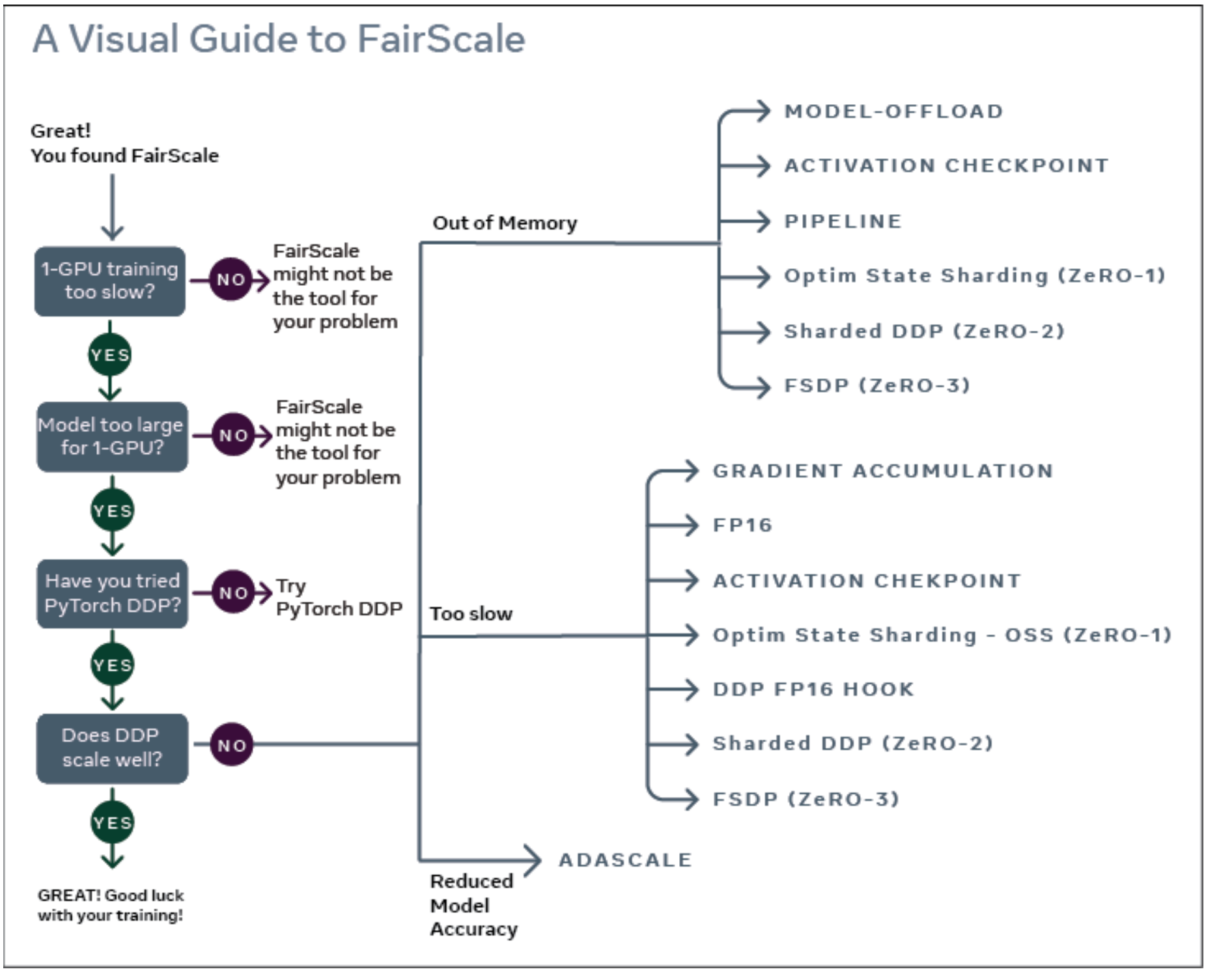
# USER WORKFLOW

User workflow Diagram with explanation of various decision points




Previous

Next

# EFFICIENT MEMORY MANAGEMENT

FairScale provides implementations inspired by the ZeRO class of algorithms in the form of modular APIs that you can plug into your model training. Zero Redundancy Optimizer is a class of algorithms that aim to tackle the tradeoff between using Data Parallel training and Model Parallel training. When using Data Parallel training, you tradeoff memory for computation/communication efficiency. On the other hand, when using Model Parallel training, you tradeoff computation/communication efficiency for memory. ZeRO attempts to solve this problem. Model training generally involves memory footprints that falls into two categories:

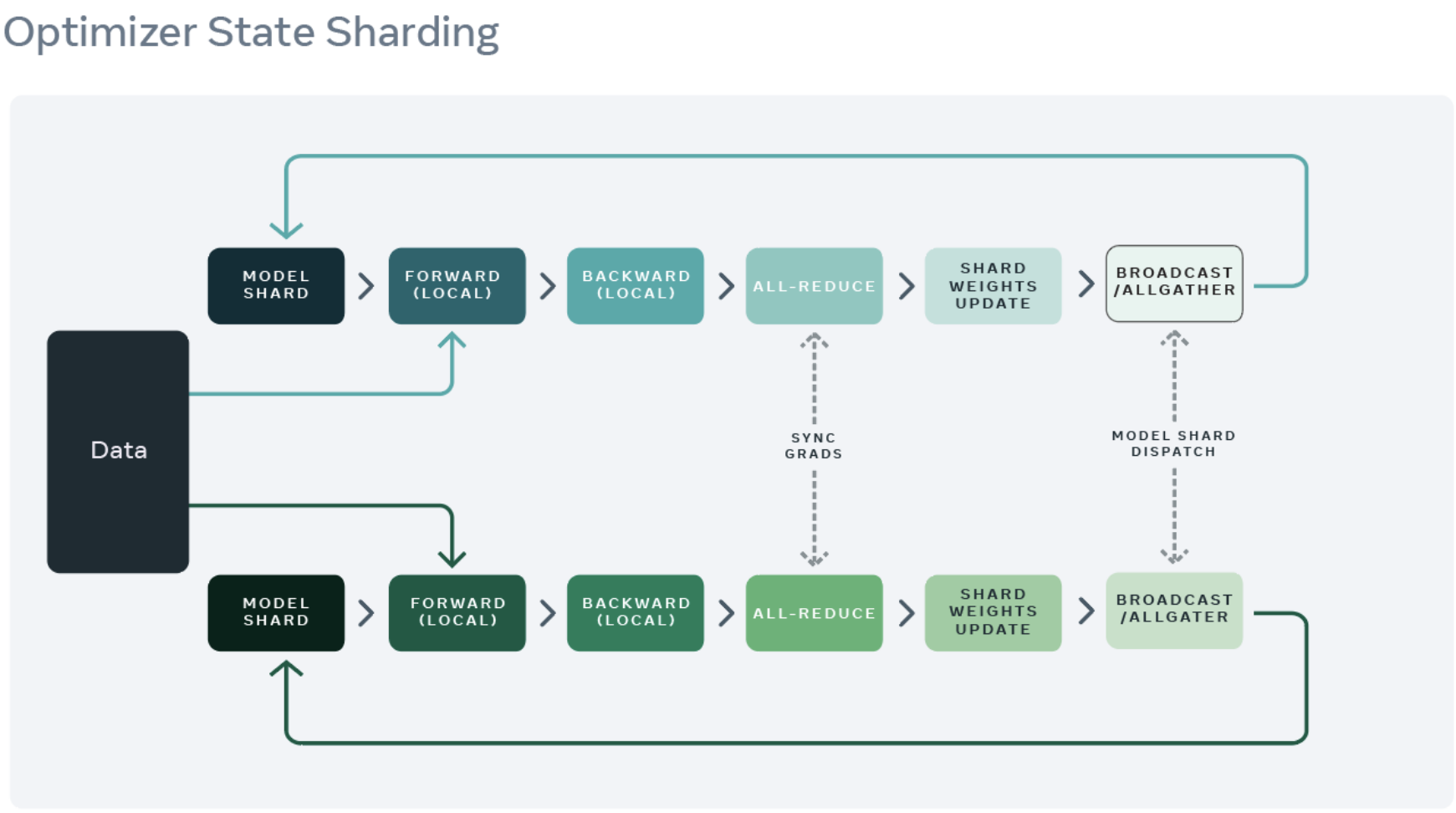
1. Model states - optimizer states, gradients, parameters
2. Residual states - activations, temp buffers, fragmented memory

To reduce redundancy in model states, three different algorithms were proposed. These have been implemented in FairScale as Optimizer State Sharding (OSS), Sharded Data Parallel (SDP) and finally Fully Sharded Data Parallel (FSDP). Let's dive deeper into the actual mechanics of each of these algorithms and understand why they provide the memory savings that they do.

## Optimizer State Sharding (OSS)

FairScale has implemented memory optimization related to optimizer memory (inspired by ZeRO-1) footprint using *fairscale.optim.OSS* API. Optimizers such as Adam usually require maintaining momentum, variance, parameters and gradients all in FP32 precision even though training can be carried out with parameters and gradients in FP16 precision. When each of the ranks update the full model, this means that a sizable part of the memory is occupied by redundant representations of the optimizer state.

To overcome this redundancy, optimizer state sharding entails partitioning the model optimization step in between the different ranks, so that each of them is only in charge of updating a unique shard of the model. This in turn makes sure that the optimizer state is a lot smaller on each rank, and that it contains no redundant information across ranks.




The training process can be modified from that carried out by DDP as follows:

1. The wrapped optimizer shards the optimizer state in a greedy fashion based on the parameter size but not the order in which it is used. This is to ensure that each rank has almost the same optimizer memory footprint.

2. The training process is similar to that used by PyTorch's Distributed Data Parallel (DDP). The forward pass completes on each of the ranks followed by the backward pass. During the backward pass, gradients are synchronized using allreduce.

3. Each rank updates the parameters for the shard of optimizer state that it is responsible for and then discards the rest.

4. After update, a broadcast or allgather follows to ensure all ranks receive the latest updated parameter values.

OSS is very useful when you are using an optimizer such as Adam that has additional state. The wrapping of the optimizer is a one-line non intrusive change that provides memory savings.

If you are using SGD or any optimizer with a limited memory footprint, it is likely that you will see a slowdown when using multiple nodes, due to the additional communication in step 4. There is also some wasteful memory used to store gradients during allreduce in step 2 that is then discarded, although this also happens with normal PyTorch (nothing extraneous here).

## Best practices for using *fairscale.optim.oss*

1. OSS exposes a broadcast_fp16 flag that you should probably use in multi-node jobs, unless this leads to accuracy issues (which is very unlikely). This can be used with or without Torch AMP. This is usually not needed in a single node experiment.

2. If your model is extremely unbalanced in terms of size (one giant tensor for instance), then this method will not be very helpful, and tensor sharding options such as *fairscale.nn.FullyShardedDataParallel* would be preferable.

3. OSS should be a drop in solution in a DDP context, and stays compatible with most of the DDP features such as the fp16 gradient compression hook, gradient accumulation and PyTorch AMP.

## Performance tips for *fairscale.optim.oss*

1. On a single node, OSS should be always faster than vanilla PyTorch, memory savings will vary depending on the optimizer being used

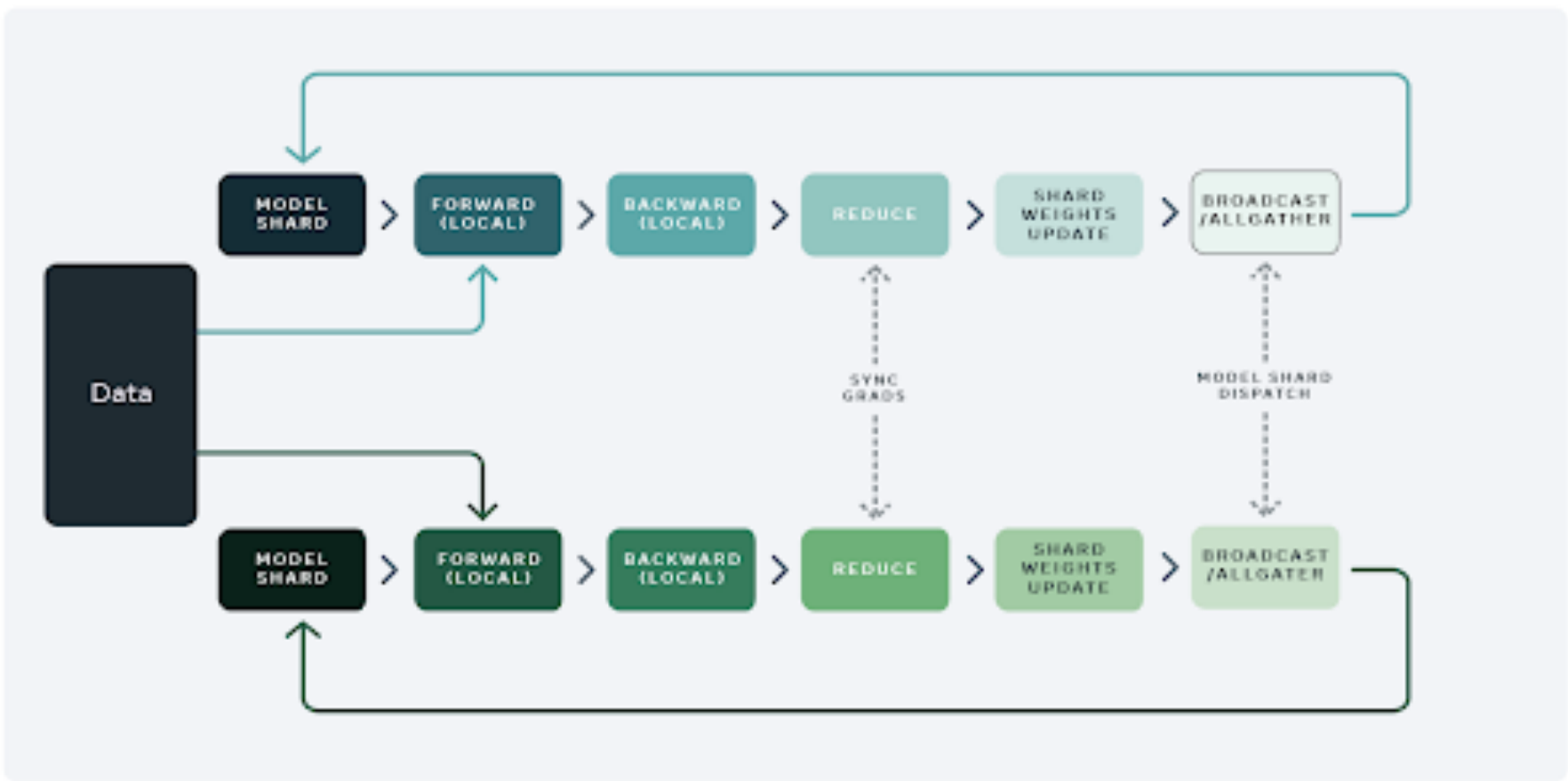
2. When using multiple nodes, OSS can alternatively be faster or slower than vanilla PyTorch, depending on the optimizer being used, and optional flags (E.g broadcast_fp16, gradient compression, gradient accumulation as mentioned above.)

3. If applicable (if your experiment can do with a bigger batch size), it's usually beneficial to reinvest the saved memory in a larger batch size and reduce the number of ranks involved, or to use gradient accumulation since this diminishes the communication cost.

# Optimizer + Gradient State Sharding

To overcome redundant gradient memory and to enable further memory savings, gradient sharding or ZeRO-2 was proposed. This has been implemented by the Sharded Data Parallel(SDP) API in FairScale. While OSS solved the redundancy problem in optimizers, the above data parallel training steps revealed a duplication of computation of gradient aggregation as well as additional memory being used for gradients are discarded.

To enable gradient sharding, each rank is assigned a set of parameters for which they are responsible for managing optimizer state as well as gradient aggregation. By assigning a model shard to a given rank we ensure that gradients are reduced to specific ranks that are in turn responsible for the update. This reduces communication as well as memory usage.




The training process is as follows:

1. As before the wrapped optimizer shards parameters across the different ranks.
2. The model is now wrapped with a Sharded Data Parallel (SDP) wrapper that allows us to add the appropriate hooks and maintain state during the training process.
3. SDP focuses on trainable parameters and adds a backward hook for each of the them.
4. During the backward pass, gradients are reduced to the rank that they are assigned to as part of the sharding process in 1. Instead of an allreduce op, a reduce op is used which reduces the communication overhead.
5. Each rank updates the parameters that they are responsible for.
6. After the update, a broadcast or allgather follows to ensure all ranks receive the latest updated parameter values.

Both the OSS and SDP APIs allow you to reduce the memory used for gradients and optimizer states. Additional communication costs can be present in slow interconnects but are useful to try as first steps when running into Out Of Memory (OOM) issues.

## Best practices for *fairscale.nn.ShardedDataParallel*

1. If using multiple nodes, make sure that SDP is using reduce buffers by specifying the *reduce_buffer_size* arg. Changing their size can be an optimization target, the best configuration could depend on the interconnect.

2. If on a single node, it's usually best not to use *reduce_buffer_size* since there is a latency cost associated with it but no memory gain. Setting this value to 0 means that this feature is not used and this is the recommended single node setting.

3. If applicable (if your experiment can do with a bigger batch size), it's usually beneficial to reinvest the saved memory in a larger batch size and reduce the number of ranks involved, or to use gradient accumulation since this diminishes the communication cost.

# Optimizer + Gradient + Horizontal Model Sharding

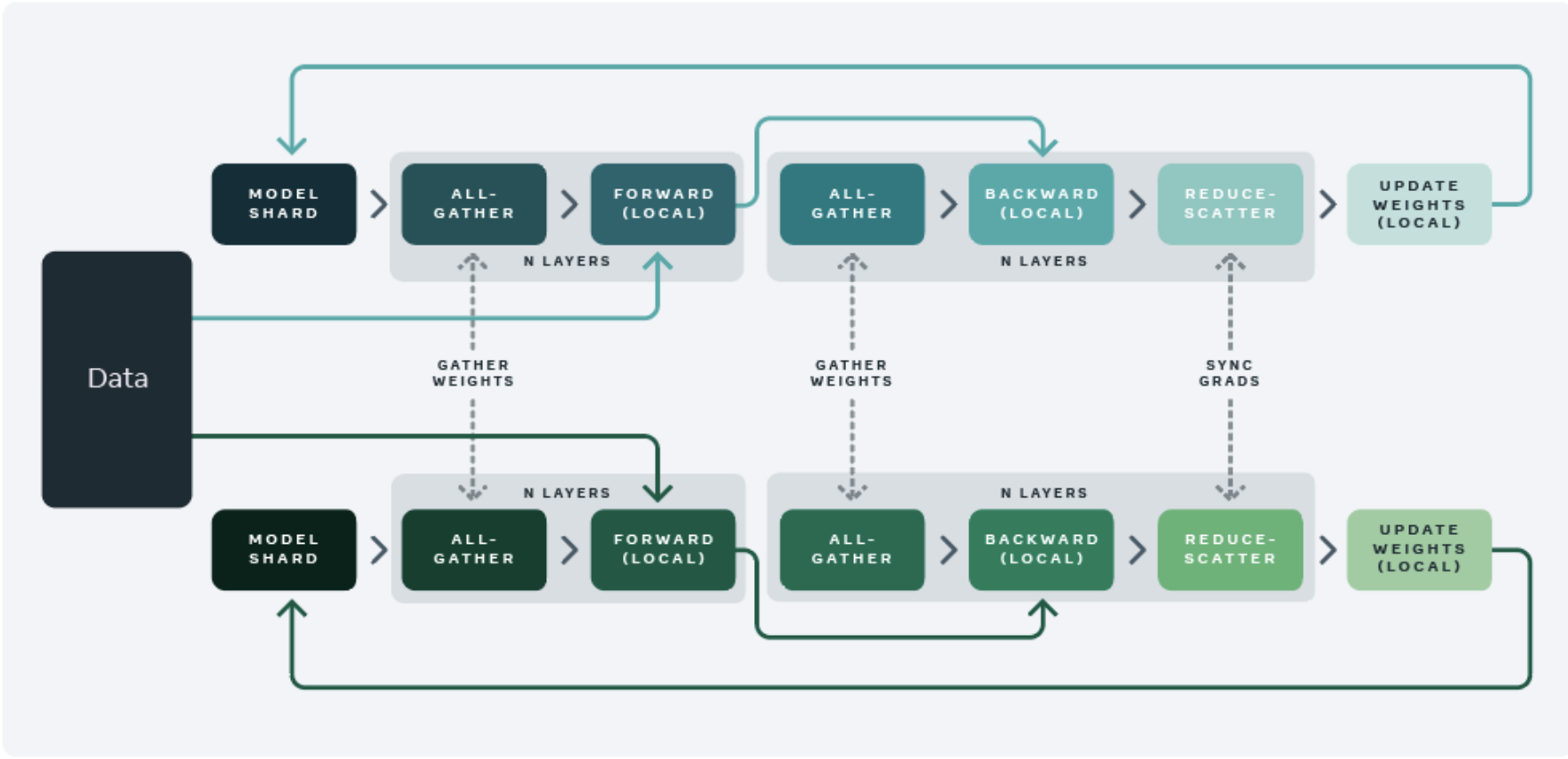
To further optimize training and achieve greater memory savings, we need to enable parameter sharding. With parameter sharding similar to gradient and optimizer states, data parallel ranks are responsible for a shard of the model parameters. FairScale implements parameter sharding by way of the Fully Sharded Data Parallel (FSDP) API which is heavily inspired by ZeRO-3. Parameter sharding is possible because of two key insights:

1. The allreduce operation can be broken up into reduce and allgather similar to the previous sharding technologies (optimizer state and gradient).

2. Individual layers can be wrapped with the FSDP API that allows us to bring in all the parameters required for a single layer onto a given GPU at a given instance, compute the forward pass and then discard the parameters not owned by that rank. Please see the tutorial section for how you can use autowrap to enable wrapping individual layers of your model.

The training process is as follows:

## Fully sharded data parallel training




1. *allgather* the parameters required for the forward pass of each of the layers of the model just before the compute of a specific layer commences.
2. Compute the forward pass.
3. *allgather* the parameters required for the backward pass of each of the layers of the model just before the backward pass of a specific layer commences.
4. Compute the backward pass.
5. *reduce* the gradients such that aggregated grads are accumulated on the ranks that are responsible for the corresponding parameters.
6. Let each rank update the parameters that have been assigned to it using the aggregated gradients.

With FSDP there are small changes one needs to make when using APIs for checkpointing and saving optimizer state. Given the sharded nature of optimizer state and parameters, any API that aims to save the model state for training or inference needs to account for saving weights from all workers. FSDP implements the required plumbing to save weights from all workers, save weights on individual workers and save optimizer state from all workers.

FSDP also supports mixed precision training where both the computation and communication are carried out in FP16 precision. If you want to reduce operations to be carried out in FP32 which is the default behavior of DDP, then you must set *fp32_reduce_scatter=True*.

To enable further memory savings, FSDP supports offloading parameters and gradients that are currently not being used onto the CPU. This can be enabled by setting *move_params_to_cpu* and *move_grads_to_cpu* to be equal to True.

### Best practices for *fairscale.nn.FullyShardedDataParallel*

1. For FSDP, it is preferable to use *model.zero_grad(set_to_none=True)* since it saves a large amount of memory after stepping.

2. *torch.cuda.amp.autocast for mixed precision* is fully compatible with FSDP. However you will need to set the *mixed_precision* arg to be True.

3. If combined with activation checkpointing, it is preferable to use FSDP(checkpoint_wrapper(module)) over checkpoint_wrapper(FSDP(module)). The latter will result in more communication and will be slower.

4. Results should be identical to DDP with pointwise Optimizers, e.g., Adam, AdamW, Adadelta, Adamax, SGD, etc.. However, the sharding will result in slightly different results when using non-pointwise Optimizers, e.g., Adagrad, Adafactor, LAMB, etc.

### Performance tips for *fairscale.nn.FullyShardedDataParallel*

1. For best memory efficiency use auto_wrap to wrap each layer in your network with FSDP and set *reshard_after_forward* to be True

2. For best training speed set *reshard_after_forward* to be False (wrapping each layer is not required, but will improve speed further)

‹ Previous

Next ›

Builds

Downloads

**On GitHub**

View

Edit

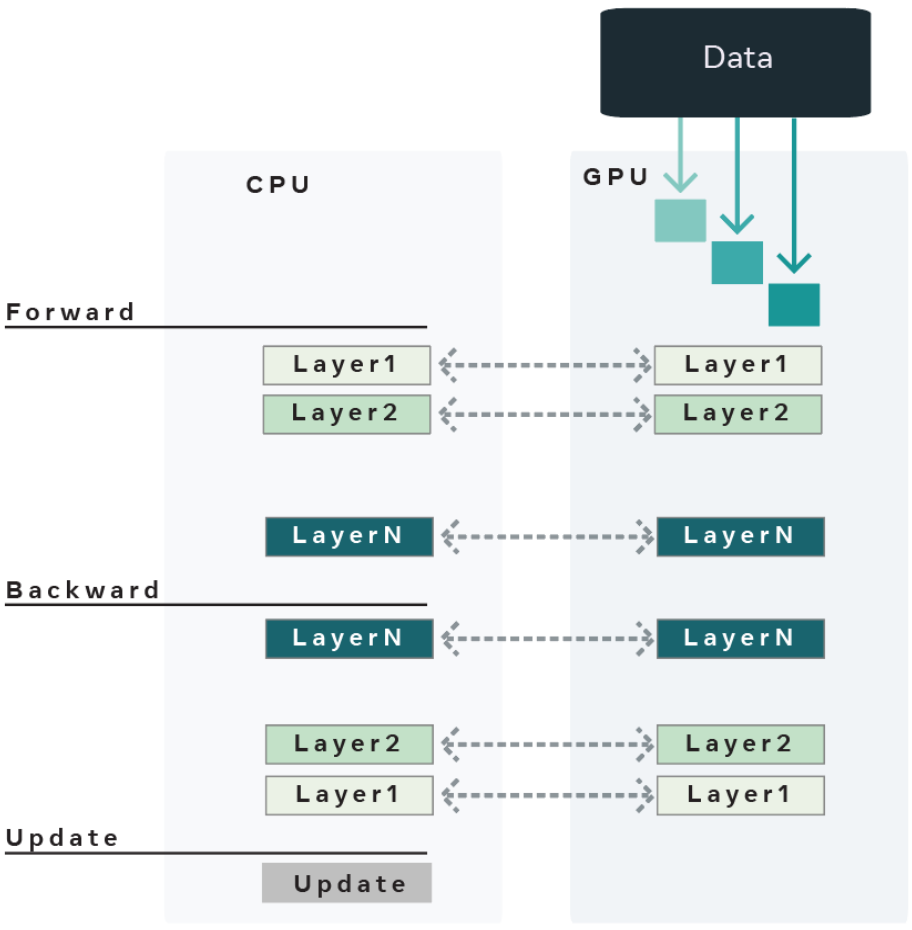
**Search**

Search docs

---

# OFFLOADMODEL

Heavily inspired by the Layer-to-Layer algorithm and Zero-Offload, OffloadModel uses the CPU to store the entire model, optimizer state and gradients. OffloadModel then brings in a layer (or a number of layers) onto the GPU for training at a time during the forward and backward pass. The intermediate activations for the layer boundaries are also stored on the CPU and copied to the GPU as needed for the backward pass. Once the backward pass is completed all the parameters are updated with the gradients present on the CPU.




Offload uses the following techniques to enable large model training:

1. The model is assumed to be nn.Sequential and sharded (almost) equally based on the number of parameters into a list of nn.Modules. Each nn.Module now contains a fraction of the whole model which we shall refer to as model shards.

2. At each iteration, each of the model shards are copied from the CPU -> GPU, FW pass is computed using the minibatch of data and the model shard is copied back from GPU -> CPU. In the BW pass, the same process is repeated.

3. The optimizer remains on the CPU and gradients and parameters are all moved onto the CPU before running optimizer.step. This ensures that the CPU is responsible for updating the parameters and holding onto the optimizer state.

4. If activation checkpointing is enabled, we use torch.autograd.Function to disable graph construction in the FW pass and copy intermediate activations from GPU -> CPU after the FW pass of a given shard is complete. The reverse copy is carried out in the BW pass.

5. Microbatches are used to enable larger throughput and offset the cost of moving model parameters and activations from CPU <-> GPU. Micro-batches allow you to specify large mini-batches which are broken down into micro-batches and fed to the model shards at each iteration. In short it is a way to allow more computation at a given time on a model shard to offset the cost of copying from CPU <-> GPU.

## Best practices for using *fairscale.experimental.nn.OffloadModel*

1. Using OffloadModel to train large models can result in loss of throughput which can be overcome by using activation checkpointing and microbatches.
2. OffloadModel currently only works for *nn.Sequential* models.

‹ Previous

Next ›

# ADASCALE

Adascale is a technique used to enable large batch training that allows you to increase batch size without loss of accuracy. When increasing batch size with the number of devices, the learning rate is typically tuned based on the batch size. With Adascale, users no longer need to modify the learning rate schedule and still achieve the desired accuracy. Adascale has been implemented as the Adascale API in FairScale. This technique typically works well for SGD (with and without momentum) The assumption is that you already have a good learning rate schedule that works well for small batch sizes. (AdaScale has not been validated to work effectively with Adam, further research in that direction is needed.)

AdaScale adapts the learning rate schedule and determines when to stop based on comparing statistics of large-batch gradients with those of small-batch gradients. Small batch gradients are gradients that have been computed on each GPU and large batch gradients are the average of gradients computed on N such GPUs. Adascale uses the concept of gain ratio which is intuitively a measure of how much the variance has reduced by averaging N small batch gradients. It is a quantity between 1 and N. In practice, the implementation tracks estimates of the gradient variance and norm-squared which are smoothed using an exponentially-weighted moving average. If T is the number of steps used to train the original small batch size before scaling, Adascale stops training once the accumulated gain ratio is greater than T. As you use more and more GPUs in the training the total steps needed to train decreases, but due to the value of gain ratio between [1, N], the total steps does not linearly decrease as you increase the GPUs. Additional training steps are taken to maintain the model accuracy, when compared with original_total_step/N (i.e. linear scaling). In other words, whenever the gain ratio is less than N, we could not take as large a step as we may have hoped for, and so the total number of iterations ends up being larger than T / N. The current implementation in FairScale supports gradient accumulation training, can be used with Optimizer State Sharding (OSS), and works with PyTorch LR scheduler classes.

The training process is as follows:

1. Compute the forward pass
2. During the backward pass, hooks attached to each of the parameters fire before the allreduce operation. This is to enable us to calculate the accumulated squares of the local gradients.
3. A final backward hook fires after all the gradients have been reduced using the allreduce op. Using the global gradient square and the accumulated local gradient square, the gradient square average and gradient variance average is calculated.
4. These values are then used to calculate the gain ratio. During the *step* call of the optimizer, the learning rate is updated using this gain ratio value.
5. The training loop terminates once maximum number of steps has been reached

## Best practices for *fairscale.optim.AdaScale*

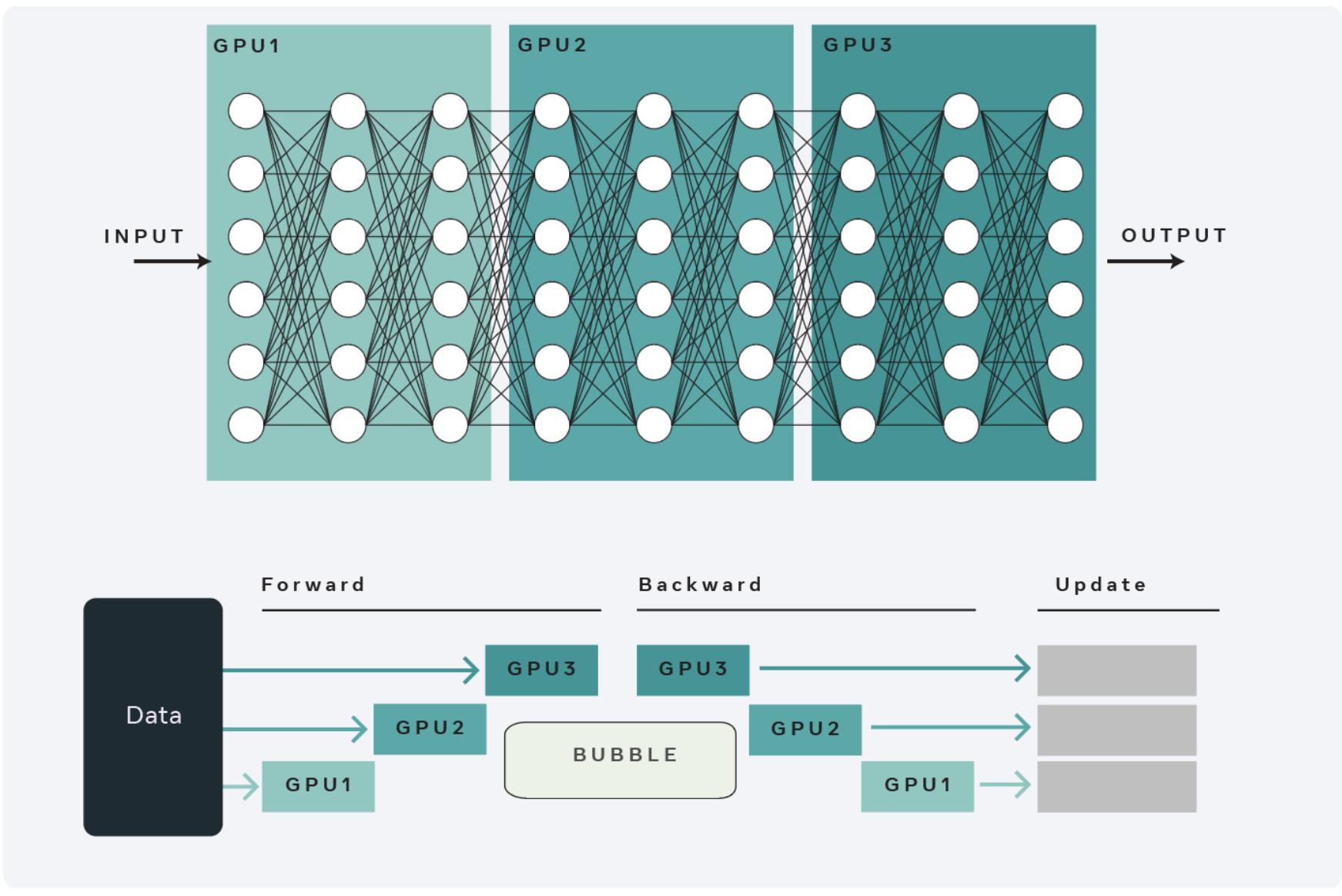
Adascale only works for the SGD optimizer (with and without momentum)

‹ Previous

Next ›

# PIPELINE PARALLELISM

Training large models can lead to out-of-memory when the size of the model is too large for a single GPU. To train such a large model, layers can be pipelined across different GPU devices as described in GPipe. The *fairscale.nn.Pipe* is an implementation of GPipe which has been adopted from torchgpipe. This API has also been upstreamed to PyTorch in the 1.8 release with the experimental tag.




Gpipe first shards the model across different devices where each device hosts a shard of the model. A shard can be a single layer or a series of layers. However Gpipe splits a mini-batch of data into micro-batches and feeds it to the device hosting the first shard. The layers on each device process the micro-batches and send the output to the following shard/device. In the meantime it is ready to process the micro batch from the previous shard/device. By pipepling the input in this way, Gpipe is able to reduce the idle time of devices.

## Best practices for using *fairscale.nn.Pipe*

1. Choice of size of micro-batches can affect GPU utilization. A smaller microbatch can reduce latency of shards waiting for previous shard outputs but a large microbatch better utilizes GPUs.
2. Sharding the model can also impact GPU utilization where layers with heavier computation can slow down the shards downstream.

‹ Previous

Next ›

# SLOWMO DISTRIBUTED DATA PARALLEL

Training neural networks in a distributed data-parallel manner results in non-linear scaling (slowdown) due to the time spent on communication between the different nodes (as well as, to a lesser extent though, synchronization between the different nodes). So, a distributed training run with 8 nodes is not 8x faster than a run with 1 node as we would expect it to be.

SlowMo Distributed Data Parallel aims to solve this by replacing the typical exact allreduce between gradients with an approximate averaging of parameters. This approximate averaging reduces both the time spent on communication as well as the synchronization between different nodes. It uses one of the following two algorithms (configurable) as a base algorithm for this purpose:

- Local SGD (papers #1 and #2). This algorithm does an allreduce of the parameters every few iterations.
- Stochastic Gradient Push (SGP). This algorithm involves one-to-one communications between nodes.

These base algorithms (LocalSGD and SGP), when used only by themselves, result in reduced model quality (measured as accuracy in a classification setting). The SlowMo algorithm alleviates this issue by doing a slow momentum step, typically, every 48 iterations.

The training process with SlowMo looks as follows:

1. Compute the forward pass.
2. Compute the backward pass.
3. During the backward pass, using a backward hook, on each node, the gradients are synchronized using allreduce across the different GPUs on that node.
4. Perform the `optimizer.step()` to update parameters on each node with the gradients of that node.
5. Approximately average the parameters using a base algorithm - one of LocalSGD or SGP (both are described above).
6. Perform the slow momentum update step once every `slowmo_frequency` (typically 48) iterations. In this step, the parameters on different nodes are (exactly) averaged, followed by a `slowmo_optimizer.step()`. Note that this `slowmo_optimizer` is different from the original optimizer, and it is done in a Zero-1 like manner to save memory.

## Best practices for using `SlowMoDistributedDataParallel`

1. SlowMo will be useful in deep learning workloads which run on more than 2 nodes in clusters with a slow interconnect, eg Ethernet.
2. SlowMo should be useful in your workload if the following condition holds:
   $\text{time\_taken\_for\_all\_reduce\_of\_gradients} \times (1 - \frac{1}{\text{localsgd\_frequency}}) > \text{time\_taken\_for\_backward\_pass}$
   Notes:
   - In case you are using SGP as the base algorithm, the value of `localsgd_frequency` can be plugged in as 2.
   - The formula above is a simplified version of:
     $\text{time\_taken\_for\_all\_reduce\_of\_gradients} > \text{time\_taken\_for\_backward\_pass} + \frac{\text{time\_taken\_for\_all\_reduce\_of\_gradients}}{\text{localsgd\_frequency}}$ The left and right hand sides denote the total backward duration (combining the computation of gradients in the backward pass and the communication cost) for DDP and SlowMo DDP, respectively. Since DDP overlaps the computation of gradients with their communication, it is bottlenecked by the latter. In contrast, there is an extra `time_taken_for_backward_pass` on the right hand side because we do not overlap the backward pass with communication in the current implementation of SlowMo.
   - In clusters with slower interconnect, `time_taken_for_all_reduce_of_gradients` will go up, leading to SlowMo being more useful. `localsgd_frequency` is also an important factor here. More details on varying that to affect performance are in tip 2 of Performance tips for SlowMoDistributedDataParallel.
3. `slowmo_momentum` will need to be tuned for obtaining good model quality. A grid search across {0.0, 0.1, 0.2, 0.4, 0.6} should be good enough for tuning. This `slowmo_momentum` value holds consistent across multiple runs with similar settings. When the number of nodes used is increased, however, a higher value of `slow_momentum` should be needed. More details about this can be found in the documentation.
4. Adding SlowMo to existing Distributed Data Parallel code involves two steps, which can be found in the tutorial.

## Performance tips for `SlowMoDistributedDataParallel`

1. `nprocs_per_node` should be set to the number of GPUs on a node (this number should be the same on each node). This allows the API to exploit the fast interconnect between different GPUs on a node.
2. Increasing the `localsgd_frequency` results in an increase in speed. However, it comes with a tradeoff of reducing the model quality. We recommend keeping the `localsgd_frequency` at 3.
3. `slowmo_memory_efficient` should typically be used (this is the default behavior). It reduces memory usage by sharding the additional slow momentum optimizer's parameters in a Zero-1 like manner.
4. A call to `model.zero_grad(set_to_none=True)` should be made after `optimizer.step()` in order to save memory for the `model.perform_slowmo()` step. More details about this can be found in the documentation for perform_slowmo().

# ENHANCED ACTIVATION CHECKPOINTING

Activation checkpointing is a technique used to reduce GPU memory usage during training. This is done by avoiding the need to store intermediate activation tensors during the forward pass. Instead, the forward pass is recomputed by keeping track of the original input during the backward pass. There is a slight increase in computation cost (about 33%) but this reduces the need to store large activation tensors which allows us to increase the batch size and thereby the net throughput of the model.

Activation checkpointing is implemented by overriding *torch.autograd.Function*. In the *forward* function which handles the forward pass of the module, using *no_grad*, we can prevent the creation of the forward graph and materialization of intermediate activation tensors for a long period of time (i.e till the backward pass). Instead, during the backward pass, the forward pass is executed again followed by the backward pass. The inputs to the forward pass are saved using a context object that is then accessed in the backward pass to retrieve the original inputs. We also save the Random Number Generator(RNG) state for the forward and backward passes as required for Dropout layers.

The above functionality is already implemented as part of the *torch.utils.checkpoint.checkpoint_wrapper* API whereby different modules in the forward pass can be wrapped. The wrapper in FairScale offers functionality beyond that provided by the PyTorch API specifically you can use *fairscale.nn.checkpoint.checkpoint_wrapper* to wrap a *nn.Module*, handle kwargs in the forward pass, offload intermediate activations to the CPU and handle non-tensor outputs returned from the forward function.

## Best practices for *fairscale.nn.checkpoint.checkpoint_wrapper*

1. Memory savings depends entirely on the model and the segmentation of checkpoint wrapping. Each backprop consists of several mini-forward and backprop passes. The gain is entirely dependent on the memory footprint of the layer's activations.

2. When using BatchNormalization you may need to freeze the calculation of statistics since we run the forward pass twice.

3. Ensure that the input tensor's *requires_grad* field is set to True. In order to trigger the backward function, the output needs to have this field set. By setting it on the input tensor we ensure that this is propagated to the output and the *backward* function is triggered.

‹ Previous

Next ›