and test step
evaluate()	evaluate the model and return the evaluation metric
predict()	make a prediction (with metrics if labels are available) on the test set

For example, to use weighted loss, rewrite compute_loss() inside Trainer.

```
from torch import nn
from transformers import Trainer

class CustomTrainer(Trainer):
    def compute_loss(self, model: nn.Module, inputs: dict[str, Union[torch.Tensor, Any
        labels = inputs.pop("labels")
        # forward pass
        outputs = model(**inputs)
        logits = outputs.get("logits")
        # compute custom loss for 3 labels with different weights
        reduction = "sum" if num_items_in_batch is not None else "mean"
        loss_fct = nn.CrossEntropyLoss(weight=torch.tensor([1.0, 2.0, 3.0], device=mod
        loss = loss_fct(logits.view(-1, self.model.config.num_labels), labels.view(-1)
        if num_items_in_batch is not None:
            loss = loss / num_items_in_batch
        return (loss, outputs) if return_outputs else loss
```

Callbacks

Callbacks are another way to customize Trainer, but they don't change anything *inside the training loop*. Instead, a callback inspects the training loop state and executes some action (early stopping, logging, etc.) depending on the state. For example, you can't implement a custom loss function with a callback because that requires overriding compute_loss().

To use a callback, create a class that inherits from TrainerCallback and implements the functionality you want. Then pass the callback to the `callback` parameter in Trainer. The example below implements an early stopping callback that stops training after 10 steps.

```
from transformers import TrainerCallback, Trainer

class EarlyStoppingCallback(TrainerCallback):
    def __init__(self, num_steps=10):
        self.num_steps = num_steps

    def on_step_end(self, args, state, control, **kwargs):
        if state.global_step >= self.num_steps:
            return {"should_training_stop": True}
        else:
            return {}

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=dataset["train"],
    eval_dataset=dataset["test"],
    processing_class=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
    callbacks=[EarlyStoppingCallback()],
)
```

Accelerate

Accelerate is a library that simplifies training in distributed environments and across different hardware. Its integration with Trainer means Trainer supports distributed training frameworks like Fully Sharded Data Parallel (FSDP) and DeepSpeed.

> Learn more about FSDP sharding strategies, CPU offloading, and more with Trainer in the Fully

> Sharded Data Parallel guide.

To use Accelerate with Trainer, run the accelerate_config command to configure your training environment. This command creates a `config_file.yaml` file that stores the configuration settings of your training environment and it's used whenever you launch your training script. Some example distributed training configurations are shown below.

DistributedDataParallel | FullyShardedDataParallel | DeepSpeed | DeepSpeed with Accelerate plugin

```yaml
compute_environment: LOCAL_MACHINE
distributed_type: MULTI_GPU
downcast_bf16: 'no'
gpu_ids: all
machine_rank: 0 #change rank as per the node
main_process_ip: 192.168.20.1
main_process_port: 9898
main_training_function: main
mixed_precision: fp16
num_machines: 2
num_processes: 8
rdzv_backend: static
same_network: true
tpu_env: []
tpu_use_cluster: false
tpu_use_sudo: false
use_cpu: false
```

Run accelerate_launch to start training with the configurations set in `config_file.yaml`. This file is saved to the Accelerate cache folder and automatically loaded when you run `accelerate_launch`.

The example below launches the run_glue.py script with the FSDP configuration shown earlier. Parameters from the `config_file.yaml` file can also be directly set in the command line.

```bash
accelerate launch \
    ./examples/pytorch/text-classification/run_glue.py \
    --model_name_or_path google-bert/bert-base-cased \
    --task_name $TASK_NAME \
    --do_train \
    --do_eval \
    --max_seq_length 128 \
    --per_device_train_batch_size 16 \
```

```
    --learning_rate 5e-5 \
    --num_train_epochs 3 \
    --output_dir /tmp/$TASK_NAME/ \
    --overwrite_output_dir
```

> Refer to the Launching your Accelerate scripts tutorial to learn more about `accelerate_launch` and custom configurations.

Optimizations

Trainer supports various optimizations to improve *training* performance - reduce memory and increase training speed - and *model* performance.

torch.compile

torch.compile can significantly speed up training and reduce computational overhead. Configure your torch.compile settings in TrainingArguments. Set `torch_compile` to `True`, and select a backend and compile mode.

```
from transformers import TrainingArguments

training_args = TrainingArguments(
    torch_compile=True,
    torch_compile_backend="inductor",
    torch_compile_mode="default",
    ...,
)
```

GaLore

Gradient Low-Rank Projection (GaLore) significantly reduces memory usage when training large language models (LLMs). One of GaLores key benefits is *full-parameter* learning, unlike low-rank adaptation methods like LoRA, which produces better model performance.

Install the GaLore and TRL libraries.

```
pip install galore-torch trl
```

Pick a GaLore optimizer (`"galore_adamw"`, `"galore_adafactor"`, `"galore_adamw_8bit"`) and pass it to the `optim` parameter in trl.SFTConfig. Use the `optim_target_modules` parameter to specify which modules to adapt (can be a list of strings, regex, or a full path).

Extra parameters supported by GaLore, `rank`, `update_proj_gap`, and `scale`, should be passed to the `optim_args` parameter in trl.SFTConfig.

The example below enables GaLore with SFTTrainer that targets the `attn` and `mlp` layers with regex.

> It can take some time before training starts (~3 minutes for a 2B model on a NVIDIA A100).

GaLore optimizer | GaLore optimizer with layerwise optimization

```
import datasets
from trl import SFTConfig, SFTTrainer

train_dataset = datasets.load_dataset('imdb', split='train')
args = SFTConfig(
    output_dir="./test-galore",
    max_steps=100,
    optim="galore_adamw",
    optim_target_modules=[r".*.attn.*", r".*.mlp.*"],
    optim_args="rank=64, update_proj_gap=100, scale=0.10",
    gradient_checkpointing=True,
)
trainer = SFTTrainer(
    model="google/gemma-2b",
    args=args,
    train_dataset=train_dataset,
)
trainer.train()
```

Only linear layers that are considered GaLore layers can be trained with low-rank decomposition. The rest of the model layers are optimized in the usual way.

Liger

Liger Kernel is a collection of layers such as RMSNorm, RoPE, SwiGLU, CrossEntropy, FusedLinearCrossEntropy, and more that have been fused into a single Triton kernel for training

LLMs. These kernels are also compatible with FlashAttention, FSDP, and DeepSpeed. As a result, Liger Kernel can increase multi-GPU training throughput and reduce memory usage. This is useful for multi-head training and supporting larger vocabulary sizes, larger batch sizes, and longer context lengths.

```
pip install liger-kernel
```

Enable Liger Kernel for training by setting `use_liger_kernel=True` in TrainingArguments. This patches the corresponding layers in the model with Ligers kernels.

> Liger Kernel supports Llama, Gemma, Mistral, and Mixtral models. Refer to the patching list for the latest list of supported models.

```
from transformers import TrainingArguments

training_args = TrainingArguments(
    output_dir="your-model",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=2,
    weight_decay=0.01,
    eval_strategy="epoch",
    save_strategy="epoch",
    load_best_model_at_end=True,
    push_to_hub=True,
    use_liger_kernel=True
)
```

You can also configure which specific kernels to apply using the `liger_kernel_config` parameter. This dict is passed as keyword arguments to the `_apply_liger_kernel_to_instance` function, allowing fine-grained control over kernel usage. Available options vary by model but typically include: `rope`, `swiglu`, `cross_entropy`, `fused_linear_cross_entropy`, `rms_norm`, etc.

```
from transformers import TrainingArguments

# Apply only specific kernels
```

```
training_args = TrainingArguments(
    output_dir="your-model",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=2,
    weight_decay=0.01,
    eval_strategy="epoch",
    save_strategy="epoch",
    load_best_model_at_end=True,
    push_to_hub=True,
    use_liger_kernel=True,
    liger_kernel_config={
        "rope": True,
        "cross_entropy": True,
        "rms_norm": False,  # Don't apply Liger's RMSNorm kernel
        "swiglu": True,
    }
)
```

NEFTune

NEFTune adds noise to the embedding vectors during training to improve model performance. Enable it in Trainer with the `neftune_noise_alpha` parameter in TrainingArguments to control how much noise is added.

```
from transformers import TrainingArguments, Trainer

training_args = TrainingArguments(..., neftune_noise_alpha=0.1)
trainer = Trainer(..., args=training_args)
```

The original embedding layer is restored after training to avoid any unexpected behavior.

</> Update on GitHub

← Inference server backends

Fine-tuning →