DeepSpeed

Release 0.14.1

Microsoft

Apr 15, 2024

CONTENTS

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()
```

deepspeed.**add_config_arguments**(*parser*)

Update the argument parser to enabling parsing of DeepSpeed command line arguments.

The set of DeepSpeed arguments include the following: 1) –deepspeed: boolean flag to enable DeepSpeed 2) –deepspeed_config <json file path>: path of a json configuration file to configure DeepSpeed runtime.

Parameters

parser – argument parser

Returns

Updated Parser

Return type

parser

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())
```

deepspeed.**initialize**(*args=None*, *model: Optional[Module] = None*, *optimizer: Optional[Union[Optimizer, Callable[[Union[Iterable[Parameter], Dict[str, Iterable]]], Optimizer]]] = None*, *model_parameters: Optional[Module] = None*, *training_data: Optional[Dataset] = None*, *lr_scheduler: Optional[Union[_LRScheduler, Callable[[Optimizer], _LRScheduler]]] = None*, *distributed_port: int = 29500*, *mpu=None*, *dist_init_required: Optional[bool] = None*, *collate_fn=None*, *config=None*, *config_params=None*)

Initialize the DeepSpeed Engine.

Parameters

- **args** – an object containing local_rank and deepspeed_config fields. This is optional if *config* is passed.
- **model** – Required: nn.module class before apply any wrappers
- **optimizer** – Optional: a user defined Optimizer or Callable that returns an Optimizer object. This overrides any optimizer definition in the DeepSpeed json config.
- **model_parameters** – Optional: An iterable of torch.Tensors or dicts. Specifies what Tensors should be optimized.
- **training_data** – Optional: Dataset of type torch.utils.data.Dataset
- **lr_scheduler** – Optional: Learning Rate Scheduler Object or a Callable that takes an Optimizer and returns a Scheduler object. The scheduler object should define a get_lr(), step(), state_dict(), and load_state_dict() methods
- **distributed_port** – Optional: Master node (rank 0)'s free port that needs to be used for communication during distributed training
- **mpu** – Optional: A model parallelism unit object that implements get_{model,data}_parallel_{rank,group,world_size}()
- **dist_init_required** – Optional: None will auto-initialize torch distributed if needed, otherwise the user can force it to be initialized or not via boolean.
- **collate_fn** – Optional: Merges a list of samples to form a mini-batch of Tensor(s). Used when using batched loading from a map-style dataset.
- **config** – Optional: Instead of requiring args.deepspeed_config you can pass your deepspeed config as an argument instead, as a path or a dictionary.
- **config_params** – Optional: Same as *config*, kept for backwards compatibility.

Returns

A tuple of `engine`, `optimizer`, `training_dataloader`, `lr_scheduler`

- `engine`: DeepSpeed runtime engine which wraps the client model for distributed training.
- `optimizer`: Wrapped optimizer if a user defined `optimizer` is supplied, or if optimizer is specified in json config else `None`.
- `training_dataloader`: DeepSpeed dataloader if `training_data` was supplied, otherwise `None`.
- `lr_scheduler`: Wrapped lr scheduler if user `lr_scheduler` is passed, or if `lr_scheduler` specified in JSON configuration. Otherwise `None`.

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.

`deepspeed.init_distributed`(*dist_backend=None*, *auto_mpi_discovery=True*, *distributed_port=29500*, *verbose=True*, *timeout=datetime.timedelta(seconds=1800)*, *init_method=None*, *dist_init_required=None*, *config=None*, *rank=-1*, *world_size=-1*)

Initialize dist backend, potentially performing MPI discovery if needed

Parameters

- **dist_backend** – Optional (str). torch distributed backend, e.g., nccl, mpi, gloo, hccl
- **Optional** (*auto_mpi_discovery*) –
- **distributed_port** – Optional (int). torch distributed backend port
- **verbose** – Optional (bool). verbose logging
- **timeout** – Optional (timedelta). Timeout for operations executed against the process group. Default value equals 30 minutes.
- **init_method** – Optional (string). Torch distributed, URL specifying how to initialize the process group. Default is "env://" if no init_method or store is specified.
- **config** – Optional (dict). DeepSpeed configuration for setting up comms options (e.g. Comms profiling)
- **rank** – Optional (int). The current manually specified rank. Some init_method like "tcp://" need the rank and world_size as well (see: https://pytorch.org/docs/stable/distributed.html#tcp-initialization)
- **world_size** – Optional (int). Desired world_size for the TCP or Shared file-system initialization.

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.

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: DtypeEnum = 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*.

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: 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: 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: 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: tuple = None

TODO: Add docs

config: 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

enabled: bool = True

Turn tensor parallelism on/off.

tp_size: int = 1

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

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

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**

enabled: bool = True

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

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

qkv: QKVQuantConfig = QKVQuantConfig(enabled=True)

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

checkpoint_dir: str = None

save_mp_checkpoint_path: str = None

base_dir: str = None

Example config:

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

deepspeed.**init_inference**(*model*, *config=None*, ***kwargs*)

Initialize the DeepSpeed InferenceEngine.

Description: all four cases are valid and supported in DS init_inference() API.

Case 1: user provides no config and no kwargs. Default config will be used.

```
generator.model = deepspeed.init_inference(generator.model)
string = generator("DeepSpeed is")
print(string)
```

Case 2: user provides a config and no kwargs. User supplied config will be used.

```
generator.model = deepspeed.init_inference(generator.model, config=config)
string = generator("DeepSpeed is")
print(string)
```

Case 3: user provides no config and uses keyword arguments (kwargs) only.

```
generator.model = deepspeed.init_inference(generator.model,
                                           tensor_parallel={"tp_size": world_size},
                                           dtype=torch.half,
                                           replace_with_kernel_inject=True)
string = generator("DeepSpeed is")
print(string)
```

Case 4: user provides config and keyword arguments (kwargs). Both config and kwargs are merged and kwargs take precedence.

```
generator.model = deepspeed.init_inference(generator.model, config={"dtype": torch.
↪half}, replace_with_kernel_inject=True)
string = generator("DeepSpeed is")
print(string)
```

Parameters

- **model** – Required: original nn.module object without any wrappers
- **config** – Optional: instead of arguments, you can pass in a DS inference config dict or path to JSON file

Returns

A deepspeed.InferenceEngine wrapped model.

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()
```

2.1.1 Forward Propagation

deepspeed.DeepSpeedEngine.**forward**(**args*, ***kwargs*)

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.

2.1.2 Backward Propagation

deepspeed.DeepSpeedEngine.**backward**(**args*, ***kwargs*)

2.1.3 Optimizer Step

deepspeed.DeepSpeedEngine.**step**(*self*, *lr_kwargs=None*)

Execute the weight update step after forward and backward propagation on effective_train_batch.

2.1.4 Gradient Accumulation

deepspeed.DeepSpeedEngine.**is_gradient_accumulation_boundary**(*self*)

Query whether the current micro-batch is at the boundary of gradient accumulation, and thus will trigger gradient reductions and an optimizer step.

Returns
if the current step is a gradient accumulation boundary.

Return type
bool

2.1.5 Model Saving

deepspeed.DeepSpeedEngine.**save_16bit_model**(*self*, *save_dir*, *save_filename='pytorch_model.bin'*, *exclude_frozen_parameters=False*)

Save 16bit model weights

This method saves the 16bit model weights at the desired destination.

Parameters

- **save_dir** – Required. Directory for saving the model
- **save_filename** – Optional. Filename to save to. Defaults to `pytorch_model.bin`
- **exclude_frozen_parameters** – Optional. Exclude frozen parameters from checkpointed state.

Returns
`True` when a model has been saved, `False` otherwise. It will not be saved if stage3_gather_16bit_weights_on_model_save is `False`.

Important: all processes must call this method and not just the process with rank 0. It is because the processes need to work in sync to gather the weights. This method will hang waiting to synchronize with other processes if it's called just for the process with rank 0.

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.

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

deepspeed.InferenceEngine.**forward**(*self*, **inputs*, ***kwargs*)

Execute forward propagation

Parameters

- ***inputs** – Variable length input list
- ****kwargs** – variable length keyword arguments

CHAPTER

FOUR

CHECKPOINTING API

4.1 Model Checkpointing

DeepSpeed provides routines for checkpointing model state during training.

4.1.1 Loading Training Checkpoints

deepspeed.DeepSpeedEngine.**load_checkpoint**(*self*, *load_dir*, *tag=None*, *load_module_strict=True*, *load_optimizer_states=True*, *load_lr_scheduler_states=True*, *load_module_only=False*, *custom_load_fn=None*)

Load training checkpoint

Parameters

- **load_dir** – Required. Directory to load the checkpoint from
- **tag** – Checkpoint tag used as a unique identifier for checkpoint, if not provided will attempt to load tag in 'latest' file
- **load_module_strict** – Optional. Boolean to strictly enforce that the keys in state_dict of module and checkpoint match.
- **load_optimizer_states** – Optional. Boolean to load the training optimizer states from Checkpoint. Ex. ADAM's momentum and variance
- **load_lr_scheduler_states** – Optional. Boolean to add the learning rate scheduler states from Checkpoint.
- **load_module_only** – Optional. Boolean to load only the model weights from the checkpoint. Ex. warmstarting.
- **custom_load_fn** – Optional. Custom model load function.

Returns

A tuple of `load_path` and `client_state`. *`load_path`: Path of the loaded checkpoint. `None` if loading the checkpoint failed. *`client_state`: State dictionary used for loading required training states in the client code.

Important: under ZeRO3, one cannot load checkpoint with `engine.load_checkpoint()` right after `engine.save_checkpoint()`. It is because `engine.module` is partitioned, and `load_checkpoint()` wants a pristine model. If insisting to do so, please reinitialize engine before `load_checkpoint()`.

4.1.2 Saving Training Checkpoints

deepspeed.DeepSpeedEngine.**save_checkpoint**(*self*, *save_dir*, *tag=None*, *client_state={}*, *save_latest=True*, *exclude_frozen_parameters=False*)

Save training checkpoint

Parameters

- **save_dir** – Required. Directory for saving the checkpoint
- **tag** – Optional. Checkpoint tag used as a unique identifier for the checkpoint, global step is used if not provided. Tag name must be the same across all ranks.
- **client_state** – Optional. State dictionary used for saving required training states in the client code.
- **save_latest** – Optional. Save a file 'latest' pointing to the latest saved checkpoint.
- **exclude_frozen_parameters** – Optional. Exclude frozen parameters from checkpointed state.

Important: all processes must call this method and not just the process with rank 0. It is because each process needs to save its master weights and scheduler+optimizer states. This method will hang waiting to synchronize with other processes if it's called just for the process with rank 0.

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*)

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

Returns

- pytorch `state_dict`

Note: this approach may not work if your application doesn't have sufficient free CPU memory and you may need to use the offline approach using the `zero_to_fp32.py` script that is saved with the checkpoint.

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␣
↪cpu
model = model.cpu() # move to cpu
```

(continues on next page)

(continued from previous page)

```
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.

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_file*, *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_file** (-) – path to the pytorch fp32 state_dict output file (e.g. path/pytorch_model.bin)
- **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)
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

deepspeed.checkpointing.**configure**(*mpu_*, *deepspeed_config=None*, *partition_activations=None*, *contiguous_checkpointing=None*, *num_checkpoints=None*, *checkpoint_in_cpu=None*, *synchronize=None*, *profile=None*)

Configure DeepSpeed Activation Checkpointing.

Parameters

- **mpu** – Optional: An object that implements the following methods get_model_parallel_rank/group/world_size, and get_data_parallel_rank/group/world_size
- **deepspeed_config** – Optional: DeepSpeed Config json file when provided will be used to configure DeepSpeed Activation Checkpointing
- **partition_activations** – Optional: Partitions activation checkpoint across model parallel GPUs when enabled. By default False. Will overwrite deepspeed_config if provided
- **contiguous_checkpointing** – Optional: Copies activation checkpoints to a contiguous memory buffer. Works only with homogeneous checkpoints when partition_activations is enabled. Must provide num_checkpoints. By default False. Will overwrite deepspeed_config if provided
- **num_checkpoints** – Optional: Number of activation checkpoints stored during the forward propagation of the model. Used to calculate the buffer size for contiguous_checkpointing Will overwrite deepspeed_config if provided
- **checkpoint_in_cpu** – Optional: Moves the activation checkpoint to CPU. Only works with partition_activation. Default is false. Will overwrite deepspeed_config if provided
- **synchronize** – Optional: Performs get_accelerator().synchronize() at the beginning and end of each call to deepspeed.checkpointing.checkpoint for both forward and backward pass. By default false. Will overwrite deepspeed_config if provided
- **profile** – Optional: Logs the forward and backward time for each deepspeed.checkpointing.checkpoint invocation. Will overwrite deepspeed_config if provided

Returns

None

deepspeed.checkpointing.**is_configured**()

True if deepspeed activation checkpointing has been configured
by calling deepspeed.checkpointing.configure, else returns false

Parameters

None –

Returns
True of configured, else False

4.2.2 Using Activation Checkpointing

deepspeed.checkpointing.**checkpoint**(*function*, **args*)

Checkpoint a model or part of the model. This has been directly copied from torch.utils.checkpoint.

deepspeed.checkpointing.**reset**()

Resets memory buffers related to contiguous memory optimizations. Should be called during eval when multiple forward propagations are computed without any backward propagation that usually clears these buffers. :param None:

Returns
None

4.2.3 Configuring and Checkpointing Random Seeds

deepspeed.checkpointing.**get_cuda_rng_tracker**()

Get cuda rng tracker.

deepspeed.checkpointing.**model_parallel_cuda_manual_seed**(*seed*)

Initialize model parallel cuda seed.

This function should be called after the model parallel is initialized. Also, no get_accelerator().manual_seed should be called after this function. Basically, this is replacement for that function. Two set of RNG states are tracked:

default state: This is for data parallelism and is the same among a
set of model parallel GPUs but different across different model parallel groups. This is used for example for dropout in the non-model-parallel regions.

model-parallel state: This state is different among a set of model
parallel GPUs, but the same across data parallel groups. This is used for example for dropout in model parallel regions.

class deepspeed.checkpointing.**CudaRNGStatesTracker**

Tracker for the cuda RNG states.

Using the *add* method, a cuda rng state is initialized based on the input *seed* and is assigned to *name*. Later, by forking the rng state, we can perform operations and return to our starting cuda state.

class deepspeed.checkpointing.**CheckpointFunction**(**args*, ***kwargs*)

This function is adapted from torch.utils.checkpoint with two main changes:

1) torch.cuda.set_rng_state is replaced with *_set_cuda_rng_state* #ignore-cuda
2) the states in the model parallel tracker are also properly tracked/set/reset.
3) Performance activation partitioning, contiguous memory optimization
4) CPU Checkpointing
5) Profile forward and backward functions

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 32-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.

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

- **minimum** = 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

- **minimum** = 0

overlap_comm: 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*.

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

- **minimum** = 0

cpu_offload_param: bool = None

Deprecated, please use `offload_param`

cpu_offload_use_pin_memory: bool = None

Deprecated, please use `offload_param` or `offload_optimizer`

cpu_offload: 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

- **minimum** = 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

- **minimum** = 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

- **minimum** = 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

- **minimum** = 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

- **minimum** = 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.

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

- **minimum** = 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

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.

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

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

device: OffloadDeviceEnum = 'none'

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

nvme_path: 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

- **minimum** = 0

buffer_size: int = 100,000,000

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

Constraints

- **minimum** = 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

- **minimum** = 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.

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: 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

- **minimum** = 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

- **minimum** = 0.0
- **maximum** = 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
        }
    },
    ...
}
```

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

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

(continues on next page)

(continued from previous page)

```
    ...
}
```

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()
```

class deepspeed.zero.**Init**(*module=None*, *data_parallel_group=None*, *mem_efficient_linear=True*, *remote_device=None*, *pin_memory=False*, *config_dict_or_path=None*, *config=None*, *enabled=True*, *dtype=None*, *mpu=None*, *zero_param_parallel_group=None*, *zero_quantized_weights=False*, *zero_quantized_nontrainable_weights=False*, *sequence_data_parallel_group=None*, *param_swapper=None*)

get_partition_rank()

subclass can overload to specify different relative rank in parameter partition group

get_dp_process_group()

Return the communication group with all data-parallel ranks

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()*.

class deepspeed.zero.**GatheredParameters**(*params*, *modifier_rank=None*, *fwd_module=None*, *enabled=True*)

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
```

deepspeed.zero.**register_external_parameter**(*module*, *parameter*)

Instruct DeepSpeed to coordinate `parameter`'s collection and partitioning in the forward and backward passes of `module`.

This is used when a parameter is accessed outside of its owning module's `forward()`. DeepSpeed must know to collect it from its partitioned state and when to release the memory.

Note: This is only applicable to training with ZeRO stage 3.

Parameters

- **module** (`torch.nn.Module`) – The module that requires `parameter` in its forward pass.
- **parameter** (`torch.nn.Parameter`) – The parameter to register.

Raises

RuntimeError – If `parameter` is not of type `torch.nn.Parameter`.

Examples

1. Register a weight that is used in another module's forward pass (line 6). Parameter `layer1.weight` is used by `layer2` (line 11).

```
1  class ModuleZ3(torch.nn.Module):
2      def __init__(self, *args):
3          super().__init__(self, *args)
4          self.layer1 = SomeLayer()
5          self.layer2 = OtherLayer()
6          deepspeed.zero.register_external_parameter(self, self.layer1.
   ↪weight)
7
8      def forward(self, input):
9          x = self.layer1(input)
10         # self.layer1.weight is required by self.layer2.forward
11         y = self.layer2(x, self.layer1.weight)
12         return y
```

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.

class deepspeed.zero.**TiledLinear**(*in_features*, *out_features*, *bias=True*, *in_splits=1*, *out_splits=1*, *input_is_already_split=False*, *combine_out_splits=True*, *linear_cls=<class 'torch.nn.modules.linear.Linear'>*, *init_linear=None*, ***kwargs*)

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.

---

copy_params_from(*other*)

Copy the weight and bias data from `other`.

This is especially useful for reproducible initialization and testing.

Equivalent to:

```
with torch.no_grad():
    self.weight.copy_(other.weight)
    if self.bias is not None:
        self.bias.copy_(other.bias)
```

Note: If ZeRO-3 is enabled, this is a collective operation and the updated parameters of data-parallel rank 0 will be visible on all ranks. See *deepspeed.zero.GatheredParameters* for more information.

Parameters

other (`torch.nn.Linear`) – the linear layer to copy from.

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: Please note that, 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.

Additionally, 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

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

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

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

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)

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

Get the fp32 partitioned parameter. :param param: A model parameter :type param: `torch.nn.Parameter`

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

Get the fp32 gradient of a partitioned parameter. :param param: A model parameter :type param: `torch.nn.Parameter`

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

Get the fp32 optimizer state of a partitioned parameter. :param param: A model parameter :type param: `torch.nn.Parameter` :param optim_state_key: Key value of optimizer state (e.g., *exp_avg* in Adam optimizer) :type optim_state_key: `string`

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) 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)


    # 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")
```

(continues on next page)

(continued from previous page)

```
    # 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 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_local_fp32_param**(*param*, *value*)

Update the partitioned fp32 parameter. :param param: A model parameter :type param: torch.nn.Parameter :param value: New value :type value: torch.Tensor

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

Update the fp32 optimizer state of a partitioned parameter. :param param: A model parameter :type param: torch.nn.Parameter :param value: New value :type value: torch.Tensor :param optim_state_key: Key value of optimizer state (e.g., *exp_avg* in Adam optimizer) :type optim_state_key: string

These routines can be used at any point after initialization of the DeepSpeed engine (i.e., deepspeed.initialize()) as shown in the following snippet.

```
[...]
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():
```

(continues on next page)

(continued from previous page)

```
    # 1. For zero stage 1 or 2, set the full fp32 and their full optim states
    zero_tensor = torch.zeros_like(lp)

    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_like(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")

[...]
```

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()
```

CHAPTER

SIX

MIXTURE OF EXPERTS (MOE)

6.1 Mixture of Experts (MoE)

6.1.1 Layer specification

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

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

class deepspeed.**DeepSpeedTransformerConfig**(*batch_size=-1*, *hidden_size=-1*, *intermediate_size=-1*, *heads=-1*, *attn_dropout_ratio=-1*, *hidden_dropout_ratio=-1*, *num_hidden_layers=-1*, *initializer_range=-1*, *layer_norm_eps=1e-12*, *local_rank=-1*, *seed=-1*, *fp16=False*, *pre_layer_norm=True*, *normalize_invertible=False*, *gelu_checkpoint=False*, *adjust_init_range=True*, *attn_dropout_checkpoint=False*, *stochastic_mode=False*, *return_tuple=False*, *training=True*)

Initialize the DeepSpeed Transformer Config.

Parameters

- **batch_size** – The maximum batch size used for running the kernel on each GPU
- **hidden_size** – The hidden size of the transformer layer
- **intermediate_size** – The intermediate size of the feed-forward part of transformer layer
- **heads** – The number of heads in the self-attention of the transformer layer
- **attn_dropout_ratio** – The ratio of dropout for the attention's output
- **hidden_dropout_ratio** – The ratio of dropout for the transformer's output
- **num_hidden_layers** – The number of transformer layers
- **initializer_range** – BERT model's initializer range for initializing parameter data
- **local_rank** – Optional: The rank of GPU running the transformer kernel, it is not required to use if the model already set the current device, otherwise need to set it so that the transformer kernel can work on the right device
- **seed** – The random seed for the dropout layers
- **fp16** – Enable half-precision computation
- **pre_layer_norm** – Select between Pre-LN or Post-LN transformer architecture

- **normalize_invertible** – Optional: Enable invertible LayerNorm execution (dropping the input activation), default is False
- **gelu_checkpoint** – Optional: Enable checkpointing of Gelu activation output to save memory, default is False
- **adjust_init_range** – Optional: Set as True (default) if the model adjusts the weight initial values of its self-attention output and layer output, False keeps the initializer_range no change. See the adjustment below:

  output_std = self.config.initializer_range / math.sqrt(2.0 * num_layers)
- **attn_dropout_checkpoint** – Optional: Enable checkpointing of attention dropout to save memory, default is False
- **stochastic_mode** – Enable for high performance, please note that this flag has some level of non-determinism and can produce different results on different runs. However, we have seen that by enabling it, the pretraining tasks such as BERT are not affected and can obtain a high accuracy level. On the other hand, for the downstream tasks, such as fine-tuning, we recommend to turn it off in order to be able to reproduce the same result through the regular kernel execution.
- **return_tuple** – Enable if using the return_tuple interface style for sending out the forward results.
- **training** – Enable for training rather than inference.

7.1.2 DeepSpeed Transformer Layer

class deepspeed.**DeepSpeedTransformerLayer**(*config*, *initial_weights=None*, *initial_biases=None*)

Initialize the DeepSpeed Transformer Layer.

Static variable:

layer_id: The layer-index counter starting from 0 and incrementing by 1 every time a layer object is instantiated, e.g. if a model has 24 transformer layers, layer_id goes from 0 to 23.

Parameters

- **config** – An object of DeepSpeedTransformerConfig
- **initial_weights** – Optional: Only used for unit test
- **initial_biases** – Optional: Only used for unit test

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*)

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, optional*) – Checkpointable layers may not be checkpointed. Defaults to None which does not additional filtering.

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.

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.

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

class deepspeed.runtime.pipe.engine.**PipelineEngine**(*has_bool_tensors=False*, **super_args*, ***super_kwargs*)

A training engine hybrid pipeline, data, and model parallel training.

This engine is created by `deepspeed.initialize()` when a `PipelineModule` is provided.

reset_activation_shape()

Reset the buffers when the shape of activation and gradient change. For example, for curriculum learning that changes the seqlen of each sample, we need to call this whenever the seqlen is going to change.

train_batch(*data_iter=None*)

Progress the pipeline to train the next batch of data. The engine will ingest `self.train_batch_size()` total samples collectively across all workers.

An iterator that over training data should be provided as an argument unless `deepspeed.initialize()` was provided a training set. In that event, the training data will automatically be read.

> **Warning:** A total of `self.gradient_accumulation_steps()` entries will be pulled from `data_iter` by each pipeline. There must be sufficient data left in `data_iter` or else a `StopIteration` will halt training.
>
> DeepSpeed provides a convenience class `deepspeed.utils.RepeatingLoader` that wraps data loaders to automatically restart upon a `StopIteration`.

Parameters
data_iter (*Iterator, optional*) – Iterator of training data.

Returns
The arithmetic mean of the losses computed this batch.

`eval_batch`(*data_iter*, *return_logits=False*, *compute_loss=True*, *reduce_output='avg'*, *bcast_loss=True*, *num_micro_batches=None*)

Evaluate the pipeline on a batch of data from `data_iter`. The engine will evaluate `self.train_batch_size()` total samples collectively across all workers.

This method is equivalent to:

```
module.eval()
with torch.no_grad():
    output = module(batch)
```

Warning: A total of `self.gradient_accumulation_steps()` entries will be pulled from `data_iter` by each pipeline. There must be sufficient data left in `data_iter` or else a `StopIteration` will halt training.

DeepSpeed provides a convenience class `deepspeed.utils.RepeatingLoader` that wraps data loaders to automatically restart upon a `StopIteration`.

Parameters
data_iter (*Iterator*) – Iterator of data to evaluate.

Returns
The arithmetic mean of the losses computed this batch.

`set_train_batch_size`(*train_batch_size*)

Adjust the global batch size by increasing or decreasing the number of micro-batches (i.e., gradient accumulation steps). The size of each micro-batch (i.e., `train_micro_batch_size_per_gpu`) is not changed. :param train_batch_size: The new global batch size for training. :type train_batch_size: int

Raises
ValueError – if `train_batch_size` is not divisible by the configured micro-batch size and data parallelism.

`is_first_stage`()

True if this process is in the first stage in the pipeline.

`is_last_stage`()

True if this process is in the last stage in the pipeline.

`set_dataiterator`(*iterator*)

Store an iterator to sample for training data.

`set_batch_fn`(*fn*)

Execute a post-processing function on input data.

Parameters
fn (*function*) – The function to run.

`is_gradient_accumulation_boundary`()

True if the engine is executing a gradient reduction or optimizer step instruction.

This is overridden from `DeepSpeedEngine` to force reductions and steps when the pipeline engine is instructed to do so.

Returns
whether reductions and optimizer steps should occur.

Return type
bool

forward(**args*, ***kwargs*)

Disabled for pipeline parallel training. See `train_batch()`.

backward(**args*, ***kwargs*)

Disabled for pipeline parallel training. See `train_batch()`.

step(**args*, ***kwargs*)

Disabled for pipeline parallel training. See `train_batch()`.

module_state_dict(*exclude_frozen_parameters=False*)

Override hack to save a pipe model and return the directory path of the save.

This method should only be called by DeepSpeed's `save_checkpoint()`. The recommended way of saving a `PipelineModule` outside of `save_checkpoint()` is `save_state_dict()`.

Returns
None

load_module_state_dict(*checkpoint*, *strict=True*, *custom_load_fn=None*, *fetch_z3_params=False*)

Override hack to instead use a directory path.

This is important because pipeline models checkpoint by layer instead of rank.

If `state_dict` is not `None` or a `str`, we revert to `super()` expecting a `dict`.

Parameters

- **state_dict** (*str, None*) – unused
- **strict** (*bool, optional*) – Strict state loading. Defaults to True.

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([
```

(continues on next page)

(continued from previous page)

```
                    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*)

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)

class deepspeed.runtime.fp16.onebit.adam.**OnebitAdam**(*params*, *deepspeed=None*, *lr=0.001*, *freeze_step=100000*, *bias_correction=True*, *betas=(0.9, 0.999)*, *eps=1e-08*, *eps_inside_sqrt=False*, *weight_decay=0.0*, *max_grad_norm=0.0*, *amsgrad=False*, *cuda_aware=False*, *comm_backend_name='nccl'*)

Implements the 1-bit Adam algorithm. Currently GPU-only. For usage example please see https://www.deepspeed.ai/tutorials/onebit-adam/ For technical details please read https://arxiv.org/abs/2102.02888

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups.
- **lr** (*float, optional*) – learning rate. (default: 1e-3)
- **freeze_step** (*int, optional*) – Number of steps for warmup (uncompressed) stage before we start using compressed communication. (default 100000)
- **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 1-bit Adam!
- **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)
- **cuda_aware** (*boolean, required*) – Set True if the underlying MPI implementation supports CUDA-Aware communication. (default: False)
- **comm_backend_name** (*string, optional*) – Set to 'mpi' if needed. (default: 'nccl')

9.1.5 ZeroOneAdam (GPU)

class deepspeed.runtime.fp16.onebit.zoadam.**ZeroOneAdam**(*params*, *deepspeed=None*, *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*, *var_freeze_step=100000*, *var_update_scaler=16*, *local_step_scaler=32678*, *local_step_clipper=16*, *amsgrad=False*, *cuda_aware=False*, *comm_backend_name='nccl'*)

Implements the 0/1 Adam algorithm. Currently GPU-only. For usage example please see https://www.deepspeed.ai/tutorials/zero-one-adam/ For technical details please read https://arxiv.org/abs/2202.06009 :param params: iterable of parameters to optimize or dicts defining

parameter groups.

Parameters

- **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)
- **var_freeze_step** (*int, optional*) – The latest step to update the variance, using the notation from https://arxiv.org/abs/2202.06009, it denotes the max{i|i in T_v}. Note that this is different from the freeze step from the 1-bit Adam. The var_freeze_step is usually the end of the learning rate warmup and thus does not require tuning. (default: 100000)
- **var_update_scaler** (*int, optional*) – The interval to update the variance. Note that the update policy for variance follows an exponential rule, where var_update_scaler denotes the kappa in the 0/1 Adam paper. (default: 16)
- **local_step_scaler** (*int, optional*) – The interval to scale the local steps interval according to the learning rate policy. (default: 32678)
- **local_step_clipper** (*int, optional*) – The largest interval for local steps with learning rate policy. This corresponds to the variable H in the 0/1 Adam paper. (default: 16)
- **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 0/1 Adam!
- **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)
- **cuda_aware** (*boolean, required*) – Set True if the underlying MPI implementation supports CUDA-Aware communication. (default: False)
- **comm_backend_name** (*string, optional*) – Set to 'mpi' if needed. (default: 'nccl')

9.1.6 OnebitLamb (GPU)

class deepspeed.runtime.fp16.onebit.lamb.**OnebitLamb**(*params*, *deepspeed=None*, *lr=0.001*, *freeze_step=100000*, *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*, *cuda_aware=False*, *comm_backend_name='nccl'*, *coeff_beta=0.9*, *factor_max=4.0*, *factor_min=0.5*, *factor_threshold=0.1*)

Implements the 1-bit Lamb algorithm. Currently GPU-only. For usage example please see https://www.deepspeed.ai/tutorials/onebit-lamb/ For technical details please see our paper https://arxiv.org/abs/2104.06069.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups.
- **lr** (*float, optional*) – learning rate. (default: 1e-3)
- **freeze_step** (*int, optional*) – Number of steps for warmup (uncompressed) stage before we start using compressed communication. (default 100000)
- **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)
- **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*) – whether to use the AMSGrad variant of this algorithm from the paper On the Convergence of Adam and Beyond (default: False) NOT SUPPORTED in 1-bit Lamb!
- **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)
- **cuda_aware** (*boolean, required*) – Set True if the underlying MPI implementation supports CUDA-Aware communication. (default: False)
- **comm_backend_name** (*string, optional*) – Set to 'mpi' if needed. (default: 'nccl')
- **coeff_beta** (*float, optional*) – coefficient used for computing running averages of lamb coefficient (default: 0.9) note that you may want to increase or decrease this beta depending on the freeze_step you choose, as 1/(1 - coeff_beta) should be smaller than or equal to freeze_step
- **factor_max** (*float, optional*) – maximum value of scaling factor to the frozen lamb coefficient during compression stage (default: 4.0)
- **factor_min** (*float, optional*) – minimum value of scaling factor to the frozen lamb coefficient during compression stage (default: 0.5)
- **factor_threshold** (*float, optional*) – threshold of how much the scaling factor can fluctuate between steps (default: 0.1)

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: Part1`_**.

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/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: 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 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:

deepspeed.runtime.zero.stage_1_and_2.**estimate_zero2_model_states_mem_needs_all_live**(*model*, *num_gpus_per_node=1*, *num_nodes=1*, *additional_buffer_factor=1.5*)

Print out estimates on memory usage requirements for ZeRO 2 params, optim states and gradients for a given `model` and hardware setup.

If you have an actual model object, use this function and everything will be derived automatically.

If it's a hypothetical model, use `estimate_zero2_model_states_mem_needs_all_cold` where you have to pass the `total_params` explicitly.

Parameters

- **model** (-) – `nn.Module` object
- **num_gpus_per_node** (-) – how many gpus per node (defaults to 1)
- **num_nodes** (-) – how many nodes (defaults to 1),
- **additional_buffer_factor** (-) – estimation factor (defaults to 1.5):

deepspeed.runtime.zero.stage_1_and_2.**estimate_zero2_model_states_mem_needs_all_cold**(*total_params*, *num_gpus_per_node=1*, *num_nodes=1*, *additional_buffer_factor=1.5*)

Print out estimates on memory usage requirements for ZeRO 2 params, optim states and gradients for a given `model` and hardware setup.

If it's a hypothetical model, use this function where you have to pass the `total_params` and `largest_layer_params` explicitly.

If you have an actual model object, use `estimate_zero2_model_states_mem_needs_all_live` and everything will be derived automatically.

Parameters

- **total_params** (-) – total model params

- **num_gpus_per_node** (-) – how many gpus per node (defaults to 1)
- **num_nodes** (-) – how many nodes (defaults to 1),
- **additional_buffer_factor** (-) – estimation factor (defaults to 1.5):

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:

deepspeed.runtime.zero.stage3.**estimate_zero3_model_states_mem_needs_all_live**(*model*, *num_gpus_per_node=1*, *num_nodes=1*, *additional_buffer_factor=1.5*)

Print out estimates on memory usage requirements for ZeRO 3 params, optim states and gradients for a given `model` and hardware setup.

If you have an actual model object, use this function and everything will be derived automatically.

If it's a hypothetical model, use `estimate_zero3_model_states_mem_needs_all_cold` where you have to pass the `total_params` and `largest_layer_params` explicitly.

Parameters

- **model** (-) – `nn.Module` object
- **num_gpus_per_node** (-) – how many gpus per node (defaults to 1)

- **num_nodes** (-) – how many nodes (defaults to 1),
- **additional_buffer_factor** (-) – estimation factor (defaults to 1.5):

deepspeed.runtime.zero.stage3.**estimate_zero3_model_states_mem_needs_all_cold**(*total_params*, *largest_layer_params*, *num_gpus_per_node=1*, *num_nodes=1*, *additional_buffer_factor=1.5*)

Print out estimates on memory usage requirements for ZeRO 3 params, optim states and gradients for a given `model` and hardware setup.

If it's a hypothetical model, use this function where you have to pass the `total_params` and `largest_layer_params` explicitly.

If you have an actual model object, use `estimate_zero3_model_states_mem_needs_all_live` and everything will be derived automatically.

Parameters

- **total_params** (-) – total model params
- **largest_layer_params** (-) – largest layer's params
- **num_gpus_per_node** (-) – how many gpus per node (defaults to 1)
- **num_nodes** (-) – how many nodes (defaults to 1),
- **additional_buffer_factor** (-) – estimation factor (defaults to 1.5):

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)'
```

(continues on next page)

(continued from previous page)

```
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)

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.

14.1.1 TensorBoard

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

Sets parameters for TensorBoard monitor.

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.

enabled: bool = False

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

group: str = None

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

team: str = None

Name for the WandB team.

project: str = 'deepspeed'

Name for the WandB project.

14.1.3 CSV Monitor

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

Sets parameters for CSV monitor.

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

INDEX

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

W

Z