Varuna

Nitika Saran

Feb 15, 2021

CONTENTS:

1 The Varuna class 3

2 Indices and tables 7

Index 9

Detailed documentation for Varuna functions and classes can be found and navigated here.

CHAPTER

ONE

THE VARUNA CLASS

The torch.nn.Module object for your DNN model should be wrapped in a *Varuna* instance for training. This class extends torch.nn.Module and handles distributed pipeline & data parallelism, mixed precision and shared parameter weights internally.

Wrapping in *Varuna* partitions the model into pipeline stages across the distributed job. For this, it uses stage allocation information that is passed by `varuna.launcher` to all worker processes. The launcher uses a string argument `stage_to_rank_map` which must be parsed and used for *Varuna* initialisation.

Sample inputs (with any batch size) also need to be passed for this automatic partitioning. These inputs are used to profile the model's computation graph and should be passed a a dictionary of keywords to args.

The model passed to *Varuna* should be on CPU. Once the profiling and partitioning are done, the model is moved to the assigned GPU. So the user need not do `model.cuda()` anywhere.

Optimizer creation should be after wrapping in *Varuna*, since it requires model parameters as input. The optimizer needs to be registered with Varuna using a setter.

Example:

```
model = MyModel()               # full model on CPU
dry_run_input = {               # sample inputs for MyModel
   'inputs': inputs,            # keyword: arg
   'mask': mask,
   'extra_norm': True
}
# parameter sharing across the model, marked as pairs of param_names
shared_weights = [("language_model.embedding.word_embeddings.weight","lm_head_weight
↪")]
model = Varuna( model, args.stage_to_rank_map, dry_run_input, global_batch_size,
                  args.chunk_size, args.fp16, local_rank=args.local_rank,
                  device=args.local_rank, shared_weights=shared_weights)

# now model is a subset of the original model, moved to the GPU on each process

optimizer = get_optimizer(model)
model.set_optimizer(optimizer)
```

class varuna.**Varuna**(*model*, *stage_to_rank_map*, *dummy_inputs*, *batch_size*, *chunk_size*, *fp16=False*, *local_rank=- 1*, *device=- 1*, *shared_weights=None*, *from_cache=True*)

Module to implement varuna training. The model must be wrapped in an instance of `Varuna` before training. This should be done before optimizer creation and the `model` passed should be on CPU.

Creating a `Varuna` instance profiles the model briefly using `dummy_inputs` and partitions it according to the distributed rank and launcher arguments. The partitioned model is then moved to the allocated cuda device. The profiling information is cached and can be re-used on resuming, unless `from_cache` is False. The `Varuna`

module performs mixed precision training internally if enabled through the `fp16` arg, no external handling is required.

Args: model (nn.Module): The model to initialize for training.

stage_to_rank_map (str): Placement of pipeline stages in the distribued job, encoded as a string. Passed by `varuna.launcher` to each worker as an argument.

dummy_inputs (dict): Sample inputs to the model as a dictionary. These are used to profile the model as `model(**dummy_inputs)`. The batch size dimention in these inputs can be any `n>=1`, but is recommended to be small for speed.

batch_size (int): Global batch size for the distributed training job.

chunk_size (int): The micro-batch size to be used for pipeline parallelism.

fp16 (bool): whether to enable mixed precision training.

local_rank (int): The local rank as passed by `varuna.launcher`. If not given, defaults to the global rank.

device (int): index of the cuda device to use. Recommended to be the same as local_rank, which is the default if not specified.

shared_weights (list or None): A list of tuples, where each each tuple is a pair of weight names (strings), such that the two weights are shared in the model (see weight sharing)

from_cache (bool): Whether to use cached profiling information if available.

Note: Optimizer initiliastion should be done after `Varuna` initialisation, so that the `param_group` s for the optimizer only contain parameters from the partitioned model. This is important both for memory usage and correctness of fp16 training. Once `Varuna` and the optimizer are initialised, *`set_optimizer()`* should be called to connect the two.

`set_optimizer`(*optimizer*, *loss_scale='dynamic'*, *init_loss_scale=1048576*, *min_loss_scale=1.0*)

Configure optimizer for training. if `fp16` is enabled, this function initializes the mixed precision state in apex.

Args: optimizer (nn.Optimizer): the optimizer for training.

loss_scale (float or "dynamic", optional): A floating point number for a static loss scale or the string "dynamic" for dynamic loss scaling.

init_loss_scale (float, optional): Initial loss scale (for dynamic scaling)

min_loss_scale (float, optional): minimum loss scale (for dynamic scaling)

`step`(*inputs*, *clip_grad_max_norm=None*)

Perform a single training step. Executes forward and backward passes for the global batch. This function must be called by all distributed workers in the training loop. After this function, the optimizer gradients are reduced accross data parallel replicas and overflow is checked for mixed precision training.

Returns average loss and a boolean for overflow.

Args: inputs (dict): The inputs to the model as a dictionary. These should be coordinated amongst workers - the global batch is sharded across data parallel replicas, so each worker should have `global_batch_size / data_parallel_depth` number of examples. And all pipeline stages of the same data parallel replica should recieve the same inputs.

clip_grad_max_norm (float or None, optional): If given, the L2 gradient norm of the entire model is clipped to this upper bound.

checkpoint(*global_store*, *step=None*, *tempdir=None*, *shard=False*, *on_demand=False*)

Writes a varuna checkpoint with model parameters, optimizer state etc. Each checkpoint is a directory, written under the given path.

Args: global_store (str): path to a folder accessible by all nodes/ranks in the training job. For example, path to a mounted blob storage. This is where the varuna checkpoint folder is written.

step (int or None): iteration number for checkpoint. If None, it'll be taken from varuna's tracked progress.

tempdir (str): path to a local directory to which to write checkpoints temporarily, and sync with the global store in the background. Lowers checkpoint write time in the critical path.

shard (bool): whether to shard checkpoint writes over data parallel workers as well. Speeds up checkpoint

load_checkpoint(*global_store*, *iteration*, *check_complete=True*)

Loads a varuna checkpoint from a shared directory. Each varuna checkpoint is a directory named as "varuna_ckpt_<iteration>". So the path under which all such checkpoints were written should be specified.

Args: global_store (str): path under which varuna checkpoints were written. Should be accessible by all workers.

iteration (int): Which iteration checkpoint to load.

check_complete (bool, optional): Check that the checkpoint is complete before loading it. A checkpoint can be incomplete if the write was interrupted.

evaluate(*inputs*)

Evaluate the model on the given inputs. Returns loss.

Args: inputs (dict): Model inputs as dictionary. The number of examples for these inputs should be the same as the batch_size defined for training.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

C

checkpoint() (*varuna.Varuna method*), 4

E

evaluate() (*varuna.Varuna method*), 5

L

load_checkpoint() (*varuna.Varuna method*), 5

S

set_optimizer() (*varuna.Varuna method*), 4
step() (*varuna.Varuna method*), 4

V

Varuna (*class in varuna*), 3