DeepSpeed

Release 0.19.2

Microsoft

Jun 11, 2026

CONTENTS

1 Model Setup **1**
- 1.1 Training Setup . . . 1
- 1.2 Inference Setup . . . 2

2 Training API **7**
- 2.1 Training API . . . 7

3 Inference API **21**
- 3.1 Inference API . . . 21

4 Checkpointing API **23**
- 4.1 Model Checkpointing . . . 23
- 4.2 Activation Checkpointing . . . 26

5 ZeRO API **27**
- 5.1 ZeRO . . . 27

6 Mixture of Experts **45**
- 6.1 AutoEP (Automatic Expert Parallelism) . . . 45
- 6.2 Mixture of Experts (DeepSpeed MoE) . . . 46

7 Transformer Kernel API **49**
- 7.1 Transformer Kernels . . . 49

8 Pipeline Parallelism **51**
- 8.1 Pipeline Parallelism . . . 51

9 Optimizers **61**
- 9.1 Optimizers . . . 61

10 Learning Rate Schedulers **65**
- 10.1 Learning Rate Schedulers . . . 65

11 Flops Profiler **71**
- 11.1 Flops Profiler . . . 71

12 Autotuning **75**
- 12.1 Autotuning . . . 75

13 Memory Usage **77**
- 13.1 Memory Requirements . . . 77

14 Monitoring **83**
14.1 Monitoring 83

15 Indices and tables **87**

Python Module Index **89**

Index **91**

CHAPTER

ONE

MODEL SETUP

1.1 Training Setup

1.1.1 Argument Parsing

DeepSpeed uses the argparse library to supply commandline configuration to the DeepSpeed runtime. Use `deepspeed.add_config_arguments()` to add DeepSpeed's builtin arguments to your application's parser.

```
parser = argparse.ArgumentParser(description='My training script.')
parser.add_argument('--local_rank', type=int, default=-1,
                    help='local rank passed from distributed launcher')
# Include DeepSpeed configuration arguments
parser = deepspeed.add_config_arguments(parser)
cmd_args = parser.parse_args()
```

1.1.2 Training Initialization

The entrypoint for all training with DeepSpeed is `deepspeed.initialize()`. Will initialize distributed backend if it is not initialized already.

Example usage:

```
model_engine, optimizer, _, _ = deepspeed.initialize(args=cmd_args,
                                                     model=net,
                                                     model_parameters=net.parameters())
```

1.1.3 Distributed Initialization

Optional distributed backend initialization separate from `deepspeed.initialize()`. Useful in scenarios where the user wants to use torch distributed calls before calling `deepspeed.initialize()`, such as when using model parallelism, pipeline parallelism, or certain data loader scenarios.

1.2 Inference Setup

The entrypoint for inference with DeepSpeed is `deepspeed.init_inference()`.

Example usage:

```
engine = deepspeed.init_inference(model=net, config=config)
```

The `DeepSpeedInferenceConfig` is used to control all aspects of initializing the `InferenceEngine`. The config should be passed as a dictionary to `init_inference`, but parameters can also be passed as keyword arguments.

class deepspeed.inference.config.**DeepSpeedInferenceConfig**

Sets parameters for DeepSpeed Inference Engine.

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

replace_with_kernel_inject: bool = False (alias 'kernel_inject')

Set to true to inject inference kernels for models such as, Bert, GPT2, GPT-Neo and GPT-J. Otherwise, the injection_dict provides the names of two linear layers as a tuple: *(attention_output projection, transformer output projection)*

dtype: dtype = torch.float16

Desired model data type, will convert model to this type. Supported target types: *torch.half*, *torch.int8*, *torch.float*

tensor_parallel: *DeepSpeedTPConfig* = {} (alias 'tp')

Configuration for tensor parallelism used to split the model across several GPUs. Expects a dictionary containing values for *DeepSpeedTPConfig*.

enable_cuda_graph: bool = False

Use this flag for capturing the CUDA-Graph of the inference ops, so that it can run faster using the graph replay method.

use_triton: bool = False

Use this flag to use triton kernels for inference ops.

triton_autotune: bool = False

Use this flag to enable triton autotuning. Turning it on is better for performance but increase the 1st runtime for autotuning.

zero: *DeepSpeedZeroConfig* = {}

ZeRO configuration to use with the Inference Engine. Expects a dictionary containing values for *DeepSpeedZeroConfig*.

triangular_masking: bool = True (alias 'tm')

Controls the type of masking for attention scores in transformer layer. Note that the masking is application specific.

moe: Union[bool, *DeepSpeedMoEConfig*] = {}

Specify if the type of Transformer is MoE. Expects a dictionary containing values for *DeepSpeedMoEConfig*.

keep_module_on_host: bool = False

When loading checkpoints to model parameters, they are moved to the device. In very large models this might fill the device and cause OOM. Setting this flag to true, will keep checkpoints on host and not move them directly to the device (giving an option to quantize checkpoint data before moving it to the device for example). Set only for models with injection policies and auto TP.

quant: *QuantizationConfig* = {}

NOTE: only works for int8 dtype. Quantization settings used for quantizing your model using the MoQ. The setting can be one element or a tuple. If one value is passed in, we consider it as the number of groups used in quantization. A tuple is passed in if we want to mention that there is extra-grouping for the MLP part of a Transformer layer (e.g. (True, 8) shows we quantize the model using 8 groups for all the network except the MLP part that we use 8 extra grouping). Expects a dictionary containing values for *QuantizationConfig*.

checkpoint: Optional[Union[str, Dict]] = None

Path to deepspeed compatible checkpoint or path to JSON with load policy.

base_dir: str = ''

This shows the root directory under which all the checkpoint files exists. This can be passed through the json config too.

set_empty_params: bool = False

specifying whether the inference-module is created with empty or real Tensor

save_mp_checkpoint_path: Optional[str] = None

The path for which we want to save the loaded model with a checkpoint. This feature is used for adjusting the parallelism degree to help alleviate the model loading overhead. It does not save any new checkpoint if no path is passed.

checkpoint_config: *InferenceCheckpointConfig* = {} (alias 'ckpt_config')

TODO: Add docs. Expects a dictionary containing values for *InferenceCheckpointConfig*.

return_tuple: bool = True

Specify whether or not the transformer layers need to return a tuple or a Tensor.

training_mp_size: int = 1

If loading a checkpoint this is the mp size that it was trained with, it may be different than what the mp size that you want to use during inference.

replace_method: str = 'auto'

injection_policy: Optional[Dict] = None (alias 'injection_dict')

Dictionary mapping a client nn.Module to its corresponding injection policy. e.g., *{BertLayer : deepspeed.inference.HFBertLayerPolicy}*

injection_policy_tuple: Optional[tuple] = None

TODO: Add docs

config: Optional[Dict] = None (alias 'args')

max_out_tokens: int = 1024 (alias 'max_tokens')

This argument shows the maximum number of tokens inference-engine can work with, including the input and output tokens. Please consider increasing it to the required token-length required for your use-case.

min_out_tokens: int = 1 (alias 'min_tokens')

This argument communicates to the runtime the minimum number of tokens you expect you will need to generate. This will cause the runtime to error if it unable to provide this and provide context on the memory pressure rather than seg-faulting or providing corrupted output.

transposed_mode: bool = False

mp_size: int = 1

Desired model parallel size, default is 1 meaning no model parallelism. Deprecated, please use the ``tensor_parallel` config to control model parallelism.

mpu: object = None

ep_size: int = 1

ep_group: object = None (alias 'expert_group')

ep_mp_group: object = None (alias 'expert_mp_group')

moe_experts: list = [1]

moe_type: MoETypeEnum = MoETypeEnum.standard

class deepspeed.inference.config.**DeepSpeedTPConfig**

Configure tensor parallelism settings

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

enabled: bool = True

Turn tensor parallelism on/off.

tp_size: int = 1

Number of devices to split the model across using tensor parallelism.

tp_grain_size: int = 64

Desired MLP/lm_head tp size granularity. DNN library favors tensor size in granularity of power of 2, we pick 64 as a default size.

mpu: object = None

A model parallelism unit object that implements `get_{model,data}_parallel_{rank,group,world_size}()`.

tp_group: object = None

class deepspeed.inference.config.**DeepSpeedMoEConfig**

Sets parameters for MoE

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

enabled: bool = True

ep_size: int = 1

The expert-parallelism size which is used for partitioning the experts across the GPUs in the expert-parallel group.

moe_experts: list = [1] (alias 'num_experts')

The global number of experts used in an MoE layer.

type: MoETypeEnum = MoETypeEnum.standard

Specify the type of MoE layer. We have two types of MoE layer: 'Standard' and 'Residual'.

ep_mp_group: object = None

ep_group: object = None (alias 'expert_group')

class deepspeed.inference.config.**QuantizationConfig**

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

enabled: bool = True

activation: ActivationQuantConfig = ActivationQuantConfig(enabled=True, num_bits=8, q_type='symmetric', q_groups=1)

weight: WeightQuantConfig = WeightQuantConfig(enabled=True, num_bits=8, q_type='symmetric', q_groups=1, quantized_initialization={}, post_init_quant={})

qkv: QKVQuantConfig = QKVQuantConfig(enabled=True)

class deepspeed.inference.config.**InferenceCheckpointConfig**

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

checkpoint_dir: Optional[str] = None

save_mp_checkpoint_path: Optional[str] = None

base_dir: Optional[str] = None

Example config:

```
config = {
    "kernel_inject": True,
    "tensor_parallel": {"tp_size": 4},
    "dtype": "fp16",
    "enable_cuda_graph": False
}
```

CHAPTER

TWO

TRAINING API

2.1 Training API

`deepspeed.initialize()` returns a *training engine* in its first argument of type `DeepSpeedEngine`. This engine is used to progress training:

```
for step, batch in enumerate(data_loader):
    #forward() method
    loss = model_engine(batch)

    #runs backpropagation
    model_engine.backward(loss)

    #weight update
    model_engine.step()
```

Note that `model_engine.backward()` accepts only a scalar loss tensor produced by a forward pass. Starting from v0.18.3, DeepSpeed also supports direct calls to `tensor.backward()`. You can now call `loss.backward()` or `tensor.backward(out_grad)` when your PyTorch version supports the necessary APIs. If your PyTorch version does not support these APIs, a direct call to `tensor.backward()` will raise an error.

2.1.1 Forward Propagation

2.1.2 Backward Propagation

Loss Scaling for Manual Backward Passes

When using mixed precision training (fp16, bf16, or torch.autocast), DeepSpeed applies loss scaling to prevent gradient underflow. If you prefer to call `loss.backward()` directly instead of `engine.backward(loss)`, you must use `engine.scale(loss)` to apply the appropriate loss scaler:

```
# Option 1: Use engine.backward() (recommended)
loss = model_engine(batch)
model_engine.backward(loss)

# Option 2: Manual backward with scaling
loss = model_engine(batch)
scaled_loss = model_engine.scale(loss)
scaled_loss.backward()
```

Both approaches produce identical gradients. The `scale()` method automatically applies the appropriate scaler based on your configuration (ZeRO optimizer scaler, torch.autocast GradScaler, etc.).

2.1.3 Optimizer Step

2.1.4 Gradient Accumulation

Coalesced Gradient Reduction

Use this when one optimizer step needs multiple `engine.backward()` calls and per-backward reduction is wasted work. Typical cases are GradCache-style cached contrastive losses that replay backward over chunked representations, and custom `torch.autograd.Function` subclasses that call `torch.autograd.backward` from inside their `forward`. Results are bit-exact against the per-backward baseline.

Under ZeRO-3, each backward inside the block leaves param-shaped gradients on the leaf modules instead of triggering the per-backward reduce-scatter. On exit, a single pass drives the reducer over the accumulated grads and restores the partitioned `averaged_gradients` for `step()`.

```
for batch in data_loader:
    chunks = batch.split(chunk_size)
    with model_engine.coalesce_grad_reduction():
        for chunk in chunks:
            loss = model_engine(chunk)
            model_engine.backward(loss)
    model_engine.step()
```

Communication

With `N` back-to-back `backward()` calls per step, ZeRO-2 and ZeRO-3 normally issue `N` gradient collectives (one per backward). Inside `coalesce_grad_reduction()` those collapse to one collective on exit. ZeRO-1 already reduces only at the accumulation boundary, so its collective count is unchanged; the context still removes the per-backward bucket setup cost.

Memory

Suppressing the per-backward reduction means each rank holds a full local gradient copy for the duration of the `with` block.

- ZeRO-2: window-resident memory equals ZeRO-1 with `deepspeed.DeepSpeedEngine.no_sync()`, one full gradient per rank held until flush. On a 2-GPU, 134M-param bf16 rig with `N=4`, peak window memory drops from 640 MiB (baseline) to 384 MiB.
- ZeRO-3: window-resident is one full gradient per rank vs the `1/world_size` partition the per-backward path holds throughout. Peak is roughly equal to baseline (the in-flight backward already needs full-grad room and the accumulator reuses it).

Constraints

- ZeRO stage 0 and pipeline parallelism raise `NotImplementedError`.
- The BF16/FP16 optimizer wrappers (`BF16_Optimizer`, `FP16_Optimizer`) route grads through their own `backward_epilogue` path and are not yet supported; the context raises `NotImplementedError` at entry. Use raw ZeRO-1/2/3 for now.
- `engine.step()` inside the `with` block raises.
- Cannot be nested inside `deepspeed.DeepSpeedEngine.no_sync()`.
- Do not split one `gradient_accumulation_steps` window across multiple `with` blocks: the flush overwrites `averaged_gradients` on each exit.

`deepspeed.DeepSpeedEngine.no_sync()` raises `AssertionError` for ZeRO-2 and ZeRO-3 (`zero_optimization_partition_gradients()` is true for stage >= 2), so it cannot collapse collectives for those stages. `coalesce_grad_reduction()` is the equivalent for ZeRO-2/3.

2.1.5 Mixed Precision Training

DeepSpeed supports mixed precision training using either native or PyTorch mechanisms. The desired mixed precision mode can be selected through the configuration dict. Mixed precision training can used with ZeRO (i.e., stages > 0) and without ZeRO (i.e., stage=0).

Native Mixed Precision

DeepSpeed provides native support for fp16 and bf16 mixed precsion training.

PyTorch Automatic Mixed Precision (AMP)

DeepSpeed provides torch-compatible automatic mixed precision (AMP) training via torch.autocast functionality. The following snippet illustrates how to enable Torch AMP.

```
{
    "torch_autocast": {
        "enabled": true,
        "dtype": "bfloat16",
        "lower_precision_safe_modules": ["torch.nn.Linear", "torch.nn.Conv2d"]
    },
    ...
}
```

Each configuration works as follows:

- `enabled`: Enable `torch.autocast` when set to `True`. You don't need to call `torch.autocast` in your code. The grad scaler is also applied in the DeepSpeed optimizer.
- `dtype`: Lower precision dtype passed to `torch.autocast`. Gradients for all-reduce (reduce-scatter) and parameters for all-gather (only for ZeRO3) of `lower_precision_safe_modules` are also downcasted to this `dtype`.
- `lower_precision_safe_modules`: The list of modules that will be downcasted for all-reduce (reduce-scatter) and all-gather (ZeRO3). The precision for PyTorch operators in forward/backward follows `torch.autocast`'s policy, not this list. If you don't set this item, DeepSpeed uses the default list: `[torch.nn.Linear, torch.nn.Conv1d, torch.nn.Conv2d, torch.nn.Conv3d]`.

Manual Backward with torch.autocast

When using `torch.autocast` with manual backward passes (`loss.backward()` instead of `engine.backward()`), you must use `engine.scale(loss)` to apply the gradient scaler:

```
# Training loop with torch.autocast and manual backward
for batch in data_loader:
    loss = model_engine(batch)

    # Apply loss scaling before manual backward
    scaled_loss = model_engine.scale(loss)
    scaled_loss.backward()

    model_engine.step()
```

The `scale()` method ensures that the `torch.amp.GradScaler` is properly applied when `torch.autocast` is enabled with fp16. For bf16 or when no mixed precision is used, `scale()` returns the loss unchanged.

If you call `loss.backward()` directly without using `engine.scale()` or `engine.backward()`, DeepSpeed will raise a `RuntimeError` to prevent training with unscaled gradients, which can lead to incorrect results or gradient underflow.

Using torch.autocast Outside the Engine

DeepSpeed applies `torch.autocast` internally during `engine.forward()`. However, you may also want autocast to cover code that runs **outside** the engine, such as a loss function or post-processing logic. In that case, wrap the entire forward-plus-loss block in your own `torch.autocast` context:

```
# Autocast covers both the engine forward AND the loss computation
with torch.autocast(device_type="cuda", dtype=torch.bfloat16):
    logits = model_engine(input_ids)
    loss = loss_fn(logits.view(-1, vocab_size), labels.view(-1))
```

Without the outer `torch.autocast`, only the model's forward pass benefits from autocast; the loss function would run in full precision.

When DeepSpeed detects a nested autocast context, it handles it as follows:

- If `torch_autocast` is **enabled** in the DeepSpeed config, the engine overrides the outer context with the dtype from the config. An info message is logged once.
- If `torch_autocast` is **disabled** in the config (i.e., you are using DeepSpeed's built-in bf16/fp16 support instead), the engine disables autocast inside `engine.forward()` and a warning is logged once.

In both cases, PyTorch's `torch.autocast` is idempotent when nested with the same dtype, so there is no performance or correctness penalty from the nesting.

deepspeed.runtime.torch_autocast.**init_autocast_params**(*engine*, *dtype: dtype*, *torch_autocast_lower_precision_safe_modules: Union[None, List[str]]*) → None

deepspeed.runtime.torch_autocast.**is_autocast_initialized**() → bool

deepspeed.runtime.torch_autocast.**get_default_autocast_lower_precision_modules**() → List[str]

2.1.6 Configuring ZeRO Leaf Modules

ZeRO-3 relies on module execution order to gather partitioned parameters. When models select submodules dynamically (for example, MoE routers), different data-parallel ranks may gather different sets of parameters, which can cause the all-gather collective to deadlock. To avoid this problem, you can designate the parent of dynamically activated submodules (e.g., MoE experts) as a "leaf" module. When a module is marked as a leaf, ZeRO gathers all of its descendants immediately and stops inserting hooks beneath it.

Programmatic API

Use `deepspeed.utils.set_z3_leaf_modules()` to flag modules by class, class name, or both. Optionally combine with `deepspeed.utils.set_z3_leaf_modules_by_name()` to target specific entries from `model.named_modules()` or `deepspeed.utils.set_z3_leaf_modules_by_suffix()` to match suffixes of those names.

```
from deepspeed.utils import (
    set_z3_leaf_modules,
    set_z3_leaf_modules_by_name,
    set_z3_leaf_modules_by_suffix,
)

# Match by class or subclass
set_z3_leaf_modules(model, [CustomMoEBlock])

# Match by fully qualified class name
set_z3_leaf_modules(model, ["my_package.layers.CustomMoEBlock"])

# Match by module name returned from model.named_modules()
set_z3_leaf_modules_by_name(model, ["transformer.layers.0.experts"])

# Match by suffix of names returned from model.named_modules()
set_z3_leaf_modules_by_suffix(model, ["experts"])
```

Configuration in DeepSpeed config

The same behavior can be controlled from the DeepSpeed config. Add a `leaf_module` block to `zero_optimization` specifying either classes, module names, or name suffixes (or any combination). While the example below shows three different ways (`classes`, `names`, and `name_suffixes`) to specify modules as leaf modules, typically you will use just one of these.

```
{
  "train_micro_batch_size_per_gpu": 1,
  "zero_optimization": {
    "stage": 3,
    "leaf_module": {
      "classes": ["my_package.layers.CustomMoEBlock"],
      "names": ["transformer.layers.0.experts"],
      "name_suffixes": ["experts"]
    }
  }
}
```

`names` must match exactly what `model.named_modules()` produces. The `name_suffixes` field compares each suffix against the end of those same module paths, making it convenient to apply a rule across repeated structures. Entries

in `classes` may be either bare class names (for example, `MixtralSparseMoeBlock`) or fully qualified dotted paths; both forms are accepted.

You can mix and match the API and configuration approaches; all referenced modules are flagged before ZeRO installs its hooks.

By default DeepSpeed marks several Hugging Face MoE blocks—including Mixtral and Qwen MoE sparse blocks so that they behave well with ZeRO3. The default class list currently contains:

- `transformers.models.mixtral.modeling_mixtral.MixtralSparseMoeBlock`
- `transformers.models.qwen2_moe.modeling_qwen2_moe.Qwen2MoeSparseMoeBlock`
- `transformers.models.qwen3_moe.modeling_qwen3_moe.Qwen3MoeSparseMoeBlock`

2.1.7 Model Saving

Additionally when a DeepSpeed checkpoint is created, a script `zero_to_fp32.py` is added there which can be used to reconstruct fp32 master weights into a single pytorch `state_dict` file.

2.1.8 Training Multiple Models

DeepSpeed supports training multiple models, which is a useful feature in scenarios such as knowledge distillation and post-training RLHF. The core approach is to create individual DeepSpeedEngines for each model.

Training Independent Models

The following code snippet illustrates independently training multiple models on the same dataset.

```
model_engines = [engine for engine, _, _, _ in [deepspeed.initialize(m, ...,) for m in␣
↪models]]
for batch in data_loader:
   losses = [engine(batch) for engine in model_engines]
   for engine, loss in zip(model_engines, losses):
      engine.backward(loss)
```

The above is similar to typical DeepSpeed usage except for the creation of multiple DeepSpeedEngines (one for each model).

Jointly Training Models With Shared Loss

The following code snippet illustrates jointly training multiple models on a shared loss value.

```
model_engines = [engine for engine, _, _, _ in [deepspeed.initialize(m, ...,) for m in␣
↪models]]
for batch in data_loader:
    losses = [engine(batch[0], batch[1]) for engine in model_engines]
    loss = sum(l / (i + 1) for i, l in enumerate(losses))
    loss.backward()

    for engine in model_engines:
        engine.step()
```

(continues on next page)

(continued from previous page)

```
    for engine in model_engines:
        engine.optimizer.zero_grad()
```

Besides the use of multiple DeepSpeedEngines, the above differs from typical usage in two key ways:

1. The **backward** call is made using the common loss value rather on individual model engines.

You can call `loss.backward()` once for the shared loss.

Note: Previously, you had to call `_backward_epilogue` on each model engine after `loss.backward()`. However, starting from v0.18.3, DeepSpeed automatically handles this internally, so you no longer need to call `_backward_epilogue` manually.

2.1.9 Automatic Tensor Parallel Training

DeepSpeed supports **Automatic Tensor Parallel (AutoTP) training** for sharding model weights across GPUs while remaining compatible with ZeRO and standard training workflows. This training API is different from the inference-only tensor parallel API exposed by `deepspeed.init_inference`.

Tensor parallelism (TP) splits the computations and parameters of large layers across multiple GPUs so each rank holds only a shard of the weight matrix. This is an efficient way to train large-scale transformer models by reducing per-GPU memory pressure while keeping the layer math distributed across the TP group.

AutoTP training is enabled by setting `tensor_parallel` in the DeepSpeed config and passing it to `deepspeed.initialize`. DeepSpeed applies AutoTP sharding during engine initialization; calling `deepspeed.tp_model_init`, which we previously used to initialize AutoTP, is now optional. See *Initialization behavior* for more details.

```
import deepspeed

ds_config = {
    "train_micro_batch_size_per_gpu": 1,
    "zero_optimization": {"stage": 2},
    "tensor_parallel": {"autotp_size": 4},
}

engine, optimizer, _, _ = deepspeed.initialize(
    model=model,
    optimizer=optimizer,
    config=ds_config,
    mpu=mpu,  # optional: TP/DP process groups
)
```

Note: AutoTP training supports ZeRO stages 0, 1, and 2. ZeRO Stage 3 is not supported.

Initialization behavior

AutoTP previously required calling `set_autotp_mode(training=True)` and `deepspeed.tp_model_init` before `deepspeed.initialize`. Now we can include all the necessary configurations in the DeepSpeed config. We still support the traditional initialization path for backward compatibility. When you use both (i.e. calling `set_autotp_mode(training=True)` and `deepspeed.tp_model_init` and passing the config to `deepspeed.initialize`), we will merge the settings at initialization. When we have conflicting settings, we will error out.

Parameter partitioning

TP sharding needs to know which parameter tensors should be partitioned and along which dimensions. AutoTP provides three ways to balance ready-to-use defaults with customizability:

- **Heuristics**: automatic sharding based on parameter names and model rules.
- **Preset**: choose a built-in model family via `preset_model`.
- **Custom specs**: define regex patterns and partition rules via `partition_config`.
- **HuggingFace tp_plan**: automatically detected from `model.config.base_model_tp_plan` or `model._tp_plan`.

HuggingFace tp_plan

Many HuggingFace models (e.g. Llama, Qwen, Gemma2) define a `base_model_tp_plan` in their model config. When present, DeepSpeed automatically extracts and converts this plan into internal partition rules. This means you do not need `preset_model` or `partition_config` for these models – just set `autotp_size`.

The resolution priority is:

1. `partition_config` (user-defined custom specs – highest priority)
2. HuggingFace `tp_plan` (from model config)
3. AutoTP heuristics / `preset_model` (lowest priority)

Currently only `colwise` and `rowwise` partition types from the HuggingFace `tp_plan` are supported. Other types (`colwise_rep`, `local_colwise`, `local_rowwise`, `local_packed_rowwise`, `gather`, `sequence_parallel`) are not yet handled and will raise an error.

Heuristic rules

Heuristics use parameter names and model-specific rules to decide how to shard layers. If you are training a supported model (see *Supported models*), the heuristic rules automatically shard the model, so you only need to add `autotp_size`.

```
{
  ...
  "tensor_parallel": {
    "autotp_size": 4
  },
  "zero_optimization": {
    ...
  },
  ...
}
```

Preset-based partitioning

You can explicitly specify the model family with `preset_model`:

```
{
  "tensor_parallel": {
    "autotp_size": 4,
    "preset_model": "llama"
  }
}
```

See *Supported models* for the supported preset names and the implementation in AutoTPPresets. If you add a new model family, you can easily add a new preset by defining patterns like the existing presets, and we welcome PRs for those additions.

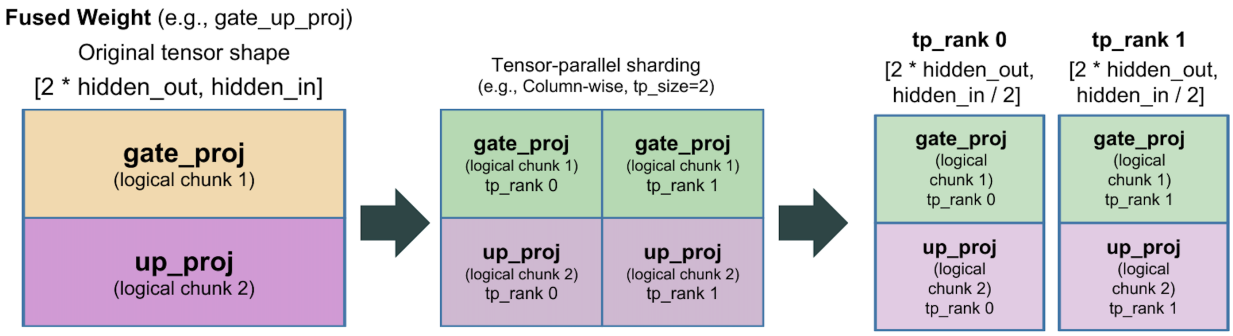
Custom layer specs

If you are training a custom model, you can use `partition_config` to specify custom regex-based patterns and partition settings.

```
{
  "tensor_parallel": {
    "autotp_size": 4,
    "partition_config": {
      "use_default_specs": false,
      "layer_specs": [
        {
          "patterns": [".*\\.o_proj\\.weight$", ".*\\.down_proj\\.weight$"],
          "partition_type": "row"
        },
        {
          "patterns": [".*\\.[qkv]_proj\\.weight$"],
          "partition_type": "column"
        },
        {
          "patterns": [".*\\.gate_up_proj\\.weight$"],
          "partition_type": "column",
          "shape": [2, -1],
          "partition_dim": 0
        }
      ]
    }
  }
}
```

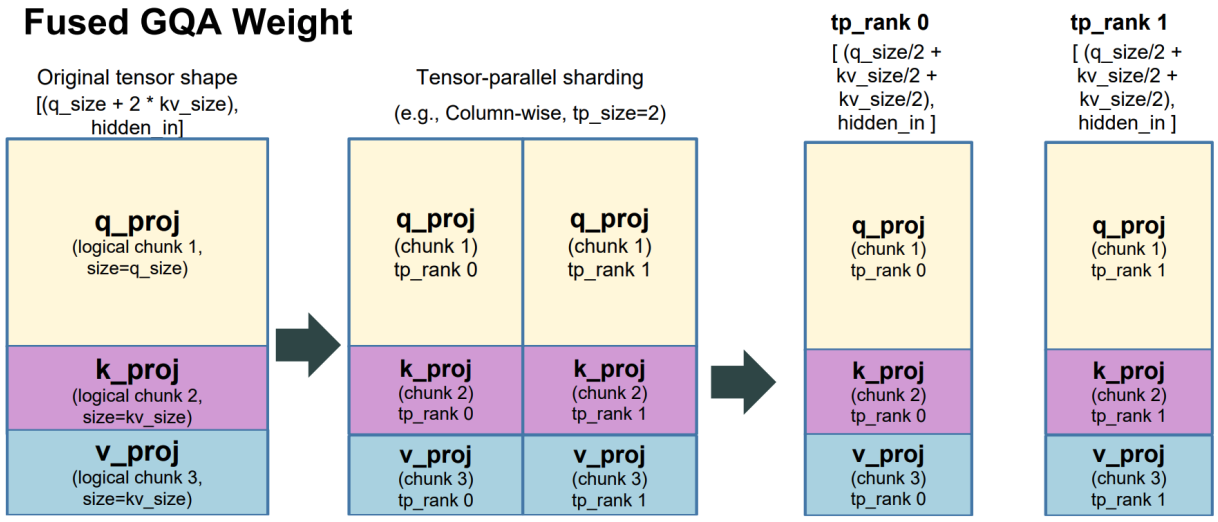
You can also set `use_default_specs` to `true` to merge your custom patterns on top of the preset (when `preset_model` is provided).

For fused or packed weights (for example QKV or gate/up projections), the `shape` and `partition_dim` options control sub-parameter partitioning. Sub-parameter partitioning lets AutoTP split a single weight tensor into logical chunks before applying tensor-parallel sharding. For example, the `gate_up_proj` weight can be viewed as two packed matrices (gate and up) by setting `shape` to `[2, -1]` and `partition_dim` to `0`; AutoTP then partitions each chunk consistently across tensor-parallel ranks.



Another example is GQA-style fused QKV weights. The tensor can contain unequal Q/K/V segments stacked along the output dimension. For example, set `shape` to the explicit sizes (for example `[(q_size, kv_size, kv_size), -1]`) and `partition_dim` to `0` so AutoTP splits the Q, K, and V regions first, then shards each region across tensor-parallel ranks.

```
{
  "patterns": [".*\\.qkv_proj\\.weight$"],
  "partition_type": "column",
  "shape": [[q_size, kv_size, kv_size], -1],
  "partition_dim": 0
}
```



Model-type filtering for shared configs

Use `model_types` when you want a single config to work across multiple model families but apply different specs. This is useful in shared training scripts or when patterns overlap across architectures.

```
{
  "tensor_parallel": {
    "autotp_size": 4,
    "partition_config": {
      "layer_specs": [
        {
```

(continues on next page)

(continued from previous page)

```
          "patterns": [".*\\.qkv_proj\\.weight$"],
          "partition_type": "column",
          "shape": [[q_size, kv_size, kv_size], -1],
          "partition_dim": 0,
          "model_types": ["llama"]
        },
        {
          "patterns": [".*\\.qkv_proj\\.weight$"],
          "partition_type": "column",
          "shape": [3, -1],
          "partition_dim": 0,
          "model_types": ["qwen2"]
        }
      ]
    }
  }
}
```

Supported models

The following model families are supported by built-in AutoTP presets:

- `llama`
- `bloom`
- `chatglm`
- `mixtral`
- `deepseek_v2`
- `qwen2`
- `phi3`

Preset definitions live in AutoTPPresets. If you add a new model family, you can easily add a new preset by defining patterns like the existing presets, and we welcome PRs for those additions.

These strings are the values accepted by `preset_model` and are matched against the model type in `model.config.model_type` (case-insensitive). When `preset_model` is not set, AutoTP uses the legacy automatic sharding rules unless you provide a custom `partition_config`. These presets are also useful when you want to extend the default patterns: set `use_default_specs` to `true` in `partition_config` to merge your custom specs on top of the selected preset.

2.1.10 Automatic Sequence Parallel Training

DeepSpeed supports **Automatic Sequence Parallel (AutoSP) training** for enabling compiler-based sequence parallelism to unlock long-context LLM training. AutoSP leverages defines custom passes to automatically shard inputs along the sequence dimension and enable Ulysses-styled sequence parallelism.

AutoSP training is enabled by setting `compile` and `passes` in the DeepSpeed config and calling `prepare_autosp_inputs()` to prepare inputs before each forward pass.

```
import deepspeed
from deepspeed.compile.passes.sp_compile import prepare_autosp_inputs

ds_config = {
    "train_micro_batch_size_per_gpu": 1,
    "zero_optimization": {"stage": 0},
    "compile": {
        "deepcompile": True,
        "passes": ["autosp"],
    }
}

engine, optimizer, _, _ = deepspeed.initialize(
    model=model,
    optimizer=optimizer,
    config=ds_config,
)

# Compile the model before training
engine.compile(backend='inductor')

for batch in dataloader:
    input_ids = prepare_autosp_inputs(
        input_id=batch["input_ids"],
        label_id=batch["labels"],
        position_id=batch.get("position_ids"),
        seq_dim=1
    )
    loss = engine(input_ids)
    engine.backward(loss)
    engine.step()
```

Note: AutoSP requires ZeRO stage 0 (no ZeRO optimization). Using AutoSP with ZeRO stages 1, 2, or 3 is not currently supported. AutoSP also requires `torch.nn.functional.scaled_dot_product_attention()` as the attention backend.

Input Preparation

Before each forward pass, inputs must be prepared using `prepare_autosp_inputs()` to mark the sequence dimension as dynamic and annotate tensors for identification during automatic sharding:

```
from deepspeed.compile.passes.sp_compile import prepare_autosp_inputs

input_ids = prepare_autosp_inputs(
    input_id=input_ids,
    label_id=labels,
    position_id=position_ids,  # optional
    attention_mask=attention_mask,  # optional
    seq_dim=1
)
```

This serves as a hint to the compiler to know which inputs should be sharded across which dimension.

Memory Optimization

AutoSP includes selective activation checkpointing that recomputes matmul operations during backpropagation while preserving attention activations. This is effective for long-context training because attention operations scale quadratically with sequence length and dominate computation latency, while matmul operations scale linearly and are relatively cheaper to recompute. This provides significant memory savings with minimal computational overhead

Limitations

AutoSP currently supports only `torch.nn.functional.scaled_dot_product_attention`. Other attention patterns require additional pattern matching logic.

AutoSP requires a fully connected computation graph without breaks. Graph breaks destroy the use-def chains across graphs and the compiler cannot propoaget sequence dimension sharding information.

CHAPTER

THREE

INFERENCE API

3.1 Inference API

`deepspeed.init_inference()` returns an *inference engine* of type `InferenceEngine`.

```
for step, batch in enumerate(data_loader):
    #forward() method
    loss = engine(batch)
```

3.1.1 Forward Propagation

CHAPTER

FOUR

CHECKPOINTING API

4.1 Model Checkpointing

DeepSpeed provides routines for checkpointing model state during training.

4.1.1 Loading Training Checkpoints

4.1.2 Saving Training Checkpoints

4.1.3 ZeRO Checkpoint fp32 Weights Recovery

DeepSpeed provides routines for extracting fp32 weights from the saved ZeRO checkpoint's optimizer states.

`deepspeed.utils.zero_to_fp32.`**`get_fp32_state_dict_from_zero_checkpoint`**(*checkpoint_dir*, *tag=None*, *exclude_frozen_parameters=False*, *lazy_mode=False*)

Convert ZeRO 2 or 3 checkpoint into a single fp32 consolidated state_dict that can be loaded with `load_state_dict()` and used for training without DeepSpeed or shared with others, for example via a model hub.

Parameters

- **`checkpoint_dir`** (-) – path to the desired checkpoint folder
- **`tag`** (-) – checkpoint tag used as a unique identifier for checkpoint. If not provided will attempt to load tag in 'latest' file. e.g., `global_step14`
- **`exclude_frozen_parameters`** (-) – exclude frozen parameters
- **`lazy_mode`** (-) – get state_dict in lazy mode. It returns a dict of pesduo tensor instead of torch tensor, which is more memory efficient. Convert the pesduo tensor to torch tensor by `.contiguous()`

Returns

- pytorch `state_dict`

A typical usage might be

```
from deepspeed.utils.zero_to_fp32 import get_fp32_state_dict_from_zero_checkpoint
# do the training and checkpoint saving
state_dict = get_fp32_state_dict_from_zero_checkpoint(checkpoint_dir) # already on␣
```

(continues on next page)

(continued from previous page)

```
↪cpu
model = model.cpu() # move to cpu
model.load_state_dict(state_dict)
# submit to model hub or save the model to share with others
```

In this example the `model` will no longer be usable in the deepspeed context of the same application. i.e. you will need to re-initialize the deepspeed engine, since `model.load_state_dict(state_dict)` will remove all the deepspeed magic from it.

If you want it all done for you, use `load_state_dict_from_zero_checkpoint` instead.

Note: the above usage may not work if your application doesn't have sufficient free CPU memory. You may need to use the offline approach using the `zero_to_fp32.py` script that is saved with the checkpoint. Or you can load state_dict in lazy mode

```
from deepspeed.utils.zero_to_fp32 import get_fp32_state_dict_from_zero_checkpoint
state_dict = get_fp32_state_dict_from_zero_checkpoint(checkpoint_dir, lazy_
↪mode=True) # not on cpu
for name, lazy_tensor in state_dict.item():
    tensor = lazy_tensor.contiguous()  # to cpu
    print(name, tensor)
    # del tensor to release memory if it no longer in use
```

deepspeed.utils.zero_to_fp32.**load_state_dict_from_zero_checkpoint**(*model*, *checkpoint_dir*, *tag=None*)

1. Put the provided model to cpu
2. Convert ZeRO 2 or 3 checkpoint into a single fp32 consolidated `state_dict`
3. Load it into the provided model

Parameters

- **model** (-) – the model object to update
- **checkpoint_dir** (-) – path to the desired checkpoint folder. (one that contains the tag-folder, like `global_step14`)
- **tag** (-) – checkpoint tag used as a unique identifier for checkpoint. If not provided will attempt to load tag in the file named `latest` in the checkpoint folder, e.g., `global_step14`

Returns

modified model

Return type

- ``model`

Make sure you have plenty of CPU memory available before you call this function. If you don't have enough use the `zero_to_fp32.py` utility to do the conversion. You will find it conveniently placed for you in the checkpoint folder.

A typical usage might be

```
from deepspeed.utils.zero_to_fp32 import load_state_dict_from_zero_checkpoint
model = load_state_dict_from_zero_checkpoint(trainer.model, checkpoint_dir)
# submit to model hub or save the model to share with others
```

Note, that once this was run, the `model` will no longer be usable in the deepspeed context of the same application. i.e. you will need to re-initialize the deepspeed engine, since `model.load_state_dict(state_dict)` will remove all the deepspeed magic from it.

`deepspeed.utils.zero_to_fp32.`**`convert_zero_checkpoint_to_fp32_state_dict`**(*checkpoint_dir*, *output_dir*, *max_shard_size='5GB'*, *safe_serialization=False*, *tag=None*, *exclude_frozen_parameters=False*)

Convert ZeRO 2 or 3 checkpoint into a single fp32 consolidated `state_dict` file that can be loaded with `torch.load(file)` + `load_state_dict()` and used for training without DeepSpeed.

Parameters

- **`checkpoint_dir`** (-) – path to the desired checkpoint folder. (one that contains the tag-folder, like `global_step14`)
- **`output_dir`** (-) – directory to the pytorch fp32 state_dict output files
- **`max_shard_size`** (-) – the maximum size for a checkpoint before being sharded, default value is 5GB
- **`safe_serialization`** (-) – whether to save the model using *safetensors* or the traditional PyTorch way (that uses *pickle*).
- **`tag`** (-) – checkpoint tag used as a unique identifier for checkpoint. If not provided will attempt to load tag in the file named `latest` in the checkpoint folder, e.g., `global_step14`
- **`exclude_frozen_parameters`** (-) – exclude frozen parameters

4.1.4 Avoiding ZeRO Checkpoint Bloat

ZeRO stage 1 and 2 checkpoints created using `torch.save()` can sometimes be larger than expected. This bloat is caused by the interaction of ZeRO's tensor flattening and torch's tensor storage management . You can avoid this problem by using the `clone_tensors_for_torch_save` utility of DeepSpeed as illustrated below.

`deepspeed.checkpoint.utils.`**`clone_tensors_for_torch_save`**(*item*, *device=device(type='cpu')*)

Returns a copy of `item` with all enclosed tensors replaced by clones on a specified device. Works on individual tensors, and tensors contained/nested in lists, tuples, and dicts.

Parameters

- **`item`** (-) – tensor to clone or (possibly nested) container of tensors to clone.
- **`device`** (-) – target device (defaults to 'cpu')

Returns

- copy of `item` with cloned tensors on target device

The following code snippet illustrates this functionality for creating a HuggingFace model checkpoint:

```
ds_config = {
  ...
}
model = AutoModelForCausalLM.from_pretrained("facebook/opt-13b", torch_dtype=torch.
↪float16)
ds_engine, _, _, _ = deepspeed.initialize(model=model, config_params=ds_config)
```

(continues on next page)

(continued from previous page)

```
lean_state_dict = deepspeed.checkpoint.utils.clone_tensors_for_torch_save(ds_engine.
↪module.state_dict())
ds_engine.module.save_pretrained("lean_after", state_dict=lean_state_dict)
```

4.1.5 Universal Checkpoints (under development)

Parallelism techniques such as ZeRO data parallelism (DP), Tensor parallelism (TP), Pipeline parallelism (TP), which shard model and/or optimizer states make it difficult to resume training with a checkpoint that was created on a different number of GPUs. DeepSpeed provides the Universal Checkpoint mechanism to address this problem. Universal Checkpoints give users the flexibility of changing the number of GPUs when training with 3D (TP, PP, and DP) parallelism, and enables more efficient use of elastic training hardware. The easiest way to get started with using Universal Checkpoints is to consult the Megatron-DeepSpeed and BLOOM examples.

4.2 Activation Checkpointing

The activation checkpointing API's in DeepSpeed can be used to enable a range of memory optimizations relating to activation checkpointing. These include activation partitioning across GPUs when using model parallelism, CPU checkpointing, contiguous memory optimizations, etc.

Please see the DeepSpeed JSON config for the full set.

Here we present the activation checkpointing API. Please see the enabling DeepSpeed for Megatron-LM tutorial for example usage.

4.2.1 Configuring Activation Checkpointing

4.2.2 Using Activation Checkpointing

4.2.3 Configuring and Checkpointing Random Seeds

CHAPTER

FIVE

ZERO API

5.1 ZeRO

The Zero Redundancy Optimizer (ZeRO) removes the memory redundancies across data-parallel processes by partitioning the three model states (optimizer states, gradients, and parameters) across data-parallel processes instead of replicating them. By doing this, it boosts memory efficiency compared to classic data-parallelism while retaining its computational granularity and communication efficiency.

1. **ZeRO Stage 1**: The optimizer states (e.g., for Adam optimizer, 32-bit weights, and the first, and second moment estimates) are partitioned across the processes, so that each process updates only its partition.
2. **ZeRO Stage 2**: The reduced 16-bit gradients for updating the model weights are also partitioned such that each process retains only the gradients corresponding to its portion of the optimizer states.
3. **ZeRO Stage 3**: The 16-bit model parameters are partitioned across the processes. ZeRO-3 will automatically collect and partition them during the forward and backward passes.

In addition, ZeRO-3 includes the *infinity offload engine* to form ZeRO-Infinity ([paper](https://arxiv.org/abs/2104.07857)), which can offload all model states to both CPU and NVMe memory for huge memory savings.

For a deep dive of our algorithms, please see our papers on ZeRO, ZeRO-Offload, and ZeRO-Infinity.

Note: DeepSpeed first included offloading capabilities with **ZeRO-Offload**, a system for offloading optimizer and gradient states to CPU memory within ZeRO-2. **ZeRO-Infinity** is the next generation of offloading capabilities, accessible to ZeRO-3. ZeRO-Infinity has all of the savings of ZeRO-Offload, plus is able to offload more the model weights and has more effective bandwidth utilization and overlapping of computation and communication.

5.1.1 Getting Started

If you are new to DeepSpeed, check out our Getting Started page.

Once you are training with DeepSpeed, enabling ZeRO-3 offload is as simple as enabling it in your DeepSpeed configuration! Below are a few examples of ZeRO-3 configurations. Please see our config guide for a complete list of options for configuration and performance tuning.

Note: ZeRO-Infinity and ZeRO-Offload work best with our heavily optimized *deepspeed.ops.adam.DeepSpeedCPUAdam* optimizer. We recommend using our optimizer config to instruct `deepspeed.initialize()` to build the optimizer for you.

ZeRO Configurations

All the settings for DeepSpeed ZeRO are set with the *DeepSpeedZeroConfig*. The dictionary provided under the `zero_optimization` entry of the main DeepSpeed configuration dict will be parsed and validated with this class. Sub-configurations for parameter offload and optimizer offload settings are parsed by *DeepSpeedZeroOffloadParamConfig* and *DeepSpeedZeroOffloadOptimizerConfig*.

class deepspeed.runtime.zero.config.**DeepSpeedZeroConfig**

Sets parameters for ZeRO optimizations.

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

stage: ZeroStageEnum = 0

Chooses different stages of ZeRO Optimizer. Stage 0, 1, 2, and 3 refer to disabled, optimizer state partitioning, and optimizer+gradient state partitioning, and optimizer+gradient+parameter partitioning, respectively.

contiguous_gradients: bool = True

Copies the gradients to a contiguous buffer as they are produced. Avoids memory fragmentation during backward pass.

reduce_scatter: bool = True

Uses reduce or reduce scatter instead of allreduce to average gradients

reduce_bucket_size: int = 500,000,000

Number of elements reduced/allreduced at a time. Limits the memory required for the allgather for large model sizes

Constraints

- **ge** = 0

use_multi_rank_bucket_allreduce: bool = True

Combine the reduce buckets of the different ranks and do an All-Reduce instead of multiple Reduce ops. This feature is useful when the model is small and we want to scale it on too many GPUs which therefore reduces the message sizes of each packet.

allgather_partitions: bool = True

Chooses between allgather collective or a series of broadcast collectives to gather updated parameters from all the GPUs at the end of each step

allgather_bucket_size: int = 500,000,000

Number of elements allgathered at a time. Limits the memory required for the allgather for large model sizes

Constraints

- **ge** = 0

overlap_comm: Optional[bool] = None

Attempts to overlap the reduction of the gradients with backward computation

load_from_fp32_weights: bool = True

Boolean indicating whether to initialize fp32 master weights from fp32 copies in checkpoint (no precision loss) or from model's fp16 copies (with precision loss). This can be used to initialize optimizer state even when checkpoint is missing optimizer state.

elastic_checkpoint: bool = False

Enable loading checkpoint that was saved by job with different GPU count. No longer supported.

offload_param: Optional[*DeepSpeedZeroOffloadParamConfig*] = None

Enable offloading of model parameters to CPU or NVMe. This frees up GPU memory for larger models or batch sizes. Valid only with stage 3. Expects a dictionary containing values for *DeepSpeedZeroOffloadParamConfig*.

offload_optimizer: Optional[*DeepSpeedZeroOffloadOptimizerConfig*] = None

Enable offloading of optimizer state to CPU or NVMe, and optimizer computation to CPU. This frees up GPU memory for larger models or batch sizes. Valid for ZeRO stage 1, 2, 3. Expects a dictionary containing values for *DeepSpeedZeroOffloadOptimizerConfig*.

zenflow: Optional[ZenFlowConfig] = None

Enable ZenFlow

sub_group_size: int = 1,000,000,000

Tile size for parameter processing to fit massive models (with trillions of parameters). Used by ZeRO3-Offload and ZeRO-Infinity

Constraints

- **ge** = 0

cpu_offload_param: Optional[bool] = None

Deprecated, please use `offload_param`

cpu_offload_use_pin_memory: Optional[bool] = None

Deprecated, please use `offload_param` or `offload_optimizer`

cpu_offload: Optional[bool] = None

Deprecated, please use `offload_optimizer`

prefetch_bucket_size: int = 50,000,000 (alias 'stage3_prefetch_bucket_size')

Maximum number of parameter elements to fetch ahead of use. Used by ZeRO3, ZeRO3-Offload, ZeRO-Infinity, and ZeRO-Inference.

Constraints

- **ge** = 0

param_persistence_threshold: int = 100,000 (alias 'stage3_param_persistence_threshold')

Do not partition parameters smaller than this threshold. Smaller values use less memory, but can greatly increase communication (especially latency-bound messages).

Constraints

- **ge** = 0

model_persistence_threshold: int = sys.maxsize (alias 'stage3_model_persistence_threshold')

Maximum number of parameter elements that can be persisted in GPU and not partitioned. This imposes an upper bound on the number of unpartitioned parameters resulting from param_persistence_threshold setting. Used by ZeRO3-Offload, ZeRO-Infinity and ZeRO-Inference.

Constraints

- **ge** = 0

`max_live_parameters: int = 1,000,000,000 (alias 'stage3_max_live_parameters')`

The maximum number of parameters resident per GPU before releasing. Smaller values use less memory, but perform more communication.

Constraints

- **ge** = 0

`max_reuse_distance: int = 1,000,000,000 (alias 'stage3_max_reuse_distance')`

Do not release a parameter if it will be reused within this threshold of parameters. Smaller values use less memory, but perform more communication.

Constraints

- **ge** = 0

`gather_16bit_weights_on_model_save: bool = False (alias 'stage3_gather_16bit_weights_on_model_save')`

Consolidate the weights before saving the model by `save_16bit_model()`. Since the weights are partitioned across GPUs, they aren't part of `state_dict`, so this function automatically gathers the weights when this option is enabled and then saves the fp16 model weights.

`module_granularity_threshold: int = 0 (alias 'stage3_module_granularity_threshold')`

The granularity of a module is determined by the ratio of "parameter_count / (1 + descendant count)". ZeRO3 classifies modules with a granularity below the threshold as fine-grained, which are treated as integral units during parameter fetching. This reduces host overhead and the separate allgather overhead introduced by hooks for fine-grained layers when fetching parameters.

`use_all_reduce_for_fetch_params: bool = False (alias 'stage3_use_all_reduce_for_fetch_params')`

Use all_reduce op when fetching module parameters at stage3. This improves performance by reducing the overhead of concatenation and slicing on the host.

`allgather_sequential: bool = False (alias 'stage3_allgather_sequential')`

Performs allgather on individual parameters sequentially, bypassing the standard parameter bucketing mechanism in stage3. This significantly reduces data copy overhead (eliminating copy-to-bucket operations) and lowers peak memory usage by avoiding the allocation of large temporary flattening buffers. Recommended for scenarios with high memory pressure.

`stage3_gather_fp16_weights_on_model_save: bool = False`

Deprecated, please use `gather_16bit_weights_on_model_save`

`ignore_unused_parameters: bool = True`

Unused parameters in modules may be unexpected in static networks, but could be normal in dynamic networks. This controls whether or not training should terminate with an error message when unused parameters are detected. This is set to `True` by default, which means unused parameters are ignored and training continues. Now is just used in stage 2.

`legacy_stage1: bool = False`

For backward-compatibility enable old ZeRO stage 1 implementation. Use at your own risk, will be deprecated soon.

`round_robin_gradients: bool = False`

Stage 1 and 2 optimization for CPU offloading that parallelizes gradient copying to CPU memory among ranks by fine-grained gradient partitioning. Performance benefit grows with gradient accumulation steps (more copying between optimizer steps) or GPU count (increased parallelism).

`zero_hpz_partition_size: int = 1`

Number of ranks in zero parameters partitioning secondary group

Constraints

- **ge** = 0

`zero_quantized_weights: bool = False`

Boolean indicating whether to quantize zero parameters (weights) for efficient all_gather comm

`zero_quantized_nontrainable_weights: bool = False`

Boolean indicating whether to quantize non-trainable zero parameters (weights) for efficient memory usage and communication. Different from zero_quantized_weights that stores the weights in original precision and only perform quantization during communication, this flag will store the weights in quantized precision. This is useful for LoRA training.

`zero_quantized_gradients: bool = False`

Boolean indicating whether to use quantized zero gradients for efficient all_2_all_reduce comm

`zeropp_loco_param: Optional[Dict[str, Any]] = None`

This dictionary contains parameters for using LoCo-Zero++, with two key parameters: - *err_beta*: A coefficient for the moving average of quantization errors before and after gradient computation. It ranges between 0 and 1, with a default value of 0.8. - *reset_T*: The number of steps after which the moving-average error buffer is cleared. The default value is 1024. These parameters can be adjusted based on performance needs. Example configuration in ds config: "zeropp_loco_param": { "err_beta": 0.8, "reset_T": 1024 }. See LoCo paper for more details: (https://arxiv.org/abs/2407.04480).

`mics_shard_size: int = -1`

`mics_hierarchical_params_gather: bool = False`

`memory_efficient_linear: bool = True`

Use memory efficient linear implementation, for Stage 3.

`pipeline_loading_checkpoint: bool = False`

`override_module_apply: bool = True`

Override nn.Module apply function, for Stage 3.

`log_trace_cache_warnings: bool = False`

Whether to log warnings from trace cache, such as invalidation events.

`enable_sanity_checks: bool = False`

Enable internal sanity checks, which could be useful for debugging

`save_muon_momentum_buffer_in_memory: bool = False`

When using the Muon optimizer with ZeRO Stage 3, keeps the Muon momentum buffer in GPU/CPU memory instead of swapping to NVMe with other optimizer states. Only relevant when using NVMe offloading.

`leaf_module: DeepSpeedZeroLeafModuleConfig [Optional]`

Configuration for modules that should be treated as ZeRO3 leaf modules.

class deepspeed.runtime.zero.config.**DeepSpeedZeroOffloadParamConfig**

Set options for parameter offload. Valid only with stage 3.

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

device: OffloadDeviceEnum = 'none'

Device memory to offload model parameters. Supported options are *cpu* and *nvme*.

nvme_path: Optional[Path] = None

Filesystem path for NVMe device for parameter offloading.

buffer_count: int = 5

Number of buffers in buffer pool for parameter offloading to NVMe.

Constraints

- **ge** = 0

buffer_size: int = 100,000,000

Size of buffers in buffer pool for parameter offloading to NVMe.

Constraints

- **ge** = 0

max_in_cpu: int = 1,000,000,000

Number of parameter elements to maintain in CPU memory when offloading to NVMe is enabled.

Constraints

- **ge** = 0

pin_memory: bool = False

Offload to page-locked CPU memory. This could boost throughput at the cost of extra memory overhead.

class deepspeed.runtime.zero.config.**DeepSpeedZeroOffloadOptimizerConfig**

Set options for optimizer offload. Valid with stage 1, 2, and 3.

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

device: OffloadDeviceEnum = 'none'

Device memory to offload optimizer state. Supported options are *cpu* and *nvme*. Optimizer computation is offload to CPU regardless of device option.

nvme_path: Optional[Path] = None

Filesystem path for NVMe device for optimizer state offloading.

buffer_count: int = 4

Number of buffers in buffer pool for optimizer state offloading to NVMe. This should be at least the number of states maintained per parameter by the optimizer. For example, Adam optimizer has 4 states (parameter, gradient, momentum, and variance).

Constraints

- **ge** = 0

pin_memory: bool = False

Offload to page-locked CPU memory. This could boost throughput at the cost of extra memory overhead.

pipeline_read: bool = False

For tile-based optimizer step processing, overlap read of next tile with computation of current tile. Used in ZeRO-Infinity.

pipeline_write: bool = False

For tile-based optimizer step processing, overlap write of previous tile with computation of current tile.

fast_init: bool = False

Enable fast optimizer initialization when offloading to NVMe.

ratio: float = 1.0

Percentage of offloaded optimizer states to CPU Adam. Only valid with ZeRO Stage 3.

Constraints

- **ge** = 0.0
- **le** = 1.0

super_offload: bool = False

Enable high performance CPU offloading for Superchips. Only valid with ZeRO Stage 3.

cpuadam_cores_perc: float = 0.8

Percentage of CPU cores to use for CPU Adam. Only valid with ZeRO Stage 3 and super_offload=True.

Constraints

- **ge** = 0.0
- **le** = 1.0

Example ZeRO-3 Configurations

1. Use ZeRO to partition the optimizer states (stage 1), gradients (stage 2), and parameters (stage 3).

```
{
    "zero_optimization": {
        "stage": 3,
    },
    "fp16": {
        "enabled": true
    },
    "optimizer": {
        "type": "AdamW",
        "params": {
        "lr": 0.001,
        "betas": [
            0.8,
            0.999
        ],
        "eps": 1e-8,
        "weight_decay": 3e-7
```

(continues on next page)

(continued from previous page)

```
        }
    },
    ...
}
```

2. Additionally offload the optimizer states and computations to the CPU with ZeRO-Infinity.

```
{
    "zero_optimization": {
        "stage": 3,
        "offload_optimizer": {
            "device": "cpu"
        }
    },
    ...
}
```

3. Save even more memory by offloading parameters to the CPU memory.

```
{
    "zero_optimization": {
        "stage": 3,
        "offload_optimizer": {
            "device": "cpu"
        }
        "offload_param": {
            "device": "cpu"
        }
    },
    ...
}
```

4. Save even MORE memory by offloading to NVMe (if available on your system):

```
{
    "zero_optimization": {
        "stage": 3,
        "offload_optimizer": {
            "device": "nvme",
            "nvme_path": "/nvme_data"
        }
        "offload_param": {
            "device": "nvme",
            "nvme_path": "/nvme_data"
        }
    },
    ...
}
```

MiCS Configurations

All MiCS configurations are set with *DeepSpeedZeroConfig*. MiCS assumes ZeRO stage 3 optimization is enabled. For now, there are two configuration fields of MiCS *mics_shard_size* and *mics_hierarchical_params_gather*. *mics_shard_size* controls how many devices are used for partitioning the model states. *mics_hierarchical_params_gather* controls whether we use a two-stage hierarchical way to gather parameters in the forward computation. *mics_hierarchical_params_gather* is useful when model states are partitioned across multiple nodes and the cross-node bandwidth is slow. By default this is turned off.

Example MiCS Configurations

1. Use MiCS to partition the model states (including optimizer states, gradients, and parameters). The following config example partitions the model states to eight devices, and assumes the eight devices are located within a single node (*mics_hierarchical_params_gather* is *False*).

```
{
    "zero_optimization": {
        "stage": 3,
        "mics_shard_size": 8,
        "mics_hierarchical_params_gather": False,
    },
    ...
}
```

Assumptions

DeepSpeed automatically coordinates the collection (*i.e.,* all-gather), partitioning (*i.e.,* scatter), and offloading of parameters at the granularity of (sub)module `forward()` methods. The backward pass is handled similarly. This strategy has two underlying assumptions:

1. The forward and backward passes of submodules must individually fit in device memory. If this not the case, `deepspeed.zero.TiledLinear` implements **memory-centric tiling** and works with ZeRO-3 to break linear layers into a sequence of smaller submodules that can fit in memory.
2. A module's parameters are only accessed within its own `__init__` and `forward()` methods. Otherwise, DeepSpeed must be instructed to collect and re-partition the parameter. See *Manual Parameter Coordination* for manually coordinating parameters.

5.1.2 Constructing Massive Models

ZeRO-3 enables massive models whose parameters exceed the size of individual nodes in a system. For the typical case of training without model parallelism, you can simply allocate your model in our context:

```
with deepspeed.zero.Init():
    model = MyLargeModel()
```

5.1.3 Manual Parameter Coordination

Most models require no modification to be trained with ZeRO-3. However, in some cases one may need to access model weights outside of the training loop, or to share weights across submodules during training. DeepSpeed has several mechanisms to coordinate partitioned weights for ZeRO-3.

Gathering Parameters

DeepSpeed provides mechanisms for collecting (or *gathering*) a partitioned parameter.

Some models partitioned with `deepspeed.zero.Init` may need to access a module's weights outside of the class constructor or its `forward()` method. We refer to these weights as **external parameters**, since these parameters are accessed outside of the module that created them. To do so, use `deepspeed.zero.GatheredParameters` or `deepspeed.zero.register_external_parameter()`.

Registering External Parameters

ZeRO-3 will automatically collect and partition the model parameters as they are needed during the forward and backward passes. However, in some cases a parameter may be used outside of its module's forward pass. We call these *external* parameters. ZeRO-3 can coordinate these parameters if they are registered either automatically or manually.

Note: DeepSpeed version `0.3.15` includes automatic external parameter discovery and registration to support the most common cases. Parameters can still be manually registered if they cannot be automatically detected.

DeepSpeed can automatically detect the following external parameter scenarios:

1. Parameter access: consider the following pattern common in language models such as GPT:

   The tensor `embeddings.weight` is used in both `embeddings.forward()` and `compute_logits()`. We call `embeddings.weight` an *external* parameter because it is used in the training loop outside of its owning module's forward pass.

```
class LanguageModel(torch.nn.Module):
    ...
    def forward(self, inputs):
        embeds = self.embeddings(inputs)
        ...
        logits = compute_logits(output, self.embeddings.weight)
        ...
```

2. Returning a parameter:

   `CustomLinear` returns both an output and its own `bias` parameter. DeepSpeed will detect the external `bias` parameter and register it with submodules that use `CustomLinear`.

```
class CustomLinear(torch.nn.Linear):
    def forward(self, *input):
        output = super().forward(*input)
        return output, self.bias
```

Overriding Module.apply

A convenient mechanism for customizing model initialization is Module.apply. With ZeRO stage 3, `Module.apply` implementations must account for parameter partitioning by `zero.Init` during model initialization. The default behavior of ZeRO stage 3 is to automatically handle this issue by overriding `Module.apply` to ensure that parameters are gathered before access by `Module.apply`. The benefit of this approach is development convenience, since users are saved the burden of manual parameter coordination in `Module.apply`. However, the downside is slow model initialization, since all the model parameters (e.g., billions) are gathered even though the common usage of `Module.apply` is to customize a few parameters. Developers can disable this default behavior by setting the `override_module_apply` configuration knob to `False`, for faster model initialization at the cost of manually handling partitioned parameters in their `Module.apply` implementations.

5.1.4 Memory-Centric Tiling

To reduce the working memory requirements of DL training for large models, ZeRO-Infinity includes technique called *memory-centric tiling* that exploits the data fetch and release pattern of ZeRO-3 to reduce the working memory requirements by breaking down a large operator into smaller tiles that can be executed sequentially. When combined with ZeRO-3, the parameter and gradients of each tile can be fetched and released one at a time, reducing the working memory proportional to the number of tiles. Therefore, ZeRO-Infinity can support operators of arbitrary sizes, without refactoring for model parallelism to fit them in limited GPU memory.

5.1.5 Debugging

Debugging ZeRO training is complicated by the partitioning of parameters, gradients, and optimizer states. None of these 3 groups of tensors (model states) can be normally accessed because of that. To overcome that DeepSpeed provides the following routines for accessing individual model states in both their partitioned (local) and unpartitioned (full) forms.

Important notes:

These APIs return tensors that are on accelerator device even if the corresponding model state is offloaded to CPU or NVMe.

To access the unpartitioned (full) form, these utilities must be called by all processes participating in the training, even if you decide to do something with the result only in the main process. If all processes don't participate these utilities will hang waiting for all processes to send their contribution.

You must be aware that these routines return correct data only in specific phases of the training. So for examples the gradients are valid after `backward` and before `step`. The optimizer states are updated after `step`. Same goes for fp32 master weights.

`deepspeed.utils.`**`safe_get_full_fp32_param`**(*param*)

Assemble and return the fp32 parameter of a low-precision (e.g., fp16) parameter.

Parameters
param (`torch.nn.Parameter`) – A model parameter

Returns
A tensor on accelerator device

Return type
Union[torch.Tensor, None]

`deepspeed.utils.`**`safe_get_full_grad`**(*param*)

Assemble and return the fp32 gradient of a low-precision (e.g., fp16) parameter. The return data type is that

used for gradient accumulation. This is usually the param data type, but could also be different (e.g., bf16 param training with fp32 gradient accumulation).

Parameters
param (torch.nn.Parameter) – A model parameter

Returns
A tensor on accelerator device

Return type
Union[torch.Tensor, None]

deepspeed.utils.**safe_get_full_optimizer_state**(*param*, *optim_state_key*)

Assemble and return the fp32 optimizer state of a low-precision (e.g., fp16) parameter.

Parameters

- **param** (torch.nn.Parameter) – A model parameter
- **optim_state_key** (string) – Key value of optimizer state (e.g., *exp_avg* in Adam optimizer)

Returns
A tensor on accelerator device

Return type
Union[torch.Tensor, None]

deepspeed.utils.**safe_get_local_fp32_param**(*param*)

Get the local partition of a ZeRO-3 partitioned parameter in fp32 precision.

Parameters
param (torch.nn.Parameter) – A model parameter.

Returns
A tensor on accelerator device

Return type
Union[torch.Tensor, None]

deepspeed.utils.**safe_get_local_grad**(*param*)

Get the local gradient partition of a ZeRO-3 partitioned parameter. The return data type is that used for gradient accumulation. This is usually the param data type, but could also be different (e.g., bf16 param training with fp32 gradient accumulation).

Parameters
param (torch.nn.Parameter) – A model parameter

Returns
A tensor on accelerator device

Return type
Union[torch.Tensor, None]

deepspeed.utils.**safe_get_local_optimizer_state**(*param*, *optim_state_key*)

Get the local optimizer state partition of ZeRO-3 partitioned parameter in fp32 precision.

Parameters

- **param** (torch.nn.Parameter) – A model parameter
- **optim_state_key** (string) – Key value of optimizer state (e.g., *exp_avg* in Adam optimizer)

Returns
A tensor on accelerator device

Return type
Union[torch.Tensor, None]

These routines can be used in a training loop as shown in the following snippet.

```
backward(loss)
[...]
from deepspeed.utils import safe_get_full_fp32_param, safe_get_full_grad, safe_get_full_
↪optimizer_state
for n, lp in model.named_parameters():
    # 1. Access the full states
    #  1.1) gradient lookup
    # For zero1 and zero2, gradient lookup must be called after `backward` and before␣
↪`step`
    # For zero3, gradient lookup must be called after `backward`
    hp_grad = safe_get_full_grad(lp)


    # 1.2) fp32 and optim states can probably be called anywhere in the training loop,␣
↪but will be updated after `step`
    hp = safe_get_full_fp32_param(lp)
    exp_avg = safe_get_full_optimizer_state(lp, "exp_avg")
    exp_avg_sq = safe_get_full_optimizer_state(lp, "exp_avg_sq")

    # 2. Access the local states (zero3)
    # For zero3, all of the parameters, gradients, and optimizer states are partitioned,
    # and each process can access its corresponding local state.
    local_hp = safe_get_local_fp32_param(lp)
    local_hp_grad = safe_get_local_grad(lp)
    local_exp_avg = safe_get_local_optimizer_state(lp, "exp_avg")
    local_exp_avg_sq = safe_get_local_optimizer_state(lp, "exp_avg_sq")

[...]
optimizer.step()
```

5.1.6 Modifying Partitioned States

Sometimes, a user may want to modify parameters, gradients, or optimizer states outside of the regular training loop. This is currently difficult in ZeRO training because of partitioning. To overcome that, DeepSpeed provides the following routines for modifying the fp32 master parameters and the fp32 optimizer states.

deepspeed.utils.**safe_set_full_fp32_param**(*param*, *value*)

Update the partitioned fp32 parameter of a low-precision (e.g., fp16) parameter.

Parameters

- **param** (torch.nn.Parameter) – A model parameter
- **value** (torch.Tensor) – New value

deepspeed.utils.**safe_set_full_optimizer_state**(*param*, *value*, *optim_state_key*)

Update the partitioned fp32 optimizer state of a low-precision (e.g., fp16) parameter.

Parameters

- **param** (torch.nn.Parameter) – A model parameter
- **value** (torch.Tensor) – New value
- **optim_state_key** (string) – Key value of optimizer state (e.g., *exp_avg* in Adam optimizer)

deepspeed.utils.**safe_set_full_grad**(*param*, *value*)

Update the partitioned gradient of a low-precision (e.g., fp16) parameter. To avoid precision issues, the update value should have the data type of gradient accumulation.

Parameters

- **param** (torch.nn.Parameter) – A model parameter
- **value** (torch.Tensor) – The un-partitioned new gradient value.

deepspeed.utils.**safe_set_local_fp32_param**(*param*, *value*)

Update the local partition of ZeRO-3 partitioned parameter.

Parameters

- **param** (torch.nn.Parameter) – A model parameter.
- **value** (torch.Tensor) – New value of local parameter partition.

deepspeed.utils.**safe_set_local_grad**(*param*, *value*)

Update the local gradient partition of a ZeRO-3 partitioned parameter. To avoid precision issues, the update value should have the data type of gradient accumulation.

Parameters

- **param** (torch.nn.Parameter) – A model parameter.
- **value** (torch.Tensor) – New value of local gradient partition.

deepspeed.utils.**safe_set_local_optimizer_state**(*param*, *value*, *optim_state_key*)

Update the local optimizer state partition of a ZeRO-3 partitioned parameter.

Parameters

- **param** (torch.nn.Parameter) – A model parameter.
- **value** (torch.Tensor) – New value of local optimizer state partition.
- **optim_state_key** (string) – Key value of optimizer state (e.g., *exp_avg* in Adam optimizer).

deepspeed.utils.**safe_update_full_grad_vectorized**(*param_list: List[Parameter]*, *update_func: Callable*)

Vectorized update of the partitioned gradients of a list of low-precision (e.g., fp16) parameters. To avoid precision issues, the update value should have the data type of gradient accumulation.

Parameters

- **param_list** (List[torch.nn.Parameter]) – List of model parameters
- **update_func** (torch.Tensor) – A function that takes current full gradient value and returns new one.

The routines for modifying parameters and optimizer states can be used at any point after initialization of the DeepSpeed engine (i.e., deepspeed.initialize()) as shown in the following snippet.

```
[...]
from deepspeed.runtime.zero.utils import is_zero_param
from deepspeed.utils import safe_set_full_fp32_param, safe_set_full_optimizer_state
from deepspeed.utils import safe_set_local_fp32_param, safe_set_local_optimizer_state
# Here is an example to zero all the fp32 parameters and optimizer states.
for n, lp in model.named_parameters():
    # 1. For zero stage 1, 2, or 3 set the full fp32 and their full optim states
    zero_tensor = torch.zeros(lp.ds_shape) if is_zero_param(lp) else torch.zeros(lp.
↪shape)

    safe_set_full_fp32_param(lp, zero_tensor)
    safe_get_full_optimizer_state(lp, zero_tensor, "exp_avg")
    safe_get_full_optimizer_state(lp, zero_tensor, "exp_avg_sq")

    # 2. For zero stage 3, each process sets its local fp32 parameters and their local␣
↪optimizer states individually
    zero_tensor_local = torch.zeros(lp.ds_tensor.shape)

    safe_set_local_fp32_param(lp, zero_tensor_local)
    safe_set_local_optimizer_state(lp, zero_tensor_local, "exp_avg")
    safe_set_local_optimizer_state(lp, zero_tensor_local, "exp_avg_sq")

[...]
```

The routines for modifying gradients can be used after `backward` but before `step` as shown in the following snippet.

```
backward(loss)
[...]
from deepspeed.runtime.zero.utils import is_zero_param
from deepspeed.utils import safe_set_full_grad, safe_set_local_grad
# Here is an example of how to zero all the gradients.
for n, lp in model.named_parameters():
    # 1. For zero stage 1, 2, or 3 set the full gradient.
    zero_tensor = torch.zeros(lp.ds_shape) if is_zero_param(lp) else torch.zeros(lp.
↪shape)

    safe_set_full_grad(lp, zero_tensor)

    # 2. For zero stage 3, each process sets its local gradient partition.
    zero_tensor_local = torch.zeros_like(lp.ds_tensor.shape)

    safe_set_local_grad(lp, zero_tensor_local)

[...]
optimizer.step()
```

5.1.7 GPU Memory Management

By default at the end of training with ZeRO stage 3 some parameters could remain unpartitioned and use up some gpu memory. This is done on purpose as an optimization should you resume training again. If you'd like to clear out the cached parameters that use up gpu memory, you can call `empty_partition_cache` method of a DeepSpeed engine.

The following code snippet illustrates this functionality.

```
with zero.Init():
    model = MyLargeModel()

ds_engine, _, _, _ = deepspeed.initialize(model, ...)
for batch in ...:
    loss = ds_engine(batch)
    ds_engine.backward(batch)
    ds_engine.step()

# Free GPU memory consumed by model parameters
ds_engine.empty_partition_cache()
```

5.1.8 Offload States

The DeepSpeed engine maintains a set of states in device memory (e.g., CUDA memory). The following API allows you to offload these states to a different device (currently, only CPU memory is supported), reducing the memory footprint on the device.

```
def offload_states(self,
                   include: Container[OffloadStateTypeEnum] = None,
                   device: OffloadDeviceEnum = OffloadDeviceEnum.cpu,
                   pin_memory: bool = True,
                   non_blocking: bool = False) -> None:
    """Offload the engine's states to the specified device.

    Arguments:
        include: Optional. The set of states to offload. If not provided, all states are␣
↪offloaded.
        device: Optional. The device to move the ZeRO optimizer buffers to. Currently␣
↪only `OffloadDeviceEnum.cpu` is supported.
        pin_memory: Optional. Whether to pin the memory of the offloaded states.
        non_blocking: Optional. Whether to offload the states asynchronously.
    """
```

You can selectively offload specific states by specifying the `OffloadStateTypeEnum` in the include argument. `OffloadStateTypeEnum` is an enum that defines the states that can be offloaded. The following states are supported:

- `OffloadStateTypeEnum.optim_states`: Optimizer states. Currently, only states of DeepSpeed's FusedAdam optimizer are supported.
- `OffloadStateTypeEnum.hp_params`: FP32 parameters.
- `OffloadStateTypeEnum.lp_params`: BF16/FP16 parameters.
- `OffloadStateTypeEnum.lp_grads`: BF16/FP16 gradients.
- `OffloadStateTypeEnum.contiguous_grad_buffer`: The contiguous gradient buffer for reduce operations.

Note that offloading states comes with a trade-off between memory savings and computational overhead. This API allows states to be reloaded back into device memory when needed.

```
def reload_states(self, non_blocking: bool = False) -> None:
    """Reload the engine states to the original device.

    Arguments:
        non_blocking: Optional. Whether to offload the states asynchronously.
    """
```

Below is an example code snippet demonstrating how to offload FP32 parameters and optimizer states to CPU memory:

```
# Offload after forward, backward, and step
ds_engine.offload_states(include=[OffloadStateTypeEnum.hp_params, OffloadStateTypeEnum.
↪optim_states])

# Do something requiring a lot of device memory
...
# Load states back to device memory
ds_engine.reload_states()
```

`deepspeed.runtime.zero.offload_states.get_state_devices` returns devices of the specified state.

```
def get_state_devices(model, state: OffloadStateTypeEnum) -> Set[torch.device]:
    """Retrieve the devices of the specified state of the model.

    Args:
        model (DeepSpeedEngine): The model whose device allocations are to be checked.
        state (OffloadStateTypeEnum): The specific state for which the devices should be␣
↪retrieved.

    Returns:
        Set[torch.device]: A set of devices of the specified state.

    """
```

CHAPTER

SIX

MIXTURE OF EXPERTS

6.1 AutoEP (Automatic Expert Parallelism)

AutoEP automatically detects MoE layers in Hugging Face models and replaces them with EP-enabled versions, requiring zero model code changes. It follows the pattern of AutoTP (Automatic Tensor Parallelism).

This API is separate from the explicit `deepspeed.moe.layer.MoE` layer API. For the explicit DeepSpeed MoE layer API, see *Mixture of Experts (DeepSpeed MoE)*.

Built-in AutoEP presets: `mixtral` (Mixtral), `qwen3_moe` (Qwen3-MoE), `qwen3_5_moe` (Qwen3.5-MoE), `deepseek_v2` (DeepSeek-V2), and `deepseek_v3` (DeepSeek-V3).

The preset name means AutoEP knows the router, expert, and weight naming patterns for that model family. Running a Hugging Face model also requires a Transformers build that exposes the matching config/model classes, `model.config.model_type` value, and fused expert layout.

Table 1: AutoEP preset compatibility by Transformers version

Preset	Minimum Transformers version	Notes
`mixtral`	`5.0.0`	
`qwen3_moe`	`5.0.0`	Also covers Qwen2-MoE when the installed Transformers build uses the validated fused expert layout. Qwen3-MoE classes appear in `4.51.3`, but the tested `4.x` builds do not match the validated AutoEP layout.
`qwen3_5_moe`	`5.2.0`	Requires the Qwen3.5 text-backbone `qwen3_5_moe_text` model type; for performance on Qwen3.5's Gated DeltaNet layers, install optimized kernels. See the Hugging Face Transformers kernel loading docs and the Qwen FlashQLA blog.
`deepseek_v2`	`4.0.0`	`load_balance_coeff` / expert-bias auxiliary-loss-free load balancing is not currently supported; non-null values are rejected.
`deepseek_v3`	`4.0.0`	`load_balance_coeff` / expert-bias auxiliary-loss-free load balancing is not currently supported; non-null values are rejected.

ZeRO compatibility: Stages 0, 1, and 2. Stage 3 is not supported.

Usage:

```
{
    "expert_parallel": {
        "enabled": true,
        "autoep_size": 4,
        "preset_model": "mixtral"
```

(continues on next page)

(continued from previous page)

```
    }
}
```

How it works:

1. During `deepspeed.initialize()`, AutoEP scans the model for MoE layers using preset-defined patterns (router name, expert name, weight shapes).
2. Detected MoE blocks are replaced with `AutoEPMoELayer`, which uses TorchTitan's grouped GEMM kernels and AllToAll token dispatch.
3. EP/EDP process groups are created automatically based on `autoep_size`.
4. Expert parameters are marked for expert-data-parallel gradient reduction; router and shared-expert parameters use standard data-parallel reduction.

Constraints:

- `autoep_size` must divide `num_experts` for all detected MoE layers.
- `autoep_size=1` is valid: all experts remain local (no AllToAll), useful for functional testing on a single GPU.
- AutoEP currently cannot be combined with AutoTP (`tensor_parallel.autotp_size > 1`) or tensor model parallelism from `mpu`; support is planned as follow-up work.
- AutoEP currently supports ZeRO stages 0, 1, and 2 only. ZeRO stage 3 and its partitioned-parameter get/set APIs are outside the scope of the current AutoEP support.
- Checkpoint save/load requires matching `autoep_size`. To change `autoep_size` across runs for the same AutoEP-detected model topology, convert the checkpoint to Universal Checkpoint format and load it with `checkpoint.load_universal`; see the Universal Checkpointing tutorial for the detailed flow and constraints.
- DeepSeek-V2 and DeepSeek-V3 AutoEP do not support load-balance expert bias yet. The built-in DeepSeek presets disable it by default; explicit non-null values fail.

6.2 Mixture of Experts (DeepSpeed MoE)

DeepSpeed provides two forms of MoE support: DeepSpeed MoE and *AutoEP (Automatic Expert Parallelism)*. DeepSpeed MoE is the explicit `deepspeed.moe.layer.MoE` API for constructing MoE layers in model code. This page introduces the DeepSpeed MoE API.

See also the Mixture of Experts (DeepSpeed MoE) tutorial for training examples and configuration details.

class deepspeed.moe.layer.**MoE**(*hidden_size: int*, *expert: Module*, *num_experts: int = 1*, *ep_size: int = 1*, *k: int = 1*, *capacity_factor: float = 1.0*, *eval_capacity_factor: float = 1.0*, *min_capacity: int = 4*, *use_residual: bool = False*, *noisy_gate_policy: Optional[str] = None*, *drop_tokens: bool = True*, *use_rts: bool = True*, *use_tutel: bool = False*, *enable_expert_tensor_parallelism: bool = False*, *top2_2nd_expert_sampling: bool = True*)

Initialize an MoE layer.

Parameters

- **hidden_size** (*int*) – the hidden dimension of the model, importantly this is also the input and output dimension.
- **expert** (*nn.Module*) – the torch module that defines the expert (e.g., MLP, torch.linear).
- **num_experts** (*int, optional*) – default=1, the total number of experts per layer.

- **ep_size** (*int, optional*) – default=1, number of ranks in the expert parallel world or group.
- **k** (*int, optional*) – default=1, top-k gating value, only supports k=1 or k=2.
- **capacity_factor** (*float, optional*) – default=1.0, the capacity of the expert at training time.
- **eval_capacity_factor** (*float, optional*) – default=1.0, the capacity of the expert at eval time.
- **min_capacity** (*int, optional*) – default=4, the minimum capacity per expert regardless of the capacity_factor.
- **use_residual** (*bool, optional*) – default=False, make this MoE layer a Residual MoE (https://arxiv.org/abs/2201.05596) layer.
- **noisy_gate_policy** (*str, optional*) – default=None, noisy gate policy, valid options are 'Jitter', 'RSample' or 'None'.
- **drop_tokens** (*bool, optional*) – default=True, whether to drop tokens - (setting to False is equivalent to infinite capacity).
- **use_rts** (*bool, optional*) – default=True, whether to use Random Token Selection.
- **use_tutel** (*bool, optional*) – default=False, whether to use Tutel optimizations (if installed).
- **enable_expert_tensor_parallelism** (*bool, optional*) – default=False, whether to use tensor parallelism for experts
- **top2_2nd_expert_sampling** (*bool, optional*) – default=True, whether to perform sampling for 2nd expert

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*hidden_states: Tensor*, *used_token: Optional[Tensor] = None*) → Tuple[Tensor, Tensor, Tensor]

MoE forward

Parameters

- **hidden_states** (*Tensor*) – input to the layer
- **used_token** (*Tensor, optional*) – default: None, mask only used tokens

Returns

A tuple including output, gate loss, and expert count.

- output (Tensor): output of the model
- l_aux (Tensor): gate loss value
- exp_counts (Tensor): expert count

CHAPTER

SEVEN

TRANSFORMER KERNEL API

7.1 Transformer Kernels

The transformer kernel API in DeepSpeed can be used to create BERT transformer layer for more efficient pre-training and fine-tuning, it includes the transformer layer configurations and transformer layer module initialization.

Here we present the transformer kernel API. Please see the BERT pre-training tutorial for usage details.

7.1.1 DeepSpeed Transformer Config

7.1.2 DeepSpeed Transformer Layer

CHAPTER

EIGHT

PIPELINE PARALLELISM

8.1 Pipeline Parallelism

8.1.1 Model Specification

class deepspeed.pipe.**PipelineModule**(*layers*, *num_stages=None*, *topology=None*, *loss_fn=None*, *seed_layers=False*, *seed_fn=None*, *base_seed=1234*, *partition_method='parameters'*, *activation_checkpoint_interval=0*, *activation_checkpoint_func=<function checkpoint>*, *checkpointable_layers=None*, *dynamic_shape=False*)

Modules to be parallelized with pipeline parallelism.

The key constraint that enables pipeline parallelism is the representation of the forward pass as a sequence of layers and the enforcement of a simple interface between them. The forward pass is implicitly defined by the module `layers`. The key assumption is that the output of each layer can be directly fed as input to the next, like a `torch.nn.Sequence`. The forward pass is implicitly:

```
def forward(self, inputs):
    x = inputs
    for layer in self.layers:
        x = layer(x)
    return x
```

Note: Pipeline parallelism is not compatible with ZeRO-2 and ZeRO-3.

Parameters

- **layers** (*Iterable*) – A sequence of layers defining pipeline structure. Can be a `torch.nn.Sequential` module.
- **num_stages** (*int, optional*) – The degree of pipeline parallelism. If not specified, `topology` must be provided.
- **topology** (`deepspeed.runtime.pipe.ProcessTopology`, optional) – Defines the axes of parallelism axes for training. Must be provided if `num_stages` is `None`.
- **loss_fn** (*callable, optional*) – Loss is computed `loss = loss_fn(outputs, label)`
- **seed_layers** (*bool, optional*) – Use a different seed for each layer. Defaults to False.

- **seed_fn** (*type, optional*) – The custom seed generating function. Defaults to random seed generator.
- **base_seed** (*int, optional*) – The starting seed. Defaults to 1234.
- **partition_method** (*str, optional*) – The method upon which the layers are partitioned. Defaults to 'parameters'.
- **activation_checkpoint_interval** (*int, optional*) – The granularity activation checkpointing in terms of number of layers. 0 disables activation checkpointing.
- **activation_checkpoint_func** (*callable, optional*) – The function to use for activation checkpointing. Defaults to `deepspeed.checkpointing.checkpoint`.
- **checkpointable_layers** (*list[str], optional*) – List of layer class names that are eligible for checkpointing. For GPT models, ParallelTransformerLayerPipe is always checkpointed regardless of this list. If None, all layers with parameters are considered checkpointable. Defaults to None.
- **dynamic_shape** – Allows dynamic shapes of inputs. This might have a performance impact.

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*forward_input*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

allreduce_tied_weight_gradients()

All reduce the gradients of the tied weights between tied stages

topology()

ProcessTopology object to query process mappings.

ckpt_prefix(*checkpoints_path*, *tag*)

Build a prefix for all checkpoint files written by this module.

ckpt_layer_path(*ckpt_dir*, *local_layer_idx*)

Customize a prefix for a specific pipeline module layer.

ckpt_layer_path_list(*ckpt_dir*, *local_layer_idx*)

Get all ckpt file list for a specific pipeline module layer.

get_additional_losses()

Returns model specific additional losses for reporting

Return a dictionary of {"loss name": loss_value} or None if no additional losses.

compile(**args*, ***kwargs*)

Compile this Module's forward using `torch.compile()`.

This Module's *__call__* method is compiled and all arguments are passed as-is to `torch.compile()`.

See `torch.compile()` for details on the arguments for this function.

class deepspeed.pipe.**LayerSpec**(*typename*, **module_args*, ***module_kwargs*)

Building block for specifying pipeline-parallel modules.

LayerSpec stores the type information and parameters for each stage in a PipelineModule. For example:

```
nn.Sequence(
    torch.nn.Linear(self.in_dim, self.hidden_dim, bias=False),
    torch.nn.Linear(self.hidden_hidden, self.out_dim)
)
```

becomes

```
layer_specs = [
    LayerSpec(torch.nn.Linear, self.in_dim, self.hidden_dim, bias=False),
    LayerSpec(torch.nn.Linear, self.hidden_hidden, self.out_dim)]
]
```

build(*log=False*)

Build the stored specification.

class deepspeed.pipe.**TiedLayerSpec**(*key*, *typename*, **module_args*, *forward_fn=None*, *tied_weight_attr=['weight']*, ***module_kwargs*)

class deepspeed.runtime.pipe.**ProcessTopology**(*axes*, *dims*)

Manages the mapping of n-dimensional Cartesian coordinates to linear indices. This mapping is used to map the rank of processes to the grid for various forms of parallelism.

Each axis of the tensor is accessed by its name. The provided ordering of the axes defines the layout of the topology. ProcessTopology uses a "row-major" layout of the tensor axes, and so axes=['x', 'y'] would map coordinates (x,y) and (x,y+1) to adjacent linear indices. If instead axes=['y', 'x'] was used, coordinates (x,y) and (x+1,y) would be adjacent.

Some methods return ProcessCoord namedtuples.

Create a mapping of n-dimensional tensor coordinates to linear indices.

Parameters

- **axes** (*list*) – the names of the tensor axes
- **dims** (*list*) – the dimension (length) of each axis of the topology tensor

get_rank(***coord_kwargs*)

Return the global rank of a process via its coordinates.

Coordinates are specified as kwargs. For example:

```
>>> X = ProcessTopology(axes=['x', 'y'], dims=[2,3])
>>> X.get_rank(x=0, y=1)
1
```

get_axis_names()

Return a list of the axis names in the ordering of the topology.

get_rank_repr(*rank*, *omit_axes=['data', 'pipe']*, *inner_sep='_'*, *outer_sep='-'*)

Return a string representation of a rank.

This method is primarily used for checkpointing model data.

For example:

```
>>> topo = Topo(axes=['a', 'b'], dims=[2, 2])
>>> topo.get_rank_repr(rank=3)
'a_01-b_01'
>>> topo.get_rank_repr(rank=3, omit_axes=['a'])
'b_01'
```

Parameters

- **rank** (*int*) – A rank in the topology.
- **omit_axes** (*list, optional*) – Axes that should not be in the representation. Defaults to [‘data’, ‘pipe’].
- **inner_sep** (*str, optional*) – [description]. Defaults to ‘_’.
- **outer_sep** (*str, optional*) – [description]. Defaults to ‘-‘.

Returns
A string representation of the coordinate owned by `rank`.

Return type
str

get_dim(*axis*)

Return the number of processes along the given axis.

For example:

```
>>> X = ProcessTopology(axes=['x', 'y'], dims=[2,3])
>>> X.get_dim('y')
3
```

get_coord(*rank*)

Return the coordinate owned by a process rank.

The axes of the returned namedtuple can be directly accessed as members. For .. rubric:: Example

```
>>> X = ProcessTopology(axes=['x', 'y'], dims=[2,3])
>>> coord = X.get_coord(rank=1)
>>> coord.x
0
>>> coord.y
1
```

get_axis_comm_lists(*axis*)

Construct lists suitable for a communicator group along axis `axis`.

Example

```
>>> topo = Topo(axes=['pipe', 'data', 'model'], dims=[2, 2, 2])
>>> topo.get_axis_comm_lists('pipe')
[
    [0, 4], # data=0, model=0
    [1, 5], # data=0, model=1
    [2, 6], # data=1, model=0
    [3, 7], # data=1, model=1
]
```

Returns
A list of lists whose coordinates match in all axes *except* `axis`.

filter_match(***filter_kwargs*)

Return the list of ranks whose coordinates match the provided criteria.

Example

```
>>> X = ProcessTopology(axes=['pipe', 'data', 'model'], dims=[2, 2, 2])
>>> X.filter_match(pipe=0, data=1)
[2, 3]
>>> [X.get_coord(rank) for rank in X.filter_match(pipe=0, data=1)]
[ProcessCoord(pipe=0, data=1, model=0), ProcessCoord(pipe=0, data=1, model=1)]
```

Parameters
****filter_kwargs** (*dict*) – criteria used to select coordinates.

Returns
The list of ranks whose coordinates match filter_kwargs.

get_axis_list(*axis*, *idx*)

Returns the list of global ranks whose coordinate in an axis is idx.

For example:

```
>>> X = ProcessTopology(axes=['x', 'y'], dims=[2,3])
>>> X.get_axis_list(axis='x', idx=0)
[0, 1, 2]
>>> X.get_axis_list(axis='y', idx=0)
[0, 3]
```

8.1.2 Training

8.1.3 Extending Pipeline Parallelism

class deepspeed.runtime.pipe.schedule.**PipeSchedule**(*micro_batches*, *stages*, *stage_id*)

Directs the execution of a pipeline engine by generating sequences of *PipeInstruction*.

Schedules are generators that yield sequences of *PipeInstruction* to process the micro-batches in one batch. Each yielded step is atomic in the sense that a barrier synchronization can be placed between successive steps without deadlock.

Below is an example schedule that implements data parallelism with gradient accumulation:

```
class DataParallelSchedule(PipeSchedule):
    def steps(self):
        for step_id in range(self.micro_batches):
            cmds = [
                LoadMicroBatch(buffer_id=0),
                ForwardPass(buffer_id=0),
                BackwardPass(buffer_id=0),
            ]
            if step_id == self.micro_batches - 1:
                cmds.extend([
                    ReduceGrads(),
                    OptimizerStep(),
                ])
            yield cmds

    def num_pipe_buffers(self):
        return 1
```

Parameters

- **micro_batches** (*int*) – The number of micro-batches that comprise a batch.
- **stages** (*int*) – The number of pipeline stages.
- **stage_id** (*int*) – The pipe stage that will execute the generated schedule.

abstract steps()

Yield a list of *PipeInstruction* for each step in the schedule.

Note: Schedules must implement `steps()` to define the schedule.

Returns

Instructions to be executed as one step of the pipeline

num_pipe_buffers()

The number of pipeline buffers that will be used by this stage.

Note: Schedules should specialize `num_pipe_buffers()` for memory savings at scale.

Returns
The number of buffers for the engine to allocate.

property stage

Stage index used to configure this schedule.

property num_stages

The number of total pipeline stages used to configure this schedule.

property num_micro_batches

The number of total micro_batches used to configure this schedule.

property is_first_stage

True if the configured `stage_id` is the first stage in the pipeline.

property is_last_stage

True if the configured `stage_id` is the last stage in the pipeline.

class deepspeed.runtime.pipe.schedule.**InferenceSchedule**(*micro_batches*, *stages*, *stage_id*)

A schedule for inferencing batches using pipeline parallelism.

num_pipe_buffers()

Only two pipeline buffers are required for inferencing.

Returns
2

class deepspeed.runtime.pipe.schedule.**TrainSchedule**(*micro_batches*, *stages*, *stage_id*)

A schedule for training a batch using hybrid parallelism.

Pipeline parallelism is extracted through gradient accumulation and thus convergence follows that of a data parallel approach with the same batch size.

num_pipe_buffers()

Return the number of pipeline buffers required for this stage.

This is equivalent to the maximum number of in-flight forward passes, since we need to remember the activations of forward passes in order to run backpropagation. For synchronous 1F1B, this is equivalent to the index difference between this stage and the last stage.

class deepspeed.runtime.pipe.schedule.**DataParallelSchedule**(*micro_batches*, *stages*, *stage_id*)

An example schedule that trains using traditional data parallelism with gradient accumulation.

num_pipe_buffers()

Only one pipeline buffer needed.

class deepspeed.runtime.pipe.schedule.**PipeInstruction**(***kwargs*)

Base class for all instructions to be executed by the pipeline engine.

All keyword arguments are stored as members similar to a `namedtuple`. These are then accessible to the `PipeEngine` during execution.

Parameters
kwargs (*optional*) – keyword arguments to store as members

class deepspeed.runtime.pipe.schedule.**OptimizerStep**(***kwargs*)

Performs one step with the optimizer and zeros gradients.

Note: Should be issued after *ReduceGrads* and *ReduceTiedGrads*.

Note: Can be a synchronization point among data-parallel ranks.

class deepspeed.runtime.pipe.schedule.**ReduceGrads**(***kwargs*)

Reduce the computed gradients among data-parallel processes within the stage.

class deepspeed.runtime.pipe.schedule.**ReduceTiedGrads**(***kwargs*)

Reduce the computed gradients of tied modules within a pipeline-parallel group.

Warning: The stages included in this synchronization point are not known until the model is partitioned among pipeline stages. In the worst case, it includes all pipeline stages. This instruction should be scheduled carefully to avoid deadlocks.

class deepspeed.runtime.pipe.schedule.**BufferOpInstruction**(*buffer_id*, ***kwargs*)

A pipeline instruction that operates on pipeline buffer(s).

Parameters

buffer_id (*int*) – the index of the pipeline buffer() to modify.

class deepspeed.runtime.pipe.schedule.**LoadMicroBatch**(*buffer_id*, ***kwargs*)

Load a micro-batch into a buffer.

Roughly:

```
buffers['inputs'][buffer_id] = next(data_iter)
```

class deepspeed.runtime.pipe.schedule.**ForwardPass**(*buffer_id*, ***kwargs*)

Compute a forward pass.

Roughly:

```
buffers['outputs'][buffer_id] = forward(buffers['inputs'][buffer_id])
```

class deepspeed.runtime.pipe.schedule.**BackwardPass**(*buffer_id*, ***kwargs*)

Compute a backward pass and accumulate gradients.

Roughly:

```
outputs = buffers['outputs'][buffer_id]
gradients = buffers['gradients'][buffer_id]
torch.autograd.backward(tensors=outputs,
                        grad_tensors=gradients)
```

class deepspeed.runtime.pipe.schedule.**SendActivation**(*buffer_id*, ***kwargs*)

Send activations to the next stage in the pipeline.

Roughly:

```
send(buffers['outputs'][buffer_id])
```

Note: The communication is blocking and must be paired with a *RecvActivation* on the next pipeline stage to avoid deadlock.

class deepspeed.runtime.pipe.schedule.**RecvActivation**(*buffer_id*, ***kwargs*)

Receive activations from the previous stage in the pipeline.

Roughly:

```
buffers['inputs'][buffer_id] = recv()
```

Note: The communication is blocking and must be paired with a *SendActivation* on the previous pipeline stage to avoid deadlock.

class deepspeed.runtime.pipe.schedule.**SendGrad**(*buffer_id*, ***kwargs*)

Send computed gradients to the previous pipeline stage. with respect to the received activations

Note: Only received tensors with `requires_grad==True` will produce gradients. Missing gradients will be replaced with `None` on the receiving stage.

Note: The communication is blocking and must be paired with a *RecvGrad* on the previous pipeline stage to avoid deadlock.

class deepspeed.runtime.pipe.schedule.**RecvGrad**(*buffer_id*, ***kwargs*)

Receive computed gradients the next pipeline stage.

Note: Only activations with `requires_grad==True` will produce gradients. Missing gradients will be replaced with `None`.

Note: The communication is blocking and must be paired with a *SendGrad* on the next pipeline stage to avoid deadlock.

CHAPTER

NINE

OPTIMIZERS

9.1 Optimizers

DeepSpeed offers high-performance implementations of `Adam` optimizer on CPU; `FusedAdam`, `FusedLamb`, `OnebitAdam`, `OnebitLamb` optimizers on GPU.

9.1.1 Adam (CPU)

class deepspeed.ops.adam.**DeepSpeedCPUAdam**(*model_params*, *lr=0.001*, *bias_correction=True*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*, *amsgrad=False*, *adamw_mode=True*, *fp32_optimizer_states=True*)

Fast vectorized implementation of two variations of Adam optimizer on CPU:

- Adam: A Method for Stochastic Optimization: (https://arxiv.org/abs/1412.6980);
- AdamW: Fixing Weight Decay Regularization in Adam (https://arxiv.org/abs/1711.05101)

DeepSpeed CPU Adam(W) provides between 5x to 7x speedup over torch.optim.adam(W). In order to apply this optimizer, the model requires to have its master parameter (in FP32) reside on the CPU memory.

To train on a heterogeneous system, such as coordinating CPU and GPU, DeepSpeed offers the ZeRO-Offload technology which efficiently offloads the optimizer states into CPU memory, with minimal impact on training throughput. DeepSpeedCPUAdam plays an important role to minimize the overhead of the optimizer's latency on CPU. Please refer to ZeRO-Offload tutorial (https://www.deepspeed.ai/tutorials/zero-offload/) for more information on how to enable this technology.

Note: We recommend using our config to allow `deepspeed.initialize()` to build this optimizer for you.

Parameters

- **model_params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups.
- **lr** (*float, optional*) – learning rate. (default: 1e-3)
- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square. (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability. (default: 1e-8)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)

- **amsgrad** (*boolean, optional*) – whether to use the AMSGrad variant of this algorithm from the paper On the Convergence of Adam and Beyond (default: False) NOT SUPPORTED in DeepSpeed CPUAdam!
- **adamw_mode** – select between Adam and AdamW implementations (default: AdamW)
- **fp32_optimizer_states** – creates momentum and variance in full precision regardless of the precision of the parameters. Set to False to keep optimizer states in the parameter dtype (e.g. bf16), which reduces the optimizer-state memory footprint at the cost of lower state precision. (default: True)

9.1.2 FusedAdam (GPU)

class deepspeed.ops.adam.**FusedAdam**(*params*, *lr=0.001*, *bias_correction=True*, *betas=(0.9, 0.999)*, *eps=1e-08*, *adam_w_mode=True*, *weight_decay=0.0*, *amsgrad=False*, *set_grad_none=True*)

Implements Adam algorithm.

Currently GPU-only. Requires Apex to be installed via `pip install -v --no-cache-dir --global-option="--cpp_ext" --global-option="--cuda_ext" ./`.

This version of fused Adam implements 2 fusions.

- Fusion of the Adam update's elementwise operations
- A multi-tensor apply launch that batches the elementwise updates applied to all the model's parameters into one or a few kernel launches.

`apex.optimizers.FusedAdam` may be used as a drop-in replacement for `torch.optim.AdamW`, or `torch.optim.Adam` with `adam_w_mode=False`:

```
opt = apex.optimizers.FusedAdam(model.parameters(), lr = ....)
...
opt.step()
```

`apex.optimizers.FusedAdam` may be used with or without Amp. If you wish to use *FusedAdam* with Amp, you may choose any `opt_level`:

```
opt = apex.optimizers.FusedAdam(model.parameters(), lr = ....)
model, opt = amp.initialize(model, opt, opt_level="O0" or "O1 or "O2")
...
opt.step()
```

In general, `opt_level="O1"` is recommended.

Warning: A previous version of *FusedAdam* allowed a number of additional arguments to `step`. These additional arguments are now deprecated and unnecessary.

Adam was been proposed in **`Adam: A Method for Stochastic Optimization`_**.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups.
- **lr** (*float, optional*) – learning rate. (default: 1e-3)

- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square. (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability. (default: 1e-8)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **amsgrad** (*boolean, optional*) – whether to use the AMSGrad variant of this algorithm from the paper On the Convergence of Adam and Beyond (default: False) NOT SUPPORTED in FusedAdam!
- **adam_w_mode** (*boolean, optional*) – Apply L2 regularization or weight decay True for decoupled weight decay(also known as AdamW) (default: True)
- **set_grad_none** (*bool, optional*) – whether set grad to None when zero_grad() method is called. (default: True)

9.1.3 FusedLamb (GPU)

class deepspeed.ops.lamb.**FusedLamb**(*params*, *lr=0.001*, *bias_correction=True*, *betas=(0.9, 0.999)*, *eps=1e-08*, *eps_inside_sqrt=False*, *weight_decay=0.0*, *max_grad_norm=0.0*, *max_coeff=10.0*, *min_coeff=0.01*, *amsgrad=False*)

Implements the LAMB algorithm. Currently GPU-only.

LAMB was proposed in `Large Batch Optimization for Deep Learning: Training BERT in 76 minutes. https://arxiv.org/abs/1904.00962

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups.
- **lr** (*float, optional*) – learning rate. (default: 1e-3)
- **bias_correction** (*bool, optional*) – bias correction (default: True)
- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square. (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability. (default: 1e-8)
- **eps_inside_sqrt** (*boolean, optional*) – in the 'update parameters' step, adds eps to the bias-corrected second moment estimate before evaluating square root instead of adding it to the square root of second moment estimate as in the original paper. (default: False)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **max_grad_norm** (*float, optional*) – value used to clip global grad norm (default: 0.0)
- **max_coeff** (*float, optional*) – maximum value of the lamb coefficient (default: 10.0)
- **min_coeff** (*float, optional*) – minimum value of the lamb coefficient (default: 0.01)
- **amsgrad** (*boolean, optional*) – NOT SUPPORTED in FusedLamb!

9.1.4 OneBitAdam (GPU)

9.1.5 ZeroOneAdam (GPU)

9.1.6 OnebitLamb (GPU)

CHAPTER

TEN

LEARNING RATE SCHEDULERS

10.1 Learning Rate Schedulers

DeepSpeed offers implementations of `LRRangeTest`, `OneCycle`, `WarmupLR`, `WarmupDecayLR`, `WarmupCosineLR` learning rate schedulers. When using a DeepSpeed's learning rate scheduler (specified in the *ds_config.json* file), DeepSpeed calls the *step()* method of the scheduler at every training step (when *model_engine.step()* is executed). When not using a DeepSpeed's learning rate scheduler:

- if the schedule is supposed to execute at every training step, then the user can pass the scheduler to *deepspeed.initialize* when initializing the DeepSpeed engine and let DeepSpeed manage it for update or save/restore.
- if the schedule is supposed to execute at any other interval (e.g., training epochs), then the user should NOT pass the scheduler to DeepSpeed during initialization and must manage it explicitly.

10.1.1 LRRangeTest

`class` deepspeed.runtime.lr_schedules.**LRRangeTest**(*optimizer: Optimizer*, *lr_range_test_min_lr: float = 0.001*, *lr_range_test_step_size: int = 2000*, *lr_range_test_step_rate: float = 1.0*, *lr_range_test_staircase: bool = False*, *last_batch_iteration: int = -1*)

Sets the learning rate of each parameter group according to learning rate range test (LRRT) policy. The policy increases learning rate starting from a base value with a constant frequency, as detailed in the paper A disciplined approach to neural network hyper-parameters: Part 1

LRRT policy is used for finding maximum LR that trains a model without divergence, and can be used to configure the LR boundaries for Cyclic LR schedules.

LRRT changes the learning rate after every batch. *step* should be called after a batch has been used for training.

Parameters

- **optimizer** (*Optimizer*) – Wrapped optimizer.
- **lr_range_test_min_lr** (*float or list*) – Initial learning rate which is the lower boundary in the range test for each parameter group.
- **lr_range_test_step_size** (*int*) – Interval of training steps to increase learning rate. Default: 2000
- **lr_range_test_step_rate** (*float*) – Scaling rate for range test. Default: 1.0
- **lr_range_test_staircase** (*bool*) – Scale in staircase fashion, rather than continuous. Default: False.

- **last_batch_iteration** (*int*) – The index of the last batch. This parameter is used when resuming a training job. Since *step()* should be invoked after each batch instead of after each epoch, this number represents the total number of *batches* computed, not the total number of epochs computed. When last_batch_iteration=-1, the schedule is started from the beginning. Default: -1

Example

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> scheduler = LRRangeTest(optimizer)
>>> data_loader = torch.utils.data.DataLoader(...)
>>> for epoch in range(10):
>>>     for batch in data_loader:
>>>         train_batch(...)
>>>         scheduler.step()
```

_A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay: https://arxiv.org/abs/1803.09820

10.1.2 OneCycle

class deepspeed.runtime.lr_schedules.**OneCycle**(*optimizer*, *cycle_min_lr*, *cycle_max_lr*, *decay_lr_rate=0.0*, *cycle_first_step_size=2000*, *cycle_second_step_size=None*, *cycle_first_stair_count=0*, *cycle_second_stair_count=None*, *decay_step_size=0*, *cycle_momentum=True*, *cycle_min_mom=0.8*, *cycle_max_mom=0.9*, *decay_mom_rate=0.0*, *last_batch_iteration=-1*)

Sets the learning rate of each parameter group according to 1Cycle learning rate policy (1CLR). 1CLR is a variation of the Cyclical Learning Rate (CLR) policy that involves one cycle followed by decay. The policy simultaneously cycles the learning rate (and momentum) between two boundaries with a constant frequency, as detailed in the paper A disciplined approach to neural network hyper-parameters.

1CLR policy changes the learning rate after every batch. *step* should be called after a batch has been used for training.

This implementation was adapted from the github repo: PyTorch.

Parameters

- **optimizer** (*Optimizer*) – Wrapped optimizer.
- **cycle_min_lr** (*float or list*) – Initial learning rate which is the lower boundary in the cycle for each parameter group.
- **cycle_max_lr** (*float or list*) – Upper learning rate boundaries in the cycle for each parameter group. Functionally, it defines the cycle amplitude (cycle_max_lr - cycle_min_lr). The lr at any cycle is the sum of cycle_min_lr and some scaling of the amplitude; therefore cycle_max_lr may not actually be reached depending on scaling function.
- **decay_lr_rate** (*float*) – Decay rate for learning rate. Default: 0.
- **cycle_first_step_size** (*int*) – Number of training iterations in the increasing half of a cycle. Default: 2000

- **cycle_second_step_size** (*int*) – Number of training iterations in the decreasing half of a cycle. If cycle_second_step_size is None, it is set to cycle_first_step_size. Default: None
- **cycle_first_stair_count** (*int*) – Number of stairs in first half of cycle phase. This means
- **0** (*lr/mom are changed in staircase fashion. Default*) –
- **disabled.** (*means staircase*) –
- **cycle_second_stair_count** (*int*) – Number of stairs in second half of cycle phase. This means
- **0** –
- **disabled.** –
- **decay_step_size** (*int*) – Intervals for applying decay in decay phase. Default: 0, means no decay.
- **cycle_momentum** (*bool*) – If `True`, momentum is cycled inversely to learning rate between 'cycle_min_mom' and 'cycle_max_mom'. Default: True
- **cycle_min_mom** (*float or list*) – Initial momentum which is the lower boundary in the cycle for each parameter group. Default: 0.8
- **cycle_max_mom** (*float or list*) – Upper momentum boundaries in the cycle for each parameter group. Functionally, it defines the cycle amplitude (cycle_max_mom - cycle_min_mom). The momentum at any cycle is the difference of cycle_max_mom and some scaling of the amplitude; therefore cycle_min_mom may not actually be reached depending on scaling function. Default: 0.9
- **decay_mom_rate** (*float*) – Decay rate for momentum. Default: 0.
- **last_batch_iteration** (*int*) – The index of the last batch. This parameter is used when resuming a training job. Since *step()* should be invoked after each batch instead of after each epoch, this number represents the total number of *batches* computed, not the total number of epochs computed. When last_batch_iteration=-1, the schedule is started from the beginning. Default: -1

Example

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> scheduler = OneCycle(optimizer, 0.0001, 0.0010)
>>> data_loader = torch.utils.data.DataLoader(...)
>>> for epoch in range(10):
>>>     for batch in data_loader:
>>>         train_batch(...)
>>>         scheduler.step()
```

10.1.3 WarmupLR

class deepspeed.runtime.lr_schedules.**WarmupLR**(*optimizer: Optimizer*, *warmup_min_lr: float = 0.0*, *warmup_max_lr: Optional[float] = None*, *warmup_num_steps: int = 1000*, *warmup_type: str = 'log'*, *last_batch_iteration: int = -1*)

Increase the learning rate of each parameter group from min lr to max lr over warmup_num_steps steps, and then fix at max lr.

Parameters

- **optimizer** (*Optimizer*) – Wrapped optimizer.
- **warmup_min_lr** (*float or list*) – minimum learning rate. Default: 0
- **warmup_max_lr** (*float or list*) – maximum learning rate. Default: 0.001
- **warmup_num_steps** (*int*) – number of steps to warm up from min_lr to max_lr. Default: 1000
- **{'log'** (*warmup_type*) – increasing function from min_lr to max_lr during warmup. Default: log
- **'linear'}** – increasing function from min_lr to max_lr during warmup. Default: log
- **last_batch_iteration** (*int*) – The index of the last batch. Default: -1.

Example

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> scheduler = WarmupLR(optimizer)
>>> data_loader = torch.utils.data.DataLoader(...)
>>> for epoch in range(10):
>>>     for batch in data_loader:
>>>         train_batch(...)
>>>         scheduler.step()
```

10.1.4 WarmupDecayLR

class deepspeed.runtime.lr_schedules.**WarmupDecayLR**(*optimizer: Optimizer*, *total_num_steps: int*, *warmup_min_lr: float = 0.0*, *warmup_max_lr: float = 0.001*, *warmup_num_steps: int = 1000*, *warmup_type: str = 'log'*, *last_batch_iteration: int = -1*)

Increase the learning rate of each parameter group from min lr to max lr over warmup_num_steps steps, and then decay at linear rate over the remaining training steps.

Parameters

- **optimizer** (*Optimizer*) – Wrapped optimizer.
- **total_num_steps** (*int*) – total number of training steps
- **warmup_min_lr** (*float or list*) – minimum learning rate. Default: 0
- **warmup_max_lr** (*float or list*) – maximum learning rate. Default: 0.001

- **warmup_num_steps** (*int*) – number of steps to warm up from min_lr to max_lr. Default: 1000
- **{'log'** (*warmup_type*) – increasing function from min_lr to max_lr during warmup. Default: log
- **'linear'}** – increasing function from min_lr to max_lr during warmup. Default: log
- **last_batch_iteration** (*int*) – The index of the last batch. Default: -1.

Example

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> scheduler = WarmupDecayLR(optimizer, 1000000)
>>> data_loader = torch.utils.data.DataLoader(...)
>>> for epoch in range(10):
>>>     for batch in data_loader:
>>>         train_batch(...)
>>>         scheduler.step()
```

10.1.5 WarmupCosineLR

class deepspeed.runtime.lr_schedules.**WarmupCosineLR**(*optimizer: Optimizer*, *total_num_steps: int*, *warmup_min_ratio: float = 0.0*, *warmup_num_steps: int = 1000*, *cos_min_ratio: float = 0.0001*, *warmup_type: str = 'log'*, *last_batch_iteration: int = -1*)

Increase the learning rate of each parameter group from min lr ratio to max lr ratio over warmup_num_steps steps, and then decay at cosine rate over the remaining training steps to min cosine ratio.

Parameters

- **optimizer** (*Optimizer*) – Wrapped optimizer.
- **total_num_steps** (*int*) – total number of training steps
- **warmup_min_ratio** (*float or list*) – warmup start learning rate ratio. Default: 0
- **warmup_num_steps** (*int*) – number of steps to warm up from warmup_min_ratio to 1.0. Default: 1000
- **{'log'** (*warmup_type*) – increasing function from min_lr to max_lr during warmup. Default: log
- **'linear'}** – increasing function from min_lr to max_lr during warmup. Default: log
- **cos_min_ratio** (*float*) – cosine end learning rate ratio. Default: 0.0001
- **last_batch_iteration** (*int*) – The index of the last batch. Default: -1.

Example

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> scheduler = WarmupCosineLR(optimizer, 1000000)
>>> data_loader = torch.utils.data.DataLoader(...)
>>> for epoch in range(10):
>>>     for batch in data_loader:
>>>         train_batch(...)
>>>         scheduler.step()
```

CHAPTER

ELEVEN

FLOPS PROFILER

Flops Profiler

The flops profiler in DeepSpeed profiles the forward pass of a model and measures its parameters, latency, and floating point operations. The DeepSpeed flops profiler can be used with the DeepSpeed runtime or as a standalone package.

When using DeepSpeed for model training, the flops profiler can be configured in the deepspeed_config file without user code changes. To use the flops profiler outside of the DeepSpeed runtime, one can simply install DeepSpeed and import the flops_profiler package to use the APIs directly.

Please see the Flops Profiler tutorial for usage details.

11.1 Flops Profiler

class deepspeed.profiling.flops_profiler.profiler.**FlopsProfiler**(*model*, *ds_engine=None*, *recompute_fwd_factor=0.0*)

Bases: `object`

Measures the latency, number of estimated floating-point operations and parameters of each module in a PyTorch model.

The flops-profiler profiles the forward pass of a PyTorch model and prints the model graph with the measured profile attached to each module. It shows how latency, flops and parameters are spent in the model and which modules or layers could be the bottleneck. It also outputs the names of the top k modules in terms of aggregated latency, flops, and parameters at depth l with k and l specified by the user. The output profile is computed for each batch of input. The DeepSpeed flops profiler can be used with the DeepSpeed runtime or as a standalone package. When using DeepSpeed for model training, the flops profiler can be configured in the deepspeed_config file and no user code change is required.

If using the profiler as a standalone package, one imports the flops_profiler package and use the APIs.

Here is an example for usage in a typical training workflow:

```
model = Model()
prof = FlopsProfiler(model)

for step, batch in enumerate(data_loader):
    if step == profile_step:
        prof.start_profile()

    loss = model(batch)
```

(continues on next page)

(continued from previous page)

```
    if step == profile_step:
        flops = prof.get_total_flops(as_string=True)
        params = prof.get_total_params(as_string=True)
        prof.print_model_profile(profile_step=profile_step)
        prof.end_profile()

    loss.backward()
    optimizer.step()
```

To profile a trained model in inference, use the *get_model_profile* API.

Parameters
object (*torch.nn.Module*) – The PyTorch model to profile.

start_profile(*ignore_list=None*)

Starts profiling.

Extra attributes are added recursively to all the modules and the profiled torch.nn.functionals are monkey patched.

Parameters
ignore_list (*list, optional*) – the list of modules to ignore while profiling. Defaults to None.

stop_profile()

Stop profiling.

All torch.nn.functionals are restored to their originals.

reset_profile()

Resets the profiling.

Adds or resets the extra attributes.

end_profile()

Ends profiling.

The added attributes and handles are removed recursively on all the modules.

get_total_flops(*as_string=False*)

Returns the total flops of the model.

Parameters
as_string (*bool, optional*) – whether to output the flops as string. Defaults to False.

Returns
The number of multiply-accumulate operations of the model forward pass.

get_total_macs(*as_string=False*)

Returns the total MACs of the model.

Parameters
as_string (*bool, optional*) – whether to output the flops as string. Defaults to False.

Returns
The number of multiply-accumulate operations of the model forward pass.

get_total_duration(*as_string=False*)

Returns the total duration of the model forward pass.

Parameters

as_string (*bool, optional*) – whether to output the duration as string. Defaults to False.

Returns

The latency of the model forward pass.

get_total_params(*as_string=False*)

Returns the total number of parameters stored per rank.

Parameters

as_string (*bool, optional*) – whether to output the parameters as string. Defaults to False.

Returns

The total number of parameters stored per rank.

print_model_profile(*profile_step=1*, *module_depth=-1*, *top_modules=1*, *detailed=True*, *output_file=None*)

Prints the model graph with the measured profile attached to each module.

Parameters

- **profile_step** (*int, optional*) – The global training step at which to profile. Note that warm up steps are needed for accurate time measurement.
- **module_depth** (*int, optional*) – The depth of the model to which to print the aggregated module information. When set to -1, it prints information from the top to the innermost modules (the maximum depth).
- **top_modules** (*int, optional*) – Limits the aggregated profile output to the number of top modules specified.
- **detailed** (*bool, optional*) – Whether to print the detailed model profile.
- **output_file** (*str, optional*) – Path to the output file. If None, the profiler prints to stdout.

print_model_aggregated_profile(*module_depth=-1*, *top_modules=1*)

Prints the names of the top top_modules modules in terms of aggregated time, flops, and parameters at depth module_depth.

Parameters

- **module_depth** (*int, optional*) – the depth of the modules to show. Defaults to -1 (the innermost modules).
- **top_modules** (*int, optional*) – the number of top modules to show. Defaults to 1.

deepspeed.profiling.flops_profiler.profiler.**get_model_profile**(*model*, *input_shape=None*, *args=[]*, *kwargs={}*, *print_profile=True*, *detailed=True*, *module_depth=-1*, *top_modules=1*, *warm_up=1*, *as_string=True*, *output_file=None*, *ignore_modules=None*, *mode='forward'*)

Returns the total floating-point operations, MACs, and parameters of a model.

Example:

```
model = torchvision.models.alexnet()
batch_size = 256
flops, macs, params = get_model_profile(model=model, input_shape=(batch_size, 3,␣
↪224, 224)))
```

Parameters

- **model** (*[torch.nn.Module]*) – the PyTorch model to be profiled.
- **input_shape** (*tuple*) – input shape to the model. If specified, the model takes a tensor with this shape as the only positional argument.
- **args** (*list*) – list of positional arguments to the model.
- **kwargs** (*dict*) – dictionary of keyword arguments to the model.
- **print_profile** (*bool, optional*) – whether to print the model profile. Defaults to True.
- **detailed** (*bool, optional*) – whether to print the detailed model profile. Defaults to True.
- **module_depth** (*int, optional*) – the depth into the nested modules. Defaults to -1 (the inner most modules).
- **top_modules** (*int, optional*) – the number of top modules to print in the aggregated profile. Defaults to 3.
- **warm_up** (*int, optional*) – the number of warm-up steps before measuring the latency of each module. Defaults to 1.
- **as_string** (*bool, optional*) – whether to print the output as string. Defaults to True.
- **output_file** (*str, optional*) – path to the output file. If None, the profiler prints to stdout.
- **ignore_modules** (*[type], optional*) – the list of modules to ignore during profiling. Defaults to None.

Returns

The number of floating-point operations, multiply-accumulate operations (MACs), and parameters in the model.

CHAPTER

TWELVE

AUTOTUNING

12.1 Autotuning

One pain point in model training is to figure out good performance-relevant configurations such as micro-batch size to fully utilize the hardware and achieve a high throughput number. This configuration exploring process is commonly done manually but is important since model training is repeated many times and benefits from using a good configuration. Not only is the hand-tuning process time-consuming, but the outcome is hardware-dependent. This means that a good configuration on one hardware might not be the best on another different hardware. The user thus has to hand tune the configuration again. With DeepSpeed, there are more configuration parameters that could potentially affect the training speed, thus making it more tedious to manually tune the configuration.

The DeepSpeed Autotuner mitigates this pain point and automatically discovers the optimal DeepSpeed configuration that delivers good training speed. The Autotuner uses model information, system information, and heuristics to efficiently tune system knobs that affect compute and memory efficiencies, such as ZeRO optimization stages, micro-batch sizes, and many other ZeRO optimization configurations. It not only reduces the time and resources users spend on tuning, but also can discover configurations better than hand-tuned methods.

Please see the Autotuning tutorial for usage details.

12.1.1 Autotuner

deepspeed.autotuning.**autotuner**

alias of <module 'deepspeed.autotuning.autotuner' from '/home/docs/checkouts/readthedocs.org/user_builds/deepspeed/checkouts

CHAPTER

THIRTEEN

MEMORY USAGE

13.1 Memory Requirements

13.1.1 API To Estimate Memory Usage

ZeRO2:

Examples:

Let's try a 3B model with just 1 node with 8 gpus, using live model:

```
python -c 'from transformers import AutoModel; \
from deepspeed.runtime.zero.stage_1_and_2 import estimate_zero2_model_states_mem_needs_
↪all_live; \
model = AutoModel.from_pretrained("t5-3b"); \
estimate_zero2_model_states_mem_needs_all_live(model, num_gpus_per_node=8, num_nodes=1)'

Estimated memory needed for params, optim states and gradients for a:
HW: Setup with 1 node, 8 GPUs per node.
SW: Model with 2851M total params.
  per CPU  |  per GPU |   Options
  127.48GB |   5.31GB | offload_optimizer=cpu
  127.48GB |  15.93GB | offload_optimizer=none
```

Now, without the actual model, which requires us to know `total_params` and `largest_layer_params`, but we got those from the run above, so future estimators are now much faster as we don't need to load the model.

```
python -c 'from deepspeed.runtime.zero.stage_1_and_2 import estimate_zero2_model_states_
↪mem_needs_all_cold; \
estimate_zero2_model_states_mem_needs_all_cold(total_params=2851e6, num_gpus_per_node=8,␣
↪num_nodes=1)'

Estimated memory needed for params, optim states and gradients for a:
HW: Setup with 1 node, 8 GPUs per node.
SW: Model with 2851M total params.
  per CPU  |  per GPU |   Options
  127.45GB |   5.31GB | offload_optimizer=cpu
  127.45GB |  15.93GB | offload_optimizer=none
```

There is a slight difference due to rounding - the actual live model has a few more params

ZeRO3:

Examples:

Let's try a 3B model with just 1 node with 8 gpus, using live model:

```
python -c 'from transformers import AutoModel; \
from deepspeed.runtime.zero.stage3 import estimate_zero3_model_states_mem_needs_all_live;
↪ \
model = AutoModel.from_pretrained("t5-3b"); \
estimate_zero3_model_states_mem_needs_all_live(model, num_gpus_per_node=8, num_nodes=1)'

Estimated memory needed for params, optim states and gradients for a:
HW: Setup with 1 node, 8 GPUs per node.
SW: Model with 2851M total params, 32M largest layer params.
  per CPU  |  per GPU |   Options
   71.71GB |   0.12GB | offload_param=cpu , offload_optimizer=cpu , zero_init=1
  127.48GB |   0.12GB | offload_param=cpu , offload_optimizer=cpu , zero_init=0
   63.74GB |   0.79GB | offload_param=none, offload_optimizer=cpu , zero_init=1
  127.48GB |   0.79GB | offload_param=none, offload_optimizer=cpu , zero_init=0
    1.47GB |   6.10GB | offload_param=none, offload_optimizer=none, zero_init=1
  127.48GB |   6.10GB | offload_param=none, offload_optimizer=none, zero_init=0
```

Now, without the actual model, which requires us to know `total_params` and `largest_layer_params`, but we got those from the run above, so future estimators are now much faster as we don't need to load the model.

```
python -c 'from deepspeed.runtime.zero.stage3 import estimate_zero3_model_states_mem_
↪needs_all_cold; \
estimate_zero3_model_states_mem_needs_all_cold(total_params=2851e6, largest_layer_
↪params=32e6, num_gpus_per_node=8, num_nodes=1)'

Estimated memory needed for params, optim states and gradients for a:
HW: Setup with 1 node, 8 GPUs per node.
SW: Model with 2851M total params, 32M largest layer params.
  per CPU  |  per GPU |   Options
   71.69GB |   0.12GB | offload_param=cpu , offload_optimizer=cpu , zero_init=1
  127.45GB |   0.12GB | offload_param=cpu , offload_optimizer=cpu , zero_init=0
   63.72GB |   0.78GB | offload_param=none, offload_optimizer=cpu , zero_init=1
  127.45GB |   0.78GB | offload_param=none, offload_optimizer=cpu , zero_init=0
    1.43GB |   6.09GB | offload_param=none, offload_optimizer=none, zero_init=1
  127.45GB |   6.09GB | offload_param=none, offload_optimizer=none, zero_init=0
```

There is a slight difference due to rounding - the actual live model has a few more params

13.1.2 Discussion

Let's look in detail how the memory estimator API calculates these numbers and also discuss some additional numbers that aren't covered by the API.

In the following discussion:

- `params` - total number of model params, which can be calculated as:

```
print(sum(dict((p.data_ptr(), p.numel()) for p in model.parameters()).values()))
```

Some models already include the number of params in the model name, e.g. t5-11b (11B params), gpt-neo-1.3B (1.3B params), etc.

Also if the model weights are stored in `fp32` the other quick way to calculate the size of the model is to simply divide the size of the `state_dict` file by 4 (fp32 == 4 bytes). For example, you can see that t5-11b's pytorch_model.bin is 42.1GB in size, so if we divide it by 4, we can immediately tell it's an 11B model.

The following calculations show how much memory is required by model params, gradients and optimizer states. In addition to those you will need enough memory to fit activation calculations and any temporary memory for intermediate calculations, which for long sequences could be very significant (e.g. could take the same amount of memory as params+grads+optim_states combined).

The optimizer states assume that `Adam` is used, where 4 bytes per parameter are used by momentum and another 4 by variance (8 in total).

Gradients at `fp32` take 4 bytes, and parameters take 2 bytes at `fp16` and 4 bytes at `fp32`.

GPU RAM

The big question is how big of a model you can fit on the hardware you have? Or rather what size of a GPU RAM do you need to fit the desired model.

- ZeRO-2:
  - `"offload_optimizer": {"device": "cpu"}`: 2 * params

    Example: a 40GB GPU can fit ~11B param model (regardless of how many GPUs are used). Here the model is loaded in `fp16` so just the model weights take about 22GB and the remaining 18GB are used by other components. You can barely fit a very small batch size in this scenario.
  - `"offload_optimizer": {"device": "none"}`: 4 * params + 16 * params/ (total number of gpus)
- ZeRO-3:

`largest_layer_memory = 4*largest_layer_params` - GPU memory needed to gather the largest layer on a single GPU. 2 bytes fp16 params are gathered and 2 bytes fp16 grads are computed (total 4x). The optimizer states and fp32 parameters are updated in partitioned form and copied to fp16 params in partitioned form. This happens during the optimizer step. After that the fp16 params are sufficient.

- case 1: `"offload_param": {"device": "none"}, "offload_optimizer": {"device": "none"}` - largest_layer_memory + 18 * params / total number of gpus across all nodes
- case 2: `"offload_param": {"device": "cpu"}, "offload_optimizer": {"device": "cpu"}`- largest_layer_memory. The main limit here is general RAM.
- case 3: `"offload_param": {"device": "none"}, "offload_optimizer": {"device": "cpu"}`- largest_layer_memory + 2 * params / total number of gpus across all nodes

  Example:

```
from transformers import AutoModel
model = AutoModel.from_pretrained("t5-large")

# shared params calculated only ones
total_params = sum(dict((p.data_ptr(), p.numel()) for p in model.parameters()).values())

largest_layer_params = 0
for m in model.modules():
    # assuming no shared params within a single layer
    layer_params = sum(p.numel() for p in m.parameters(recurse=False))
    largest_layer_params = max(largest_layer_params, layer_params)

largest_layer_memory = (4*largest_layer_params)
```

(continues on next page)

(continued from previous page)

```
total_gpus = 4

case1 = largest_layer_memory + int(18*total_params/total_gpus)
case2 = largest_layer_memory
case3 = largest_layer_memory + int(2*total_params/total_gpus)

print(f"total params:          {total_params/1e6:6.2f}M")
print(f"largest layer params: {largest_layer_params/1e6:6.2f}M")
print(f"largest layer memory: {largest_layer_memory>>20:6}MB")
print(f"case1 gpu memory: {(case1)>>20:6}MB")
print(f"case2 gpu memory: {(case2)>>20:6}MB")
print(f"case3 gpu memory: {(case3)>>20:6}MB")

total params:          737.67M
largest layer params:  32.90M
largest layer memory:    125MB
case1 gpu memory:   3291MB
case2 gpu memory:    125MB
case3 gpu memory:    477MB
```

General RAM:

One of the key features of ZeRO is its CPU offload which can dramatically extend the total memory pool accessible to the project by using general RAM. One can easily expand their general RAM by 10x times, at a significantly lower cost than what it'd take to have the same GPU RAM. And often, it's not even possible to buy GPUs with a lot of RAM (112GB GPU anybody?) since they simply don't yet exist.

In the following calculations we will use:

- `additional_buffer_factor=1.5` as an additional buffer factor to be conservative
- `n_gpus` the number of GPUs on a single node (machine)
- `total_gpus` the total number of GPUs across all nodes
- `params` - total number of model params (see above for how to get this number)
- ZeRO-2:
  - `"offload_optimizer": {"device": "none"}`:

    params * 4 * n_gpus * additional_buffer_factor - this is the memory needed only at the beginning to initialize the model on CPU memory
  - `"offload_optimizer": {"device": "cpu"}`:

    params * max(4 * n_gpus, 16) * additional_buffer_factor

  Example: xxx
- ZeRO-3:

  gpus_factor = n_gpus / total_gpus
  - case 1: `"offload_param": {"device": "none"}, "offload_optimizer": {"device": "none"}`:

    Without `zero.Init`:

params * 4 * n_gpus * additional_buffer_factor

this is the memory needed only at the beginning to initialize the model on CPU memory. Once the model is transferred to GPUs this memory is freed.

With `zero.Init`:

largest_layer_params * 4 * n_gpus * additional_buffer_factor

assuming Pytorch is deallocating the memory once the tensors are moved to the GPU by ZeRO.Init

- case 2: `"offload_param": {"device": "cpu"}, "offload_optimizer": {"device": "cpu"}`:

  Without `zero.Init`:

  params * max(4 * n_gpus, 18 * gpus_factor) * additional_buffer_factor

  With `zero.Init`:

  params * 18 * gpus_factor * additional_buffer_factor

- case 3: `"offload_param": {"device": "none"}, "offload_optimizer": {"device": "cpu"}`:

  Without `zero.Init`:

  params * max(4 * n_gpus, 16 * gpus_factor) * additional_buffer_factor

  With `zero.Init`:

  params * 16 * gpus_factor * additional_buffer_factor

Here is a breakdown for the 16 and 18 multipliers (b = bytes):

4 (in `4*n_gpus`):

- when pytorch creates a model it creates it in fp32 by default (4 bytes)

16:

- 16b for fp32: 4b params, 4b grads, 4b momentum and 4b variance per parameter

18:

- 16b for fp32: 4b params, 4b grads, 4b momentum and 4b variance per parameter
- +2b for fp16 params

Note about gradients: While gradients are stored in fp16 (2 bytes), during the weight update, all of them are converted into fp32 before doing the weight updates since the weight updates are done at almost the entire model granularity (param_group granularity) in FusedAdam Optimizer in DeepSpeed. So after that conversion we would need the 4 bytes per gradient for nearly the entire set of weights.

Pinned Memory

Pinned general RAM is included in normal general RAM allocations (i.e. this is not extra memory allocations but simply shows how much of the general RAM is pinned)

- ZeRO-2: can't be controlled
- ZeRO-3

To enable add: `"cpu_offload_use_pin_memory" : true`

Now there are 2 sub-cases:

1. `"cpu_offload_params": true`:

- 6 * params (2b for fp16 params + 4b for fp32 gradients)
- if `gradient_accumulation_steps > 1` an additional 2b for fp16 gradients are pinned

2. `"cpu_offload_params": false`:
    - 4b for fp32 gradients

Activation Memory

XXX: For Transformers is probably around (2* seq * attn_heads + 16 * hidden_size) * sequence * batch/gpu

This needs to be completed.

CHAPTER

FOURTEEN

MONITORING

14.1 Monitoring

Deepspeed's Monitor module can log training details into a Tensorboard-compatible file, to WandB, or to simple CSV files. Below is an overview of what DeepSpeed will log automatically.

Table 1: Automatically Logged Data

Field	Description	Condition
Train/Samples/train_loss	The training loss.	None
Train/Samples/lr	The learning rate during training.	None
Train/Samples/loss_scale	The loss scale when training using *fp16*.	*fp16* must be enabled.
Train/Eigenvalues/ModelBlockParam_{i}	Eigen values per param block.	*eigenvalue* must be enabled.
Train/Samples/elapsed_time_ms_forward	The global duration of the forward pass.	*flops_profiler.enabled* or *wall_clock_breakdown*.
Train/Samples/elapsed_time_ms_backward	The global duration of the forward pass.	*flops_profiler.enabled* or *wall_clock_breakdown*.
Train/Samples/elapsed_time_ms_backward_inner	The backward time that does not include the gradient reduction time. Only in cases where the gradient reduction is not overlapped, if it is overlapped then the inner time should be about the same as the entire backward time.	*flops_profiler.enabled* or *wall_clock_breakdown*.
Train/Samples/elapsed_time_ms_backward_allreduce	The global duration of the allreduce operation.	*flops_profiler.enabled* or *wall_clock_breakdown*.
Train/Samples/elapsed_time_ms_step	The optimizer step time.	*flops_profiler.enabled* or *wall_clock_breakdown*.

14.1.1 TensorBoard

class deepspeed.monitor.config.**TensorBoardConfig**

Sets parameters for TensorBoard monitor.

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

enabled: bool = False

Whether logging to Tensorboard is enabled. Requires *tensorboard* package is installed.

output_path: str = ''

Path to where the Tensorboard logs will be written. If not provided, the output path is set under the training script's launching path.

job_name: str = 'DeepSpeedJobName'

Name for the current job. This will become a new directory inside *output_path*.

14.1.2 WandB

class deepspeed.monitor.config.**WandbConfig**

Sets parameters for WandB monitor.

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

enabled: bool = False

Whether logging to WandB is enabled. Requires *wandb* package is installed.

group: Optional[str] = None

Name for the WandB group. This can be used to group together runs.

team: Optional[str] = None

Name for the WandB team.

project: str = 'deepspeed'

Name for the WandB project.

14.1.3 Comet

class deepspeed.monitor.config.**CometConfig**

Sets parameters for Comet monitor. For logging data Comet uses experiment object. https://www.comet.com/docs/v2/api-and-sdk/python-sdk/reference/Experiment/

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

enabled: bool = False

Whether logging to Comet is enabled. Requires *comet_ml* package is installed.

samples_log_interval: int = 100

Metrics will be submitted to Comet after processing every *samples_log_intervas* samples

project: Optional[str] = None

Comet project name. Can be set through .comet.config file or environment variable COMET_PROJECT_NAME https://www.comet.com/docs/v2/guides/experiment-management/configure-sdk/#explore-comet-configuration-options

workspace: Optional[str] = None

Comet workspace name. Can be set through .comet.config file or environment variable COMET_WORKSPACE https://www.comet.com/docs/v2/guides/experiment-management/configure-sdk/#explore-comet-configuration-options

api_key: Optional[str] = None

Comet API key. Can be set through .comet.config file or environment variable COMET_API_KEY https://www.comet.com/docs/v2/guides/experiment-management/configure-sdk/#explore-comet-configuration-options

experiment_name: Optional[str] = None

The name for comet experiment to be used for logging. Can be set through .comet.config file or environment variable COMET_EXPERIMENT_NAME https://www.comet.com/docs/v2/guides/experiment-management/configure-sdk/#explore-comet-configuration-options

experiment_key: Optional[str] = None

The key for comet experiment to be used for logging. Must be an alphanumeric string whose length is between 32 and 50 characters. Can be set through .comet.config or environment variable COMET_EXPERIMENT_KEY https://www.comet.com/docs/v2/guides/experiment-management/configure-sdk/#explore-comet-configuration-options

online: Optional[bool] = None

If True, the data will be logged to Comet server, otherwise it will be stored locally in offline experiment Defaults to True.

mode: Optional[str] = None

Control how the Comet experiment is started, 3 options are possible.:

- "get": Continue logging to an existing experiment identified by the *experiment_key* value.
- "create": Always creates of a new experiment, useful for HPO sweeps.
- "get_or_create" (default): Starts a fresh experiment if required, or persists logging to an existing one.

14.1.4 CSV Monitor

class deepspeed.monitor.config.**CSVConfig**

Sets parameters for CSV monitor.

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

enabled: bool = False

Whether logging to local CSV files is enabled.

output_path: str = ''

Path to where the csv files will be written. If not provided, the output path is set under the training script's launching path.

job_name: str = 'DeepSpeedJobName'

Name for the current job. This will become a new directory inside *output_path*.

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

deepspeed.profiling.flops_profiler.profiler, 71
deepspeed.runtime.pipe.schedule, 56

INDEX

A

activation (*deepspeed.inference.config.QuantizationConfig attribute*), 5
allgather_bucket_size (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 28
allgather_partitions (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 28
allgather_sequential (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 30
allreduce_tied_weight_gradients() (*deepspeed.pipe.PipelineModule method*), 52
api_key (*deepspeed.monitor.config.CometConfig attribute*), 85
autotuner (*in module deepspeed.autotuning*), 75

B

BackwardPass (*class in deepspeed.runtime.pipe.schedule*), 58
base_dir (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 3
base_dir (*deepspeed.inference.config.InferenceCheckpointConfig attribute*), 5
buffer_count (*deepspeed.runtime.zero.config.DeepSpeedZeroOffloadOptimizerConfig attribute*), 32
buffer_count (*deepspeed.runtime.zero.config.DeepSpeedZeroOffloadParamConfig attribute*), 32
buffer_size (*deepspeed.runtime.zero.config.DeepSpeedZeroOffloadParamConfig attribute*), 32
BufferOpInstruction (*class in deepspeed.runtime.pipe.schedule*), 58
build() (*deepspeed.pipe.LayerSpec method*), 53

C

checkpoint (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 3
checkpoint_config (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 3
checkpoint_dir (*deepspeed.inference.config.InferenceCheckpointConfig attribute*), 5
ckpt_layer_path() (*deepspeed.pipe.PipelineModule method*), 52
ckpt_layer_path_list() (*deepspeed.pipe.PipelineModule method*), 52
ckpt_prefix() (*deepspeed.pipe.PipelineModule method*), 52
clone_tensors_for_torch_save() (*in module deepspeed.checkpoint.utils*), 25
compile() (*deepspeed.pipe.PipelineModule method*), 52
config (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 3
contiguous_gradients (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 28
convert_zero_checkpoint_to_fp32_state_dict() (*in module deepspeed.utils.zero_to_fp32*), 25
cpu_offload (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 29
cpu_offload_param (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 29
cpu_offload_use_pin_memory (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 29
cpuadam_cores_perc (*deepspeed.runtime.zero.config.DeepSpeedZeroOffloadOptimizerConfig attribute*), 33

D

DataParallelSchedule (*class in deepspeed.runtime.pipe.schedule*), 57
deepspeed.profiling.flops_profiler.profiler module, 71
deepspeed.runtime.pipe.schedule module, 56
DeepSpeedCPUAdam (*class in deepspeed.ops.adam*), 61
device (*deepspeed.runtime.zero.config.DeepSpeedZeroOffloadOptimizerConfig attribute*), 32
device (*deepspeed.runtime.zero.config.DeepSpeedZeroOffloadParamConfig*

attribute), 32
dtype (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 2

E

elastic_checkpoint (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 29
enable_cuda_graph (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 2
enable_sanity_checks (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 31
enabled (*deepspeed.inference.config.DeepSpeedMoEConfig attribute*), 4
enabled (*deepspeed.inference.config.DeepSpeedTPConfig attribute*), 4
enabled (*deepspeed.inference.config.QuantizationConfig attribute*), 5
enabled (*deepspeed.monitor.config.CometConfig attribute*), 84
enabled (*deepspeed.monitor.config.CSVConfig attribute*), 85
enabled (*deepspeed.monitor.config.TensorBoardConfig attribute*), 84
enabled (*deepspeed.monitor.config.WandbConfig attribute*), 84
end_profile() (*deepspeed.profiling.flops_profiler.profiler.FlopsProfiler method*), 72
ep_group (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 4
ep_group (*deepspeed.inference.config.DeepSpeedMoEConfig attribute*), 5
ep_mp_group (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 4
ep_mp_group (*deepspeed.inference.config.DeepSpeedMoEConfig attribute*), 5
ep_size (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 4
ep_size (*deepspeed.inference.config.DeepSpeedMoEConfig attribute*), 4
experiment_key (*deepspeed.monitor.config.CometConfig attribute*), 85
experiment_name (*deepspeed.monitor.config.CometConfig attribute*), 85

F

fast_init (*deepspeed.runtime.zero.config.DeepSpeedZeroOffloadOptimizerConfig attribute*), 33
filter_match() (*deepspeed.runtime.pipe.ProcessTopology method*), 55
FlopsProfiler (*class in deepspeed.profiling.flops_profiler.profiler*), 71
forward() (*deepspeed.moe.layer.MoE method*), 47
forward() (*deepspeed.pipe.PipelineModule method*), 52
ForwardPass (*class in deepspeed.runtime.pipe.schedule*), 58
FusedAdam (*class in deepspeed.ops.adam*), 62
FusedLamb (*class in deepspeed.ops.lamb*), 63

G

gather_16bit_weights_on_model_save (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 30
get_additional_losses() (*deepspeed.pipe.PipelineModule method*), 52
get_axis_comm_lists() (*deepspeed.runtime.pipe.ProcessTopology method*), 54
get_axis_list() (*deepspeed.runtime.pipe.ProcessTopology method*), 55
get_axis_names() (*deepspeed.runtime.pipe.ProcessTopology method*), 53
get_coord() (*deepspeed.runtime.pipe.ProcessTopology method*), 54
get_default_autocast_lower_precision_modules() (*in module deepspeed.runtime.torch_autocast*), 10
get_dim() (*deepspeed.runtime.pipe.ProcessTopology method*), 54
get_fp32_state_dict_from_zero_checkpoint() (*in module deepspeed.utils.zero_to_fp32*), 23
get_model_profile() (*in module deepspeed.profiling.flops_profiler.profiler*), 73
get_rank() (*deepspeed.runtime.pipe.ProcessTopology method*), 53
get_rank_repr() (*deepspeed.runtime.pipe.ProcessTopology method*), 53
get_total_duration() (*deepspeed.profiling.flops_profiler.profiler.FlopsProfiler method*), 72
get_total_flops() (*deepspeed.profiling.flops_profiler.profiler.FlopsProfiler method*), 72
get_total_macs() (*deepspeed.profiling.flops_profiler.profiler.FlopsProfiler method*), 72
get_total_params() (*deepspeed.profiling.flops_profiler.profiler.FlopsProfiler*

method), 73
group (*deepspeed.monitor.config.WandbConfig attribute*), 84

I

ignore_unused_parameters (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 30
InferenceSchedule (*class in deepspeed.runtime.pipe.schedule*), 57
init_autocast_params() (*in module deepspeed.runtime.torch_autocast*), 10
injection_policy (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 3
injection_policy_tuple (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 3
is_autocast_initialized() (*in module deepspeed.runtime.torch_autocast*), 10
is_first_stage (*deepspeed.runtime.pipe.schedule.PipeSchedule property*), 57
is_last_stage (*deepspeed.runtime.pipe.schedule.PipeSchedule property*), 57

J

job_name (*deepspeed.monitor.config.CSVConfig attribute*), 86
job_name (*deepspeed.monitor.config.TensorBoardConfig attribute*), 84

K

keep_module_on_host (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 2

L

LayerSpec (*class in deepspeed.pipe*), 52
leaf_module (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 31
legacy_stage1 (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 30
load_from_fp32_weights (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 28
load_state_dict_from_zero_checkpoint() (*in module deepspeed.utils.zero_to_fp32*), 24
LoadMicroBatch (*class in deepspeed.runtime.pipe.schedule*), 58
log_trace_cache_warnings (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 31
LRRangeTest (*class in deepspeed.runtime.lr_schedules*), 65

M

max_in_cpu (*deepspeed.runtime.zero.config.DeepSpeedZeroOffloadParam attribute*), 32
max_live_parameters (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 30
max_out_tokens (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 3
max_reuse_distance (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 30
memory_efficient_linear (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 31
mics_hierarchical_params_gather (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 31
mics_shard_size (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 31
min_out_tokens (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 3
mode (*deepspeed.monitor.config.CometConfig attribute*), 85
model_persistence_threshold (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 29
module
    deepspeed.profiling.flops_profiler.profiler, 71
    deepspeed.runtime.pipe.schedule, 56
module_granularity_threshold (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 30
MoE (*class in deepspeed.moe.layer*), 46
moe (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 2
moe_experts (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 4
moe_experts (*deepspeed.inference.config.DeepSpeedMoEConfig attribute*), 4
moe_type (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 4
mp_size (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 4

mpu (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 4
mpu (*deepspeed.inference.config.DeepSpeedTPConfig attribute*), 4

N

num_micro_batches (*deepspeed.runtime.pipe.schedule.PipeSchedule property*), 57
num_pipe_buffers() (*deepspeed.runtime.pipe.schedule.DataParallelSchedule method*), 57
num_pipe_buffers() (*deepspeed.runtime.pipe.schedule.InferenceSchedule method*), 57
num_pipe_buffers() (*deepspeed.runtime.pipe.schedule.PipeSchedule method*), 56
num_pipe_buffers() (*deepspeed.runtime.pipe.schedule.TrainSchedule method*), 57
num_stages (*deepspeed.runtime.pipe.schedule.PipeSchedule property*), 57
nvme_path (*deepspeed.runtime.zero.config.DeepSpeedZeroOffloadOptimizerConfig attribute*), 32
nvme_path (*deepspeed.runtime.zero.config.DeepSpeedZeroOffloadParamConfig attribute*), 32

O

offload_optimizer (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 29
offload_param (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 29
OneCycle (*class in deepspeed.runtime.lr_schedules*), 66
online (*deepspeed.monitor.config.CometConfig attribute*), 85
OptimizerStep (*class in deepspeed.runtime.pipe.schedule*), 57
output_path (*deepspeed.monitor.config.CSVConfig attribute*), 86
output_path (*deepspeed.monitor.config.TensorBoardConfig attribute*), 84
overlap_comm (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 28
override_module_apply (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 31

P

param_persistence_threshold (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 29
pin_memory (*deepspeed.runtime.zero.config.DeepSpeedZeroOffloadOptimizerConfig attribute*), 33
pin_memory (*deepspeed.runtime.zero.config.DeepSpeedZeroOffloadParamConfig attribute*), 32
PipeInstruction (*class in deepspeed.runtime.pipe.schedule*), 57
pipeline_loading_checkpoint (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 31
pipeline_read (*deepspeed.runtime.zero.config.DeepSpeedZeroOffloadOptimizerConfig attribute*), 33
pipeline_write (*deepspeed.runtime.zero.config.DeepSpeedZeroOffloadOptimizerConfig attribute*), 33
PipelineModule (*class in deepspeed.pipe*), 51
PipeSchedule (*class in deepspeed.runtime.pipe.schedule*), 56
prefetch_bucket_size (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 29
print_model_aggregated_profile() (*deepspeed.profiling.flops_profiler.profiler.FlopsProfiler method*), 73
print_model_profile() (*deepspeed.profiling.flops_profiler.profiler.FlopsProfiler method*), 73
ProcessTopology (*class in deepspeed.runtime.pipe*), 53
project (*deepspeed.monitor.config.CometConfig attribute*), 85
project (*deepspeed.monitor.config.WandbConfig attribute*), 84

Q

qkv (*deepspeed.inference.config.QuantizationConfig attribute*), 5
quant (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 3

R

ratio (*deepspeed.runtime.zero.config.DeepSpeedZeroOffloadOptimizerConfig attribute*), 33
RecvActivation (*class in deepspeed.runtime.pipe.schedule*), 59
RecvGrad (*class in deepspeed.runtime.pipe.schedule*), 59
reduce_bucket_size (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 28
reduce_scatter (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 28
ReduceGrads (*class in deepspeed.runtime.pipe.schedule*), 58

ReduceTiedGrads (*class in deepspeed.runtime.pipe.schedule*), 58
replace_method (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 3
replace_with_kernel_inject (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 2
reset_profile() (*deepspeed.profiling.flops_profiler.profiler.FlopsProfiler method*), 72
return_tuple (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 3
round_robin_gradients (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 30

S

safe_get_full_fp32_param() (*in module deepspeed.utils*), 37
safe_get_full_grad() (*in module deepspeed.utils*), 37
safe_get_full_optimizer_state() (*in module deepspeed.utils*), 38
safe_get_local_fp32_param() (*in module deepspeed.utils*), 38
safe_get_local_grad() (*in module deepspeed.utils*), 38
safe_get_local_optimizer_state() (*in module deepspeed.utils*), 38
safe_set_full_fp32_param() (*in module deepspeed.utils*), 39
safe_set_full_grad() (*in module deepspeed.utils*), 40
safe_set_full_optimizer_state() (*in module deepspeed.utils*), 39
safe_set_local_fp32_param() (*in module deepspeed.utils*), 40
safe_set_local_grad() (*in module deepspeed.utils*), 40
safe_set_local_optimizer_state() (*in module deepspeed.utils*), 40
safe_update_full_grad_vectorized() (*in module deepspeed.utils*), 40
samples_log_interval (*deepspeed.monitor.config.CometConfig attribute*), 85
save_mp_checkpoint_path (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 3
save_mp_checkpoint_path (*deepspeed.inference.config.InferenceCheckpointConfig attribute*), 5
save_muon_momentum_buffer_in_memory (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 31
SendActivation (*class in deepspeed.runtime.pipe.schedule*), 58
SendGrad (*class in deepspeed.runtime.pipe.schedule*), 59
set_empty_params (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 3
stage (*deepspeed.runtime.pipe.schedule.PipeSchedule property*), 57
stage (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 28
stage3_gather_fp16_weights_on_model_save (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 30
start_profile() (*deepspeed.profiling.flops_profiler.profiler.FlopsProfiler method*), 72
steps() (*deepspeed.runtime.pipe.schedule.PipeSchedule method*), 56
stop_profile() (*deepspeed.profiling.flops_profiler.profiler.FlopsProfiler method*), 72
sub_group_size (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 29
super_offload (*deepspeed.runtime.zero.config.DeepSpeedZeroOffloadOptimizerConfig attribute*), 33

T

team (*deepspeed.monitor.config.WandbConfig attribute*), 84
tensor_parallel (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 2
TiedLayerSpec (*class in deepspeed.pipe*), 53
topology() (*deepspeed.pipe.PipelineModule method*), 52
tp_grain_size (*deepspeed.inference.config.DeepSpeedTPConfig attribute*), 4
tp_group (*deepspeed.inference.config.DeepSpeedTPConfig attribute*), 4
tp_size (*deepspeed.inference.config.DeepSpeedTPConfig attribute*), 4
training_mp_size (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 3
TrainSchedule (*class in deepspeed.runtime.pipe.schedule*), 57
transposed_mode (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 4
triangular_masking (*deepspeed.inference.config.DeepSpeedInferenceConfig*

attribute), 2
triton_autotune (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 2
type (*deepspeed.inference.config.DeepSpeedMoEConfig attribute*), 5

U

use_all_reduce_for_fetch_params (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 30
use_multi_rank_bucket_allreduce (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 28
use_triton (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 2

W

WarmupCosineLR (*class in deepspeed.runtime.lr_schedules*), 69
WarmupDecayLR (*class in deepspeed.runtime.lr_schedules*), 68
WarmupLR (*class in deepspeed.runtime.lr_schedules*), 68
weight (*deepspeed.inference.config.QuantizationConfig attribute*), 5
workspace (*deepspeed.monitor.config.CometConfig attribute*), 85

Z

zenflow (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 29
zero (*deepspeed.inference.config.DeepSpeedInferenceConfig attribute*), 2
zero_hpz_partition_size (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 31
zero_quantized_gradients (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 31
zero_quantized_nontrainable_weights (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 31
zero_quantized_weights (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 31
zeropp_loco_param (*deepspeed.runtime.zero.config.DeepSpeedZeroConfig attribute*), 31