SageMaker Tensorflow 训练场景调优实战总结

梁宇辉

议程

- 这个总结的适用性
- Sagemaker + Tensorflow 的各种训练场景
- 总结

这个总结的适用性

适用性

- 虽然这里都是讲TF的，但是**优化的思路是通用的**。
  - 思路可以扩展到其他框架的模型的训练速度优化。
- 虽然这里的例子是用Deepfm模型，但是优化的经验**对于TF的其他模型也是通用的**。
  - 这个片子是根据从多个客户做的实际项目归纳总结的。
  - 里面包括了很多的踩坑总结。
- 这个片子适用的人群：
  - 对TF API的使用有一定经验。
  - 对分布式训练的原理有一定理解。
  - 对linux OS有一定理解。
  - 对ML有一定理解。

需求和目标

- **尽量少的改动代码**。
- **迁移的方式对于SageMaker prebuilt-in Tensorflow支持的版本是通用的**，而不只是针对DeepFM模型。
- **所有可能的训练方式都要尝试**。
- 在模型评价效果基本不降低的情况下，**加速训练速度**，从而节省成本。

Sagemaker + Tensorflow 的各种训练场景

训练场景

- 单机训练 VS 多机训练
- 纯CPU训练 VS GPU训练
- 单机单卡 VS 单机多卡
- 多机单卡 VS 多机多卡
- 分布式训练方式：
  - Parameter server
  - Horovod
  - Native tensorflow distributed strategy
- File mode Vs pipe mode
- S3 side data shard or tensorflow side data shard
- Tensorflow input data pipeline：
  - Libsvm格式 vs tfrecord格式
  - Scalar map vs Vectorized map
  - Tensorflow Dataset API的各种transformation API的合理调用顺序
- 训练中的CPU或GPU使用率优化
- 自动混合精度训练

单机训练

- 问题：
  - 纯CPU训练和用GPU训练对代码改动一样吗？
  - 代码中不做对设备的感知，能使用单机的多个CPU或者多个GPU吗？
  - 用什么方法可以做单机多CPU或者单机多GPU卡训练呢？哪种方法更好呢？
  - 用GPU训练就一定比用纯CPU训练性价比高吗？
  - 选用Sagemaker的file mode还是pipe mode呢？
  - 如何更好的利用单机的多个CPU或者多个GPU？
    - Data input pipeline是瓶颈吗？
    - CPU或者GPU使用率上不去，怎么办呢？
    - **CPU训练实例的机型越大，训练速度就越快吗**？
    - **8卡一定比4卡的训练速度快吗？**
      - 拥有8卡的机器，用8卡训练和用4卡训练的对比
      - 一个8卡机器用8卡训练与1个4卡的机器用4卡训练的对比

Continue….

- 基于Sagemaker内建tensorflow容器的**单机多CPU训练**可选择方式：
  - 代码不做CPU设备感知的处理
    - Tensorflow会自动选择合适数量的线程来并行训练过程的operation的计算。
  - 代码对CPU设备感知做处理：
    - 利用tensorflow提供的intra_op线程池和inter_op并行度设置。
    - 配合MKL-DNN提供的环境变量对OS线程的binding设置。
  - 利用tower方式：
    - 原生的Tensorflow estimator API提供towerOptimizer
  - 利用原生tensorflow的distributed strategy mirrorstrategy方式
  - 利用Sagemaker集成的horovod方式

Continue.....

- 代码对CPU设备感知做处理：
  - 设置tensorflow的intra_op线程池和inter_op并行度：

```
    num_cpus = int(os.environ['SM_NUM_CPUS'])
    config = tf.estimator.RunConfig().replace(session_config =
tf.ConfigProto(allow_soft_placement=True, device_count={'CPU': num_cpus},
intra_op_parallelism_threads=num_cpus, inter_op_parallelism_threads=num_cpus))
```

  - 配合Intel MKL-DNN的环境变量设置一般能让CPU使用率更高，训练速度更快：

```
os.environ["KMP_AFFINITY"]= "verbose,disabled"
#os.environ["KMP_AFFINITY"]= "granularity=fine,compact,1,0"
#os.environ["KMP_AFFINITY"]= "granularity=fine,verbose,scatter,1,0"
os.environ['OMP_NUM_THREADS'] = str(num_cpus)
os.environ['KMP_SETTINGS'] = '1'
```

Continue…..

- Tips：
  - MKL-DNN的环境变量“KMP_AFFINITY”缺省设置为 **“granularity=fine,compact,1,0“**，然后把TF的intra和inter都设置为当前实例的最大的VCPU数量后，CPU使用率上限差不多就是训练实例的物理核数。
  - 而设置了**os.environ[“KMP_AFFINITY”]= “verbose,disabled”**之后，也就是没有把 OS的线程bind到硬件的超线程之后，CPU使用率超过了物理核数。
    - 之所以bind后的效果不好，我的理解是如果需要消费min batch的计算线程与prepare数据的线程都bind到同一个超线程上并且它们都处于线程ready可以run的情况，那么它们因为share同一个VCPU而互相等待CPU时间片，而且linux kernel没有办法对可以run的已经绑定的OS线程做CPU load balance了。

Continue……

- **并不是一定要disable binding**，这个要case by case来测试。
  - 我测试的几个场景，相同的mini-batch size（1024），同样的三层全连接层结构‘256,128,64’的deepfm模型，不管输入文件格式是libsvm还是tfrecord，都是disable binding以后CPU使用率更高。
- **关于TF intra并行度，TF inter并行度以及MKLDNN线程数量的设置**：
  - 这三个参数的不同组合会产生不同的训练速度。
  - **对于不同的模型，不同的batch size，不同的训练实例，上面三个参数组合产生的效果也不同。因此要case by case来调试**。
    - 之前有个项目，把三者都设置为VCPU数量的一半的训练速度最快；而另一个项目，把三者设置为VCPU的数量训练速度最快。
    - 分别有两个项目，一个项目的batch size是4K，c5.18xlarge在各种参数设置下都比c5.9xlarge速度慢，但是当batch size是64K的时候，c5.18xlarge的速度超过了c5.9xlarge；而另一个项目的batch size不管是4K还是64K，c5.18xlarge都比c5.9xlarge的训练速度快。
    - 同一个项目，batch size不同对于同一个CPU实例的CPU使用率影响很大。比如batch size是4K的时候，c5.9xlarge的CPU使用率能超过21K；而当batch size是64K的时候，其他上下文都一样的情况下，c5.9xlarge的CPU使用率才刚超过16K。
      - 但是**CPU使用率高不见得整个训练速度就快，因为batch size是64K的话需要更少的step，所以仍然可能用更少的时间训练完**。

Continue….

- 利用tower方式，涉及的代码修改如下：
  - 手动设置CPU device：

```
devices_list = []
if manual_CPU_device_set:
    cpu_prefix='/cpu:'
    for i in range(1, num_cpus):
        devices_list.append(cpu_prefix + str(i))
```

  - 用replicate_model_fn对原model_fn做wrapper：

```
DeepFM = tf.estimator.Estimator(model_fn=tf.contrib.estimator.replicate_model_fn(model_fn, devices=devices_list), model_dir=FLAGS.model_dir, params=model_params, config=config)
```

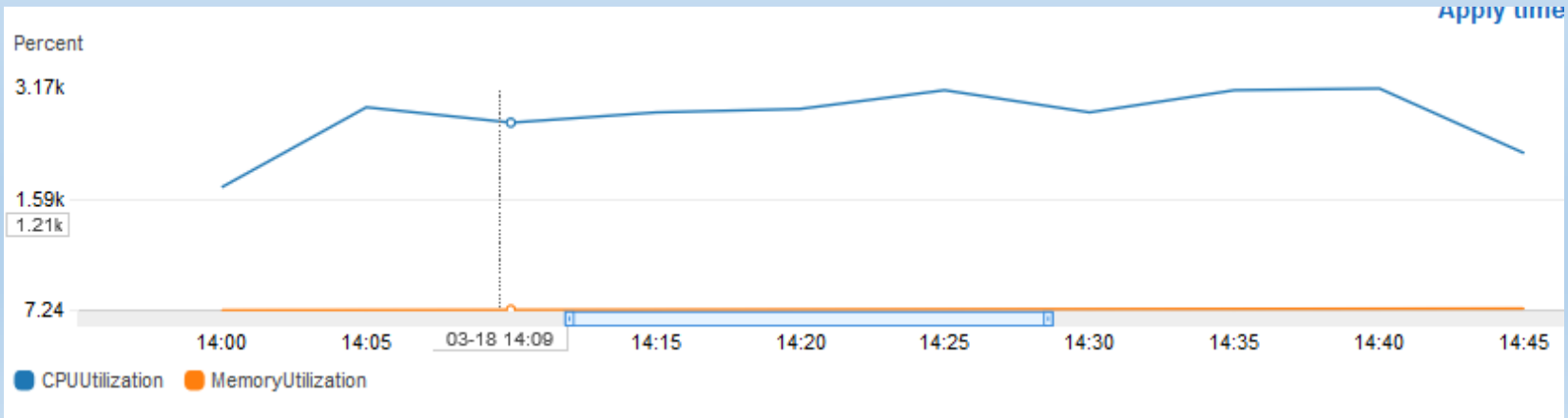
  - 用TowerOptimizer对optimizer做wrapper：

```
optimizer = tf.contrib.estimator.TowerOptimizer(optimizer)
```

  - 用 **with tf.variable_scope('deepfm_model', reuse=tf.AUTO_REUSE):** 来wrapper你的model_fn函数实现。
  - Scaling batch size为（CPU数量 - 1）

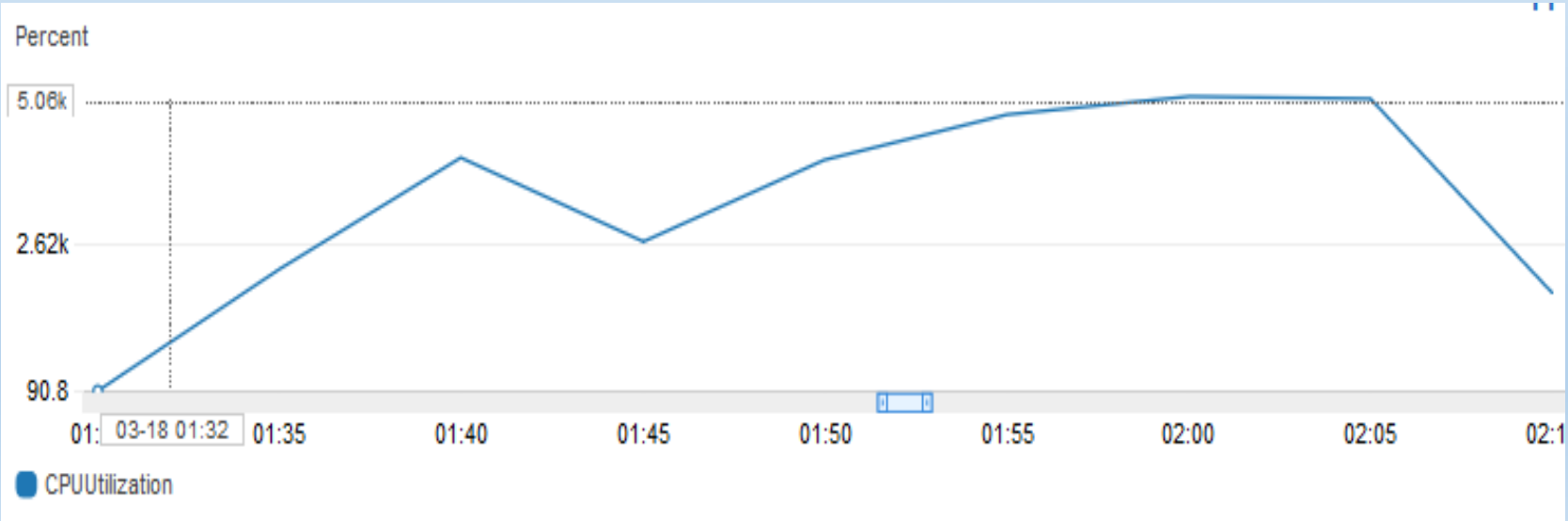
Continue….

- Tips:
  - **若没有手动设置CPU device**，Tensorflow的实现并没有实际做tower的梯度平均，**已经退化为没有tower的情况**。**这个时候不要scaling batch size**。
  - 不使用Tower + Libsvm格式 + pipe mode + MKL-DNN disable binding + intra/inter设置为最大VCPU数量， 单机多CPU(ml.c5.18xlarge 72个VCPU)使用率如下图：



Continue……

- Tower + 手动设置CPU device + Libsvm格式 + pipe mode + MKL-DNN disable binding + intra/inter设置为最大VCPU数量， 单机多CPU(ml.c5.18xlarge 72个VCPU)使用率如下图（**很明显比不用tower方式的CPU使用率要高，而且超过了物理core数量**）：



Continue.....

- 利用原生tensorflow的mirrorstrategy，涉及的代码修改如下：
  - 手动设置CPU device：
    - TF的mirrostrategy+手动设置CPU设备的时候，同样会在/CPU:0上做reduce，所以这里把/CPU:0设备不作为replica。

```
devices_list = []
if manual_CPU_device_set:
    cpu_prefix='/cpu:'
    for i in range(1, num_cpus):
        devices_list.append(cpu_prefix + str(i))
    mirrored_strategy = tf.distribute.MirroredStrategy(devices=devices_list)
else :
    mirrored_strategy = tf.distribute.MirroredStrategy()
```

  - 把这个strategy加入config：

```
config = tf.estimator.RunConfig(train_distribute=mirrored_strategy,
                                eval_distribute=mirrored_strategy,
                                session_config = session_config)
```

Continue.....

- Tips：
  - **若没有手动设置CPU device**，Tensorflow的实现是只有一个CPU设备是worker，**基本和没有设置mirrorstrategy的情况差不多。**
  - **使用mirrorstrategy的时候需要对batch size做scaling，也就是这个时候送入的batch size是global batch size。**
    - **参考： https://www.tensorflow.org/guide/distributed_training**
  - 在手动设置CPU device的情况下，且tf.distribute.MirroredStrategy() API中如果没有设置cross_device_ops 参数, TF的实现是对于CPU设备用ReductionToOneDevice子策略，也就是只是在单个设备上做reduce而不是all reduce。

Continue……

- 在手动设置CPU device的情况下，且手动设置子策略为**HierarchicalCopyAllReduce，**当前的TF实现是对于CPU设备HierarchicalCopyAllReduce不做ALL reduce，仍然是在/CPU:0做单一的reduce。

- 当使用tensorflow的dataset API与distribute strategy联合使用的时候，**input_fn需要返回dataset**而不是返回特征和label。
- 和tower方式相比，其他上下文相同的情况下，mirrostrategy的训练速度比较慢。**因此在单机多CPU上训练时，不建议使用mirrorstrategy方式**。

Continue.....

- 利用horovod做单机多CPU训练，代码改动需要注意的地方：
  - 初始化horovod即调用hvd.init()；
  - 如果是用CPU来训练，若想把worker绑定到物理core需要在自己写的调用Sagemaker API的helper code中设置distributions参数：

```
hvd_processes_per_host = 35
distributions = {'mpi': {
                    'enabled': True,
                    'processes_per_host': hvd_processes_per_host,
                    'custom_mpi_options': '--bind-to core -verbose  -x OMPI_MCA_btl_vader_single_copy_mechanism=none'
                        }
                }
```

  - 如果是用GPU卡训练，把每个GPU pin到每个worker进程：

    config = tf.ConfigProto()

    config.gpu_options.visible_device_list = str(hvd.local_rank())
  - 从Rank 0的master把模型参数的初始值广播给所有的其他worker来保证大家一致性的初始化，即调用hvd.BroadcastGlobalVariablesHook(0)。
  - 用Horovod Distributed Optimizer对原始optimizer进行wrapper。
  - **scale learning rate by the number of workers**.

Continue……

- Tips：
  - horovod rank 0的master进行checkpoint和模型的保存，以及模型的评估。
  - **Horovod训练方式不需要scaling batch size。**
  - **在pipe mode下，horovod的同一个训练实例上的不同的worker进程需要使用不同的channel**，对应了一个Linux的FIFO命名管道，否则会hung住。
    - 原因是第一个worker进程读取了FIFO的数据，同一个实例上的其他的worker进程从同一个FIFO读不到数据，因此horovod的工作就不正常了。
  - Horovod for CPU训练的时候，每个机器一个worker，配合上TF inter和intra线程池设置 +MKL-DNN环境变量的设置可能是一个不错的尝试起点。
  - horovod for CPU训练的时候，如果每个机器使用多个worker，那么设置--bind-to core 可能对训练速度好一些，这个core指的是物理core不是超线程。
    - 如果不设置，默认是bind to socket，这个socket指的是物理socket，一个socket会有多个物理core，如果是bind to socket的话每个实例能使用的worker进程数就少了。

Continue.....

- 使用horovod训练的时候，可能会遇到如下的错误：
  - One or more tensors were submitted to be reduced, gathered or broadcasted by subset of ranks and are waiting for remainder of ranks for more than 60 seconds. **This may indicate that different ranks are trying to submit different tensors or that only subset of ranks is submitting tensors, which will cause deadlock**.
  - 这个可能的原因是**某个rank比如rank 0干活慢或者干的比别人多，导致大家都长时间等待他**。
    - Horovod适合的是每个rank/worker训练时的计算图是一样的。
    - Rank 0干的活多一些，但是要注意它多干的事情不能太久。比如对验证集的评估和训练中的checkpoint的保存，如果不可避免这些操作时间比较长，那么workaround就是就所有worker都来做checkpoint保存和/或者验证集评估。
  - 之前还遇到过这个问题是因为TF与horovod，openmpi的版本导致的。
    - 用sagemaker TF2.0 + horovod一直报错，换成Sagemaker TF2.1+horovod就正常了。

Continue.....

- 使用pipe mode + horovod的时候，训练集的channel根据每个训练实例的worker数量来设置，**至少同一个实例上的不同worker进程要消费不同的channel，不同实例上的worker可以消费相同的channel。**
- 评估集的channel设置一个，因为只有horovod master进程评估模型。
- **在使用horovod的时候，如果不同worker的训练集的数据量不均衡，可能会引发问题**，报如下的错误（这是个known issue）：
  - Horovod has been shut down. This was caused by an exception on one of the ranks or an attempt to allreduce, allgather or broadcast a tensor after one of the ranks finished execution. If the shutdown was caused by an exception, you should see the exception.
  - 相对来说，Tensorflow的distribute strategy能比较好的处理每个worker数据量不均衡的情况。

Continue.....

- **因此在使用horovod的时候最好给每个worker基本相同的数据量**：
  - 对于Sagemaker file mode，很容易通过tensorflow的dataset API的shard功能来实现；
  - 对于Sagemaker pipe mode，要自己给每个训练的channel基本等同的数据量，然后在每个训练实例上对同一个channel来做基于tensorflow dataset API的shard功能，目的是让**每个训练实例的每个work都处理几乎相同的数据量，而且处理的是不同的数据。其实就是把手动shard + tensorflow side shard结合起来用。**

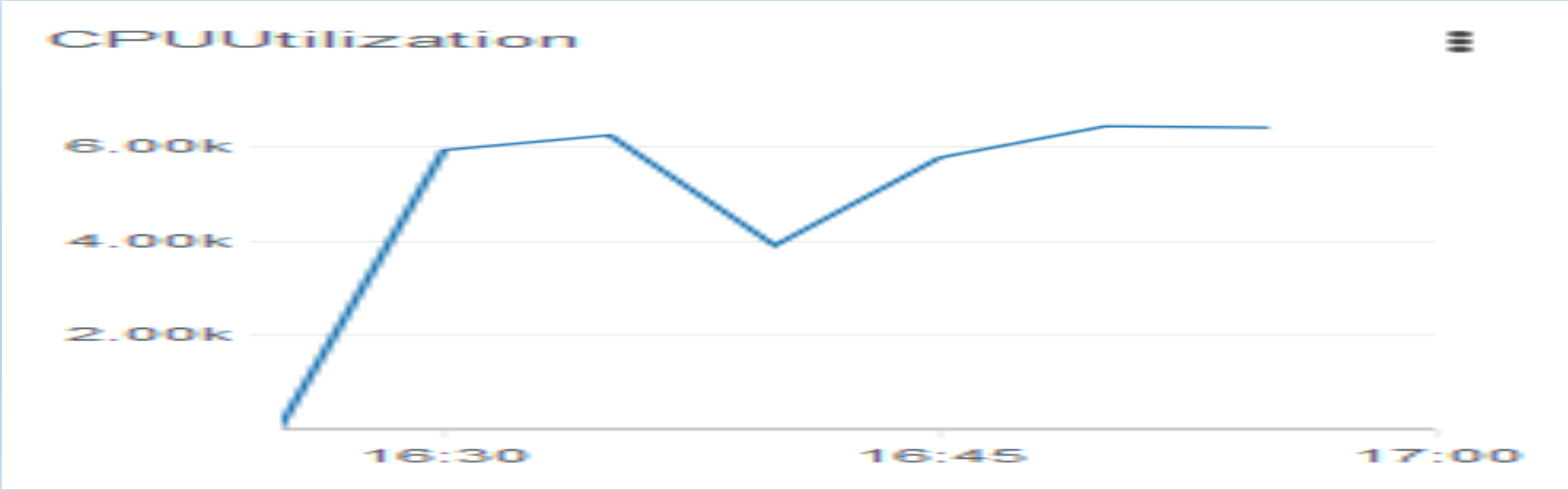
```
        dataset = PipeModeDataset(channel, record_format='TextLine')
        number_host = len(FLAGS.hosts)
        #liangaws: horovod + pipe mode下，如果每个训练实例有多个worker，需要每个worker对应一个不同的channel，因此建议每个
channel中的数据集是提前经过切分好的。只要在多个训练实例上并且每个训练实例是多个worker进程的情况下，才需要对不同训练实例上的同一个
channel的数据做shard。
        if number_host > 1 and hvd.size() > number_host:
            #liangaws: 在Sagemaker horovod方式下，不同训练实例的current-host都是一样的，而sagemaker PS方式下，不同训练实例的
current-host是不一样的。
            #index = FLAGS.hosts.index(FLAGS.current_host)
            index = hvd.rank() // FLAGS.worker_per_host
            dataset = dataset.shard(number_host, index)
```

Continue….

- 上面提到的tensorflow dataset shard是在自己代码里面做的shard，还可以让sagemaker在S3侧自动做shard（通过对每个训练集的channel的S3distributetype设置S3shardbykey），这个功能可以在调用fit API的时候利用sagemaker.session.s3_input API来设置。
  - 如果是file mode + 所有训练channel都设置S3shardbykey，同一个训练实例上的不同worker再调用TF的shard API来进一步shard。
  - 如果是pipe mode + 所有训练channel都设置S3shardbykey，horovod的所有worker都处理不同的数据集一部分，不要在调用TF的shard API了。
- 使用horovod的时候如果worker数量很多，那么学习率会被scaling很大，这样容易造成训练效果不好比如loss比较大，accuracy/AUC比较低。
  - **这个时候要适当减少scaling前的学习率**。
- 可以**使用horovod的一些调优参数来尝试加速效果**。
  - 比如autotune，fusion-threshold，cycle-time等等这样的参数。
  - 参考：https://aws.amazon.com/cn/blogs/machine-learning/reducing-training-time-with-apache-mxnet-and-horovod-on-amazon-sagemaker/

Continue……

- **对于单机多CPU，多机多CPU，单机多GPU卡，多机多GPU卡这些场景的horovod训练，BYOS的代码基本差不多，主要区别就是worker对CPU和GPU的绑定方式不一样。**
- 使用ml.c5.18xlarge(72个VCPU)做单机多CPU horovod训练libsvm格式的数据（并没有设置TF的intra线程池和inter并行度），CPU使用率如下图（差不多能到60K）：



Continue.....

- 基于Sagemaker的内建tensorflow容器的**单机多GPU训练**可以选择的方式：
  - 利用tower方式：
    - 涉及的代码改动类似tower做单机多CPU训练，只是不需要手动设置GPU设备。
    - **Scaling batch size为GPU数量。**
  - 利用原生tensorflow的distributed stragety mirrorstrategy：
    - GPU训练时默认子策略用的是**NcclAllReduce**：

```
mirrored_strategy = tf.distribute.MirroredStrategy()
config = tf.estimator.RunConfig(train_distribute=mirrored_strategy,
                                eval_distribute=mirrored_strategy)
```

    - **Scaling batch size为GPU数量。**
  - 利用Sagemaker集成的horovod方式

Continue….

- Tips:
  - 使用mirrorstrategy for GPU或者tower for GPU训练的时候，如果在tf.ConfigProto中设置log_device_placement=True会出错（用的Sagemaker TF 1.14版本），解决办法有两种：
    - 不要设置log_device_placement参数
    - 设置log_device_placement=True的同时，设置allow_soft_placement=True
  - 其实在用Tensorflow BYOS方式训练过程中，只要出现类似variables或operation放置设备出错，都可以**通过设置allow_soft_placement=True**来尝试解决。
  - 对于**单机单卡训练，直接训练**就可以了。TF缺省只会用实例的单个GPU卡做计算，但是可以使用实例上的所有GPU卡的显存。
    - 这个对于模型太大比如embedding table太大或者网络结构复杂的情况，以至于单个GPU卡显存放不下的情况，同时不想做复杂的模型并行或者embedding table切分的代码修改，可以考虑在多卡的实例上的单卡训练。

多机训练

- 问题：
  - 和单机训练有什么区别吗？
  - 什么时候考虑使用多机训练呢？
  - 多机训练都有哪些方式呢？哪种方法更好呢？
  - 多机训练如何充分利用每个机器上的多个GPU卡？

Continue.....

- 基于Sagemaker内建tensorflow容器的**多机多CPU训练**可以选择的方式：
  - Parameter server分布式训练方式 + 代码不做特殊CPU设备感知的处理。
  - Parameter server分布式训练方式 + 代码对CPU设备感知做处理。
    - 设置Tensorflow的intra op线程池和inter op并行度，配置MKL-DNN的环境变量来修改绑定策略。
  - Parameter server分布式训练方式 + tower方式 + 手动设置CPU device。
    - 注意：这里**需要在tf.ConfigProto中设置allow_soft_placement=True，否则**计算图的operation的设备置放会冲突而失败。
  - horovod分布式训练方式

Continue.....

- Tips：
  - 前面提到的几种多机多CPU分布式训练的方法，需要的代码改动和单机多CPU训练的改动几乎一样。
  - 在**使用parameter server分布式训练方式的时候，并且用的是tf.estimator API来的话，checkpoint的路径必须设置为可以share的比如用S3**，否则训练刚开始就会失败（具体细节请参考本页下面的注释。）
    - 但是**如果是parameter server + tf.keras的组合，checkpoint路径设置为本地路径是可以的。**
    - **而horovod方式的话，checkpoint路径可以设置为本地路径。**
    - **tf.estimator + parameter server的组合下，checkpoint的路径除了可以设置为S3的路径，也可以设置为EFS mapping到容器中的本地路径。**
      - EFS作为一个channel提供给Sagemaker，Sagemaker当前把EFS mapping到了**/opt/ml/input/data/{channel_name}**

Continue…..

- 使用tf.estimator + parameter server的组合，如果模型本身比较大（比如超过2GB），在保存checkpoint到S3的时候可能会出问题：
  - 如果在使用TF S3 file system的时候遇到问题，可以首先尝试设置如下的环境变量：
    - export AWS_REGION=us-east-1 #你准备访问的S3的桶所在的region
    - export S3_ENDPOINT=s3.us-east-1.amazonaws.com #你准备访问的S3的桶的包含region信息的完整的域名
    - export S3_USE_HTTPS=1
    - export S3_VERIFY_SSL=0
    - export S3_REQUEST_TIMEOUT_MSEC=6000000
    - export S3_CONNECT_TIMEOUT_MSEC=6000000
  - **如果发现master一直hung在saving checkpoint to S3，可以尝试下面的workaround：**
    - **使用Sagemaker内建的TF1.15**
    - **或者使用EFS作为share的checkpoint路径。**

Continue….

- 在使用parameter server进行多机训练的时候，可能会出现每个ps上的参数load不均衡的情况（尤其是在有比较大的embedding table变量的时候）：
  - **如何快速知道ps上的参数不均匀**？
    - 通过查看shard的checkpoint的文件大小便可知。
      - 一般每个ps对应一个shard的checkpoint文件。
  - **可以使用partitioner功能来尽量让每个ps的参数均匀分布**，使用如下的代码对你的变量进行wrapper：
    - with tf.variable_scope('deepfm_model', reuse=tf.AUTO_REUSE, partitioner = tf.fixed_size_partitioner(num_shards=len(FLAGS.hosts))):
    - 使用这个方法用内建的TF1.14/TF1.15都能work，但是内建的TF1.13会出问题。

Continue……

- 原生的tensorflow的多机分布式训练策略MultiWorkerMirroredStrategy在Sagemaker中使用时，可能因为版本的关系总报错（Sagemaker内建的TF1.14和multiworkermirrorStrategy配合就有问题）：
  - **使用Sagemaker 内建的TF1.15配合multiworkermirrorstrategy可以训练**（参考如下的代码），只是训练速度和Parameter server方式对比就太慢了：

```
num_cpus = int(os.environ['SM_NUM_CPUS'])
TF_CONFIG = os.environ.get('TF_CONFIG')

if TF_CONFIG and '"master"' in TF_CONFIG:

    os.environ['TF_CONFIG'] = TF_CONFIG.replace('"master"', '"chief"')

    print(os.environ['TF_CONFIG'])

strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()

config = tf.estimator.RunConfig(train_distribute=strategy,
log_step_count_steps=10).replace(session_config =
tf.ConfigProto(allow_soft_placement=True, device_count={'CPU': num_cpus},
intra_op_parallelism_threads=num_cpus, inter_op_parallelism_threads=num_cpus))
```

Continue.....

- 利用parameter server方式训练的时候，TF的实现就是只有master worker会保存checkpoint和对验证集进行验证。因此如果验证集比较大的话，master worker会长时间卡在验证这里。
  - 因此**如果发现master worker卡住了，要看是保存S3 checkpoint的时候卡了还是验证集做评估的时候卡了**。
  - 正常的日志参考如下（使用tf.estimator API的情况）：
    - I0317 13:45:05.493547 140587113436928 basic_session_run_hooks.py:262] loss = 0.69646865, step = 1
    - I0317 13:53:47.497896 140587113436928 basic_session_run_hooks.py:606] **Saving checkpoints for 88 into** s3://liang200/deepfm-dataset-tfrecord-vectorized_map90812013336223344898 75ullCPUbyTowerDropsmaller11110/model.ckpt**.**
    - I0317 13:54:08.917733 140587113436928 evaluation.py:255] **Starting evaluation at** 2020-03-17T13:54:08Z
    - I0317 13:54:09.268356 140587113436928 saver.py:1286] **Restoring parameters from** s3://liang200/deepfm-dataset-tfrecord-vectorized_map90812013336223344898 75ullCPUbyTowerDropsmaller11110/model.ckpt-88
    - I0317 14:03:30.263931 140587113436928 evaluation.py:275] **Finished evaluation at** 2020-03-17-14:03:30
- 利用parameter server训练的时候，使用tf.estimator API，并设置了session config，master worker最后会hung住，具体参考本页注释。

Continue.....

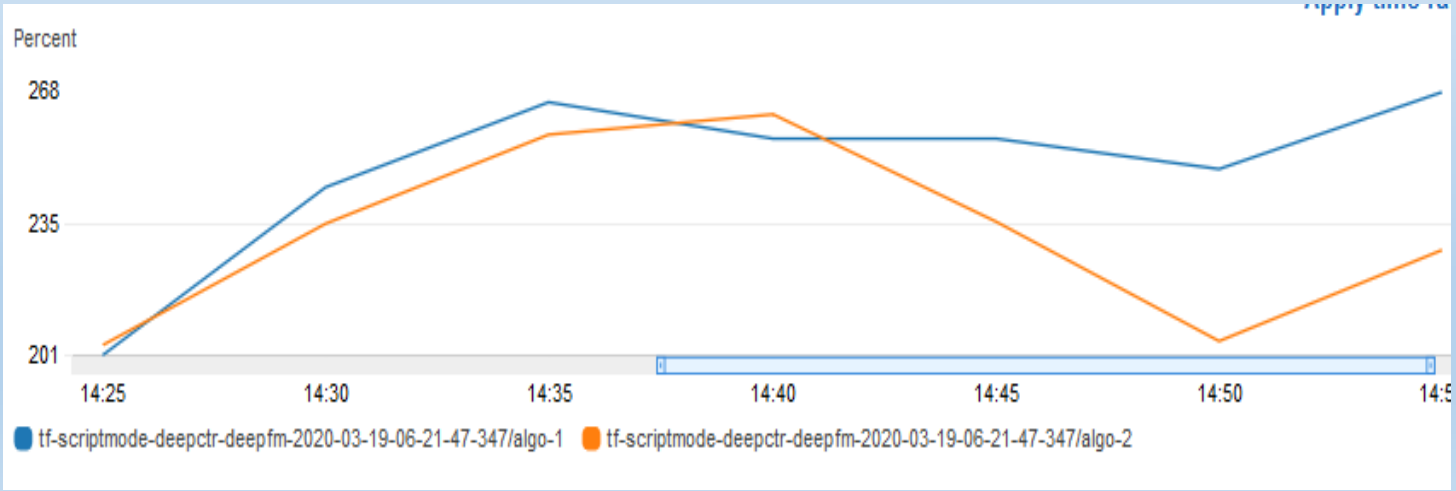
- 基于Sagemaker中的内建tensorflow容器的**多机单GPU卡训练**可以选择的方式：
  - Parameter server分布式训练 ：
    - Sagemaker内建的parameter server训练方式是每个训练实例启动一个parameter server进程和一个worker进程（**每个parameter server只是负责模型参数的一部分**），所以缺省是多机单卡训练的。
    - Sagemaker内建的parameter server训练是采用的异步梯度更新方式，目前没有办法设置为同步更新方式。
      - **为了减少异步更新对训练收敛性的影响，建议减少学习率。**
    - 用这种方式要使用单个实例上的所有GPU卡的话，需要配合上tower来实现。
  - horovod分布式训练：
    - horovod方式的多机单卡的训练，只需要在helper code中调用Sagemaker high level API来设置distributions参数中的processes_per_host为1。

Continue.....

- 基于Sagemaker内建tensorflow容器的**多机多GPU卡训练**可以选择的方式：
  - Parameter server分布式训练 + tower方式：
    - 代码改动基本和单机多卡的tower方式是一样的，同样不需要自己手动设置GPU device。
  - horovod分布式训练：
    - 只需要在helper code中调用Sagemaker high level API的设置distributions参数中的processes_per_host为训练实例的GPU数量。
    - 再次强调：**对于pipe mode方式，horovod设置的每个训练实例的worker数量要和训练的channel数量能匹配上。**
    - Horovod方式训练时，**checkpoint保存路径设置可以是本地路径**。
      - 如果模型比较大而且checkpoint保存相对频繁， checkpoint个数上限也比较大的话，要把**train_volume_size**设置大一些。

Continue….

- 利用2台P3.8xlarge只使用每台实例的3个GPU卡，horovod+libsvm格式+pipe mode+三个训练channel的情况下，且模型比较大（deep_layer = '4096,4096,4096'），GPU使用率还不错（比相同设置的小模型deep_layer = '256,128,64'的GPU使用率高很多），如下图：



训练速度优化

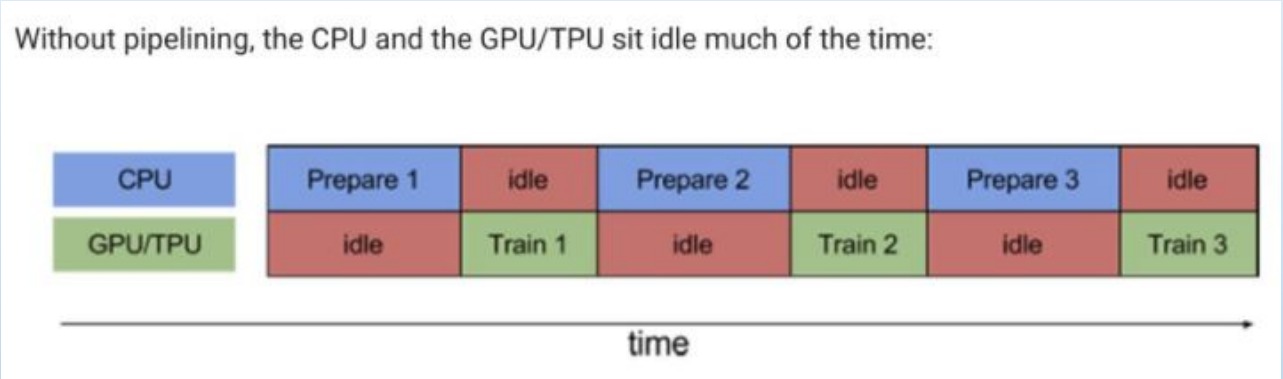
- 为什么要优化训练速度呢？
  - 钱
  - 时间
  - 更快的看到模型效果调优结果
- 什么迹象表明可能需要优化速度？
  - GPU或者CPU使用率
- 如何进行优化呢？有什么建议的方法吗？

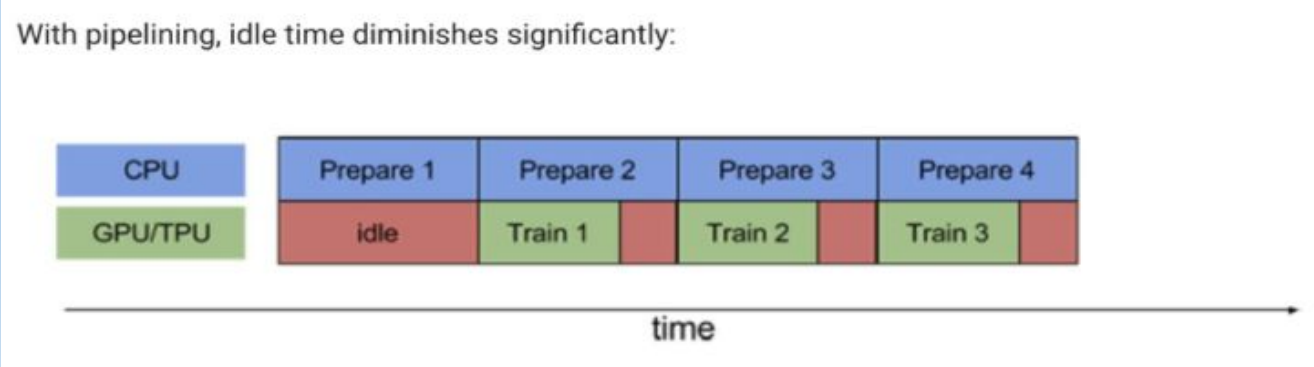
Continue.....

- **影响训练速度的因素是非常多的**：
  - TF data input pipeline的优化
    - cache() API，Prefetch API
    - 数量很多的小文件 vs 数量很少的大文件
  - TF inter_op线程池，TF intra_op线程池，MKLDNN的环境变量的设置
    - 常见的会把他们设置为VCPU数量或者物理core的数量，具体哪种设置好是case by case的。
  - CPU实例or GPU实例，实例类型大小的选择
    - 并不是机型越大，训练速度越快
  - 分布式训练的方式的选择，使用训练实例的数量
    - 常见的是Parameter for CPU， horovod for GPU
    - 训练速度并不是一定随训练实例数量的增加而增加，会有一个极值出现。
  - 模型大小，batch size大小
    - Batch size不同，不管是对于CPU还是GPU训练，速度都会有影响。
  - TF框架的API的特性的使用
    - 是否设置XLA编译计算图
    - 是否disable eager
    - 是否使用了tensorboard callback来频繁的profile，直方图数据收集等等
  - File mode还是pipe mode
  - Sagemaker pipemode的正确使用姿势

Continue.....

- 使用GPU实例训练过程中GPU和CPU的交互：





Continue….

- 如何让input data pipeline更高效呢？
  - 从原始样本抽取出特征和label的函数实现要尽量简单。
  - 提前预取一些原始样本到主存。
  - 减少不必要的disk IO和或networking IO
  - 在主存中缓存经过处理已经提取出的特征和label
  - 减少CPU和GPU之间的拷贝数据次数
  - 让同一个机器上的不同worker处理不同的训练集中的部分数据。
  - 尽量减少数据转换函数的调用次数。
  - ……
- **可喜又可悲**的是Tensorflow提供的dataset 各种transformer API提供了上面所有的功能。

Continue.....

- TF的各种transformation API的**调用顺序**对训练速度的影响很大。

```
    def decode_tfrecord(batch_examples):
        # The feature definition here should BE consistent with LibSVM TO TFRecord process.
        features = tf.parse_example(batch_examples,
                                    features={
                                        "label": tf.FixedLenFeature([], tf.float32),
                                        "ids": tf.FixedLenFeature(dtype=tf.int64, shape=
[FLAGS.field_size]),
                                        "values": tf.FixedLenFeature(dtype=tf.float32, shape=
[FLAGS.field_size])
                                    })

        batch_label = features["label"]
        batch_ids = features["ids"]
        batch_values = features["values"]

        return {"feat_ids": batch_ids, "feat_vals": batch_values}, batch_label
```

```
def decode_libsvm(line):
    columns = tf.string_split([line], ' ')
    labels = tf.string_to_number(columns.values[0], out_type=tf.float32)
    splits = tf.string_split(columns.values[1:], ':')
    id_vals = tf.reshape(splits.values,splits.dense_shape)
    feat_ids, feat_vals = tf.split(id_vals,num_or_size_splits=2,axis=1)
    feat_ids = tf.string_to_number(feat_ids, out_type=tf.int32)
    feat_vals = tf.string_to_number(feat_vals, out_type=tf.float32)
    return {"feat_ids": feat_ids, "feat_vals": feat_vals}, labels
```

```
if FLAGS.pipe_mode == 0:
    dataset = tf.data.Dataset.from_tensor_slices(filenames)
    dataset = dataset.interleave(lambda x:
                                 tf.data.TFRecordDataset(x),
                                 cycle_length=len(filenames), block_length=16,
                                 num_parallel_calls=tf.data.experimental.AUTOTUNE)
    dataset = dataset.batch(batch_size, drop_remainder=True) # Batch size to use
    dataset = dataset.map(decode_tfrecord,
                          num_parallel_calls=tf.data.experimental.AUTOTUNE)
    dataset = dataset.cache()
    if num_epochs > 1:
        dataset = dataset.repeat(num_epochs)
    dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)

    iterator = dataset.make_one_shot_iterator()
    batch_features, batch_labels = iterator.get_next()
    return batch_features, batch_labels

else :
    dataset = PipeModeDataset(channel, record_format='TFRecord')
    dataset = dataset.batch(batch_size, drop_remainder=True)
    dataset = dataset.map(decode_tfrecord,
                          num_parallel_calls=tf.data.experimental.AUTOTUNE)
    dataset = dataset.cache()
    if num_epochs > 1:
        dataset = dataset.repeat(num_epochs)
    dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
    return dataset
```

Continue.....

- Tips：
  - 关于TF dataset 的transformation API的调用顺序最好以测试结果为准，但是有几条是通用的：
    - **Prefetch API在放在transformation API的最后。**
    - **尽量用vectorized map来解析原始样本，也就是先调用batch，再调用map。**
      - 这样map中提供的自定义解析函数比如前面代码的decode_tfrecord就是对一个mini batch的数据进行解析。
      - 先map后batch就是scalar map，自定义解析函数中处理的就是单个样本。
    - **尽量使用cache API来缓存处理完的特征和label。**
      - **cache API要在repeat API之前，否则每个epoch训练都会近线性增加RAM内存的使用。**
      - **如果数据集比较大比如和RAM大小差不多，不要用cache() API。**
    - **TF的dataset API如果从linux的本地文件系统读取文件或者cache数据集到本地文件系统，是否是direct IO？**（具体细节参考本页的注释）
      - **不是Direct IO，因此读写文件都会用到Linux Kernel的page cache。**
      - 使用file mode的时候，需要保证shard到每台训练实例上的数据集大小比单个实例的RAM小；
      - 使用pipe mode的时候，若满足上面这个条件，则可以利用cache API把数据集缓存在用户态内存从第二个epoch开始加速训练。

Continue……

- **当需要TF dataset的cache API和shuffle API同时调用的时候**，可以参考如下的顺序：
  - **create dataset----cache----shuffle--batch—map----repeat----prefetch**
  - 具体细节和原因参考本页的注释
- 对于TF dataset中的transformation API涉及到的并行度以及size相关的参数，**建议使用tf.data.experimental.AUTOTUNE来让TF来自动动态调整作为起点**。
  - 设置了tf.data.experimental.AUTOTUNE，TF相对要保守一点，可以自己手动调整对应的size或者并行度，可能训练速度更好。
- **使用tfrecord格式文件比libsvm格式文件训练效率更高**。
  - （Libsvm格式转换为tfrecord格式的时间 + Tfrecord格式文件训练的时间） vs 直接用libsvm格式文件训练的时间

Continue.....

- 当训练文件的数量比较多的时候，file mode vs pipe mode谁更快？
  - 拿TF的tfrecorddataset API来举例，在其他的dataset API基本一样的前提下，file mode更快。主要的区别就是pipemodedataset API和tfrecorddataset API。
    - tfrecorddataset API可以设置num_parallel_reads来并行读取多个文件的数据，还可以设置buffer_size来优化数据读取。
    - Pipemodedataset API则没有类似上面的参数来加速数据的读取。
      - 也就是说**pipemode更适合读取数量不多，但是每个文件都很大的场景**。
  - **优化建议**：
    - 在训练job外面比如用Sagemaker processing job 来利用serverless spark集群做如下的处理：
      - **把训练数据集的多个文件根据类别拼接为更大的文件**；
      - 把训练集的类别通过过采样来弄的尽量均衡；
      - 对类别均衡的样本集进行Shuffle并保存到S3；
      - 最后交给Sagemaker 来训练。

Continue.....

- 经过了尽可能的input data pipeline优化后，CPU或者GPU设备的训练使用率仍然不高？
  - **很可能的原因就是模型本身太小**（尤其是对于GPU使用率低的情况），单个step的计算很快就结束。也就是单个step的计算相对于prepare batch来说太快。
    - 这个时候**把模型本身变深或者变宽能看到GPU使用率的明显提升**。当然要综合考虑模型评价效果。
    - **不要为了提升GPU使用率而刻意把模型变大，这个是需要权衡的。一般来说模型大小满足业务需求指标就可以。**
  - **对于纯CPU训练方式来说，更建议使用前面提到的使用TF的intra和inter并行度设置配合MKL-DNN的绑定策略环境变量来调整**。
  - 适当的提高mini batch size也能一定程度提升GPU或者CPU使用率，但是mini batch size和learning rate以及模型评价效果有很大关系，要慎重调整或者利用超参数调优。

Continue….

- 混合精度训练：
  - **用GPU在TF上进行自动混合精度训练，能在模型精度几乎不怎么降低的情况下显著提升训练速度**。
  - P3实例的GPU是支持tensor core的，而tensor core就是用来做混合精度计算的。为了能activate tensor core来计算，在TF中需要：
    - **使用能支持自动开启混合精度训练的框架的版本**，比如tensorflow从1.14开始支持，只需要用下面的语句来wrapper你原本的optimizer：
      - tf.train.experimental.enable_mixed_precision_graph_rewrite(optimizer)
    - 神经网络中的一些与size有关的参数，比如对于全连接层的batch size， input size，output size都需要是8的倍数，对于卷积层的input channel，output channel（就是filter个数）需要是8的倍数，对于RNN的batch size需要是8的倍数。
  - **在模型复杂度比较低，GPU使用率比较少的时候，混合精度训练的优势体现不出来**。因为那个时候在CPU上准备数据占每个step训练时间的大部头。
  - **当模型复杂度比较高，GPU使用率高的时候，混合精度训练提速就很明显**。（可以参考本页注释中的日志）

总结

总结

- 在Sagemaker中进行训练的话，**建议的训练尝试顺序**：
  - 先单机多CPU或者单机单GPU来训练，如果CPU或者GPU卡的使用率很高比如快到90%，尝试下一步。
  - 单机更多的CPU或者单机更多的GPU卡（当前AWS的GPU实例最多是8卡）来训练，继续观察CPU和GPU卡的使用率是否很高比如达到单机最大的CPU数量或者单机最大的GPU卡数量，如果是尝试下一步。
  - 多机多CPU或者多机多GPU卡的分布式训练：
    - 不管使用PS还是horovod方式，都需要修改一些代码，而且对于tensorflow三种不同的API（session-based API，tf.estimator，tf.keras）修改的方法都不太一样，根据实际情况来进行选择使用哪一种。
    - **PS和horovod哪种训练速度一定快呢？不一定，所以有时间和成本的话，都可以尝试一下。**

Continue….

- 简单说就是：
  - **先scaling up再scaling out；**
  - **scaling之前查看GPU或者CPU的使用率，先优化数据预处理pipeline，让单机的CPU和GPU处理能在时间上overlap从而真正并行算力；**
  - **如果在优化之后GPU利用率实在不怎么高，考虑尝试使用CPU训练。**
    - **太多次的案例用CPU训练比GPU训练的总的训练时间又短又省钱。**