```
import argparse
import collections
import logging
import math
import os
import re
import time
from concurrent.futures import ProcessPoolExecutor

import numpy as np
import smdistributed.modelparallel
import smdistributed.modelparallel.torch as smp
import torch
import torch.nn as nn
import torch.utils.data
import transformers
from data_pipeline import create_pretraining_dataloader
from fp16 import FP16_Module, FP16_Optimizer, load_fp16_optimizer,
save_fp16_optimizer
from learning_rates import AnnealingLR
from smdistributed.modelparallel.torch.nn import FusedLayerNorm as
LayerNorm
from smdistributed.modelparallel.torch.nn.huggingface.gpt2 import (
    translate_hf_state_dict_to_smdistributed,
    translate_state_dict_to_hf_gpt2,
)
from torch import optim
from torch.nn.parallel.distributed import DistributedDataParallel
from transformers import (
    CONFIG_MAPPING,
    MODEL_FOR_CAUSAL_LM_MAPPING,
    AutoConfig,
    AutoModelForCausalLM,
    AutoTokenizer,
    GPT2Config,
    default_data_collator,
    set_seed,
)
from transformers.trainer_utils import is_main_process

logger = logging.getLogger(__name__)


def get_learning_rate_scheduler(optimizer, args):

    # Add linear learning rate scheduler.
    if args.lr_decay_iters is not None:
        num_iters = args.lr_decay_iters
    else:
        num_iters = args.max_steps
    num_iters = max(1, num_iters)
    init_step = 0
    warmup_iter = args.warmup * num_iters
    plateau_iter = warmup_iter + args.plateau * num_iters
```

```
    lr_scheduler = AnnealingLR(
        optimizer,
        start_lr=args.lr,
        warmup_iter=warmup_iter,
        plateau_iter=plateau_iter,
        total_iters=num_iters,
        decay_style=args.lr_decay_style,
        last_iter=init_step,
        min_lr=args.min_lr,
        use_checkpoint_lr_scheduler=args.load_partial or args.load_full,
        override_lr_scheduler=False,
    )

    return lr_scheduler


def get_param_groups_by_weight_decay(module):
    weight_decay_params = {"params": []}
    no_weight_decay_params = {"params": [], "weight_decay": 0.0}
    param_ids = set()
    for module_ in module.modules():
        if isinstance(module_, LayerNorm):
            for p in list(module_._parameters.values()):
                if p is not None and id(p) not in param_ids:
                    no_weight_decay_params["params"].append(p)
                    param_ids.add(id(p))
        else:
            for n, p in list(module_._parameters.items()):
                if p is not None and n != "bias" and id(p) not in
param_ids:
                    weight_decay_params["params"].append(p)
                    param_ids.add(id(p))
            for n, p in list(module_._parameters.items()):
                if p is not None and n == "bias" and id(p) not in
param_ids:
                    no_weight_decay_params["params"].append(p)
                    param_ids.add(id(p))
    return weight_decay_params, no_weight_decay_params


# SMP modification: Define smp.step. Return any tensors needed outside.
@smp.step
def train_step(model, optimizer, input_ids, attention_mask, args):
    if args.logits_output:
        output = model(input_ids=input_ids,
attention_mask=attention_mask, labels=input_ids)
        loss = output["loss"]
    else:
        loss = model(input_ids=input_ids, attention_mask=attention_mask,
labels=input_ids)["loss"]
    if args.fp16:
        optimizer.backward(loss, update_master_grads=False)
    else:
        model.backward(loss)
```

```
    if args.logits_output:
        return output
    return loss


# SMP modification: Define smp.step. Return any tensors needed outside.
@smp.step
def test_step(model, input_ids, attention_mask):
    loss = model(input_ids=input_ids, attention_mask=attention_mask,
labels=input_ids)["loss"]
    return loss


def save_ckptsum(args, model, optimizer, filename):
    results = collections.defaultdict(dict)
    model_result = collections.defaultdict(dict)

    if args.fp16:
        from fp16.fp16util import register_optimizer_hooks

        register_optimizer_hooks(model)

    def _get_optimizer_result(optimizer_states):
        _optimizer_result = collections.defaultdict(dict)
        for param_idx, state in optimizer_states.items():
            for key, val in state.items():
                if isinstance(val, torch.Tensor):
                    _optimizer_result["tensors"][f"{param_idx}_{key}"] =
torch.sum(val)
                else:
                    _optimizer_result["scalars"][f"{param_idx}_{key}"] =
val
        return _optimizer_result

    if not args.shard_optimizer_state:
        optimizer_result =
_get_optimizer_result(optimizer.local_state_dict()["state"])
    else:
        local_state_dict = optimizer.local_state_dict()["state"]
        if smp.rdp_rank() == 0:
            optimizer_result = []
            for partial_local_state_dict in local_state_dict:

optimizer_result.append(_get_optimizer_result(partial_local_state_dict))

    for param_name, param in model.local_state_dict().items():
        if isinstance(param, torch.Tensor):
            model_result["tensors"][param_name] = torch.sum(param)
        else:
            model_result["scalars"][param_name] = param

    if smp.rdp_rank() == 0:
        results["optimizer"] = optimizer_result
        results["model"] = model_result
```

```
        smp.save(results, filename)


def load_and_verify_ckptsum(args, model, optimizer, filename):
    results = smp.load(filename)
    optimizer_result = (
        results["optimizer"]
        if not args.shard_optimizer_state
        else results["optimizer"][smp.rdp_rank()]
    )
    model_result = results["model"]

    def opt_check_fn(mod, opt):
        loaded_opt_states = (
            opt.orig_state_dict()["state"]
            if args.shard_optimizer_state
            else opt.local_state_dict()["state"]
        )
        for param_idx, state in loaded_opt_states.items():
            for key, val in state.items():
                if isinstance(val, torch.Tensor):
                    assert torch.isclose(
                        torch.sum(val),
optimizer_result["tensors"][f"{param_idx}_{key}"]
                    ), f"mismatch for param_idx: {param_idx}, key is
{key}"
                else:
                    assert (
                        val ==
optimizer_result["scalars"][f"{param_idx}_{key}"]
                    ), f"mismatch for param_idx: {param_idx}, key is
{key}"
        print("Optimizer save/load check passed successfully")

    def model_check_fn(mod, opt):
        for param_name, param in mod.local_state_dict().items():
            if isinstance(param, torch.Tensor):
                assert torch.isclose(
                    torch.sum(param), model_result["tensors"][param_name]
                ), f"mismatch for param_name: {param_name}"
            else:
                assert (
                    param == model_result["scalars"][param_name]
                ), f"mismatch for param_name: {param_name}"
        print("Model save/load check passed successfully")

    model.register_post_partition_hook(model_check_fn)
    model.register_post_step_hook(opt_check_fn)


def save(
    output_save_file,
    model,
    optimizer,
```

```
    lr_scheduler,
    model_config,
    num_params,
    total_steps,
    curr_train_path_index,
    args,
    partial=True,
    translate_to_hf=False,
    seq_length=1024,
    batch_idx=0,
):
    save_fn = save_fp16_optimizer
    save_dict = {
        "cli_args": args.__dict__,
        "num_params": num_params,
        "total_steps": total_steps,
        "curr_train_path_index": curr_train_path_index,
        "model_config": model_config,
        "batch_idx": batch_idx,
    }

    if lr_scheduler is not None:
        save_dict["lr_scheduler"] = lr_scheduler.state_dict()
    if partial:
        # SMP modification: check if using optimizer state sharding or
tensor parallelism
        if args.gather_if_shard > 0 or smp.rdp_rank() == 0:
            # if not gather the opt checkpoint, only save the model for
rdp rank 0
            save_dict["model"] = model.local_state_dict()
    else:
        model_state_dict = model.state_dict(gather_to_rank0=True)
        if smp.rank() == 0:
            save_dict["model"] = (
                translate_state_dict_to_hf_gpt2(model_state_dict,
seq_length)
                if translate_to_hf
                else model_state_dict
            )

    if args.fp16:
        if not partial and args.skip_full_optimizer:
            print("Skipping saving the final optimizer state")
        else:
            if args.shard_optimizer_state == 0 or partial:
                save_dict["optimizer"] = save_fn(args, model, optimizer,
partial=partial)
            else:
                print(
                    "Saving the full optimizer state does not work with
shard_optimizer_state > 0! Skipping..."
                )
    else:
        # fp32
```

```
        if partial:
            save_dict["optimizer"] = optimizer.local_state_dict()
        else:
            if not args.skip_full_optimizer:
                save_dict["optimizer"] = optimizer.state_dict()
            else:
                print("Skipping saving of full optimizer state")

    # SMP modification: criteria for checkpointing the zeroth rank for
    # pipeline parallelism, checkpointing the zeroth reduced data
parallel
    # rank for tensor parallelism, and preventing checkpointing if
optimizer
    # state sharding is enabled
    if not args.gather_if_shard or (smp.rdp_rank() == 0 and partial) or
smp.rank() == 0:
        smp.save(save_dict, output_save_file, partial=partial, v3=not
args.gather_if_shard)

    print(f"Finished checkpointing after {total_steps} steps:
{output_save_file}")


def load_model_and_optimizer(
    output_dir,
    model,
    optimizer,
    lr_scheduler,
    partial,
    args,
    translate_from_hf=False,
    seq_length=1024,
    load_model=True,
    load_optimizer=True,
    num_params=0,
):
    # Find longest-trained checkpoint
    re_pattern = f"trained_gpt_nparams-{num_params}_steps-
(?P<total_steps>\d+)\.pt"
    if partial:
        re_pattern += "_(?P<rank>\d+)"
    else:
        re_pattern += "$"

    ckpt_paths = sorted(
        [
            (int(re.match(re_pattern, p).group("total_steps")),
os.path.join(output_dir, p))
            for p in os.listdir(output_dir)
            if re.match(re_pattern, p)
        ],
        reverse=True,
    )
    if not ckpt_paths:
```

```
        raise Exception(
            f'No checkpoints could be found in "{output_dir}".
Candidates: {os.listdir(output_dir)}'
        )

    local_ckpt_path = ckpt_paths[0][1]

    if partial:
        # need to pass prefix without ranks to smp
        local_ckpt_path = local_ckpt_path.split(".pt")[0] + ".pt"

    if args.gather_if_shard > 0:
        # Should expect v2 checkpoint here
        checkpoint = smp.load(local_ckpt_path, partial=partial)
    else:
        # Loading separately for model and opt
        checkpoint =
torch.load(f"{local_ckpt_path}_{smp.pp_rank()}_{smp.tp_rank()}_0")
        if smp.rdp_rank() != 0:
            opt_checkpoint = torch.load(

f"{local_ckpt_path}_{smp.pp_rank()}_{smp.tp_rank()}_{smp.rdp_rank()}"
            )

    if load_model:
        checkpointed_model = (
            translate_hf_state_dict_to_smdistributed(checkpoint["model"],
seq_length)
            if translate_from_hf
            else checkpoint["model"]
        )
        model.load_state_dict(checkpointed_model,
same_partition_load=args.same_partition_load > 0)
        if lr_scheduler is not None:
            lr_scheduler.load_state_dict(checkpoint["lr_scheduler"])

    if load_optimizer:
        # Loading loss scale eagerly
        opt_state_dict = checkpoint["optimizer"]
        optimizer.loss_scaler = opt_state_dict["loss_scaler"]
        optimizer.loss_scaler.model = model
        optimizer.dynamic_loss_scale =
opt_state_dict["dynamic_loss_scale"]
        optimizer.overflow = opt_state_dict["overflow"]
        optimizer.first_closure_call_this_step =
opt_state_dict["first_closure_call_this_step"]

        def opt_load_hook(mod, opt):
            load_fn = load_fp16_optimizer
            if args.fp16:
                if not partial and args.skip_full_optimizer:
                    print(
                        "Skipping loading the final optimizer state, and
reloading master_params from model_params"
```

```
                    )
                    opt.reload_model_params()
                else:
                    load_fn(args, mod, opt, checkpoint, partial=partial)
            else:
                # fp32
                if not partial and args.skip_full_optimizer:
                    print("Skipping loading the final optimizer state")
                else:
                    opt.load_state_dict(checkpoint["optimizer"])

        model.register_post_step_hook(opt_load_hook)

    print(f'Loaded model from "{local_ckpt_path}"')

    batch_idx = 0
    if "batch_idx" in checkpoint:
        batch_idx = checkpoint["batch_idx"]

    return (
        model,
        optimizer,
        checkpoint["total_steps"],
        checkpoint["curr_train_path_index"],
        batch_idx,
    )


def delete_oldest_ckpt(args, delete_on_rank0_only=False):
    to_delete = smp.rank() == 0 if delete_on_rank0_only else
smp.local_rank() == 0
    if to_delete:
        re_pattern = "trained_gpt_nparams-(?P<num_params>\d+)_steps-
(?P<total_steps>\d+)\.pt"

        # partial
        re_pattern += "_(?P<pp_rank>\d+)_(?P<tp_rank>\d+)"

        paths_per_step = collections.defaultdict(list)

        for p in os.listdir(args.checkpoint_dir):
            if re.match(re_pattern, p):
                step = int(re.match(re_pattern, p).group("total_steps"))
                path = os.path.join(args.checkpoint_dir, p)
                paths_per_step[step].append(path)

        if paths_per_step:
            oldest_step = sorted(paths_per_step.keys())[0]
            num_parts = len(paths_per_step[oldest_step])
        if len(paths_per_step) > args.num_kept_checkpoints:
            # delete oldest step
                for p in paths_per_step[oldest_step]:
                    os.remove(p)
```

```
        # else We still haven't reached maximum number of checkpoints --
no need to delete older ones
    return None


def eval_model(model, dataloader, num_batches, use_wiki_data):
    model = model.eval()
    n_batches = 0
    loss = 0.0

    with torch.no_grad():
        for batch_idx, input_data in enumerate(dataloader):
            if use_wiki_data:
                input_ids, _, attention_mask, _, _ = input_data
            else:
                input_ids, attention_mask = input_data
            if batch_idx >= num_batches:
                break

            loss += test_step(model, input_ids,
attention_mask).reduce_mean()
            n_batches += 1

    if n_batches > 0:
        torch.distributed.all_reduce(loss,
group=smp.get_dp_process_group())
        loss /= smp.dp_size()
        loss /= n_batches
        loss = loss.item()
        ppl = math.exp(loss)
    else:
        loss = -1.0
        ppl = -1.0

    return loss, ppl


def train(
    model,
    optimizer,
    lr_scheduler,
    model_config,
    start_train_path_index,
    start_batch_index,
    num_params,
    total_steps,
    args,
):
    model.train()
    if args.parallel_proc_data_processing:
        pool = ProcessPoolExecutor(1)

    dp_rank = smp.dp_rank() if not args.prescaled_batch else
smp.rdp_rank()
```

```
    dp_size = smp.dp_size() if not args.prescaled_batch else
smp.rdp_size()
    data_type = "wiki" if args.use_wiki_data else "openwebtext"

    if args.use_wiki_data:
        train_paths = sorted(
            [
                os.path.join(args.training_dir, p)
                for p in os.listdir(args.training_dir)
                if os.path.isfile(os.path.join(args.training_dir, p)) and
"training" in p
            ]
        )
    else:
        if args.zipped_data > 0:
            file_extension = ".json.gz"
        else:
            file_extension = ".json"
        train_paths = sorted(
            [
                os.path.join(args.training_dir, p)
                for p in os.listdir(args.training_dir)
                if p.endswith(file_extension)
            ]
        )

    train_dataloader = create_pretraining_dataloader(
        [train_paths[start_train_path_index]],
        args.train_batch_size,
        args.max_context_width,
        seed=args.seed,
        dp_rank=dp_rank,
        dp_size=dp_size,
        shuffle=args.same_seed < 1,
        zipped=args.zipped_data > 0,
        use_last_file_only=args.fast_validation > 0,
        data_type=data_type,
    )

    if args.validation_freq is not None:
        # load all validation examples
        if smp.rank() == 0:
            print("Creating val dataloader")
        if args.use_wiki_data:
            val_paths = sorted(
                [
                    os.path.join(args.test_dir, p)
                    for p in os.listdir(args.test_dir)
                    if os.path.isfile(os.path.join(args.test_dir, p)) and
"testing" in p
                ]
            )

        else:
```

```
            if args.zipped_data > 0:
                file_extension = ".json.gz"
            else:
                file_extension = ".json"
            val_paths = sorted(
                [
                    os.path.join(args.test_dir, p)
                    for p in os.listdir(args.test_dir)
                    if p.endswith(file_extension)
                ]
            )
        val_dataloader = create_pretraining_dataloader(
            val_paths,
            args.val_batch_size,
            args.max_context_width,
            seed=args.seed,
            dp_rank=dp_rank,
            dp_size=dp_size,
            shuffle=True,
            zipped=args.zipped_data > 0,
            use_last_file_only=args.fast_validation > 0,
            data_type=data_type,
        )
        if smp.rank() == 0:
            print("Created val dataloader")

    start = time.time()
    throughput = None
    to_save = {"loss": [], "val_loss": []}
    loss_metric = 0

    def should_record():
        # only record the ranks that in the tp group that contains global
rank 0
        if smp.tp_size() > 1:
            tp_group = smp.get_tp_group()
            return 0 in tp_group
        else:
            return smp.rank() == 0

    # Set the same seed for computation
    set_seed(args.seed)

    for index in range(start_train_path_index, args.epochs *
len(train_paths)):
        next_train_path_index = (index + 1) % len(train_paths)
        curr_train_path_index = index % len(train_paths)

        if total_steps >= args.max_steps:
            break

        if args.parallel_proc_data_processing:
            dataset_future = pool.submit(
                create_pretraining_dataloader,
```

```
                [train_paths[next_train_path_index]],
                args.train_batch_size,
                args.max_context_width,
                seed=args.seed,
                dp_rank=dp_rank,
                dp_size=dp_size,
                shuffle=args.same_seed < 1,
                zipped=args.zipped_data > 0,
                use_last_file_only=args.fast_validation > 0,
                data_type=data_type,
            )

        if smp.rank() == 0:
            if args.use_wiki_data:
                print(f"Reading data from training path
{train_dataloader.dataset.input_file}")
            else:
                print(f"Reading data from training path
{train_dataloader.dataset.input_paths}")

        for batch_idx, input_data in enumerate(train_dataloader):
            if batch_idx < start_batch_index:
                if smp.rank() == 0:
                    print(
                        f"Resuming from saved batch index
{start_batch_index}, skipping batch {batch_idx}..."
                    )
                if start_batch_index == len(train_dataloader):
                    # If saving at the last batch of the file, read from
the next file
                    start_batch_index = 0
                    break
                continue
            else:
                start_batch_index = 0

            if args.use_wiki_data:
                input_ids, _, attention_mask, _, _ = input_data
            else:
                input_ids, attention_mask = input_data

            if total_steps >= args.max_steps:
                break

            step_start = time.time()

            if args.fp16:
                optimizer.zero_grad(set_grads_to_None=True)
            else:
                optimizer.zero_grad()

            if args.logits_output:
                train_output = train_step(model, optimizer, input_ids,
attention_mask, args)
```

```
                loss_mb = train_output["loss"]
                logits_mb = train_output["logits"]
                if smp.tp_size() > 1:
                    logits = torch.cat(tuple(logits_mb.outputs), dim=1)
                else:
                    logits = torch.cat(tuple(logits_mb.outputs), dim=0)
            else:
                # Return value, loss_mb is a StepOutput object
                loss_mb = train_step(model, optimizer, input_ids,
attention_mask, args)

            # SMP modification: Average the loss across microbatches.
            loss = loss_mb.reduce_mean()
            if not args.validation_freq:
                loss_metric = loss.item()

            if args.fp16:
                optimizer.update_master_grads()
                optimizer.clip_master_grads(args.grad_clip)
                optimizer.step()
                overflow = optimizer.overflow
            else:
                optimizer.step()

            if not (args.fp16 and overflow):
                lr_scheduler.step()

            if args.enable_memory_profiling > 0:
                memory_status(msg="After_opt_step")

            total_steps += 1
            time_elapsed = time.time() - start
            step_time = time.time() - step_start
            sample_processed = input_ids.shape[0] * dp_size
            throughput = sample_processed / step_time
            if smp.rank() == 0 and not total_steps % args.logging_freq:
                print(
                    f"({int(time_elapsed)}s), Batch {total_steps - 1}
Loss: {loss.item()}, Speed: {throughput} samples/sec"
                )

            # evaluate on validation
            if args.validation_freq and not (total_steps %
args.validation_freq):
                cur_state = np.random.get_state()
                model = model.eval()
                val_loss, val_ppl = eval_model(
                    model, val_dataloader, args.validation_batches,
args.use_wiki_data
                )
                if is_main_process(smp.rank()):
                    print(
                        f"({int(time.time()-start)}s) Batch {total_steps
- 1} Validation loss: {val_loss}"
```

```
                    )
                    print(
                        f"({int(time.time()-start)}s) Batch {total_steps
- 1} Validation perplexity: {val_ppl}"
                    )
                loss_metric = val_loss
                if args.logits_output:
                    to_save["val_loss"].append(val_loss)
                model = model.train()
                if args.preserve_np_state > 0:
                    np.random.set_state(cur_state)

            # checkpoint
            if not (total_steps % args.checkpoint_freq):
                base_path = f"trained_gpt_nparams-{num_params}_steps-
{total_steps}.pt"
                out_path = os.path.join(args.checkpoint_dir, base_path)
                total_ckpts = total_steps // args.checkpoint_freq

                delete_oldest_ckpt(args,
delete_on_rank0_only=args.use_fsx > 0)

                # save_or_verify_ckptsum if this is the last checkpoint
                if (args.save_or_verify_ckptsum and total_steps >=
args.max_steps) or (
                    (total_ckpts + 1) * args.checkpoint_freq
                ) > args.max_steps:
                    # Save optimizer and model tensor sums and scalars
before saving
                    save_ckptsum(
                        args,
                        model,
                        optimizer,
                        filename=os.path.join(args.model_dir,
"saved_partial_sum"),
                    )

                save(
                    out_path,
                    model,
                    optimizer,
                    lr_scheduler,
                    model_config,
                    num_params,
                    total_steps,
                    curr_train_path_index,
                    args,
                    partial=True,
                    batch_idx=batch_idx + 1,
                )

            if args.logits_output:
                to_save["loss"].append(loss.item())
```

```
        if total_steps >= args.max_steps:
            if should_record() and args.logits_output:
                to_save["logits"] = logits.detach().cpu()
                output_file = f"rank_{smp.rank()}_" + args.logits_output
                torch.save(to_save, os.path.join(args.model_dir,
output_file))
                print(f"logits and loss saved at
{os.path.join(args.model_dir, output_file)}")
            break

        del train_dataloader

        if args.parallel_proc_data_processing:
            s = time.time()
            train_dataloader = dataset_future.result(timeout=None)
            wait_time = time.time() - s
            if wait_time > 1:
                # TODO if this happens, we should try num_workers>1 in
dataloader
                print(
                    f"[{smp.rank()}] Waited {wait_time} for data loader
to be ready. Please check if dataloader performance can be improved to
avoid these waits."
                )
        else:
            train_dataloader = create_pretraining_dataloader(
                [train_paths[next_train_path_index]],
                args.train_batch_size,
                args.max_context_width,
                seed=args.seed,
                dp_rank=dp_rank,
                dp_size=dp_size,
                shuffle=args.same_seed < 1,
                zipped=args.zipped_data > 0,
                use_last_file_only=args.fast_validation > 0,
                data_type=data_type,
            )

    return total_steps, throughput, loss_metric


def parse_args():
    parser = argparse.ArgumentParser()

    # hyperparameters sent by the client are passed as command-line
arguments to the script.

    opt_grp = parser.add_argument_group(
        title="optimization", description="arguments for optimization"
    )
    opt_grp.add_argument(
        "--train_batch_size",
        type=int,
        default=4,
```

```
        help="batch size per dp rank, for tensor parallelism degree 8
with pipeline parallel degree 1 this means 8*this batch size per node",
    )
    opt_grp.add_argument("--val_batch_size", type=int, default=4)
    opt_grp.add_argument("--max_steps", type=int, default=5000)
    opt_grp.add_argument("--seed", type=int, default=12345)
    opt_grp.add_argument("--same_seed", type=int, default=0)
    opt_grp.add_argument("--n_gpus", type=str,
default=os.environ["SM_NUM_GPUS"])
    opt_grp.add_argument("--fp16", default=0, type=int, help="automatic
mixed precision training")
    opt_grp.add_argument(
        "--fp32_grad_accumulation", default=0, type=int, help="Enable
FP32 Grad accumulation"
    )
    opt_grp.add_argument("--grad_clip", default=1.0, type=float,
help="gradient clipping")
    opt_grp.add_argument("--weight_decay", default=0.01, type=float,
help="weight decay")
    opt_grp.add_argument(
        "--beta1", default=0.9, type=float, help="beta1 parameter for
Adam optimizer"
    )
    opt_grp.add_argument(
        "--beta2", default=0.95, type=float, help="beta2 parameter for
Adam optimizer"
    )
    opt_grp.add_argument(
        "--activation_checkpointing",
        type=int,
        default=1,
        help="enable gradient checkpointing to reduce memory
consumption",
    )
    parser.add_argument(
        "--logging_freq", type=int, default=1, help="number of iterations
between logging"
    )

    # I/O
    io_grp = parser.add_argument_group(title="io", description="location
for input and output")
    io_grp.add_argument("--use_wiki_data", type=int, default=0, help="use
wiki corpus data for training")
    io_grp.add_argument("--zipped_data", type=int, default=1, help="input
data is zipped files")
    io_grp.add_argument(
        "--epochs", type=int, default=3, help="times of iterating over
the training dataset"
    )
    io_grp.add_argument("--output-data-dir", type=str,
default=os.environ["SM_OUTPUT_DATA_DIR"])
    io_grp.add_argument(
        "--checkpoint-dir",
```

```
        type=str,
        default="/opt/ml/checkpoints",
        help="Saves partial checkpoints (model, optimizer) to this dir,
and loads latest checkpoint from this if load_partial is specified.",
    )
    io_grp.add_argument(
        "--model-dir",
        type=str,
        default=os.environ["SM_MODEL_DIR"],
        help="Saves full model for inference to this dir. Also used if
load_full is given to load the model. Note the lack of optimizer state
here.",
    )
    io_grp.add_argument("--training-dir", type=str,
default=os.environ["SM_CHANNEL_TRAIN"])
    io_grp.add_argument("--test-dir", type=str,
default=os.environ["SM_CHANNEL_TEST"])
    io_grp.add_argument(
        "--parallel_proc_data_processing",
        type=int,
        default=0,
        help="Load data in parallel with a different process. At any
point a process can have two files in memory. With tensor parallelism,
each of the 8 processes on an instance will then have 2 files in memory.
Depending on file sizes this may or may not be feasible. With pipeline
parallelism this was not a problem as only 1 rank on an instance loaded
data.",
    )
    io_grp.add_argument(
        "--save_final_full_model",
        type=int,
        default=0,
        help="Enabling this will save a combined model only at the end",
    )
    io_grp.add_argument(
        "--skip_full_optimizer",
        type=int,
        default=1,
        help="Disabling this will also save the full optimizer state",
    )
    io_grp.add_argument("--load_partial", type=int, default=0, help="Load
from partial checkpoints")
    io_grp.add_argument("--load_full", type=int, default=0, help="Load
from full checkpoints")
    io_grp.add_argument(
        "--logits_output", type=str, default="", help="Path to save
logits and loss"
    )
    io_grp.add_argument("--prescaled_batch", type=int, default=1,
help="use prescaled batch")

    # configure model size
    model_grp = parser.add_argument_group(
```

```
        title="model", description="arguments to describe model
configuration"
    )
    model_grp.add_argument("--max_context_width", type=int, default=1024)
    model_grp.add_argument("--vocab_size", type=int, default=50257)
    model_grp.add_argument("--hidden_width", type=int, default=768)
    model_grp.add_argument("--num_layers", type=int, default=12)
    model_grp.add_argument("--num_heads", type=int, default=12)
    model_grp.add_argument("--resid_pdrop", type=float, default=0.1)
    model_grp.add_argument("--embd_pdrop", type=float, default=0.1)
    model_grp.add_argument("--attn_pdrop", type=float, default=0.1)
    model_grp.add_argument("--summary_first_pdrop", type=float,
default=0.1)
    model_grp.add_argument("--use_adamw", type=int, default=0, help="Use
adamw optimizer")

    smp_grp = parser.add_argument_group(title="smp", description="smp")
    smp_grp.add_argument("--tensor_parallel_degree", type=int, default=8)
    smp_grp.add_argument("--pipeline_parallel_degree", type=int,
default=1)
    smp_grp.add_argument("--microbatches", type=int, default=1)
    smp_grp.add_argument("--active_microbatches", type=int, default=None)
    smp_grp.add_argument("--optimize", type=str, default="speed")
    smp_grp.add_argument("--activation_strategy", type=str,
default="each")
    smp_grp.add_argument("--shard_optimizer_state", type=int, default=0)
    smp_grp.add_argument("--offload_activations", type=int, default=0)
    smp_grp.add_argument("--fast_mode", type=int, default=0)
    smp_grp.add_argument("--static_mode", type=int, default=0)
    smp_grp.add_argument("--delayed_param", type=int, default=0)
    smp_grp.add_argument("--same_partition_load", type=int, default=0)
    smp_grp.add_argument("--attention_in_fp32", type=int, default=0)
    smp_grp.add_argument("--placement_strategy", type=str,
default="cluster")
    smp_grp.add_argument("--activation_loading_horizon", type=int,
default=4)
    smp_grp.add_argument("--skip_tracing", type=int, default=0)
    smp_grp.add_argument("--query_key_layer_scaling", type=int,
default=1)
    smp_grp.add_argument("--fused_softmax", type=int, default=1)
    smp_grp.add_argument("--fused_bias_gelu", type=int, default=1)

    parser.add_argument(
        "--num_kept_checkpoints",
        type=int,
        default=5,
        help="how many checkpoints to keep before deleting",
    )
    parser.add_argument(
        "--checkpoint_freq",
        type=int,
        default=10000,
        help="number of iterations between checkpointing",
    )
```

```
    parser.add_argument(
        "--validation_freq",
        type=int,
        default=None,
        help="number of iterations to print validation loss",
    )
    parser.add_argument(
        "--validation_batches",
        type=int,
        default=10,
        help="number of batches to estimate validation loss",
    )
    parser.add_argument(
        "--manual_partition",
        type=int,
        default=0,
        help="evenly distribute layers across the partitions",
    )
    parser.add_argument(
        "--match_weights", type=int, default=0, help="Get weights from
the original model"
    )
    parser.add_argument(
        "--preserve_np_state",
        type=int,
        default=0,
        help="Perserve the numpy random state between validation",
    )
    parser.add_argument(
        "--fast_validation",
        type=int,
        default=1,
        help="Running validation only with the last data file for faster
speed",
    )
    parser.add_argument(
        "--gather_if_shard",
        type=int,
        default=1,
        help="When sharding opt states is enabled, gather the opt
checkpoint to rdp rank 0 during saving",
    )
    parser.add_argument(
        "--clean_cache",
        type=int,
        default=0,
        help="Clean torch reserved memory at he end of every step",
    )
    parser.add_argument("--use_fsx", type=int, default=0, help="Using FSx
for checkpointing")
    parser.add_argument(
        "--enable_memory_profiling", type=int, default=0, help="Enable
memory profile"
    )
```

```
    # learning rate
    lr_grp = parser.add_argument_group(
        title="lr", description="arguments for learning rate schedule"
    )
    lr_grp.add_argument("--lr", type=float, default=None, help="Initial
learning rate.")
    lr_grp.add_argument(
        "--lr_decay_style",
        type=str,
        default="linear",
        choices=["constant", "linear", "cosine", "exponential",
"plateau"],
        help="Learning rate decay function.",
    )
    lr_grp.add_argument(
        "--lr_decay_iters",
        type=int,
        default=None,
        help="number of iterations to decay learning rate over," " If
None defaults to train iters",
    )
    lr_grp.add_argument(
        "--min_lr",
        type=float,
        default=0.0,
        help="Minumum value for learning rate. The scheduler" "clip
values below this threshold.",
    )
    lr_grp.add_argument(
        "--warmup",
        type=float,
        default=0.01,
        help="Percentage of total iterations to warmup on "
        "(.01 = 1 percent of all training iters).",
    )
    lr_grp.add_argument(
        "--plateau",
        type=float,
        default=0.4,
        help="Percentage of total iterations to keep at max if using
plateau lr",
    )

    ci_grp = parser.add_argument_group(title="ci", description="ci
related settings")
    ci_grp.add_argument("--ci", default=False, action="store_true",
help="Whether enable ci")
    ci_grp.add_argument("--time_to_train", type=int, help="time to train
threshold")
    ci_grp.add_argument("--throughput", type=float, help="throughput
threshold")
    ci_grp.add_argument("--loss", type=float, help="loss threshold")
    ci_grp.add_argument(
```

```
        "--save_or_verify_ckptsum", default=False, action="store_true",
help="Whether to save sum"
    )

    args, _ = parser.parse_known_args()
    return args


def main():
    args = parse_args()

    if args.shard_optimizer_state > 0 and not args.skip_full_optimizer:
        raise ValueError(
            "If shard_optimizer_state is enabled, skip_full_optimizer
must also be enabled. Full optimizer saving is currently not supported
under optimizer state sharding."
        )

    # any value here is overriden by the config set in notebook when
launching the sagemaker job
    smp_config = {
        "ddp": True,
        "tensor_parallel_degree": args.tensor_parallel_degree,
        "pipeline_parallel_degree": args.pipeline_parallel_degree,
        "microbatches": args.microbatches,
        # if activation_checkpointing true checkpoints transformer layers
below
        "checkpoint_attentions": False if args.activation_checkpointing
else True,
        "shard_optimizer_state": args.shard_optimizer_state > 0,
        "prescaled_batch": args.prescaled_batch > 0,
        "_match_weights": args.match_weights > 0,
        "fp16_params": args.fp16 > 0,
        "offload_activations": args.offload_activations > 0,
        "optimize": args.optimize,
        "placement_strategy": args.placement_strategy,
        "activation_loading_horizon": args.activation_loading_horizon,
        "skip_tracing": args.skip_tracing > 0,
        "auto_partition": False if args.manual_partition else True,
        "default_partition": 0,
        "_fp32_grad_accumulation": args.fp32_grad_accumulation > 0,
        "static_mode": args.static_mode > 0,
        "fast_mode": args.fast_mode > 0,
    }
    if args.active_microbatches is not None:
        smp_config["active_microbatches"] = args.active_microbatches

    smp.init(smp_config) # initialize modelparallel library

    if smp.rank() == 0:
        print("Arguments:", args.__dict__)
        print(f"Transformers version: {transformers.__version__}")
        print(f"smdistributed.modelparallel version:
{smdistributed.modelparallel.__version__}")
```

```
        print(f"smdistributed config: {smp_config}")

    if args.save_final_full_model and smp.rank() == 0:
        print(
            f"[Warning] Note that save_final_full_model only saves the
final model at the end of all steps. It does not save optimizer state.
Optimizer state is only saved with partial models which are saved at
checkpointing_freq during training. If you want to restart training you
need partial checkpoints."
        )

    if smp.local_rank() == 0:
        for path in [args.model_dir, args.checkpoint_dir]:
            if not os.path.exists(path):
                os.makedirs(path, exist_ok=True)

    model_config = GPT2Config(
        vocab_size=args.vocab_size,
        n_positions=args.max_context_width,
        n_embd=args.hidden_width,
        n_layer=args.num_layers,
        n_head=args.num_heads,
        n_inner=None,
        activation_function="gelu_new",
        resid_pdrop=args.resid_pdrop,
        embd_pdrop=args.embd_pdrop,
        attn_pdrop=args.attn_pdrop,
        layer_norm_epsilon=1e-05,
        initializer_range=0.02,
        summary_type="cls_index",
        summary_use_proj=True,
        summary_activation=None,
        summary_proj_to_labels=True,
        summary_first_dropout=args.summary_first_pdrop,
        # gradient_checkpointing=args.gradient_checkpointing > 0,
        use_cache=False,
        bos_token_id=50256,
        eos_token_id=50256,
        return_dict=True,
    )

    # the following improves start-up time by skipping proper
initialization
    # of weights in the original model. this is not a problem because
DistributedModel
    # will override those weights anyway when tensor_parallel_degree > 1.
    if smp.tp_size() > 1 and args.match_weights < 1:
        from transformers.modeling_utils import PreTrainedModel

        PreTrainedModel.init_weights = lambda x: None

    set_seed(args.seed)

    if args.fp16:
```

```
        torch.set_default_dtype(torch.float16)
    with smp.tensor_parallelism(
        enabled=smp.tp_size() > 1,
        attention_in_fp32=args.attention_in_fp32 > 0,
        query_key_layer_scaling=args.query_key_layer_scaling > 0,
        fused_softmax=args.fused_softmax > 0,
        fused_bias_gelu=args.fused_bias_gelu > 0,
    ):
        with smp.delay_param_initialization(
            enabled=(smp.tp_size() > 1 and args.match_weights < 1 and
args.delayed_param > 0)
        ):
            model = AutoModelForCausalLM.from_config(model_config)

    torch.set_default_dtype(torch.float32)

    if args.fp16:
        model = FP16_Module(model)

    num_params = sum([np.prod(p.size()) for p in model.parameters()])
    if smp.rank() == 0:
        print(f"# total parameters: {num_params}")

    # SMP modification: Set the device to the GPU ID used by the current
process.
    # Input tensors should be transferred to this device.
    torch.cuda.set_device(smp.local_rank()) # restrict each process to its own device
    device = torch.device("cuda")

    if not args.same_seed:
        # Set seed by tp_rank to prevent weights from being the same on
different tp_ranks
        set_seed(args.seed + smp.tp_rank())

    # SMP modification: Use the DistributedModel container to provide the
model
    # to be partitioned across different ranks. For the rest of the
script,
    # the returned DistributedModel object should be used in place of
    # the model provided for DistributedModel class instantiation.
    if args.fp16:
        torch.set_default_dtype(torch.float16)
    model = smp.DistributedModel(model, trace_device="gpu") # wrap the model with smp.DistributedModel

    if args.fp16:
        m = model.module
    else:
        m = model

    if smp.tp_size() > 1:
        transformer_layers = m.module.module.transformer.seq_layers
    else:
        transformer_layers = m.module.module.transformer.h
```

```
    if args.manual_partition:
        print(f"Manual partition enabled")
        # evenly distribute layers across all partitions
        div, rem = divmod(args.num_layers, smp.pp_size())
        get_num_layers = lambda x: (div + 1 if x >= smp.pp_size() - rem
else div)
        assignments = []
        for pp_rank in range(smp.pp_size()):
            nl = get_num_layers(pp_rank)
            print(f"{nl} layers assigned to partition {pp_rank}")
            assignments += [pp_rank for _ in range(nl)]

        for i, c in enumerate(transformer_layers.children()):
            smp.set_partition(c, assignments[i])

    torch.set_default_dtype(torch.float32)

    iter_model = model
    # Build parameter groups (weight decay and non-decay).
    while isinstance(iter_model, (DistributedDataParallel, FP16_Module)):
        iter_model = iter_model.module
    param_groups = get_param_groups_by_weight_decay(iter_model)

    if args.use_adamw > 0:
        optimizer = optim.AdamW(
            param_groups, betas=(args.beta1, args.beta2), lr=args.lr,
weight_decay=args.weight_decay
        )
    else:
        optimizer = optim.Adam(
            param_groups, betas=(args.beta1, args.beta2), lr=args.lr,
weight_decay=args.weight_decay
        )

    if args.activation_checkpointing:
        kwargs = {}
        if isinstance(transformer_layers, nn.Sequential):
            kwargs["pack_args_as_tuple"] = True
            kwargs["strategy"] = args.activation_strategy
            smp.set_activation_checkpointing(transformer_layers,
**kwargs)
        else:
            for c in transformer_layers.children():
                smp.set_activation_checkpointing(c)

    if args.fp16:
        optimizer = FP16_Optimizer(
            model,
            optimizer,
            static_loss_scale=None,
            dynamic_loss_scale=True,
            use_smp=True,
            dynamic_loss_args={"scale_window": 1000, "min_scale": 1,
"delayed_shift": 2},
```

```
            params_have_main_grad=args.fp32_grad_accumulation > 0,
            shard_optimizer_state=args.shard_optimizer_state > 0,
        )

    optimizer = smp.DistributedOptimizer(optimizer) # wrap the optimizer with smp.DistributedOptimizer
    lr_scheduler = get_learning_rate_scheduler(optimizer, args)

    if args.fp16:
        model.register_post_step_hook(lambda model, optimizer:
optimizer.init_master_params())

    if args.enable_memory_profiling > 0:
        model.register_post_partition_hook(
            lambda model, optimizer: memory_status(msg="After_partition")
        )

    # load after wrapping model and optimizer with smp Distributed...
    if args.load_full or args.load_partial:
        if args.load_partial and args.load_full:
            print(
                "Since both --load_partial and --load_full set, will try
to load from full checkpoint."
                "If the intention is to load from partial checkpoint,
please don't set --load_full"
            )
        partial = not args.load_full
        path = args.checkpoint_dir if partial else args.model_dir
        translate_from_hf = not partial
        model, optimizer, total_steps, start_train_path_index,
start_batch_index = load_model_and_optimizer(
            path,
            model,
            optimizer,
            lr_scheduler,
            partial,
            args,
            translate_from_hf=translate_from_hf,
            seq_length=args.max_context_width,
            load_model=True,
            load_optimizer=args.load_partial > 0,
            num_params=num_params,
        )
        if args.save_or_verify_ckptsum:
            filename = "saved_sum" if args.load_full else
"saved_partial_sum"
            load_and_verify_ckptsum(
                args, model, optimizer,
filename=os.path.join(args.model_dir, filename)
            )
    else:
        total_steps = 0
        start_train_path_index = 0
        start_batch_index = 0
```

```
    start = time.time()
    total_steps, throughput, loss = train(
        model,
        optimizer,
        lr_scheduler,
        model_config,
        start_train_path_index,
        start_batch_index,
        num_params,
        total_steps,
        args,
    )
    time_to_train = time.time() - start
    if args.ci:
        print(f"[SMP_METRIC]__GPT2__Time_to_train__{time_to_train}")
        print(f"[SMP_METRIC]__GPT2__samples/second__{throughput}")
        print(f"[SMP_METRIC]__GPT2__Loss__{loss}")
        if not args.load_partial and not args.load_full:
            assert time_to_train < args.time_to_train
            assert throughput > args.throughput
            if args.loss:
                assert loss < args.loss

    if args.save_final_full_model:
        # saves full model at the end

        base_path = f"trained_gpt_nparams-{num_params}_steps-
{total_steps}.pt"
        out_path = os.path.join(args.model_dir, base_path)
        if args.save_or_verify_ckptsum:
            # Save optimizer and model tensor sums and scalars before
saving
            save_ckptsum(args, model, optimizer,
filename=os.path.join(args.model_dir, "saved_sum"))

        if smp.rdp_rank() == 0:
            save(
                out_path,
                model,
                optimizer,
                lr_scheduler,
                model_config,
                num_params,
                total_steps,
                -1,
                args,
                partial=False,
                translate_to_hf=smp.tp_size() > 1,
                seq_length=args.max_context_width,
            )

    smp.barrier()
    if smp.rank() == 0:
        print("SMP training finished successfully")
```

```
if __name__ == "__main__":
    main()
```