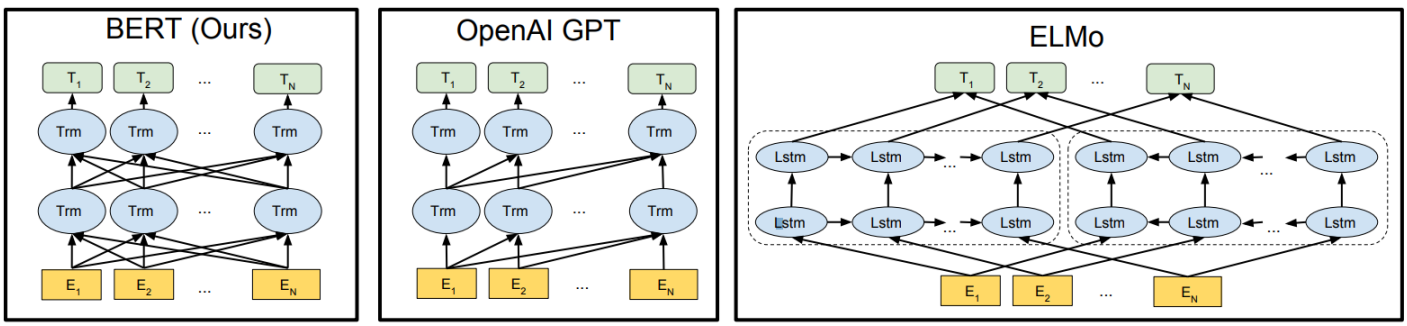
第二讲：模型领域性增强及分布式训练

小象学院

Bert模型结构及其变种

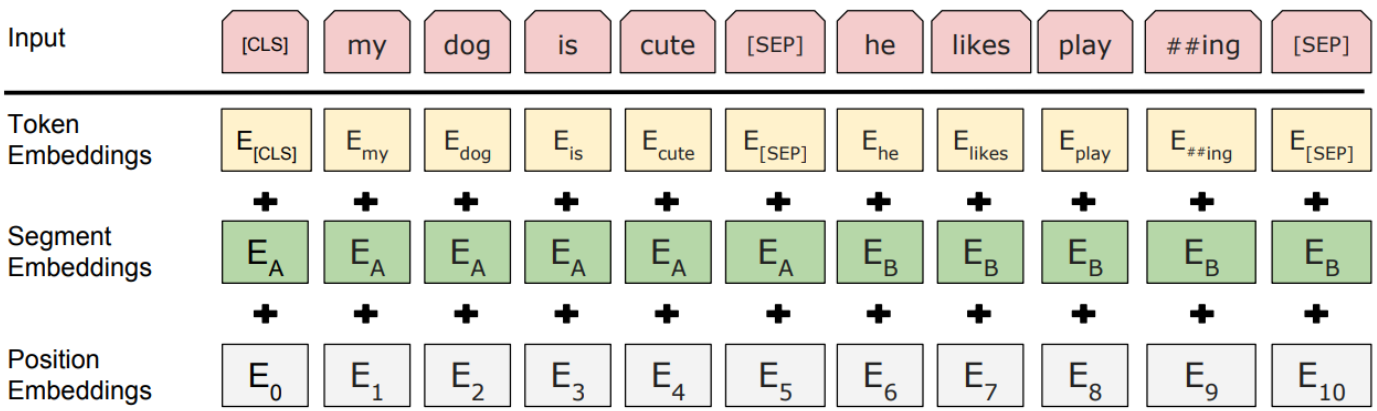
Bert

BERT是2018年10月**由Google AI研究院提出的一种预训练模型。**可谓是一经问世，便**惊艳全场**，首先是在机器阅读理解顶级水平测试SQuAD1.1中表现出惊人的成绩: **全部两个衡量指标上全面超越人类，并且在11种不同NLP测试中创出SOTA表现**，包括将GLUE基准推高至80.4% (绝对改进7.6%)，MultiNLI准确度达到86.7% (绝对改进5.6%)，成为NLP发展史上的里程碑式的模型成就。



关于Bert训练过程中的关键点

输入表征



- 词块嵌入（token embedding）
- 段嵌入（segment embedding）
- 位嵌入（position embedding）

输入表征通过对相应词块的词块嵌入（word embedding）、段嵌入（segment embedding）和位嵌入（position embedding）求和来构造。即

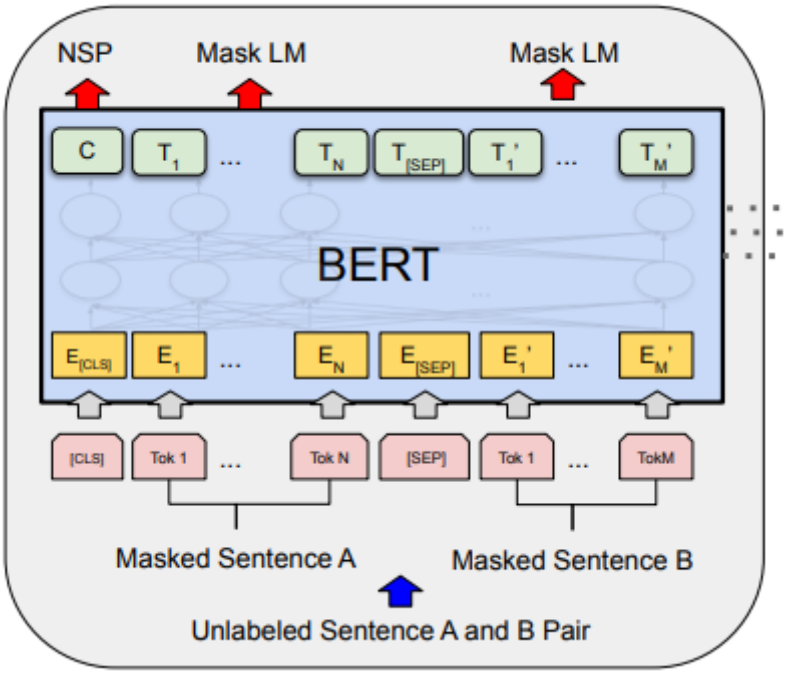
MLM预训练

随机遮蔽输入序列中15%的单词，然后仅预测那些被遮蔽词块，这个任务可以通过模型训练得到更深深层次的语义表征，整个mask的过程相当于完形填空任务，其中在mask的过程中，有区分如下：

- 有80%的概率用“[mask]”标记来替换
- 有10%的概率用随机采样的一个单词来替换
- 有10%的概率不做替换（虽然不做替换，但是还是要预测哈）

NSP

BERT中另一个重要且轻量级的任务为Next Sentence Prediction，即学习句间关系，判断两个句子是否是连续的上下文两个句子。



- 正负样本对的构造
- 句子级负采样
- nsp任务的潜在弊端

RoBERTa: A Robustly Optimized BERT Pretraining Approach

论文：https://arxiv.org/pdf/1907.11692.pdf

- 训练时间更长，batch size更大，训练数据更多
- 移除了next predict loss
- dynamic masking
- 调整Adam优化器的参数

Macbert

MLM as correction, 使用校正做为Mask的语言模型, 通过用相似的单词mask，减轻了预训练和微调阶段两者之间的差距

Revisiting Pre-trained Models for Chinese Natural Language Processing

论文：https://arxiv.org/pdf/2004.13922.pdf

	Chinese	English
Original Sentence	使用语言模型来预测下一个词的概率。	we use a language model to predict the probability of the next word.
+ CWS	使用 语言 模型 来 预测 下 一个 词 的 概率 。	-
+ BERT Tokenizer	使 用 语 言 模 型 来 预 测 下 一 个 词 的 概 率 。	we use a language **model** to **pre ##di ##ct** the **pro ##ba ##bility** of the next word .
Original Masking	使 用 语 言 **[M]** 型 来 **[M]** 测 下 一 个 词 的 概 率 。	we use a language **[M]** to **[M] ##di ##ct** the **pro [M] ##bility** of the next word .
+ WWM	使 用 语 言 **[M] [M]** 来 **[M] [M]** 下 一 个 词 的 概 率 。	we use a language **[M]** to **[M] [M] [M]** the **[M] [M] [M]** of the next word .
++ N-gram Masking	使 用 **[M] [M] [M] [M]** 来 **[M] [M]** 下 一 个 词 的 概 率 。	we use a **[M] [M]** to **[M] [M] [M]** the **[M] [M] [M] [M] [M]** next word .
+++ Mac Masking	使 用 语 法 建 模 来 预 见 下 一 个 词 的 几 率 。	we use a **text system** to **ca ##lc ##ulate** the **po ##si ##bility** of the next word .

- Ngram mask策略；
- 相似单词替换；

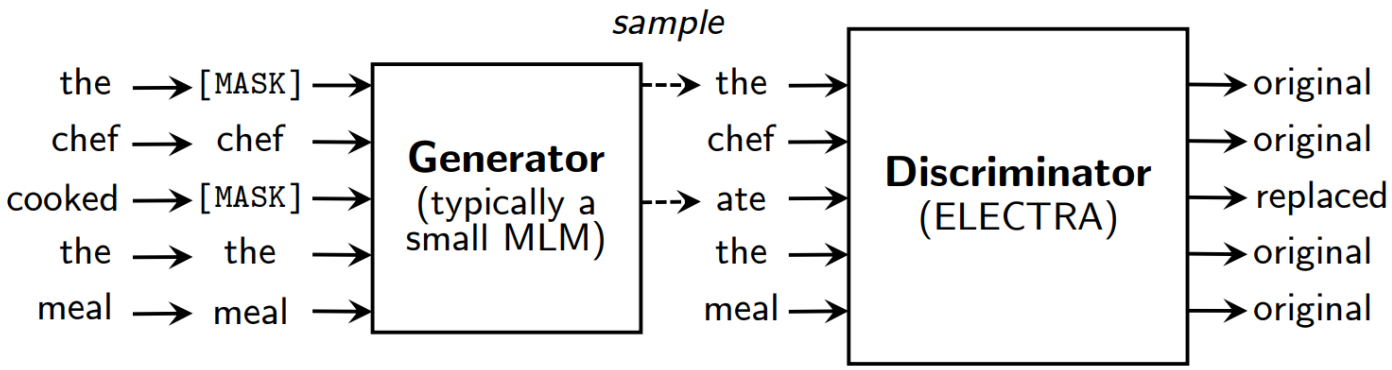
总结一下， mask类语言模型预训练任务依赖于两个方面：

- 选择要mask的token；
- 替换所选token；
- 如何根据下游具体任务设计合适的策略，进行定制化；

ELECTRA

论文地址：https://openreview.net/pdf?id=r1xMH1BtvB

ELECTRA最主要的贡献是提出了新的预训练任务和框架，把这种Masked language model(MLM)预训练任务改成了判别式的Replaced token detection(RTD)任务，判断当前token是否被语言模型替换过。



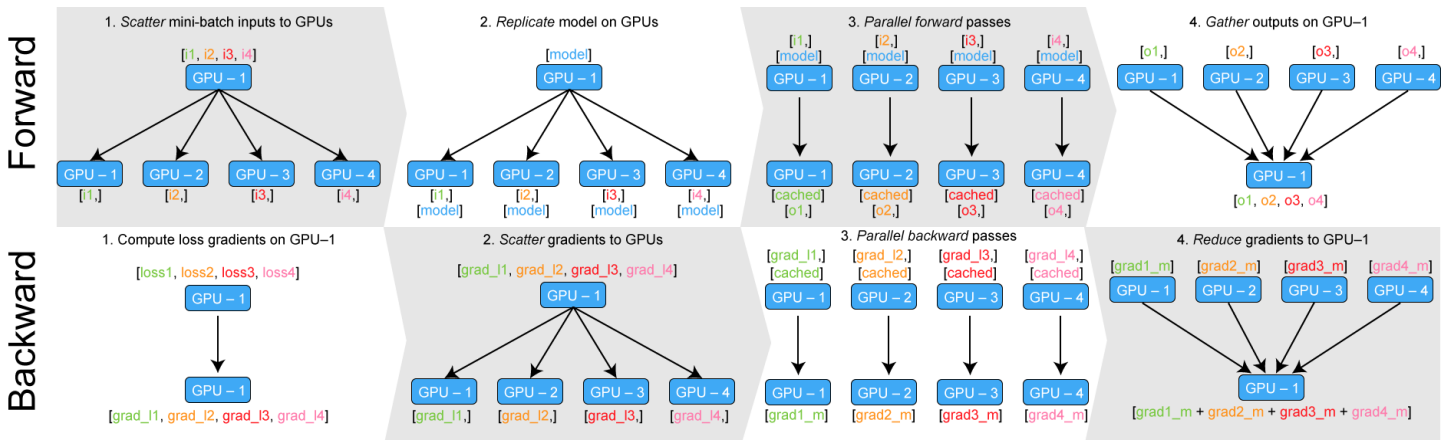
ELECTRA提出了一套新的预训练框架，其中包括两个部分：Generator和Discriminator，Generator: 一个小的MLM模型，在[MASK]的位置预测原来的词。Generator将用来把输入文本做部分词的替换，Discriminator: 判断输入句子中的每个词是否被替换，即使用Replaced Token Detection (RTD)预训练任务，取代了BERT原始的Masked Language Model (MLM)。需要注意的是这里并没有使用Next Sentence Prediction (NSP)任务，在预训练阶段结束之后，我们只使用Discriminator作为下游任务精调的基模型。

多GPU下分布式训练

DataParallel

DataParallel 是torch早期推出的用于分布式训练的包，DataParallel 使用起来非常方便，我们只需要用 DataParallel 包装模型，再设置一些参数即可。需要定义的参数包括：参与训练的 GPU 有哪些，device_ids=gpus；用于汇总梯度的 GPU 是哪个，output_device=gpus[0] 。DataParallel 会自动帮我们将数据切分 load 到相应 GPU，将模型复制到相应 GPU，进行正向传播计算梯度并汇总：

```
1  model = nn.DataParallel(model.cuda(), device_ids=gpus, output_device=gpus[0])
```



DataParallel的并行处理机制

1. 数据和模型首先加载到主gpu上，再由主gpu将模型和数据复制到其他gpu上；
2. 每个gpu在独立的线程上 对自己的数据独立的进行 forward计算；
3. 每个gpu上独立进行backforward并计算梯度；
4. 所有梯度汇总到主gpu上，然后梯度下降 权重更新，然后再将更新好的权重分发到每个gpu上。

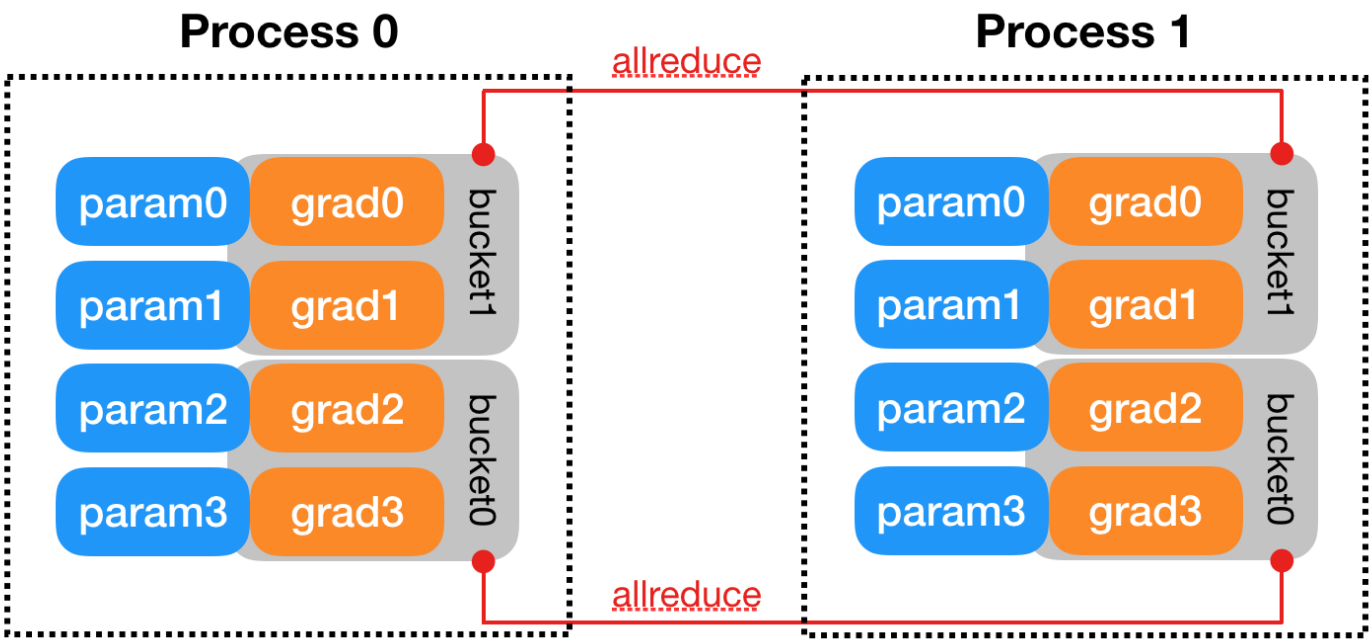
存在的问题：

- 负载不均衡
- 网络通信瓶颈

DistributedDataParallel

在 pytorch 1.0 之后，官方终于对分布式的常用方法进行了封装，支持 all-reduce，broadcast，send 和 receive 等等。通过 MPI 实现 CPU 通信，通过 NCCL 实现 GPU 通信。官方也曾经提到用 DistributedDataParallel 解决 DataParallel 速度慢，GPU 负载不均衡的问题，目前已经很成熟了。

与 DataParallel 的单进程控制多 GPU 不同，在 distributed 的帮助下，我们只需要编写一份代码，torch 就会自动将其分配给n个进程，分别在n个 GPU 上运行。



运行机制：

- 每个进程独立占有一张显卡，独立的加载自己的数据和模型，不需要数据广播、也不需要模型广播（每个gpu都有一个相同的模型副本），其中，分布式数据采样器（DistributedSampler）可确保加载的数据在各个进程之间不重叠；
- 每个gpu独立进行forward，计算网络的输出，每个gpu独立计算loss，进行反向计算梯度， 各进程独立将梯度进行汇总平均；

- 每个gpu用相同的梯度 独立更新参数。因为每个gpu都是从一个相同的模型副本开始的，初始参数一致，并且下降的梯度相同，所以所有gpu上的权重更新都是相同的。因此不需要模型同步。

执行步骤：

1. 在 API 层面，pytorch 为我们提供了 torch.distributed.launch 启动器，用于在命令行分布式地执行 python 文件。在执行过程中，启动器会将当前进程的（其实就是 GPU的）index 通过参数传递给 python，我们可以这样获得当前进程的 index：

```
local_rank = torch.distributed.get_rank()
torch.cuda.set_device(local_rank)
print(local_rank)
```

2. 接着，使用 init_process_group 设置GPU 之间通信使用的后端和端口：

```
torch.distributed.init_process_group(backend='nccl')
```

3. 然后使用 DistributedSampler 对数据集进行划分。如此前我们介绍的那样，它能帮助我们将每个 batch 划分成几个 partition，在当前进程中只需要获取和 local_rank 对应的那个 partition 进行训练.

```
train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=..., sampler=
```

4. 使用 DistributedDataParallel 包装模型，它能帮助我们为不同 GPU 上求得的梯度进行 all reduce（即汇总不同 GPU 计算所得的梯度，并同步计算结果）。all reduce 后不同 GPU 中模型的梯度均为 all reduce 之前各 GPU 梯度的均值。

```
model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[args.local_ra
```

Apex 加速

Apex 是 NVIDIA 开源的用于混合精度训练和**分布式训练库**。Apex 除了对混合精度训练的过程进行了封装，改两三行配置就可以进行混合精度的训练，从而大幅度降低显存占用，节约运算时间。此外，Apex 也提供了对分布式训练的封装，针对 NVIDIA 的 NCCL 通信库进行了优化。

在混合精度训练上，Apex 的封装十分优雅。直接使用 amp.initialize 包装模型和优化器，apex 就会自动帮助我们管理模型参数和优化器的精度了，根据精度需求不同可以传入其他配置参数。

在分布式训练的封装上，Apex 改动并不大，主要是优化了 NCCL 的通信。**因此，大部分代码仍与 torch.distributed 保持一致**。使用的时候只需要将 torch.nn.parallel.DistributedDataParallel 替换为 apex.parallel.DistributedDataParallel 用于包装模型。在 API 层面，相对于 torch.distributed ，它可以自动管理一些参数（可以少传一点）：

```
from apex.parallel import DistributedDataParallel
model = DistributedDataParallel(model)
# # torch.distributed
# model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[args.local_
# model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[args.local_
```

在使用时，调用 torch.distributed.launch 启动器启动：

```
CUDA_VISIBLE_DEVICES=0,1,2,3 python -m torch.distributed.launch --nproc_per_node=4
```

需要注意梯度溢出的问题。

Horovod 分布式训练

Horovod 是 Uber 开源的深度学习工具，它的发展吸取了 Facebook "Training ImageNet In 1 Hour" 与百度 "Ring Allreduce" 的优点，可以无痛与 PyTorch/Tensorflow 等深度学习框架结合，实现并行训练。

在 API 层面，Horovod 和 torch.distributed 十分相似。在 mpirun 的基础上，Horovod 提供了自己封装的 horovodrun 作为启动器。

与 torch.distributed.launch 相似，我们只需要编写一份代码，horovodrun 启动器就会自动将其分配给 n个进程，分别在n 个 GPU 上运行。在执行过程中，启动器会将当前进程的（其实就是 GPU的）index 注入 hvd，我们可以这样获得当前进程的 index：

```
import horovod.torch as hvd
hvd.local_rank()
```

与 init_process_group 相似，Horovod 使用 init 设置GPU 之间通信使用的后端和端口:

```
hvd.init()
```

接着，使用 DistributedSampler 对数据集进行划分。如此前我们介绍的那样，它能帮助我们将每个 batch 划分成几个 partition，在当前进程中只需要获取和 rank 对应的那个 partition 进行训练：

```
train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=..., sampler=
```

之后，使用 broadcast_parameters 包装模型参数，将模型参数从编号为 root_rank 的 GPU 复制到所有其他 GPU 中：

```
hvd.broadcast_parameters(model.state_dict(), root_rank=0)
```

然后，使用 DistributedOptimizer 包装优化器。它能帮助我们为不同 GPU 上求得的梯度进行 all reduce（即汇总不同 GPU 计算所得的梯度，并同步计算结果）。all reduce 后不同 GPU 中模型的梯度均为 all reduce 之前各 GPU 梯度的均值：

```
hvd.DistributedOptimizer(optimizer, named_parameters=model.named_parameters(), com
```

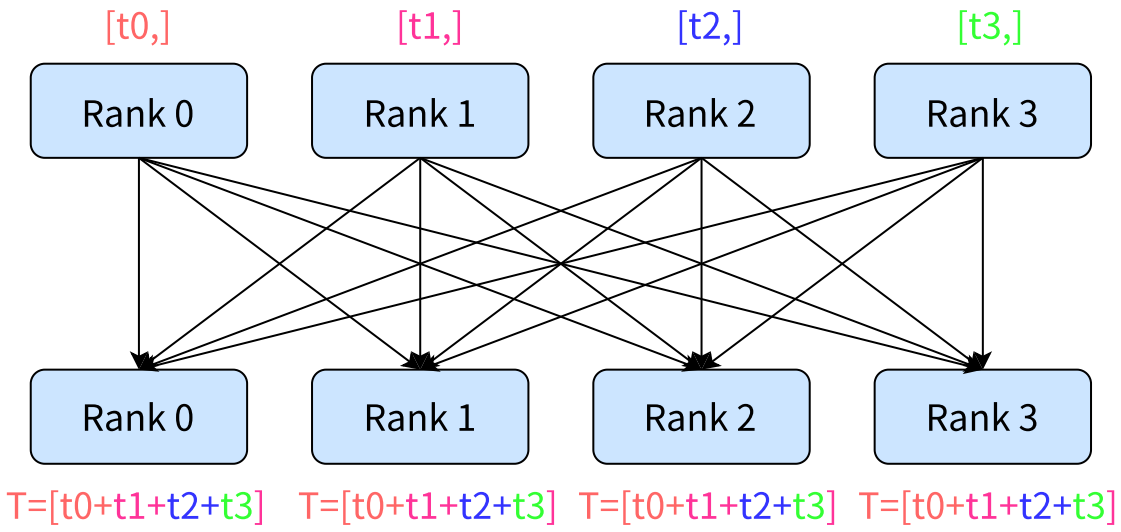
在使用时，调用 horovodrun 启动器启动：

```
CUDA_VISIBLE_DEVICES=0,1,2,3 horovodrun -np 4 -H localhost:4 python main.py
```

分布式训练进程间的交互

到目前为止，我们已经了解了distributed 中一些比较基础和底层的 API的用法，这些 API 可以帮助我们控制进程之间的交互，控制 GPU 数据的传输，例如：

Distributed Sampler 能够帮助我们分发数据，DistributedDataParallel、hvd.broadcast_parameters 能够帮助我们分发模型，并在框架的支持下解决梯度汇总和参数更新的问题，具体如下：



具体来说，它的交互过程包含以下三步：

1. 通过调用 `all_reduce(tensor, op=...)`，当前进程会向其他进程发送 `tensor`（例如 rank 0 会发送 rank 0 的 tensor 到 rank 1、2、3）
2. 接受其他进程发来的 `tensor`（例如 rank 0 会接收 rank 1 的 tensor、rank 2 的 tensor、rank 3 的 tensor）。
3. 在全部接收完成后，当前进程（例如rank 0）会对当前进程的和接收到的 `tensor`（例如 rank 0 的 `tensor`、rank 1 的 `tensor`、rank 2 的 `tensor`、rank 3 的 `tensor`）进行 `op`（例如求和）操作。