```
import gzip
import json
import os
import h5py
from typing import List, Tuple
import random

import numpy as np
import smdistributed.modelparallel.torch as smp
import torch


class WikiPretrainingDataset(torch.utils.data.Dataset):
    def __init__(self, input_file, max_pred_length):
        self.input_file = input_file
        self.max_pred_length = max_pred_length
        f = h5py.File(input_file, "r")
        keys = [
            "input_ids",
            "input_mask",
            "segment_ids",
            "masked_lm_positions",
            "masked_lm_ids",
            "next_sentence_labels",
        ]
        self.inputs = [np.asarray(f[key][:]) for key in keys]
        f.close()

    def __len__(self):
        "Denotes the total number of samples"
        return len(self.inputs[0])

    def __getitem__(self, index):
        [
            input_ids,
            input_mask,
            segment_ids,
            masked_lm_positions,
            masked_lm_ids,
            next_sentence_labels,
        ] = [
            torch.from_numpy(input[index].astype(np.int64))
            if indice < 5
            else
torch.from_numpy(np.asarray(input[index].astype(np.int64)))
            for indice, input in enumerate(self.inputs)
        ]

        masked_lm_labels = torch.ones(input_ids.shape, dtype=torch.long)
* -1
        index = self.max_pred_length
        # store number of  masked tokens in index
        padded_mask_indices = (masked_lm_positions ==
0).nonzero(as_tuple=False)
```

```
        if len(padded_mask_indices) != 0:
            index = padded_mask_indices[0].item()
        masked_lm_labels[masked_lm_positions[:index]] =
masked_lm_ids[:index]

        return [input_ids, segment_ids, input_mask, masked_lm_labels,
next_sentence_labels]


###### Load Openwebtext pretraining data ######
class OpenwebtextPretrainingDataset(torch.utils.data.Dataset):
    def __init__(self, input_paths: List[str], max_sequence_length=None,
zipped=True, use_last_file_only=False):
        self.input_paths = input_paths
        self.max_sequence_length = max_sequence_length
        self.zipped = zipped
        self.use_last_file_only = use_last_file_only

        self.__read_examples(self.input_paths)

    def __read_examples(self, paths: List[str]):

        self.input_data = []
        if self.zipped:
            if self.use_last_file_only:
                with gzip.open(paths[-1], "rt") as f:
                    self.input_data = [ln for _, ln in enumerate(f, 1)]
            else:
                for path in paths:
                    with gzip.open(path, "rt") as f:
                        self.input_data.extend([ln for _, ln in
enumerate(f, 1)])
        else:
            if self.use_last_file_only:
                with open (paths[-1], "r") as f:
                    self.input_data = [ln for ln in f]
            else:
                for path in paths:
                    with open (path, "r") as f:
                        self.input_data.extend([ln for ln in f])

        # print(f'__Finished building pretraining dataset with
{self.iids.shape[0]} rows__')

    def __len__(self) -> int:
        return len(self.input_data)

    def __getitem__(self, index: int) -> Tuple[torch.Tensor,
torch.Tensor]:
        obj = json.loads(self.input_data[index])
        iids = torch.tensor(obj["input_ids"], dtype=torch.long)
        attns = torch.tensor(obj["attention_mask"], dtype=torch.long)
        self.actual_sequence_length = len(obj["input_ids"])
        if self.actual_sequence_length > self.max_sequence_length:
            s_idx = np.random.randint(0, self.actual_sequence_length - 
self.max_sequence_length)
            e_idx = s_idx + self.max_sequence_length
            iids = iids[s_idx:e_idx]
            attns = attns[s_idx:e_idx]
        return iids, attns


class DummyDataset(torch.utils.data.dataset.Dataset):
    def __init__(self, length, data_type="openwebtext"):
        if data_type == "openwebtext":
            self.batch = (torch.Tensor(0), torch.Tensor(0))
        elif data_type == "wiki":
            self.batch = (torch.Tensor(0), torch.Tensor(0), 
torch.Tensor(0), torch.Tensor(0), torch.Tensor(0))
        self.length = length

    def __getitem__(self, index):
        return self.batch

    def __len__(self):
        return self.length


def create_pretraining_dataloader(
    input_paths: List[str],
    batch_size: int,
    max_sequence_length: int,
    seed: int,
    dp_rank: int,
    dp_size: int,
    shuffle: bool = False,
    zipped: bool = True,
    use_last_file_only: bool = False,
    data_type: str = "openwebtext",
):
    if smp.pp_rank() == 0:
        if data_type == "openwebtext":
            data = OpenwebtextPretrainingDataset(
                input_paths=input_paths, 
max_sequence_length=max_sequence_length, zipped=zipped, 
use_last_file_only=use_last_file_only
            )
        elif data_type == "wiki":
            if len(input_paths) > 1:
                print(f"Wiki data only support single file when calling 
create_pretraining_dataloader, reading the first file instead..")
            data = WikiPretrainingDataset(input_file=input_paths[0], 
max_pred_length=max_sequence_length)
        else:
            raise ValueError(f"Unsupported data type {data_type}")
        sampler = torch.utils.data.DistributedSampler(
            data,
            shuffle=shuffle,
            seed=seed,
            rank=dp_rank,
            num_replicas=dp_size,
            drop_last=True,
        )
        dataloader = torch.utils.data.DataLoader(
            data,
            sampler=sampler,
            batch_size=batch_size,
            num_workers=0,
            pin_memory=True,
            drop_last=True,  # skip a batch in the training loop if the batch size is not divisible by the number of microbatches
        )
        smp.broadcast(len(dataloader), smp.PP_GROUP)
    else:
        data_len = smp.recv_from(0, smp.RankType.PP_RANK)
        dataset = DummyDataset(data_len * batch_size,
data_type=data_type)
        dataloader = torch.utils.data.DataLoader(dataset,
batch_size=batch_size, drop_last=True)

    return dataloader
```