horovod (2) --- 从使用者角度切入

0x01 Horovod 简介

Horovod 是Uber于2017年发布的一个易于使用的高性能的分布式训练框架，支持TensorFlow，Keras，PyTorch和MXNet。Horovod 的名字来自于俄国传统民间舞蹈，舞者手牵手围成一个圈跳舞，与分布式 TensorFlow 流程使用 Horovod 互相通信的场景很像。

因为各个机器学习框架对于底层集合通信库（ nccl，openmpi，gloo 等等）的利用水平可能各不相同，使得他们无法充分利用这些底层集合通信库的威力。因而，hovorod 就整合这些框架，提供一个易用高效的解决方案。

Uber的工程师就是根据FaceBook的一篇paper："Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour"和百度的一篇"Bringing HPC Techniques to Deep Learning" 改进并发布了开源框架Horovod。

Horovod 相比于百度的工作，并无学术上的贡献。但是 Horovod 扎实的工程实现，使得它受到了更多的关注。它最大的优势在于对 RingAllReduce 进行了更高层次的抽象，使其支持多种不同的框架。同时引入了 Nvidia NCCL，对 GPU 更加友好。

Horovod依赖于Nvidia的 NCCL2 做 All Reduce，依赖于MPI做进程间通信，简化了同步多 GPU 或多节点分布式训练的开发流程。由于使用了NCCL2，Horovod也可以利用以下功能：NVLINK，RDMA，GPUDirectRDMA，自动检测通信拓扑，能够回退到 PCIe 和 TCP/IP 通信。

我们需要几个问题来引导分析：

- Hovorod 怎么进行数据分割？
- Hovorod 怎么进行训练代码分发？
- Hovorod 启动时候，python 和 C++ 都做了什么？
- 如何确保 Hovorod 启动时候步骤一致；

0x02 Hovorod 机制概述

2.1 Horovod 机制

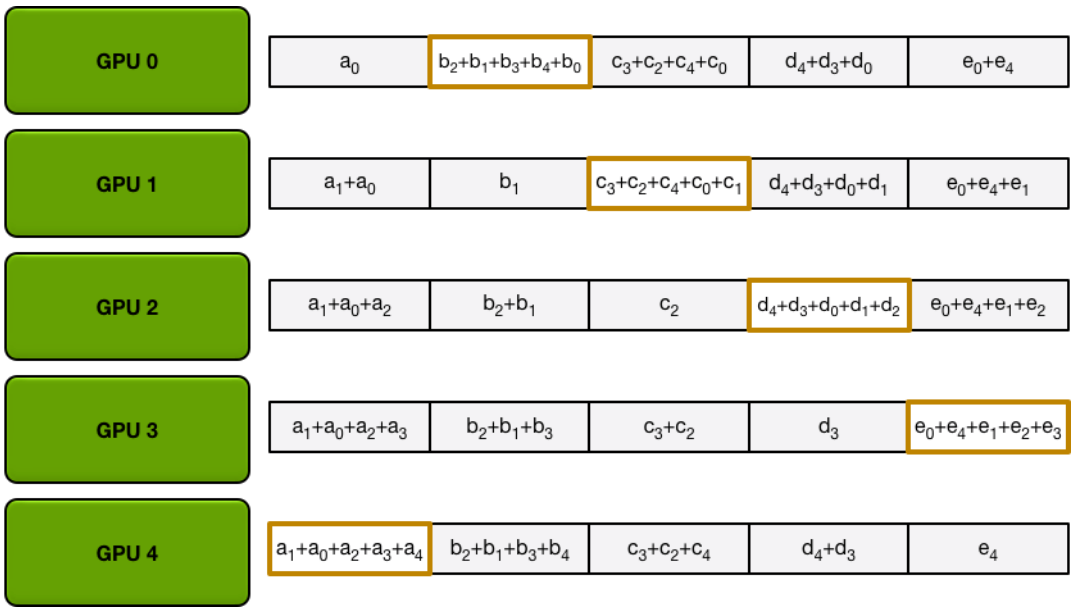
Horovod使用**数据并行化**策略在GPU上分配训练。

在数据并行化中，作业中的每个GPU都会接收其自己的数据批处理的独立切片，即它的"批处理切片"。每个GPU都使用自己分配到的数据来独立计算，进行梯度更新。

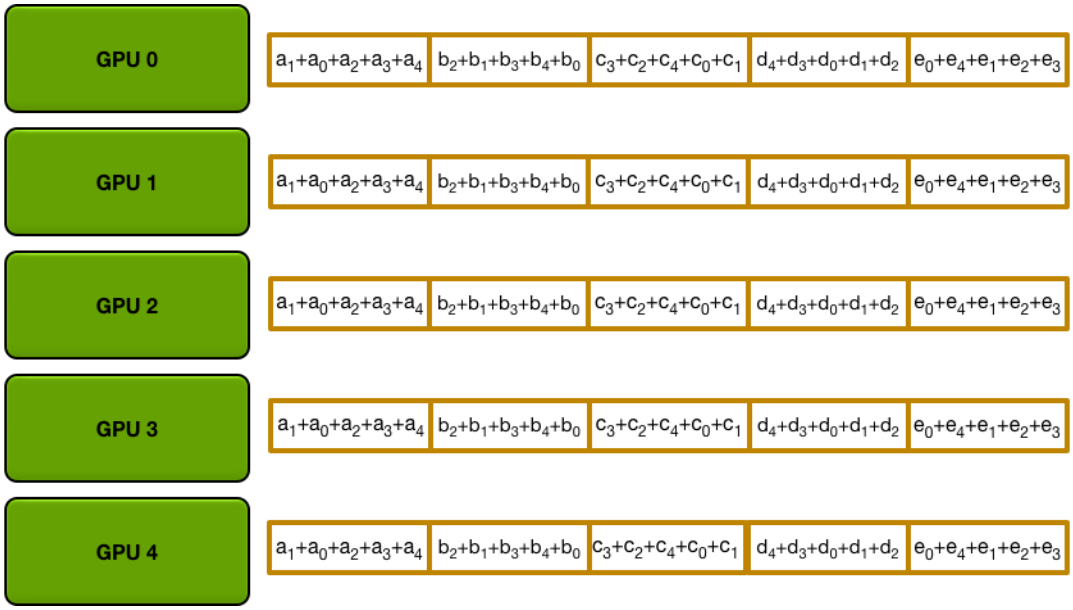
假如使用两个GPU，批处理大小为32，则第一个GPU将处理前16条记录的正向传播和向后传播，以及第二个GPU处理后16条记录的正向传播和向后传播。然后，这些梯度更新将在GPU之间平均在一起，最后应用于模型。

每一个迭代的操作方法如下：

1. 每个 worker 将维护自己的模型权重副本和自己的数据集副本。
2. 收到执行信号后，每个工作进程都会从数据集中提取一个不相交的批次，并计算该批次的梯度。
3. Workers 使用ring all-reduce算法来同步彼此的梯度，从而在本地所有节点上计算同样的平均梯度。
    1. 将每个设备上的梯度 tensor 切分成长度大致相等的 num_devices 个分片，后续每一次通信都将给下一个邻居发送一个自己的分片（同时从上一个邻居接受一个新分片）。
    2. ScatterReduce 阶段：通过 num_devices - 1 轮通信和相加，在每个 device 上都计算出**一个** tensor 分片的和，即每个 device 将有一个块，其中包含所有device 中该块中所有值的总和；具体如下：



3. AllGather 阶段：通过 num_devices - 1 轮通信和覆盖，将上个阶段计算出的每个 tensor 分片的和 广播到其他 device；最终所有节点都拥有**所有**tensor分片和。具体如下：



4. 在每个设备上合并分片，得到梯度和，然后除以 num_devices，得到平均梯度；

4. 每个 worker 将 梯度更新 应用于其模型的本地副本。

5. 执行下一个batch。

0x03 示例代码

3.1 摘要代码

我们此处给出官网示例代码部分摘要，具体分析参见下面代码中的注释。

```
import tensorflow as tf
import horovod.tensorflow.keras as hvd

# Horovod: initialize Horovod.
hvd.init() # 初始化 Horovod，启动相关线程和MPI线程

# Horovod: pin GPU to be used to process local rank (one GPU per process)
# 依据 local rank 为不同的进程分配不同的GPU
gpus = tf.config.experimental.list_physical_devices('GPU')
```

```
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
if gpus:
    tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')

(mnist_images, mnist_labels), _ = \
    tf.keras.datasets.mnist.load_data(path='mnist-%d.npz' % hvd.rank())

# 切分数据
dataset = tf.data.Dataset.from_tensor_slices(
    (tf.cast(mnist_images[..., tf.newaxis] / 255.0, tf.float32),
             tf.cast(mnist_labels, tf.int64))
)
dataset = dataset.repeat().shuffle(10000).batch(128)

mnist_model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, [3, 3], activation='relu'),
    ......
    tf.keras.layers.Dense(10, activation='softmax')
])

# Horovod: adjust learning rate based on number of GPUs.
scaled_lr = 0.001 * hvd.size() # 根据Worker的数量增加学习率的大小
opt = tf.optimizers.Adam(scaled_lr)

# Horovod: add Horovod DistributedOptimizer.
# 把常规TensorFlow Optimizer通过Horovod包装起来，进而使用 ring-allreduce 来得到平均梯度
opt = hvd.DistributedOptimizer(
    opt, backward_passes_per_step=1, average_aggregated_gradients=True)

# Horovod: Specify `experimental_run_tf_function=False` to ensure TensorFlow
# uses hvd.DistributedOptimizer() to compute gradients.
mnist_model.compile(loss=tf.losses.SparseCategoricalCrossentropy(),
                    optimizer=opt, metrics=['accuracy'],
                    experimental_run_tf_function=False)

callbacks = [
    hvd.callbacks.BroadcastGlobalVariablesCallback(0), # 广播初始化，将模型的参数从第一个设备传向其他设备，以保证初始化模型参数的一致性
    hvd.callbacks.MetricAverageCallback(),
    hvd.callbacks.LearningRateWarmupCallback(initial_lr=scaled_lr,
warmup_epochs=3, verbose=1),
]

# Horovod: save checkpoints only on worker 0 to prevent other workers from
corrupting them. # 只有设备0需要保存模型参数作为checkpoint
if hvd.rank() == 0:
    callbacks.append(tf.keras.callbacks.ModelCheckpoint('./checkpoint-
{epoch}.h5'))

# Horovod: write logs on worker 0.
verbose = 1 if hvd.rank() == 0 else 0

# Train the model.
# Horovod: adjust number of steps based on number of GPUs.
mnist_model.fit(dataset, steps_per_epoch=500 // hvd.size(), callbacks=callbacks,
epochs=24, verbose=verbose)
```

3.2 horovodrun

Horovod训练脚本 **未** 作为Python脚本启动。 例如，您不能使用 `python train.py` 运行此培训脚本。需要采用特殊的CLI命令 `horovodrun` 来启动（训练代码 train.py 需要手动拷贝到各个节点上，且目录相同）：

```
$ horovodrun -np 4 -H localhost:4 python train.py
```

0x04 运行逻辑

我们按照顺序梳理，看看在程序初始化过程背后都做了什么。

4.1 引入python文件

如下代码会引入各种相关python文件。

```
import tensorflow as tf
import horovod.tensorflow.keras as hvd
```

4.2 初始化 in python

python 世界的初始化位于 horovod-master/horovod/mxnet/mpi_ops.py

4.2.1 引入SO库

4.2.1.1 SO库

horovod/tensorflow/mpi_ops.py 之中会引入SO库。

比如 dist-packages/horovod/tensorflow/mpi_lib.cpython-36m-x86_64-linux-gnu.so。

SO库 就是 horovod 中 C++ 代码编译出来的结果。

```
def _load_library(name):
    """Loads a .so file containing the specified operators.
    """
    filename = resource_loader.get_path_to_datafile(name)
    library = load_library.load_op_library(filename)
    return library

# Check possible symbol not found error from tensorflow version mismatch
try:
    MPI_LIB = _load_library('mpi_lib' + get_ext_suffix())
except Exception as e:
    check_installed_version('tensorflow', tf.__version__, e)
    raise e
else:
    check_installed_version('tensorflow', tf.__version__)
```

4.2.2.2 SO作用

引入库的作用是获取到 C++ 的函数，并且用 python 封装一下，这样就可以在 python 世界使用 C++代码了。

由下文可以看出来，python 的 _allreduce 函数就会把功能转发给 C++，由 MPI_LIB.horovod_allreduce 完成。

```
def _allreduce(tensor, name=None, op=Sum, prescale_factor=1.0,
postscale_factor=1.0,
               ignore_name_scope=False):
    if name is None and not _executing_eagerly():
        name = 'HorovodAllreduce_%s' % _normalize_name(tensor.name)
    return MPI_LIB.horovod_allreduce(tensor, name=name, reduce_op=op,
                                     prescale_factor=prescale_factor,
                                     postscale_factor=postscale_factor,
                                     ignore_name_scope=ignore_name_scope)
```

4.2.2 初始化配置

我们摘录了主要部分，就是初始化 _HorovodBasics，然后从 _HorovodBasics 内获取各种函数，变量和配置，比如是否编译了mpi，gloo等等。

```
from horovod.common.basics import HorovodBasics as _HorovodBasics

_basics = _HorovodBasics(__file__, 'mpi_lib')

# import basic methods
init = _basics.init
size = _basics.size
local_size = _basics.local_size
rank = _basics.rank
local_rank = _basics.local_rank
mpi_built = _basics.mpi_built
gloo_enabled = _basics.gloo_enabled
......
```

4.2.3 hvd.init() 初始化

首先需要用 hvd.init() 来初始化，`horovod` 管理的所有状态都会传到 hvd 对象中。

```
# Horovod: initialize Horovod.
hvd.init()
```

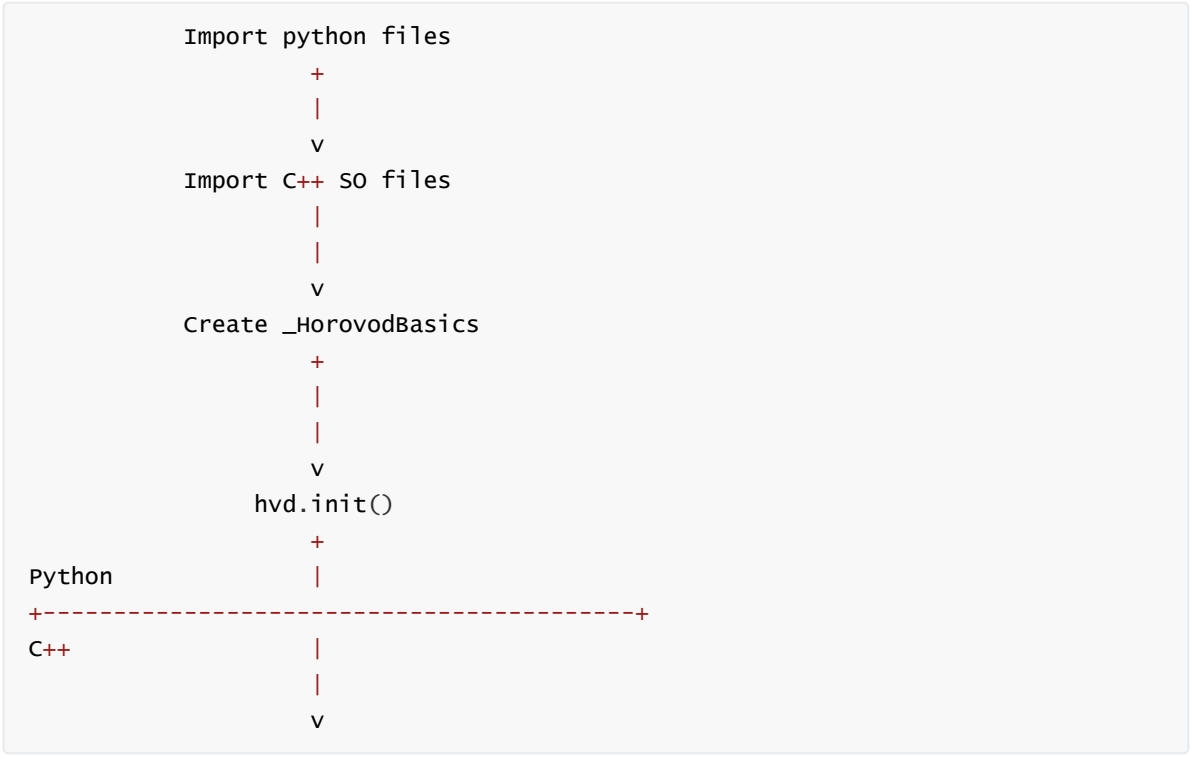
此处调用的是 HorovodBasics 中的函数，我们看看做了什么。

可以看到，这部分会一直深入到 C++世界，调用了大量的 MPI_LIB_CTYPES 函数，所以我们接下来就要进入到 C++的世界看看。

```
def init(self, comm=None):
    """A function that initializes Horovod.
    """
    atexit.register(self.shutdown)

    if not isinstance(comm, list):
        mpi_built = self.MPI_LIB_CTYPES.horovod_mpi_built()

        from mpi4py import MPI
        if MPI._sizeof(MPI.Comm) == ctypes.sizeof(ctypes.c_int):
            MPI_Comm = ctypes.c_int
        else:
            MPI_Comm = ctypes.c_void_p
            self.MPI_LIB_CTYPES.horovod_init_comm.argtypes = [MPI_Comm]
```

```
        comm_obj = MPI_Comm.from_address(MPI._addressof(comm))
        self.MPI_LIB_CTYPES.horovod_init_comm(comm_obj)
    else:
        comm_size = len(comm)
        self.MPI_LIB_CTYPES.horovod_init(
            (ctypes.c_int * comm_size)(*comm), ctypes.c_int(comm_size))
```

目前逻辑如下图：



4.3 初始化 in C++

4.3.1 horovod_init_comm

在初始化的时候，Horovod 会：

- 调用 MPI_Comm_dup 获取一个 Communicator，这样就有了和 MPI 协调的基础。
- 然后调用 InitializeHorovodOnce。

```
void horovod_init_comm(MPI_Comm comm) {
  MPI_Comm_dup(comm, &mpi_context.mpi_comm);
  InitializeHorovodOnce(nullptr, 0);
}
```

4.3.2 InitializeHorovodOnce

InitializeHorovodOnce 是初始化的主要工作，主要是：

- 依据是否编译了 mpi 或者 gloo，对各自的 context 进行处理，为 globalstate 创建对应的 controller；
- 启动了后台线程 BackgroundThreadLoop 用来在各个worker之间协调；

```
void horovod_init(const int* ranks, int nranks) {
  InitializeHorovodOnce(ranks, nranks);
}
```

```
void InitializeHorovodOnce(const int* ranks, int nranks) {
  // Ensure background thread is only started once.
  if (!horovod_global.initialize_flag.test_and_set()) {
    horovod_global.control_operation = ParseControllerOpsFromEnv();
    horovod_global.cpu_operation = ParseCPUOpsFromEnv();

#if HAVE_MPI // 依据是否编译了MPI进行处理
    // Enable mpi is it's used either in cpu data transfer or controller
    if (horovod_global.cpu_operation == LibType::MPI ||
        horovod_global.control_operation == LibType::MPI) {
      mpi_context.Enable();
    }

    if (horovod_global.control_operation == LibType::MPI){
      // 创建一个 MPIController 对象
      horovod_global.controller.reset(new MPIController(
          horovod_global.response_cache,
          horovod_global.tensor_queue, horovod_global.timeline,
          horovod_global.parameter_manager, horovod_global.group_table,
          mpi_context));
      horovod_global.controller->SetRanks(ranks, nranks);
    }
#endif

#if HAVE_GLOO // 依据是否编译了 GLOO 进行处理
    // Enable gloo is it's used either in cpu data transfer or controller
    if (horovod_global.cpu_operation == LibType::GLOO ||
        horovod_global.control_operation == LibType::GLOO) {
      gloo_context.Enable();
    }

    if (horovod_global.control_operation == LibType::GLOO) {
      horovod_global.controller.reset(new GlooController(
          horovod_global.response_cache,
          horovod_global.tensor_queue, horovod_global.timeline,
          horovod_global.parameter_manager, horovod_global.group_table,
          gloo_context));
    }
#endif
    // Reset initialization flag
    // 启动后台线程
    horovod_global.initialization_done = false;
    horovod_global.background_thread = std::thread(
        BackgroundThreadLoop, std::ref(horovod_global));
  }

  // Wait to ensure that the background thread has finished initializing MPI.
  while (!horovod_global.initialization_done) {
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
  }
}
```

4.3.3 HorovodGlobalState

在 C++ 世界，HorovodGlobalState 起到了集中管理各种全局变量的作用。

HorovodGlobalState 在 horovod 中是一个全局变量，其中的元素可以供不同的线程访问。HorovodGlobalState 在加载 C++ 的代码时候就已经创建了，同时创建的还有各种 context（mpi_context, nccl_context, gpu_context）。

Horovod 主要会在backgroundThreadLoop 中完成 HorovodGlobalState 不同元素初始化，比较重要的有：

- **controller** 管理总体通信控制流；
- **tensor_queue** 会处理从前端过来的通信需求（allreduce，broadcast 等）；

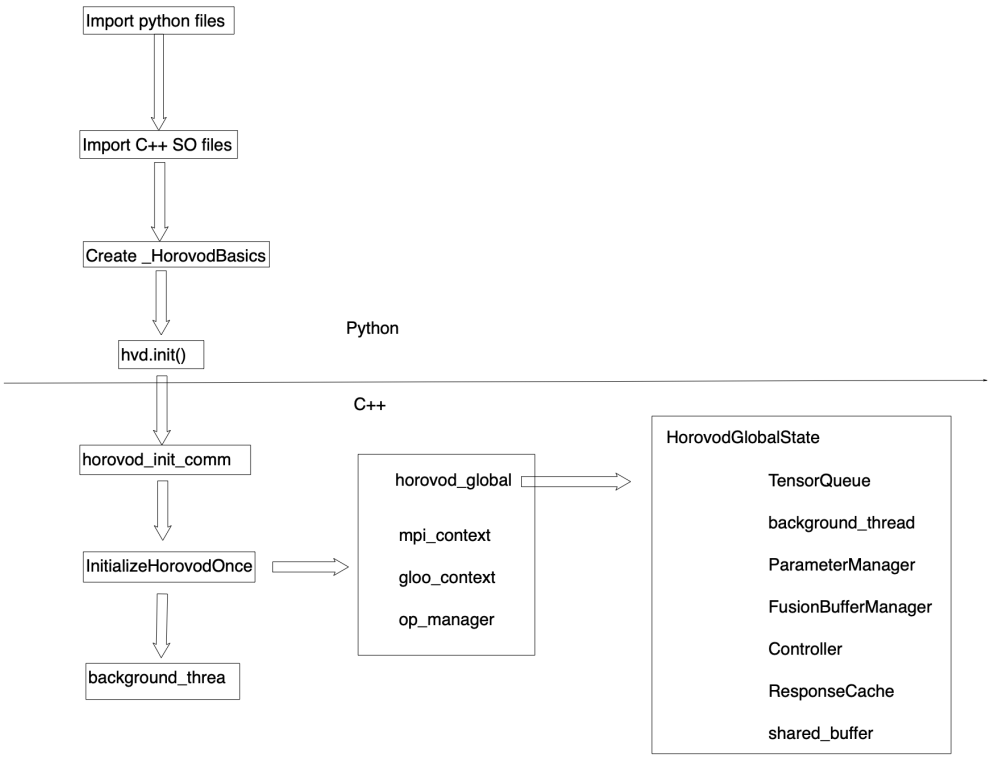
```
// All the Horovod state that must be stored globally per-process.
HorovodGlobalState horovod_global;

#if HAVE_MPI
MPIContext mpi_context;
#endif

#if HAVE_GLOO
GlooContext gloo_context;
#endif

....

std::unique_ptr<OperationManager> op_manager;
```

HorovodGlobalState 摘要如下：

```
struct HorovodGlobalState {

  // Background thread running MPI communication.
  std::thread background_thread; // 后台线程，用来在各个worker之间协调

  ParameterManager parameter_manager; // 维护后台总体参数配置

  // Encapsulates the fusion buffers, handles resizing and auto-tuning of buffer
  // size.
  FusionBufferManager fusion_buffer; // 融合tensor，以便缩减通信开销

  std::shared_ptr<Controller> controller; //管理总体通信控制流

  TensorQueue tensor_queue; //处理从前端过来的通信需求（allreduce，broadcast 等）

  // Pointer to shared buffer for allgather
  void* shared_buffer = nullptr;

  // LRU cache of Responses
  ResponseCache response_cache;

  // Information on registered groups.
  GroupTable group_table;

  ~HorovodGlobalState() {
    // Make sure that the destructor of the background thread is safe to
```

```
    // call. If a thread is still joinable (not detached or complete) its
    // destructor cannot be called.
    if (background_thread.joinable()) {
      shut_down = true;
      background_thread.join();
    }
  }
};
```

目前具体逻辑如下：



至此，horovod 已经初始化完成，用户代码可以使用了。

4.4 hvd 概念

在用户代码中，接下来是rank概念。

```
hvd.local_rank()

hvd.rank()
```

我们介绍下几个相关概念：

- Horovod为设备上的每个GPU启动了该训练脚本的一个副本。**local rank** 就是分配给 某一台计算机 上每个执行训练 的唯一编号（也可以认为是进程号或者GPU设备的ID号），范围是 0 到 n-1，其中 n 是该计算机上GPU设备的数量。
- rank

  可以认为是代表分布式任务里的一个执行训练的唯一全局编号（用于进程间通讯）。Rank 0 在Horovod中通常具有特殊的意义：它是负责此同步的设备。
  - 在百度的实现中，不同 Rank 的角色是不一样的，Rank 0 会充当 coordinator 的角色。它会协调来自其他 Rank 的 MPI 请求，是一个工程上的考量。这一设计也被后来的 Horovod 采

用。
    - Rank 0 也用来把参数广播到其他进程 & 存储 checkpoint。
- **world_size**：进程总数量，会等到所有world_size个进程就绪之后才会开始训练。

hvd.init 这部分的目的就是让并行进程们可以知道自己被分配的 rank / local rank 等信息，于是后续可以根据 local rank（所在节点上的第几张 GPU 卡） 来设置所需的显存分配。

4.5 数据处理

接下来是数据处理。

```
dataset = tf.data.Dataset.from_tensor_slices(
    (tf.cast(mnist_images[..., tf.newaxis] / 255.0, tf.float32),
             tf.cast(mnist_labels, tf.int64))
)
dataset = dataset.repeat().shuffle(10000).batch(128)
```

这里有几点需要说明：

- 首先，训练的数据需要放置在任何节点都能访问的地方。
- 其次，Horovod 需要对数据进行分片处理，需要在不同机器上按Rank进行切分，以保证每个GPU进程训练的数据集是不一样的。
- 数据集本体需要出于数据并行性的需求而被拆分为多个分片，Horovod的不同工作节点都将分别读取自己的数据集分片。

从 PyTorch 示例脚本看得更加清楚。

```
# Horovod: use DistributedSampler to partition the training data.
train_sampler = torch.utils.data.distributed.DistributedSampler(
    train_dataset, num_replicas=hvd.size(), rank=hvd.rank())
train_loader = torch.utils.data.DataLoader(
    train_dataset, batch_size=args.batch_size, sampler=train_sampler, **kwargs)
```

- `DataLoader`的采样器组件从要绘制的数据集中返回可迭代的索引。 PyTorch中的默认采样器是顺序的，返回序列`0, 1, 2, …, n`。 Horovod使用其`DistributedSampler`覆盖了此行为，该`DistributedSampler`处理跨计算机的数据集分区。 `DistributedSampler`本身接受两个参数作为输入： `hvd.size()`(GPU的总数，例如16)和`hvd.rank()`(从总体列表中分配给该设备的ID，例如0...15)。
- Pytorch使用的是数据分布式训练，每个进程实际上是独立加载数据的，所以需要加载相同数据集后用一定的规则根据rank来顺序切割获取不同的数据子集，DistributedSampler就是用来确保dataloader只会load到整个数据集的一个特定子集的做法(实际上不用Pytorch提供的DistributedSampler工具，自己做加载数据后切分word_size个子集按rank顺序拿到子集效果也是一样)。
- 同时为了能够按顺序划分数据子集，拿到不同部分数据，所以数据集不能够进行随机打散，所以用了参数 'shuffle': False。

4.6 广播初始化变量

以下代码完成广播初始化的功能。

```
hvd.callbacks.BroadcastGlobalVariablesCallback(0)
```

这句代码保证的是 rank 0 上的所有参数只在 rank 0 初始化，然后广播给其他节点，即变量从第一个流程向其他流程传播，以实现参数一致性初始化。

下面就介绍下 Horvod 之中广播的使用。

4.6.1 广播定义

广播的具体实现是：

```
class BroadcastGlobalVariablesCallbackImpl(object):
    def __init__(self, backend, root_rank, device='', *args):
        super(BroadcastGlobalVariablesCallbackImpl, self).__init__(*args)
        self.backend = backend
        self.root_rank = root_rank
        self.device = device
        self.broadcast_done = False

    def on_batch_end(self, batch, logs=None):
        if self.broadcast_done:
            return

        with tf.device(self.device):
            if hvd._executing_eagerly() and hasattr(self.model, 'variables'):
                # TensorFlow 2.0 or TensorFlow eager
                hvd.broadcast_variables(self.model.variables,
                                        root_rank=self.root_rank)
                hvd.broadcast_variables(self.model.optimizer.variables(),
                                        root_rank=self.root_rank)
            else:
                bcast_op = hvd.broadcast_global_variables(self.root_rank)
                self.backend.get_session().run(bcast_op)

        self.broadcast_done = True
```

4.6.2 broadcast_variables

broadcast_variables 调用了 _make_broadcast_group_fn 完成功能，可以看到对于 执行图 的每个变量，调用了 broadcast。

```
def broadcast_variables(variables, root_rank):
    """Broadcasts variables from root rank to all other processes.

    Arguments:
        variables: variables for broadcast
        root_rank: rank of the process from which global variables will be
broadcasted
                   to all other processes.
    """
    broadcast_group = _make_broadcast_group_fn()
    return broadcast_group(variables, root_rank)
```

以及

```
@_cache
def _make_broadcast_group_fn():
    if _executing_eagerly():
        # Eager mode will parallelize independent control flow
        def broadcast_group(variables, root_rank):
            for var in variables:
```

```
                var.assign(broadcast(var, root_rank))

        return _make_subgraph(broadcast_group)
    else:
        # Graph mode requires an Op
        def broadcast_group(variables, root_rank):
            return tf.group(*[var.assign(broadcast(var, root_rank))
                              for var in variables])

        return broadcast_group
```

4.6.3 调用 MPI

broadcast 就是调用了 MPI 函数真正完成了功能。

```
def broadcast(tensor, root_rank, name=None, ignore_name_scope=False):
    """An op which broadcasts the input tensor on root rank to the same input
tensor
    on all other Horovod processes.

    The broadcast operation is keyed by the name of the op. The tensor type and
    shape must be the same on all Horovod processes for a given name. The
broadcast
    will not start until all processes are ready to send and receive the tensor.

    Returns:
      A tensor of the same shape and type as `tensor`, with the value
broadcasted
      from root rank.
    """
    if name is None and not _executing_eagerly():
        name = 'HorovodBroadcast_%s' % _normalize_name(tensor.name)
    return MPI_LIB.horovod_broadcast(tensor, name=name, root_rank=root_rank,
                                     ignore_name_scope=ignore_name_scope)
```

4.6.4 同步参数

在后台进程中，会根据情况定期同步参数。

```
bool RunLoopOnce(HorovodGlobalState& state) {
    // 业务逻辑功能

  if (state.parameter_manager.IsAutoTuning()) {
    bool should_sync =
        state.parameter_manager.Update(tensor_names, total_tensor_size);
    // 看看是否需要同步，如果需要，就同步。
    if (should_sync) {
      state.controller->SynchronizeParameters();
    }
  }
  ......
}
```

同步参数代码也是调用了 Bcast 功能完成。

```
void Controller::SynchronizeParameters() {
```

```
  ParameterManager::Params param;
  if (is_coordinator_) { // rank 0 执行操作
    param = parameter_manager_.GetParams();
  }

  void* buffer = (void*)(&param);
  size_t param_size = sizeof(param);
  Bcast(buffer, param_size, 0, Communicator::GLOBAL);

  if (!is_coordinator_) { // worker 执行操作
    parameter_manager_.SetParams(param);
  }
}
```

4.7 DistributedOptimizer

最后需要配置DistributedOptimizer，这就是关键点之一。

```
# Horovod: add Horovod DistributedOptimizer.
opt = hvd.DistributedOptimizer(
    opt, backward_passes_per_step=1, average_aggregated_gradients=True)
```

TF Optimizer 是模型训练的关键API，可以获取到每个OP的梯度并用来更新权重。HVD 在原始 TF Optimizer的基础上包装了**hvd.DistributedOptimizer**。

`DistributedOptimizer`包装器将原始优化器作为输入，将梯度计算委托给它。 即`DistributedOptimizer`会调用原始优化器进行梯度计算。这样，在集群中每台机器都会用原始优化器得到自己的梯度（Local Gradient）。

Horovod `DistributedOptimizer`接下来会使用all-reduce或all-gather来完成全局梯度归并，然后将这些平均梯度应用于所有设备。

我们梳理下其中的调用关系：

- hvd.DistributedOptimizer继承 keras Optimizer，在计算时候，依然由传入的原始优化器做计算。
- 在得到计算的梯度之后，调用 hvd.allreduce 或者 hvd.allgather 来计算。
- 最后实施这些平均之后的梯度。从而实现整个集群的梯度归并操作。

具体后文会详细介绍。

4.8 未来可能

Horovod 目前架构的基础是：机器学习的模型参数在一张 GPU 上可以存下。

未来是否可以把模型分片结合进来，是一个很大的看点。

另外，如果模型的全连接层较多，则全连接层的强耦合性结合 allreduce 类似 bsp 的同步机制，还是会让网络通信时间成为瓶颈。因此，在 ring-allreduce 环境下，同步协议的改造，比如利用 SSP 来替换 BSP，或者利用梯度压缩来加快 allreduce 进程也是值得探索的方向。

0x05 总结

针对文初提出的几个问题，我们现在回答如下：

- Hovorod 怎么进行数据分割?

    - 答案：有的框架可以自动做数据分割。如果框架不提供，则需要用户自己进行数据分割，以保证每个GPU进程训练的数据集是不一样的。
- Hovorod 怎么进行模型分发?
    - 用户需要手动拷贝训练代码到各个节点上。
- Hovorod 启动时候，python 和 C++ 都做了什么?
    - 答案：python 会引入 C++库，初始化各种变量和配置。C++部分会对 MPI，GLOO上下文进行初始化，启动后台进程处理内部通信。
- 如何确保 Hovorod 启动时候步骤一致；
    - 答案： rank 0 上的所有参数只在 rank 0 初始化，然后广播给其他节点，即变量从第一个流程向其他流程传播，以实现参数一致性初始化。