```python
# coding=utf-8
# Copyright 2020-present the HuggingFace Inc. team.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""
Torch utilities for the Trainer class.
"""

import datetime
import json
import math
import os
import warnings
from contextlib import contextmanager
from dataclasses import dataclass
from typing import Dict, Iterator, List, Optional, Union

import numpy as np
import torch
from packaging import version
from torch.utils.data.dataset import Dataset, IterableDataset
from torch.utils.data.distributed import DistributedSampler
from torch.utils.data.sampler import RandomSampler, Sampler

from .file_utils import is_sagemaker_dp_enabled, is_sagemaker_mp_enabled, is_torch_tpu_available
from .utils import logging


if is_sagemaker_dp_enabled():
    import smdistributed.dataparallel.torch.distributed as dist
else:
    import torch.distributed as dist


if is_torch_tpu_available():
    import torch_xla.core.xla_model as xm

# this is used to suppress an undesired warning emitted by pytorch versions 1.4.2-1.7.0
try:
    from torch.optim.lr_scheduler import SAVE_STATE_WARNING
except ImportError:
    SAVE_STATE_WARNING = ""

logger = logging.get_logger(__name__)


def torch_pad_and_concatenate(tensor1, tensor2, padding_index=-100):
    """Concatenates `tensor1` and `tensor2` on first axis, applying padding on the second if necessary."""
    if len(tensor1.shape) == 1 or tensor1.shape[1] == tensor2.shape[1]:
        return torch.cat((tensor1, tensor2), dim=0)

    # Let's figure out the new shape
    new_shape = (tensor1.shape[0] + tensor2.shape[0], max(tensor1.shape[1], tensor2.shape[1])) + tensor1.shape[2:]

    # Now let's fill the result tensor
    result = tensor1.new_full(new_shape, padding_index)
    result[: tensor1.shape[0], : tensor1.shape[1]] = tensor1
    result[tensor1.shape[0] :, : tensor2.shape[1]] = tensor2
    return result


def numpy_pad_and_concatenate(array1, array2, padding_index=-100):
    """Concatenates `array1` and `array2` on first axis, applying padding on the second if necessary."""
    if len(array1.shape) == 1 or array1.shape[1] == array2.shape[1]:
        return np.concatenate((array1, array2), axis=0)

    # Let's figure out the new shape
    new_shape = (array1.shape[0] + array2.shape[0], max(array1.shape[1], array2.shape[1])) + array1.shape[2:]

    # Now let's fill the result tensor
    result = np.full_like(array1, padding_index, shape=new_shape)
    result[: array1.shape[0], : array1.shape[1]] = array1
    result[array1.shape[0] :, : array2.shape[1]] = array2
    return result


def nested_concat(tensors, new_tensors, padding_index=-100):
    """
    Concat the `new_tensors` to `tensors` on the first dim and pad them on the second if needed. Works for tensors or
    nested list/tuples of tensors.
    """
    assert type(tensors) == type(
        new_tensors
    ), f"Expected `tensors` and `new_tensors` to have the same type but found {type(tensors)} and {type(new_tensors)}."
    if isinstance(tensors, (list, tuple)):
        return type(tensors)(nested_concat(t, n, padding_index=padding_index) for t, n in zip(tensors, new_tensors))
    elif isinstance(tensors, torch.Tensor):
        return torch_pad_and_concatenate(tensors, new_tensors, padding_index=padding_index)
    elif isinstance(tensors, np.ndarray):
        return numpy_pad_and_concatenate(tensors, new_tensors, padding_index=padding_index)
    else:
        raise TypeError(f"Unsupported type for concatenation: got {type(tensors)}")


def find_batch_size(tensors):
    """
    Find the first dimension of a tensor in a nested list/tuple/dict of tensors.
    """
    if isinstance(tensors, (list, tuple)):
        for t in tensors:
            result = find_batch_size(t)
            if result is not None:
                return result
    elif isinstance(tensors, dict):
        for key, value in tensors.items():
            result = find_batch_size(value)
            if result is not None:
                return result
    elif isinstance(tensors, torch.Tensor):
```

```python
        return tensors.shape[0] if len(tensors.shape) >= 1 else None
    elif isinstance(tensors, np.ndarray):
        return tensors.shape[0] if len(tensors.shape) >= 1 else None


def nested_numpify(tensors):
    "Numpify `tensors` (even if it's a nested list/tuple of tensors)."
    if isinstance(tensors, (list, tuple)):
        return type(tensors)(nested_numpify(t) for t in tensors)
    return tensors.cpu().numpy()


def nested_detach(tensors):
    "Detach `tensors` (even if it's a nested list/tuple of tensors)."
    if isinstance(tensors, (list, tuple)):
        return type(tensors)(nested_detach(t) for t in tensors)
    return tensors.detach()


def nested_xla_mesh_reduce(tensors, name):
    if is_torch_tpu_available():
        import torch_xla.core.xla_model as xm

        if isinstance(tensors, (list, tuple)):
            return type(tensors)(nested_xla_mesh_reduce(t, f"{name}_{i}") for i, t in enumerate(tensors))
        return xm.mesh_reduce(name, tensors, torch.cat)
    else:
        raise ImportError("Torch xla must be installed to use `nested_xla_mesh_reduce`")


def distributed_concat(tensor: "torch.Tensor", num_total_examples: Optional[int] = None) -> torch.Tensor:
    try:
        if isinstance(tensor, (tuple, list)):
            return type(tensor)(distributed_concat(t, num_total_examples) for t in tensor)
        output_tensors = [tensor.clone() for _ in range(dist.get_world_size())]
        dist.all_gather(output_tensors, tensor)
        concat = torch.cat(output_tensors, dim=0)

        # truncate the dummy elements added by SequentialDistributedSampler
        if num_total_examples is not None:
            concat = concat[:num_total_examples]
        return concat
    except AssertionError:
        raise AssertionError("Not currently using distributed training")


def distributed_broadcast_scalars(
    scalars: List[Union[int, float]], num_total_examples: Optional[int] = None
) -> torch.Tensor:
    try:
        tensorized_scalar = torch.tensor(scalars).cuda()
        output_tensors = [tensorized_scalar.clone() for _ in range(dist.get_world_size())]
        dist.all_gather(output_tensors, tensorized_scalar)
        concat = torch.cat(output_tensors, dim=0)

        # truncate the dummy elements added by SequentialDistributedSampler
        if num_total_examples is not None:
            concat = concat[:num_total_examples]
        return concat
    except AssertionError:
        raise AssertionError("Not currently using distributed training")


def reissue_pt_warnings(caught_warnings):
    # Reissue warnings that are not the SAVE_STATE_WARNING
    if len(caught_warnings) > 1:
        for w in caught_warnings:
            if w.category != UserWarning or w.message != SAVE_STATE_WARNING:
                warnings.warn(w.message, w.category)


@contextmanager
def torch_distributed_zero_first(local_rank: int):
    """
    Decorator to make all processes in distributed training wait for each local_master to do something.

    Args:
        local_rank (:obj:`int`): The rank of the local process.
    """
    if local_rank not in [-1, 0]:
        dist.barrier()
    yield
    if local_rank == 0:
        dist.barrier()


class DistributedSamplerWithLoop(DistributedSampler):
    """
    Like a :obj:torch.utils.data.distributed.DistributedSampler` but loops at the end back to the beginning of the
    shuffled samples to make each process have a round multiple of batch_size samples.

    Args:
        dataset (:obj:`torch.utils.data.Dataset`):
            Dataset used for sampling.
        batch_size (:obj:`int`):
            The batch size used with this sampler
        kwargs:
            All other keyword arguments passed to :obj:`DistributedSampler`.
    """

    def __init__(self, dataset, batch_size, **kwargs):
        super().__init__(dataset, **kwargs)
        self.batch_size = batch_size

    def __iter__(self):
        indices = list(super().__iter__())
        remainder = 0 if len(indices) % self.batch_size == 0 else self.batch_size - len(indices) % self.batch_size
        # DistributedSampler already added samples from the beginning to make the number of samples a round multiple
        # of the world size, so we skip those.
        start_remainder = 1 if self.rank < len(self.dataset) % self.num_replicas else 0
        indices += indices[start_remainder : start_remainder + remainder]
        return iter(indices)


class SequentialDistributedSampler(Sampler):
    """
    Distributed Sampler that subsamples indices sequentially, making it easier to collate all results at the end.

    Even though we only use this sampler for eval and predict (no training), which means that the model params won't
    have to be synced (i.e. will not hang for synchronization even if varied number of forward passes), we still add
```

```python
    extra samples to the sampler to make it evenly divisible (like in `DistributedSampler`) to make it easy to `gather`
    or `reduce` resulting tensors at the end of the loop.
    """

    def __init__(self, dataset, num_replicas=None, rank=None, batch_size=None):
        warnings.warn(
            "SequentialDistributedSampler is deprecated and will be removed in v5 of Transformers.",
            FutureWarning,
        )
        if num_replicas is None:
            if not dist.is_available():
                raise RuntimeError("Requires distributed package to be available")
            num_replicas = dist.get_world_size()
        if rank is None:
            if not dist.is_available():
                raise RuntimeError("Requires distributed package to be available")
            rank = dist.get_rank()
        self.dataset = dataset
        self.num_replicas = num_replicas
        self.rank = rank
        num_samples = len(self.dataset)
        # Add extra samples to make num_samples a multiple of batch_size if passed
        if batch_size is not None:
            self.num_samples = int(math.ceil(num_samples / (batch_size * num_replicas))) * batch_size
        else:
            self.num_samples = int(math.ceil(num_samples / num_replicas))
        self.total_size = self.num_samples * self.num_replicas
        self.batch_size = batch_size

    def __iter__(self):
        indices = list(range(len(self.dataset)))

        # add extra samples to make it evenly divisible
        indices += indices[: (self.total_size - len(indices))]
        assert (
            len(indices) == self.total_size
        ), f"Indices length {len(indices)} and total size {self.total_size} mismatched"

        # subsample
        indices = indices[self.rank * self.num_samples : (self.rank + 1) * self.num_samples]
        assert (
            len(indices) == self.num_samples
        ), f"Indices length {len(indices)} and sample number {self.num_samples} mismatched"

        return iter(indices)

    def __len__(self):
        return self.num_samples


def get_tpu_sampler(dataset: torch.utils.data.dataset.Dataset, bach_size: int):
    if xm.xrt_world_size() <= 1:
        return RandomSampler(dataset)
    return DistributedSampler(dataset, num_replicas=xm.xrt_world_size(), rank=xm.get_ordinal())


def nested_new_like(arrays, num_samples, padding_index=-100):
    """Create the same nested structure as `arrays` with a first dimension always at `num_samples`."""
    if isinstance(arrays, (list, tuple)):
        return type(arrays)(nested_new_like(x, num_samples) for x in arrays)
    return np.full_like(arrays, padding_index, shape=(num_samples, *arrays.shape[1:]))


def expand_like(arrays, new_seq_length, padding_index=-100):
    """Expand the `arrays` so that the second dimension grows to `new_seq_length`. Uses `padding_index` for padding."""
    result = np.full_like(arrays, padding_index, shape=(arrays.shape[0], new_seq_length) + arrays.shape[2:])
    result[:, : arrays.shape[1]] = arrays
    return result


def nested_truncate(tensors, limit):
    "Truncate `tensors` at `limit` (even if it's a nested list/tuple of tensors)."
    if isinstance(tensors, (list, tuple)):
        return type(tensors)(nested_truncate(t, limit) for t in tensors)
    return tensors[:limit]


class DistributedTensorGatherer:
    """
    A class responsible for properly gathering tensors (or nested list/tuple of tensors) on the CPU by chunks.

    If our dataset has 16 samples with a batch size of 2 on 3 processes and we gather then transfer on CPU at every
    step, our sampler will generate the following indices:

        :obj:`[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0, 1]`

    to get something of size a multiple of 3 (so that each process gets the same dataset length). Then process 0, 1 and
    2 will be responsible of making predictions for the following samples:

        - P0: :obj:`[0, 1, 2, 3, 4, 5]`
        - P1: :obj:`[6, 7, 8, 9, 10, 11]`
        - P2: :obj:`[12, 13, 14, 15, 0, 1]`

    The first batch treated on each process will be

        - P0: :obj:`[0, 1]`
        - P1: :obj:`[6, 7]`
        - P2: :obj:`[12, 13]`

    So if we gather at the end of the first batch, we will get a tensor (nested list/tuple of tensor) corresponding to
    the following indices:

        :obj:`[0, 1, 6, 7, 12, 13]`

    If we directly concatenate our results without taking any precautions, the user will then get the predictions for
    the indices in this order at the end of the prediction loop:

        :obj:`[0, 1, 6, 7, 12, 13, 2, 3, 8, 9, 14, 15, 4, 5, 10, 11, 0, 1]`

    For some reason, that's not going to roll their boat. This class is there to solve that problem.

    Args:

        world_size (:obj:`int`):
            The number of processes used in the distributed training.
        num_samples (:obj:`int`):
            The number of samples in our dataset.
        make_multiple_of (:obj:`int`, `optional`):
            If passed, the class assumes the datasets passed to each process are made to be a multiple of this argument
            (by adding samples).
```

```python
        padding_index (:obj:`int`, `optional`, defaults to -100):
            The padding index to use if the arrays don't all have the same sequence length.
    """

    def __init__(self, world_size, num_samples, make_multiple_of=None, padding_index=-100):
        warnings.warn(
            "DistributedTensorGatherer is deprecated and will be removed in v5 of Transformers.",
            FutureWarning,
        )
        self.world_size = world_size
        self.num_samples = num_samples
        total_size = world_size if make_multiple_of is None else world_size * make_multiple_of
        self.total_samples = int(np.ceil(num_samples / total_size)) * total_size
        self.process_length = self.total_samples // world_size
        self._storage = None
        self._offsets = None
        self.padding_index = padding_index

    def add_arrays(self, arrays):
        """
        Add :obj:`arrays` to the internal storage, Will initialize the storage to the full size at the first arrays
        passed so that if we're bound to get an OOM, it happens at the beginning.
        """
        if arrays is None:
            return
        if self._storage is None:
            self._storage = nested_new_like(arrays, self.total_samples, padding_index=self.padding_index)
            self._offsets = list(range(0, self.total_samples, self.process_length))

        slice_len, self._storage = self._nested_set_tensors(self._storage, arrays)
        for i in range(self.world_size):
            self._offsets[i] += slice_len

    def _nested_set_tensors(self, storage, arrays):
        if isinstance(arrays, (list, tuple)):
            result = [self._nested_set_tensors(x, y) for x, y in zip(storage, arrays)]
            return result[0][0], type(arrays)(r[1] for r in result)
        assert (
            arrays.shape[0] % self.world_size == 0
        ), f"Arrays passed should all have a first dimension multiple of {self.world_size}, found {arrays.shape[0]}."

        slice_len = arrays.shape[0] // self.world_size
        for i in range(self.world_size):
            if len(arrays.shape) == 1:
                storage[self._offsets[i] : self._offsets[i] + slice_len] = arrays[i * slice_len : (i + 1) * slice_len]
            else:
                # Expand the array on the fly if needed.
                if len(storage.shape) > 1 and storage.shape[1] < arrays.shape[1]:
                    storage = expand_like(storage, arrays.shape[1], padding_index=self.padding_index)
                storage[self._offsets[i] : self._offsets[i] + slice_len, : arrays.shape[1]] = arrays[
                    i * slice_len : (i + 1) * slice_len
                ]
        return slice_len, storage

    def finalize(self):
        """
        Return the properly gathered arrays and truncate to the number of samples (since the sampler added some extras
        to get each process a dataset of the same length).
        """
        if self._storage is None:
            return
        if self._offsets[0] != self.process_length:
            logger.warning("Not all data has been set. Are you sure you passed all values?")
        return nested_truncate(self._storage, self.num_samples)

from typing import Optional
from matplotlib import pyplot as plt
import numpy as np
from numpy.typing import NDArray
from pydantic import BaseModel, Extra, Field, validate_arguments


class FCM(BaseModel):
    """Fuzzy C-means Model
    Attributes:
        n_clusters (int): The number of clusters to form as well as the number
        of centroids to generate by the fuzzy C-means.
        max_iter (int): Maximum number of iterations of the fuzzy C-means
        algorithm for a single run.
        m (float): Degree of fuzziness: $m \in (1, \infty)$.
        error (float): Relative tolerance with regards to Frobenius norm of
        the difference
        in the cluster centers of two consecutive iterations to declare
        convergence.
        random_state (Optional[int]): Determines random number generation for
        centroid initialization.
        Use an int to make the randomness deterministic.
        trained (bool): Variable to store whether or not the model has been
        trained.
    Returns:
        FCM: A FCM model.
    Raises:
        ReferenceError: If called without the model being trained
    """

    class Config:
        extra = Extra.allow
        arbitrary_types_allowed = True

    n_clusters: int = Field(5, ge=1, le=100)
    max_iter: int = Field(150, ge=1, le=1000)
    m: float = Field(2.0, ge=1.0)
    error: float = Field(1e-5, ge=1e-9)
    random_state: Optional[int] = None
    trained: bool = Field(False, const=True)

    @validate_arguments(config=dict(arbitrary_types_allowed=True))
    def fit(self, X: NDArray) -> None:
        """Train the fuzzy-c-means model
        Args:
            X (NDArray): Training instances to cluster.
        """
        self.rng = np.random.default_rng(self.random_state)
        n_samples = X.shape[0]
        self.u = self.rng.uniform(size=(n_samples, self.n_clusters))
        self.u = self.u / np.tile(
            self.u.sum(axis=1)[np.newaxis].T, self.n_clusters
        )
        for _ in range(self.max_iter):
            u_old = self.u.copy()
```

```python
            self._centers = FCM._next_centers(X, self.u, self.m)
            self.u = self.soft_predict(X)
            # Stopping rule
            if np.linalg.norm(self.u - u_old) < self.error:
                break
        self.trained = True

    @validate_arguments(config=dict(arbitrary_types_allowed=True))
    def soft_predict(self, X: NDArray) -> NDArray:
        """Soft predict of FCM
        Args:
            X (NDArray): New data to predict.
        Returns:
            NDArray: Fuzzy partition array, returned as an array with
            n_samples rows and n_clusters columns.
        """
        temp = FCM._dist(X, self._centers) ** (2 / (self.m - 1))
        denominator_ = temp.reshape((X.shape[0], 1, -1)).repeat(
            temp.shape[-1], axis=1
        )
        denominator_ = temp[:, :, np.newaxis] / denominator_
        return 1 / denominator_.sum(2)

    @validate_arguments(config=dict(arbitrary_types_allowed=True))
    def predict(self, X: NDArray) -> NDArray:
        """Predict the closest cluster each sample in X belongs to.
        Args:
            X (NDArray): New data to predict.
        Raises:
            ReferenceError: If it called without the model being trained.
        Returns:
            NDArray: Index of the cluster each sample belongs to.
        """
        if self._is_trained():
            X = np.expand_dims(X, axis=0) if len(X.shape) == 1 else X
            return self.soft_predict(X).argmax(axis=-1)
        raise ReferenceError(
            "You need to train the model. Run `.fit()` method to this."
        )

    def _is_trained(self) -> bool:
        if self.trained:
            return True
        return False

    @staticmethod
    def _dist(A: NDArray, B: NDArray) -> NDArray:
        """Compute the euclidean distance two matrices"""
        return np.sqrt(np.einsum("ijk->ij", (A[:, None, :] - B) ** 2))

    @staticmethod
    def _next_centers(X: NDArray, u: NDArray, m: float):
        """Update cluster centers"""
        um = u**m
        return (X.T @ um / np.sum(um, axis=0)).T

    @property
    def centers(self) -> NDArray:
        if self._is_trained():
            return self._centers
        raise ReferenceError(
            "You need to train the model. Run `.fit()` method to this."
        )

    @property
    def partition_coefficient(self) -> float:
        """Partition coefficient
        Equation 12a of
        [this paper](https://doi.org/10.1016/0098-3004(84)90020-7).
        """
        if self._is_trained():
            return np.mean(self.u**2)
        raise ReferenceError(
            "You need to train the model. Run `.fit()` method to this."
        )
    @property
    def partition_coefficient_dev(self) -> float:
        if self._is_trained():
            return self.u**2
        raise ReferenceError(
            "You need to train the model. Run `.fit()` method to this."
        )

    @property
    def partition_entropy_coefficient(self):
        if self._is_trained():
            return -np.mean(self.u * np.log2(self.u))
        raise ReferenceError(
            "You need to train the model. Run `.fit()` method to this."
        )
    fcm.fit(BigBirdModel.Buffer())


@dataclass
class LabelSmoother:
    """
    Adds label-smoothing on a pre-computed output from a Transformers model.

    Args:
        epsilon (:obj:`float`, `optional`, defaults to 0.1):
            The label smoothing factor.
        ignore_index (:obj:`int`, `optional`, defaults to -100):
            The index in the labels to ignore when computing the loss.
    """

    epsilon: float = 0.1
    ignore_index: int = -100

    def __call__(self, model_output, labels):
        logits = model_output["logits"] if isinstance(model_output, dict) else model_output[0]
        log_probs = -torch.nn.functional.log_softmax(logits, dim=-1)
        if labels.dim() == log_probs.dim() - 1:
            labels = labels.unsqueeze(-1)

        padding_mask = labels.eq(self.ignore_index)
        # In case the ignore_index is -100, the gather will fail, so we replace labels by 0. The padding_mask
        # will ignore them in any case.
        labels.clamp_min_(0)
        nll_loss = log_probs.gather(dim=-1, index=labels)
        # works for fp16 input tensor too, by internally upcasting it to fp32
```

```python
        smoothed_loss = log_probs.sum(dim=-1, keepdim=True, dtype=torch.float32)

        nll_loss.masked_fill_(padding_mask, 0.0)
        smoothed_loss.masked_fill_(padding_mask, 0.0)

        # Take the mean over the label dimensions, then divide by the number of active elements (i.e. not-padded):
        num_active_elements = padding_mask.numel() - padding_mask.long().sum()
        nll_loss = nll_loss.sum() / num_active_elements
        smoothed_loss = smoothed_loss.sum() / (num_active_elements * log_probs.shape[-1])
        return (1 - self.epsilon) * nll_loss + self.epsilon * smoothed_loss+ torch.var([torch.Tensor(min(1-partition_coefficient_dev[1],partition_coefficient_dev[0])), torch.Ter



def get_length_grouped_indices(lengths, batch_size, mega_batch_mult=None, generator=None):
    """
    Return a list of indices so that each slice of :obj:`batch_size` consecutive indices correspond to elements of
    similar lengths. To do this, the indices are:

    - randomly permuted
    - grouped in mega-batches of size :obj:`mega_batch_mult * batch_size`
    - sorted by length in each mega-batch

    The result is the concatenation of all mega-batches, with the batch of :obj:`batch_size` containing the element of
    maximum length placed first, so that an OOM happens sooner rather than later.
    """
    # Default for mega_batch_mult: 50 or the number to get 4 megabatches, whichever is smaller.
    if mega_batch_mult is None:
        mega_batch_mult = min(len(lengths) // (batch_size * 4), 50)
        # Just in case, for tiny datasets
        if mega_batch_mult == 0:
            mega_batch_mult = 1

    # We need to use torch for the random part as a distributed sampler will set the random seed for torch.
    indices = torch.randperm(len(lengths), generator=generator)
    megabatch_size = mega_batch_mult * batch_size
    megabatches = [indices[i : i + megabatch_size].tolist() for i in range(0, len(lengths), megabatch_size)]
    megabatches = [list(sorted(megabatch, key=lambda i: lengths[i], reverse=True)) for megabatch in megabatches]

    # The rest is to get the biggest batch first.
    # Since each megabatch is sorted by descending length, the longest element is the first
    megabatch_maximums = [lengths[megabatch[0]] for megabatch in megabatches]
    max_idx = torch.argmax(torch.tensor(megabatch_maximums)).item()
    # Switch to put the longest element in first position
    megabatches[0][0], megabatches[max_idx][0] = megabatches[max_idx][0], megabatches[0][0]

    return sum(megabatches, [])


class LengthGroupedSampler(Sampler):
    r"""
    Sampler that samples indices in a way that groups together features of the dataset of roughly the same length while
    keeping a bit of randomness.
    """

    def __init__(
        self,
        dataset: Dataset,
        batch_size: int,
        lengths: Optional[List[int]] = None,
        model_input_name: Optional[str] = None,
    ):
        self.dataset = dataset
        self.batch_size = batch_size
        self.model_input_name = model_input_name if model_input_name is not None else "input_ids"
        if lengths is None:
            if not isinstance(dataset[0], dict) or self.model_input_name not in dataset[0]:
                raise ValueError(
                    "Can only automatically infer lengths for datasets whose items are dictionaries with an "
                    f"'{self.model_input_name}' key."
                )
            lengths = [len(feature[self.model_input_name]) for feature in dataset]
        self.lengths = lengths

    def __len__(self):
        return len(self.lengths)

    def __iter__(self):
        indices = get_length_grouped_indices(self.lengths, self.batch_size)
        return iter(indices)


class DistributedLengthGroupedSampler(DistributedSampler):
    r"""
    Distributed Sampler that samples indices in a way that groups together features of the dataset of roughly the same
    length while keeping a bit of randomness.
    """
    # Copied and adapted from PyTorch DistributedSampler.
    def __init__(
        self,
        dataset: Dataset,
        batch_size: int,
        num_replicas: Optional[int] = None,
        rank: Optional[int] = None,
        seed: int = 0,
        drop_last: bool = False,
        lengths: Optional[List[int]] = None,
        model_input_name: Optional[str] = None,
    ):
        if num_replicas is None:
            if not dist.is_available():
                raise RuntimeError("Requires distributed package to be available")
            num_replicas = dist.get_world_size()
        if rank is None:
            if not dist.is_available():
                raise RuntimeError("Requires distributed package to be available")
            rank = dist.get_rank()
        self.dataset = dataset
        self.batch_size = batch_size
        self.num_replicas = num_replicas
        self.rank = rank
        self.epoch = 0
        self.drop_last = drop_last
        # If the dataset length is evenly divisible by # of replicas, then there
        # is no need to drop any data, since the dataset will be split equally.
        if self.drop_last and len(self.dataset) % self.num_replicas != 0:
            # Split to nearest available length that is evenly divisible.
            # This is to ensure each rank receives the same amount of data when
            # using this Sampler.
            self.num_samples = math.ceil((len(self.dataset) - self.num_replicas) / self.num_replicas)
        else:
```

```python
            self.num_samples = math.ceil(len(self.dataset) / self.num_replicas)
        self.total_size = self.num_samples * self.num_replicas
        self.seed = seed
        self.model_input_name = model_input_name if model_input_name is not None else "input_ids"

        if lengths is None:
            if not isinstance(dataset[0], dict) or self.model_input_name not in dataset[0]:
                raise ValueError(
                    "Can only automatically infer lengths for datasets whose items are dictionaries with an "
                    f"'{self.model_input_name}' key."
                )
            lengths = [len(feature[self.model_input_name]) for feature in dataset]
        self.lengths = lengths

    def __iter__(self) -> Iterator:
        # Deterministically shuffle based on epoch and seed
        g = torch.Generator()
        g.manual_seed(self.seed + self.epoch)
        indices = get_length_grouped_indices(self.lengths, self.batch_size, generator=g)

        if not self.drop_last:
            # add extra samples to make it evenly divisible
            indices += indices[: (self.total_size - len(indices))]
        else:
            # remove tail of data to make it evenly divisible.
            indices = indices[: self.total_size]
        assert len(indices) == self.total_size

        # subsample
        indices = indices[self.rank : self.total_size : self.num_replicas]
        assert len(indices) == self.num_samples

        return iter(indices)


class ShardSampler(Sampler):
    """
    Sampler that shards batches between several processes. Dispatches indices batch by batch: on 2 processes with batch
    size 4, the first two batches are :obj:`[0, 1, 2, 3, 4, 5, 6, 7]` and :obj:`[8, 9, 10, 11, 12, 13, 14, 15]`, which
    shard into :obj:`[0, 1, 2, 3]` and :obj:`[8, 9, 10, 11]` for GPU-0 and :obj:`[4, 5, 6, 7]` and :obj:`[12, 13, 14,
    15]` for GPU-1.

    The sampler thus yields :obj:`[0, 1, 2, 3, 8, 9, 10, 11]` on GPU-0 and :obj:`[4, 5, 6, 7, 12, 13, 14, 15]` on
    GPU-1.
    """

    def __init__(
        self,
        dataset: Dataset,
        batch_size: int = 1,
        drop_last: bool = False,
        num_processes: int = 1,
        process_index: int = 0,
    ):
        self.dataset = dataset
        self.batch_size = batch_size
        self.drop_last = drop_last
        self.num_processes = num_processes
        self.process_index = process_index

        self.total_batch_size = total_batch_size = batch_size * num_processes

        num_batches = len(dataset) // total_batch_size if drop_last else math.ceil(len(dataset) / total_batch_size)
        self.total_num_samples = num_batches * total_batch_size

    def __iter__(self):
        indices = list(range(len(self.dataset)))

        # Add extra samples to make it evenly divisible. While loop is there in the edge case we have a tiny dataset
        # and it needs to be done several times.
        while len(indices) < self.total_num_samples:
            indices += indices[: (self.total_num_samples - len(indices))]

        result = []
        for batch_start in range(self.batch_size * self.process_index, self.total_num_samples, self.total_batch_size):
            result += indices[batch_start : batch_start + self.batch_size]

        return iter(result)

    def __len__(self):
        # Each shard only sees a fraction of total_num_samples.
        return self.total_num_samples // self.num_processes


class IterableDatasetShard(IterableDataset):
    """
    Wraps a PyTorch :obj:`IterableDataset` to generate samples for one of the processes only. Instances of this class
    will always yield a number of samples that is a round multiple of the actual batch size (which is :obj:`batch_size
    x num_processes`). Depending on the value of the :obj:`drop_last` attribute, it will either stop the iteration at
    the first batch that would be too small or loop with indices from the beginning.

    On two processes with an iterable dataset yielding of :obj:`[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]` with a batch
    size of 2:

    - the shard on process 0 will yield :obj:`[0, 1, 4, 5, 8, 9]` so will see batches :obj:`[0, 1]`, :obj:`[4, 5]`,
      :obj:`[8, 9]`
    - the shard on process 1 will yield :obj:`[2, 3, 6, 7, 10, 11]` so will see batches :obj:`[2, 3]`, :obj:`[6, 7]`,
      :obj:`[10, 11]`

    .. warning:

        If your IterableDataset implements some randomization that needs to be applied the same way on all processes
        (for instance, a shuffling), you should use a :obj:`torch.Generator` in a :obj:`generator` attribute of the
        :obj:`dataset` to generate your random numbers and call the
        :meth:`~transformers.trainer_pt_utils.IterableDatasetShard.set_epoch` method of this object. It will set the
        seed of this :obj:`generator` to :obj:`seed + epoch` on all processes before starting the iteration.
        Alternatively, you can also implement a :obj:`set_epoch()` method in your iterable dataset to deal with this.


    Args:
        dataset (:obj:`torch.utils.data.dataset.IterableDataset`):
            The batch sampler to split in several shards.
        batch_size (:obj:`int`, `optional`, defaults to 1):
            The size of the batches per shard.
        drop_last (:obj:`bool`, `optional`, defaults to :obj:`False`):
            Whether or not to drop the last incomplete batch or complete the last batches by using the samples from the
            beginning.
        num_processes (:obj:`int`, `optional`, defaults to 1):
            The number of processes running concurrently.
        process_index (:obj:`int`, `optional`, defaults to 0):
```

```
            The index of the current process.
        seed (:obj:`int`, `optional`, defaults to 0):
            A random seed that will be used for the random number generation in
            :meth:`~transformers.trainer_pt_utils.IterableDatasetShard.set_epoch`.
    """

    def __init__(
        self,
        dataset: IterableDataset,
        batch_size: int = 1,
        drop_last: bool = False,
        num_processes: int = 1,
        process_index: int = 0,
        seed: int = 0,
    ):
        self.dataset = dataset
        self.batch_size = batch_size
        self.drop_last = drop_last
        self.num_processes = num_processes
        self.process_index = process_index
        self.seed = seed
        self.epoch = 0
        self.num_examples = 0

    def set_epoch(self, epoch):
        self.epoch = epoch
        if hasattr(self.dataset, "set_epoch"):
            self.dataset.set_epoch(epoch)

    def __iter__(self):
        self.num_examples = 0
        if (
            not hasattr(self.dataset, "set_epoch")
            and hasattr(self.dataset, "generator")
            and isinstance(self.dataset.generator, torch.Generator)
        ):
            self.dataset.generator.manual_seed(self.seed + self.epoch)
        real_batch_size = self.batch_size * self.num_processes
        process_slice = range(self.process_index * self.batch_size, (self.process_index + 1) * self.batch_size)

        first_batch = None
        current_batch = []
        for element in self.dataset:
            self.num_examples += 1
            current_batch.append(element)
            # Wait to have a full batch before yielding elements.
            if len(current_batch) == real_batch_size:
                for i in process_slice:
                    yield current_batch[i]
                if first_batch is None:
                    first_batch = current_batch.copy()
                current_batch = []

        # Finished if drop_last is True, otherwise complete the last batch with elements from the beginning.
        if not self.drop_last and len(current_batch) > 0:
            if first_batch is None:
                first_batch = current_batch.copy()
            while len(current_batch) < real_batch_size:
                current_batch += first_batch
            for i in process_slice:
                yield current_batch[i]


# In order to keep `trainer.py` compact and easy to understand, place any secondary PT Trainer
# helper methods here


def _get_learning_rate(self):
    if self.deepspeed:
        # with deepspeed's fp16 and dynamic loss scale enabled the optimizer/scheduler steps may
        # not run for the first few dozen steps while loss scale is too large, and thus during
        # that time `get_last_lr` will fail if called during that warm up stage, so work around it:
        try:
            last_lr = self.lr_scheduler.get_last_lr()[0]
        except AssertionError as e:
            if "need to call step" in str(e):
                logger.warning("tried to get lr value before scheduler/optimizer started stepping, returning lr=0")
                last_lr = 0
            else:
                raise
    else:
        last_lr = (
            # backward compatibility for pytorch schedulers
            self.lr_scheduler.get_last_lr()[0]
            if version.parse(torch.__version__) >= version.parse("1.4")
            else self.lr_scheduler.get_lr()[0]
        )
    return last_lr


def _secs2timedelta(secs):
    """
    convert seconds to hh:mm:ss.msec, msecs rounded to 2 decimals
    """

    msec = int(abs(secs - int(secs)) * 100)
    return f"{datetime.timedelta(seconds=int(secs))}.{msec:02d}"


def metrics_format(self, metrics: Dict[str, float]) -> Dict[str, float]:
    """
    Reformat Trainer metrics values to a human-readable format

    Args:
        metrics (:obj:`Dict[str, float]`):
            The metrics returned from train/evaluate/predict

    Returns:
        metrics (:obj:`Dict[str, float]`): The reformatted metrics
    """

    metrics_copy = metrics.copy()
    for k, v in metrics_copy.items():
        if "_mem_" in k:
            metrics_copy[k] = f"{ v >> 20 }MB"
        elif "_runtime" in k:
            metrics_copy[k] = _secs2timedelta(v)
        elif k == "total_flos":
            metrics_copy[k] = f"{ int(v) >> 30 }GF"
        elif type(metrics_copy[k]) == float:
```

```
            metrics_copy[k] = round(v, 4)

    return metrics_copy


def log_metrics(self, split, metrics):
    """
    Log metrics in a specially formatted way

    Under distributed environment this is done only for a process with rank 0.

    Args:
        split (:obj:`str`):
            Mode/split name: one of ``train``, ``eval``, ``test``
        metrics (:obj:`Dict[str, float]`):
            The metrics returned from train/evaluate/predictmetrics: metrics dict

    Notes on memory reports:

    In order to get memory usage report you need to install ``psutil``. You can do that with ``pip install psutil``.

    Now when this method is run, you will see a report that will include: ::

        init_mem_cpu_alloc_delta   =     1301MB
        init_mem_cpu_peaked_delta  =      154MB
        init_mem_gpu_alloc_delta   =      230MB
        init_mem_gpu_peaked_delta  =        0MB
        train_mem_cpu_alloc_delta  =     1345MB
        train_mem_cpu_peaked_delta =        0MB
        train_mem_gpu_alloc_delta  =      693MB
        train_mem_gpu_peaked_delta =        7MB

    **Understanding the reports:**

    - the first segment, e.g., ``train__``, tells you which stage the metrics are for. Reports starting with ``init_``
      will be added to the first stage that gets run. So that if only evaluation is run, the memory usage for the
      ``__init__`` will be reported along with the ``eval_`` metrics.
    - the third segment, is either ``cpu`` or ``gpu``, tells you whether it's the general RAM or the gpu0 memory
      metric.
    - ``*_alloc_delta`` - is the difference in the used/allocated memory counter between the end and the start of the
      stage - it can be negative if a function released more memory than it allocated.
    - ``*_peaked_delta`` - is any extra memory that was consumed and then freed - relative to the current allocated
      memory counter - it is never negative. When you look at the metrics of any stage you add up ``alloc_delta`` +
      ``peaked_delta`` and you know how much memory was needed to complete that stage.

    The reporting happens only for process of rank 0 and gpu 0 (if there is a gpu). Typically this is enough since the
    main process does the bulk of work, but it could be not quite so if model parallel is used and then other GPUs may
    use a different amount of gpu memory. This is also not the same under DataParallel where gpu0 may require much more
    memory than the rest since it stores the gradient and optimizer states for all participating GPUS. Perhaps in the
    future these reports will evolve to measure those too.

    The CPU RAM metric measures RSS (Resident Set Size) includes both the memory which is unique to the process and the
    memory shared with other processes. It is important to note that it does not include swapped out memory, so the
    reports could be imprecise.

    The CPU peak memory is measured using a sampling thread. Due to python's GIL it may miss some of the peak memory if
    that thread didn't get a chance to run when the highest memory was used. Therefore this report can be less than
    reality. Using ``tracemalloc`` would have reported the exact peak memory, but it doesn't report memory allocations
    outside of python. So if some C++ CUDA extension allocated its own memory it won't be reported. And therefore it
    was dropped in favor of the memory sampling approach, which reads the current process memory usage.

    The GPU allocated and peak memory reporting is done with ``torch.cuda.memory_allocated()`` and
    ``torch.cuda.max_memory_allocated()``. This metric reports only "deltas" for pytorch-specific allocations, as
    ``torch.cuda`` memory management system doesn't track any memory allocated outside of pytorch. For example, the
    very first cuda call typically loads CUDA kernels, which may take from 0.5 to 2GB of GPU memory.

    Note that this tracker doesn't account for memory allocations outside of :class:`~transformers.Trainer`'s
    ``__init__``, ``train``, ``evaluate`` and ``predict`` calls.

    Because ``evaluation`` calls may happen during ``train``, we can't handle nested invocations because
    ``torch.cuda.max_memory_allocated`` is a single counter, so if it gets reset by a nested eval call, ``train``'s
    tracker will report incorrect info. If this `pytorch issue <https://github.com/pytorch/pytorch/issues/16266>`__
    gets resolved it will be possible to change this class to be re-entrant. Until then we will only track the outer
    level of ``train``, ``evaluate`` and ``predict`` methods. Which means that if ``eval`` is called during ``train``,
    it's the latter that will account for its memory usage and that of the former.

    This also means that if any other tool that is used along the :class:`~transformers.Trainer` calls
    ``torch.cuda.reset_peak_memory_stats``, the gpu peak memory stats could be invalid. And the
    :class:`~transformers.Trainer` will disrupt the normal behavior of any such tools that rely on calling
    ``torch.cuda.reset_peak_memory_stats`` themselves.

    For best performance you may want to consider turning the memory profiling off for production runs.
    """
    if not self.is_world_process_zero():
        return

    logger.info(f"***** {split} metrics *****")
    metrics_formatted = self.metrics_format(metrics)
    k_width = max(len(str(x)) for x in metrics_formatted.keys())
    v_width = max(len(str(x)) for x in metrics_formatted.values())
    for key in sorted(metrics_formatted.keys()):
        logger.info(f"  {key: <{k_width}} = {metrics_formatted[key]:>{v_width}}")


def save_metrics(self, split, metrics, combined=True):
    """
    Save metrics into a json file for that split, e.g. ``train_results.json``.

    Under distributed environment this is done only for a process with rank 0.

    Args:
        split (:obj:`str`):
            Mode/split name: one of ``train``, ``eval``, ``test``, ``all``
        metrics (:obj:`Dict[str, float]`):
            The metrics returned from train/evaluate/predict
        combined (:obj:`bool`, `optional`, defaults to :obj:`True`):
            Creates combined metrics by updating ``all_results.json`` with metrics of this call

    To understand the metrics please read the docstring of :meth:`~transformers.Trainer.log_metrics`. The only
    difference is that raw unformatted numbers are saved in the current method.

    """
    if not self.is_world_process_zero():
        return

    path = os.path.join(self.args.output_dir, f"{split}_results.json")
    with open(path, "w") as f:
        json.dump(metrics, f, indent=4, sort_keys=True)

    if combined:
```

```python
        path = os.path.join(self.args.output_dir, "all_results.json")
        if os.path.exists(path):
            with open(path, "r") as f:
                all_metrics = json.load(f)
        else:
            all_metrics = {}

        all_metrics.update(metrics)
        with open(path, "w") as f:
            json.dump(all_metrics, f, indent=4, sort_keys=True)


def save_state(self):
    """
    Saves the Trainer state, since Trainer.save_model saves only the tokenizer with the model

    Under distributed environment this is done only for a process with rank 0.
    """
    if not self.is_world_process_zero():
        return

    path = os.path.join(self.args.output_dir, "trainer_state.json")
    self.state.save_to_json(path)


def get_parameter_names(model, forbidden_layer_types):
    """
    Returns the names of the model parameters that are not inside a forbidden layer.
    """
    result = []
    for name, child in model.named_children():
        result += [
            f"{name}.{n}"
            for n in get_parameter_names(child, forbidden_layer_types)
            if not isinstance(child, tuple(forbidden_layer_types))
        ]
    # Add model specific parameters (defined with nn.Parameter) since they are not in any child.
    result += list(model._parameters.keys())
    return result


if is_sagemaker_mp_enabled():
    import smdistributed.modelparallel.torch as smp

    @smp.step()
    def smp_forward_backward(model, inputs, gradient_accumulation_steps=1, scaler=None):
        with torch.cuda.amp.autocast(enabled=(scaler is not None)):
            outputs = model(**inputs)

        loss = outputs["loss"] if isinstance(outputs, dict) else outputs[0]
        loss /= gradient_accumulation_steps
        if scaler is not None:
            loss = scaler.scale(loss).squeeze()

        model.backward(loss)
        return loss

    @smp.step()
    def smp_forward_only(model, inputs):
        return model(**inputs)

    def smp_gather(tensor):
        if isinstance(tensor, (list, tuple)):
            return type(tensor)(smp_gather(t) for t in tensor)
        elif isinstance(tensor, dict):
            return type(tensor)({k: smp_gather(v) for k, v in tensor.items()})
        elif not isinstance(tensor, torch.Tensor):
            raise TypeError(
                f"Can't gather the values of type {type(tensor)}, only of nested list/tuple/dicts of tensors."
            )
        all_tensors = smp.allgather(tensor, smp.CommGroup.DP_GROUP)
        return torch.cat([t.cpu() for t in all_tensors], dim=0)

    def smp_nested_concat(tensor):
        if isinstance(tensor, (list, tuple)):
            return type(tensor)(smp_nested_concat(t) for t in tensor)
        elif isinstance(tensor, dict):
            return type(tensor)({k: smp_nested_concat(v) for k, v in tensor.items()})
        # It doesn't seem possible to check here if `tensor` is a StepOutput because StepOutput lives in `smp.step`
        # which is also the name of the decorator so Python is confused.
        return tensor.concat().detach().cpu()
```