DeepSpeed

Release 0.3.0

Feb 23, 2021

Contents

CHAPTER 1

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 intialized already.

Example usage:

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

deepspeed.**initialize**(*args*, *model*, *optimizer=None*, *model_parameters=None*, *training_data=None*, *lr_scheduler=None*, *mpu=None*, *dist_init_required=None*, *collate_fn=None*, *config_params=None*)

Initialize the DeepSpeed Engine.

Parameters

- **args** – a dictionary containing local_rank and deepspeed_config file location
- **model** – Required: nn.module class before apply any wrappers
- **optimizer** – Optional: a user defined optimizer, this is typically used instead of defining an optimizer 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. It should define a get_lr(), step(), state_dict(), and load_state_dict() methods
- **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.

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 initializating 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='nccl'*, *auto_mpi_discovery=True*, *distributed_port=29500*, *verbose=True*, *timeout=datetime.timedelta(seconds=1800)*, *init_method=None*)

Initialize torch.distributed backend, potentially performing MPI discovery if needed

Parameters

- **dist_backend** – Optional (str). torch distributed backend, e.g., nccl, mpi, gloo
- **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.

CHAPTER 2

Configuration

2.1 DeepSpeed Configuration

2.1.1 Configurations

Training Setup

class deepspeed.config.**TrainingConfig**(***kwargs*)
Top-level configuration for all aspects of training with DeepSpeed.

batch = None
Batch configuration, see *BatchConfig*

fp16 = None
FP16 training, see *FP16Config*

class deepspeed.config.**BatchConfig**(***kwargs*)
Batch size related parameters.

train_batch_size = None
The effective training batch size.

This is the number of data samples that leads to one step of model update. *train_batch_size* is aggregated by the batch size that a single GPU processes in one forward/backward pass (a.k.a., train_step_batch_size), the gradient accumulation steps (a.k.a., *gradient_accumulation_steps*), and the number of GPUs.

train_micro_batch_size_per_gpu = None
The batch size to be processed per device each forward/backward step.

When specified, gradient_accumulation_steps is automatically calculated using train_batch_size and the number of devices. Should not be concurrently specified with gradient_accumulation_steps.

gradient_accumulation_steps = None
The number of training steps to accumulate gradients before averaging and applying them.

This feature is sometimes useful to improve scalability since it results in less frequent communication of gradients between steps. Another impact of this feature is the ability to train with larger batch sizes per GPU. When specified, `train_step_batch_size` is automatically calculated using `train_batch_size` and number of GPUs. Should not be concurrently specified with `train_step_batch_size`.

resolve()
Complete batch configuration so long as two are provided.

is_valid()
Resolve any missing configurations and determine in the configuration is valid.

Returns Whether the config and all sub-configs are valid.

Return type bool

class deepspeed.config.**FP16Config**(***kwargs*)
FP16 configuration.

enabled = None
Enable/disable FP16

clip = None
Gradient clipping

Training Optimizations

class deepspeed.config.**FP16Config**(***kwargs*)
FP16 configuration.

enabled = None
Enable/disable FP16

clip = None
Gradient clipping

2.1.2 Extending Configurations

class deepspeed.config.**Config**(***kwargs*)
Base class for DeepSpeed configurations.

`Config` is a struct with subclassing. They are initialized from dictionaries and thus also keyword arguments:

```
>>> c = Config(verbose=True)
>>> c.verbose
True
>>> c['verbose']
True
```

You can initialize them from dictionaries:

```
>>> myconf = {'verbose' : True}
>>> c = Config.from_dict(myconf)
>>> c.verbose
True
```

Configurations should be subclassed to group arguments by topic.

resolve()
Infer any missing arguments, if possible.

This is useful for configs such as *BatchConfig* in only a subset of arguments are required to complete a valid config.

is_valid()
Resolve any missing configurations and determine in the configuration is valid.

Returns Whether the config and all sub-configs are valid.

Return type bool

CHAPTER 3

Training API

3.1 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()
```

3.1.1 Forward Propagation

`deepspeed.DeepSpeedEngine.`**`forward`**(*self*, **inputs*, ***kwargs*)

Execute forward propagation

Parameters

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

3.1.2 Backward Propagation

`deepspeed.DeepSpeedEngine.`**`backward`**(*self*, *loss*, *allreduce_gradients=True*, *release_loss=False*)

Execute backward pass on the loss

Parameters

- **loss** – Torch tensor on which to execute backward propagation
- **allreduce_gradients** – If this is False, then gradient averaging will be skipped. Default is True.

3.1.3 Optimizer Step

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

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

3.1.4 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

CHAPTER 4

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 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.

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.

4.1.2 Saving Training Checkpoints

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

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.

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 torch.cuda.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 torch.cuda.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 paralle 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*
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 5

Transformer Kernel API

5.1 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.

5.1.1 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*, *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*, *huggingface=False*, *training=True*)

Initialize the DeepSpeed Transformer Config.

Parameters

- **batch_size** – The maximum batch size used for running the kernel on each GPU
- **max_seq_length** – The sequence-length of the model being trained with DeepSpeed
- **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.
- **huggingface** – Enbale if using the HuggingFace interface style for sending out the forward results.
- **training** – Enable for training rather than inference.

5.1.2 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 6

Pipeline Parallelism

6.1 Pipeline Parallelism

6.1.1 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>*)

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
```

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** (`deepseed.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)`
- **base_seed** (*int, optional*) – [description]. Defaults to 1234.
- **partition_method** (*str, optional*) – [description]. 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`.

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.

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*)

6.1.2 Training

class deepspeed.runtime.pipe.engine.**PipelineEngine**(**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.

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*)

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.

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.

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()
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(*state_dict*, *strict=True*)
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.

set_batch_fn(*fn*)
Execute a post-processing function on input data.

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

6.1.3 Extending Pipeline Parallelism

class deepspeed.runtime.pipe.schedule.**PipeSchedule**(*micro_batches*, *stages*, *stage_id*)
Directs the execution of a pipeline engine by generating sequences of *PipeInstruction*.

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

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

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

    def num_pipe_buffers(self):
        return 1
```

Parameters

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

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.

stage
Stage index used to configure this schedule.

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

num_micro_batches
The number of total micro_batches used to configure this schedule.

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

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()
As many buffers as the distance from this stage to 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['ouputs'][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['ouputs'][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 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

L

M

N

O

P

R

S

T