我不太懂BERT系列——BERT预训练实操总结

原创 邱震宇 AINLP 昨天

作者：邱震宇（华泰证券股份有限公司 算法工程师）
知乎专栏：我的ai之路

通过本文章，你可以了解以下内容：

了解bert预训练会遇到的坑，包括但不限于数据预处理的正确姿势、数据预处理的高效实现、bert单机多卡分布式训练的基本实现，以及如何debug并提升使用单机多卡来进行深度学习训练的性能瓶颈。

本文篇幅有点长，大家可以就自己感兴趣的点挑选章节阅读。

> 纸上得来终觉浅，绝知此事要躬行。——陆游

近期在做一件我一直很想尝试的事情——BERT预训练。之前由于硬件条件和数据量的限制，一直没有机会。这次，借助我们公司强大的AI平台能力以及长时间积累到的大量金融新闻数据，我终于从头到尾实践了一番，可以说是收货颇丰。事实上，在开始做预训练之前，我是信心很足的，之前看了很多预训练的论文，包括BERT的原始论文、roberta以及其他一些变种bert的论文，总结了一些预训练的细节以及注意点，感觉只

要有GPU和大量的语料，很快就可以训起来了。然而，现实总是很残酷，结果证明我还是太天真了。在整个任务阶段，遇到了不少工程实现问题，有性能方面的，也有数据处理方面的。通过这次BERT预训练实践，我也提升了自己在工程方面的能力。本文就详细地总结一下我在做BERT预训练任务中的心得体会。需要注意的是，本文着重描述实践方面的内容，原理内容点到为止，不过我会将相关参考链接贴上供大家深入研究。

本文参考预训练的工作流，主要分为以下三个内容：

1、数据预处理以及训练数据生成。

2、预训练性能优化。

3、预训练效果调优。

另外说一下我使用的主要硬件配置：4*V100(16G)，8核CPU，128G内存。使用的开发框架是tensorflow，借鉴了谷歌官方的bert项目

google-research/bert github.com

，以及brightmart的roberta-chinese项目

brightmart/roberta_zh github.com

数据预处理

我这次使用的训练数据主要是金融领域里面的新闻、研报以及部分公告。之前，熵简科技已经发布了一个金融领域下的预训练模型，具体可参考：

李渔：熵简科技 AI Lab 开源金融领域中文预训练语言模型 FinBERT
zhuanlan.zhihu.com

我这次做的工作与他们的比较相似，不过由于缺少标注人员，我们没有引入监督训练阶段，还是采用经典的roberta训练方式，即去除next sentence prediction任务后的masked_language_model任务。整体的新闻document数据量在五百万条左右，数据量还是很可观的。当然，在将数据应用到训练任务前，还有很多预处理工作要做，包括数据清洗、数据分块、训练数据生成。

数据清洗

由于大部分的新闻数据均是从各大新闻平台爬取的，因此其中存在很多HTML标签字符。本阶段主要工作就是将这些无意义字符过滤掉。常用的方法就是使用正则表达式，将</>包住的字符串过滤掉。然而，这种方式会有一些case没办法cover。比如一些HTML标签中包含比较复杂的css样式。经过一番搜索，发现了一个比较好的处理方法，即借用Beautifulshop中的html格式化方法，将style属性也过滤掉，代码如下：

```
def remove_html_tags(html):
    soup = BeautifulSoup(html,"html.parser")
    for s in soup(['script', 'style']):
        s.decompose()
    return ' '.join(soup.stripped_strings)
```

其中，decompose方法递归删除符合条件的所有标签。

然而，通过上述方式还是会有一些乱七八糟的字符串序列无法过滤，例如md5哈希后的字符串下面这种：

> 3frfd44ee233ddfs/

这种情况下，为了保证训练数据的质量，可以使用一些规则比如连续出现n次非中文字符的文本就直接过滤掉。

另外，文本还会存在一些非法的unicode字符，比如\ue000-\uf8ff范围的字符。可以使用正则表达式来进行过滤，也可以使用unicodedata.normalize来进行标准化，不过，我在实践的过程中，发现有些unicode字符还是没办法通过normalize来转换，只能使用正则表达式来过滤。

数据分块

这部分内容其实没有什么技术要点，之所以要做这一步是为了在后续模型训练的时候使用shard来优化模型数据读取的性能。简单来说，就是将原始的数据按照一定份数进行均分，将大文件拆成一些较小的文件，这样做还可以在一定程度上提升后续制作训练数据流程的性能，因为bert的训练数据制作涉及到很多循环操作，具体的内容将在下一小节进行描述。

训练数据生成

这部分内容是整个数据预处理中最重要的一环，也是我花费时间最多的。碰到的坑主要有两大点：基于中文分词的WWM训练数据制作以及训练数据制作的性能优化。下面分别详细说明。

1、关于第一点，在很多论文里也说明了（例如arxiv.org/pdf/1906.0810），对于中文语料来说，英文的wordpiece切词方式几乎等同于按照字符分词，后续做MLM任务的时候，只会随机mask单个中文字符。然而，中文里面，很多语义都是通过词来表达，因此对于中文的MLM任务，需要引入中文词汇的信息。这就是引入WWM（whole word mask）机制的原因。具体的做法为先使用中文的分词工具将文本进行分词，然后在制作mlm训练任务的训练数据时，还是先用wordpiece进行切词，并对该切词结果进行随机mask，当碰到一个完整单词中的字符时，把从属于该单词中的所有字符都mask掉。但是需要注意一个单词中替换的字符并不一致，即一个单词中的每个字符还是按照80%的概率用[MASK]替换，否则按照90%的概率不变，10%的概率用随机字符替换。整个句子的替换比例还是不变，按照原来的做法。具体WWM举例如下：

原始文本	使用语言模型来预测下一个词的probability。
分词文本	使用 语言 模型 来 预测 下 一个 词 的 probability 。
全 词 mask 结果	使 用 语 言 [MASK] 型 来 [MASK] 我 下 一 个 词 的 [MASK] [MASK] ##lity 。

具体的说明可以参考

ymcui/Chinese-BERT-wwm github.com

上述原理很容易理解，然而在实现的时候还是有一些细节需要注意。具体实现时，首先需要在中文分词的时候，将每个字符与其所属的词的对应的关系使用一个index列表或者字典来记录下来，类似于得到[[c1,c2],[c3,c4,c5]]；然后在做mask替换的时候,对前述的列表进行shuffle，然后对子列表中的字符进行mask操作，直到整个句子的mask比例达到设定值。brightmart的开源实现就是上述思想。但是，我在实践过程中，发现其代码会有一个问题：

```
masked_token = tokens[index][2:] if len(re.findall('##[\u4E00-\u9FA5]', to
```

即在中文分词的时候，为了标识一个词的非首字符，会在该字符前面增加'##'，而这也是英文wordpiece切词时对子词的操作。当进行mask的时候，如果碰到保持原子词的情况下，代码中会对中文字符进行'##'的过滤处理，防止其在词表中找不到。然而，有

些英文字符串会和其他中文字符一起构成一个词，此时该英文字符串不会在上述的过滤范围内，最后导致报词找不到的错误。举例如下：

> bert分词：'顺', '利', '的', '无', '创', 'dna'
> jieba分词：'顺', '##利', '的', '无', '##创', '##dna'

dna与无创是属于一个词的，但是##dna并不在bert词表中。具体可见

https://github.com/brightmart/roberta_zh/issues/70 github.com

那么有什么办法可以规避上述问题呢？大概有两种做法，一种是参考苏神的bert4keras的方式，先进行中文分词，然后shuffle词列表，之后对每个中文词进行wordpiece切词，最后对切词后的子词列表进行mask操作。还有一种是可以用不同的字符来表示一个词的非首字符（例如@@），在整合wordpiece切词结果和中文分词结果时，将##和@@开头的字符都进行与所属词的整合操作。在mask的时候，只要注意处理带@@开头的子词即可。这样就可以实现中英文混合的WWM。

2、另外，根据roberta的论文所述，建议使用dynamic mask代替原始的static mask，简单来说就是对于一个训练样本，每个epoch都重新随机mask生成新的训练数据，这样可以促进模型对数据的学习更加透彻。然而，现实条件下，训练时动态生成新数据不太现实，因此有一个折中的办法就是在训练数据生成阶段就将每个样本复制N（一般这个数值叫做dup_factor）遍，每个副本都用重新mask生成，这样就相当于对原始数据进行了扩充，间接实现了dynamic mask。

3、此外，我还尝试引入了一些金融领域的一些知识。我们之前维护了一个国内国外的一个公司实体词的知识库，包括公司全称、简称以及股票代码。这些我都作为中文分词的增强词库，配置到了中文分词工具中，期望能够为模型引入金融领域的结构化信息。

数据生成性能优化

经过上述改造后，整个代码已经可以正确无误生成bert训练数据了。不过在我以为可以端着茶安心等待数据生成的时候，却又碰到个大问题：数据生成效率太低。按照当前的执行效率，在dup_factor为5的情况下，处理30万篇文档，预估要半天时间，也就是说我要花10天左右的时间才能生成所有数据。。。这个效率是我不能接受的。因此我详细得分析了原始的数据生成代码，大概总结了几处性能瓶颈的地方：

1、原始代码中，对于原始数据格式的要求是每个文章先分句，然后每个句子一行，文章与文章之间使用空行来分隔。而在制作训练数据时，又需要通过循环读取每行文本，

将同一篇文章的句子进行聚合，代码如下：

```
for input_file in input_files:
        with tf.gfile.GFile(input_file, "r") as reader:
            while True:
                line = tokenization.convert_to_unicode(reader.readline().r
                if not line:
                    break
                line = line.strip()

                # Empty lines are used as document delimiters
                if not line:
                    all_documents.append([])
                tokens = tokenizer.tokenize(line)
                if tokens:
                    all_documents[-1].append(tokens)
```

这种方式相当于会对大数据量的列表进行循环，效率比较低。因此，我建议可以在前面做数据清洗的时候就进行分句，然后将一篇文章转换为句子列表，最后整体外面再包一层列表，中间临时保存文件可以保存成json文件。上述代码即可直接用json的load来代替。

2、将每个文章句子列表整合后，就会对文档列表循环，执行wwm的核心逻辑。这里还是需要有大量的循环操作，效率还是非常低的。此时，要优化性能的话，可以引入python的多进程，跑并发任务。我的CPU是8核的，因此并发数可以设为8。具体多进程实现可以参考苏神的bert4keras中的实现：

```
def parallel_apply(
    func,
    iterable,
    workers,
    max_queue_size,
    callback=None,
    dummy=False,
    random_seeds=True
):
    """多进程或多线程地将func应用到iterable的每个元素中。
    注意这个apply是异步且无序的，也就是说依次输入a,b,c，但是
    输出可能是func(c), func(a), func(b)。
    参数：
```

```
        callback: 处理单个输出的回调函数;
        dummy: False是多进程/线性，True则是多线程/线性;
        random_seeds: 每个进程的随机种子。
    """
```

大概原理就是使用python的Pool机制，将每个样本的处理操作封装成一个work_step。另外定义两个queue，一个用于存放输入数据，一个用于存放work_step输出的结果。最后可以定义一些后处理操作比如保存到最终结果列表等。

通过上述多进程改造后，性能提升非常明显，提升率足有200%-300%。同等情况下，处理30万篇文档，现在只要1.5小时。

3、另外一个可以使用多进程优化的地方在write_instance_to_example_files。原始代码中，它主要是遍历每个instance，然后生成tf的Example写到tfrecord中。这里的循环也可以使用多进程来改造优化，不过有一点需要注意，即将tf.Example写到tfrecord的操作最好放在后处理的function中，如果放在work_step中，会导致并发执行的时候，写文件紊乱，最后生成tfrecord会格式错误。通过本步骤的优化，同样能够让性能提升200%。结合上述步骤，处理30万篇文档，现在只要50分钟左右。

4、原始代码使用的tokenization也是一个可以优化性能的点。我使用了huggingface的tokenizer代替了原始的tokenization，它是有ruby开发的一个高性能的切词工具，里面内置了bert的wordpiece分词模型，相对于原始python实现的分词方法，它在性能上能够带来20-30%左右的提升。

5、中文分句工具方面也是一个可以关注的点。原本我是准备用百度的lac来做分词，但是发现它的执行效率还是比不上结巴分词。另外，我在github上还发现了结巴分词的性能提升版fast_jieba，它是用C++来重新实现的，因此效率上更加高效。最后该项改造能够为整体性能带来10%左右的提升。

最终，通过上述的努力，我将整体的训练数据生成流程由10天缩短到了1-2天，基本上满足了我对性能的需求。

另外，如果你们的内存不是很大的话，还可以在节省内存损耗方面进行优化。原始代码对于大数据量来说，内存的需求量是很大的，因此秉持着能省就省的原则，我尝试过以下几个改造方法：

1、对于循环操作，我改用yield生成器，即用到该样本的时候才会把它读到内存。然而，这种方式相当于时间换空间，执行效率远远不如预期，**因此最后没有采用。**

2、使用recordclass方式代替所有的orderdict，或者类实例。recordclass相对于后面两种形式，能够占用更少的空间。具体可参考

Jackpop：从青铜到王者，一文教你节省90%内存占用
zhuanlan.zhihu.com

3、最后这个改造影响比较小，不过我个人比较推荐。原始bert输入数据包括input_mask，它是一个长度为max_seq_len的int64列表。其实它可以用每个sequence的真实长度来代替。这样相当于用一个int64的值节省了max_seq_len个int64的空间。我们只需要对bert的modeling.py进行小改造，在里面增加一行代码tf.sequence_mask，即可在训练时得到input_mask，这个改造还可以节省硬盘空间，用tensorflow的op进行操作也不会有太大性能上的损失。

预训练性能优化

当前开源的中文通用BERT模型，大多是使用谷歌的TPU训练的，这种情况下，一般是不需要考虑性能调优的工作，你尽可以将batch size设为很大的值，然后使用LAMB优化器来加速训练收敛。TPU的显存一般至少得有128G，而且针对深度学习训练有专门的优化。然而，对于我们公司场景来说，使用TPU不现实。一个是成本太高，另一个是我们公司的数据属于隐私数据，是不能随意外传的，因此只能使用传统的GPU来训练。我参考了英伟达的一些深度学习高性能训练的wiki，从以下几个角度分别对bert的训练过程进行优化，最终目标都是希望能尽量缩短bert训练的时间。详细可以参考

NVIDIA Deep Learning Performance
docs.nvidia.com

> 如果你能直接上TPU的话，下面的内容你可以不看了。

多卡训练

首先是利用多卡进行多卡分布式训练。我们公司的平台是能够提供虚拟化的镜像资源来做深度学习训练，因此对于分配给我的镜像来说，相当于是单机多卡的环境，因此下面所说的都是单机多卡的技术，至于多机多卡的实践，暂时没有涉猎。

我使用的是horovod+tensorflow的搭配，其中horovod（horovod/horovod）是uber开源的一款支持分布式深度学习训练的框架，它的最大优点就是能够与各种不同的深度学习训练框架无缝对接，包括pytorch，tensorflow等，而且它对于单机模型训练

的代码改造量不会太大，很适合对底层开发不太深入的算法人员（比如我）。具体的训练改造在其examples上都可以找到，主要注意以下几点：

1、optimization优化器中使用horovod本身的Distributedoptimizer

2、train的主逻辑代码中，记得进行horovod的初始化，根据当前的GPU卡数量，设定真实的训练步数。另外，如果使用esitmator的话，需要添加一个BroadcastGlobalVariablesHook。这个hook很重要，因为它会在初始训练或者每次加载模型参数的时候，让在GPU设备号为0的参数分配到其他设备上，保证大家都是用一套参数。

除了上述注意点外，还有一些细节配置就不多提了，大家可以参考github上的实例代码。

当我按照github上的readme进行安装和改造后，虽然成功将代码跑起来了，模型训练的loss变化也没问题，然而我却发现GPU的使用率特别低，六张卡的使用率经常处于20-30%左右。最后算下来，居然跟用单卡训练的效率差不多。之后，经过很长很长时间的不断摸索和尝试，发现了很多可以提升性能的地方，下面我会逐个列出，希望能给其他同学在做性能瓶颈debug时作为参考。

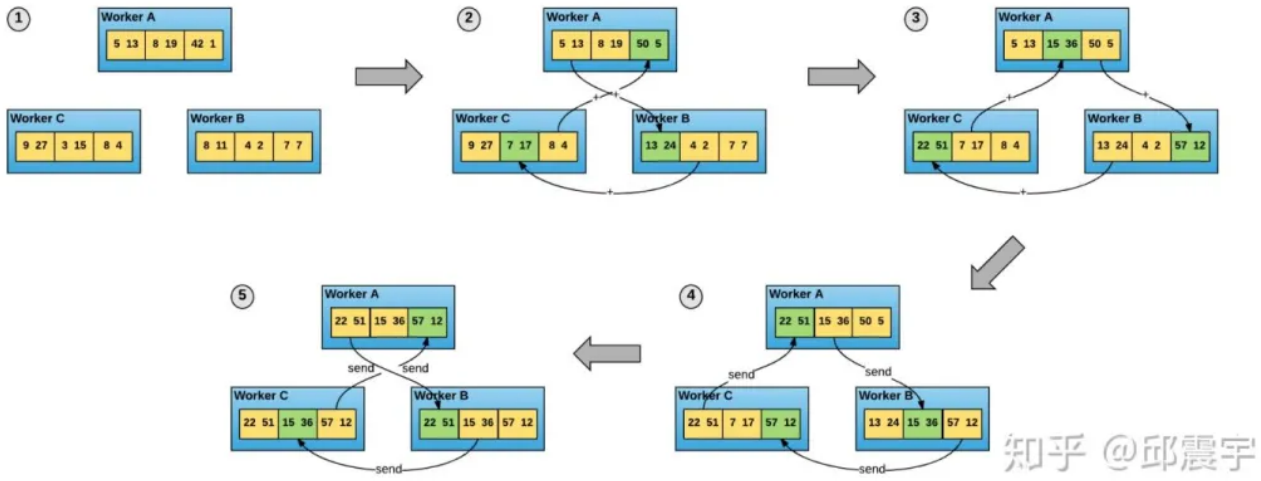
正确安装horovod

有的同学会问，安装horovod不是只要按照它readme上的指示就可以了，很简单啊，它的坑在哪里呢？我遇到的坑是**安装horovod时并没有与我的NCCL库进行链接**。NCCL是NVIDIA开发的一款用于GPU之间通信的库。它本身支持了多种集合通信原语（例如allreduce），它可以帮助我们在不同GPU设备进程之间进行参数的分发、聚合等。

为什么没有了它的帮助，horovod就没办法发挥理想中的效果呢？这个就要简单说一下分布式训练中的通信拓扑问题。众所周知，分布式训练牵扯到很多细节，其中，由于分布式训练利用多个工作节点同时训练、相互合作来加速学习过程，那么通信就成为训练中不可缺少的环节。由于网络传输速度往往跟不上设备内部的数据计算和传输性能，因此通信经常成为分布式系统的瓶颈，尤其是大规模的深度学习训练，要传输大量的权重参数、计算中间结果等。因此，需要谨慎设计多工作节点的通信拓扑，使得节点间的通信能够尽量高效。比较经典的结构是AllReduce，可以使用MPI(消息通信接口)来实现。

> 常见的MPI实现有open MPI，这也是horovod需要安装的依赖包

它主要负责多个设备进程间的信息同步，支持所有符合Reduce规则的运算，例如求和、平均、求最大最小值等，而分布式机器学习中的基本模型聚合方法主要就是加和与平均，因此正好使用AllReduce逻辑来处理。当然AllReduce本身是一种思想，它有不同的实现架构。而horovod主要实现的是百度在2017年提出的Ring-AllReduce算法，具体图例如下：



它主要实现原则是每个worker节点只会与它的左节点和右节点进行通信，其中每个节点上会先开辟一个buffer存储要通信的数据，每次通信传输的都是buffer中的某块数据。假设一共有N个节点，那么每个节点会与它的左右邻居节点通信2*（N-1）次。在第一轮的N-1次通信时，每个节点会把接收到的数据库与对应index上的数据快进行相加（图上1-3），这个阶段一般叫scatter-reduce；在第二轮的N-1次通信时，每个节点会将收到的数据块直接替换掉对应index的数据块（图上4-5），这个阶段一般叫allgather，最后所有 worker节点都会得到完整的融合后的数据。

我们可以使用MPI来实现上述的通信机制，然而MPI的一些开源实现（比如open MPI）都是适配通用的硬件设备，并没有专门针对GPU设备进行性能的调优。而NCCL是NVIDIA专门为GPU之间的通信而设计，它的通信原语与MPI几乎一样，同时它针对GPU通信做了很多优化，因此如果你有多个GPU设备的话，使用它是比较适合的。

那么怎么判断你的horovod是否正确链接了NCCL呢？答案是使用horovodrun --check-build命令，如果安装好的话，应该会显示以下结果：

```
Available Frameworks:
    [X] TensorFlow
    [ ] PyTorch
    [ ] MXNet

Available Controllers:
    [X] MPI
    [X] Gloo

Available Tensor Operations:
    [X] NCCL
    [ ] DDL
    [ ] CCL
    [X] MPI
    [X] Gloo
```


有哪些原因会导致没有正确链接NCCL呢?

1、NCCL库本身没有安装好。这个可以直接参考英伟达官网的NCCL安装指导，一般有RPM安装、tar包手动安装（此时需要手动配置环境变量，将nccl.h放到LD_LIBRARY_PATH中，以及配置nccl的so库）。是否安装好NCCL，可以参考NVIDIA的另一个nccl-test项目github.com/NVIDIA/nccl-。里面的测试脚本会构造一些数据包，让NCCL进行GPU之间的数据传输，测试连通性以及通信的带宽。

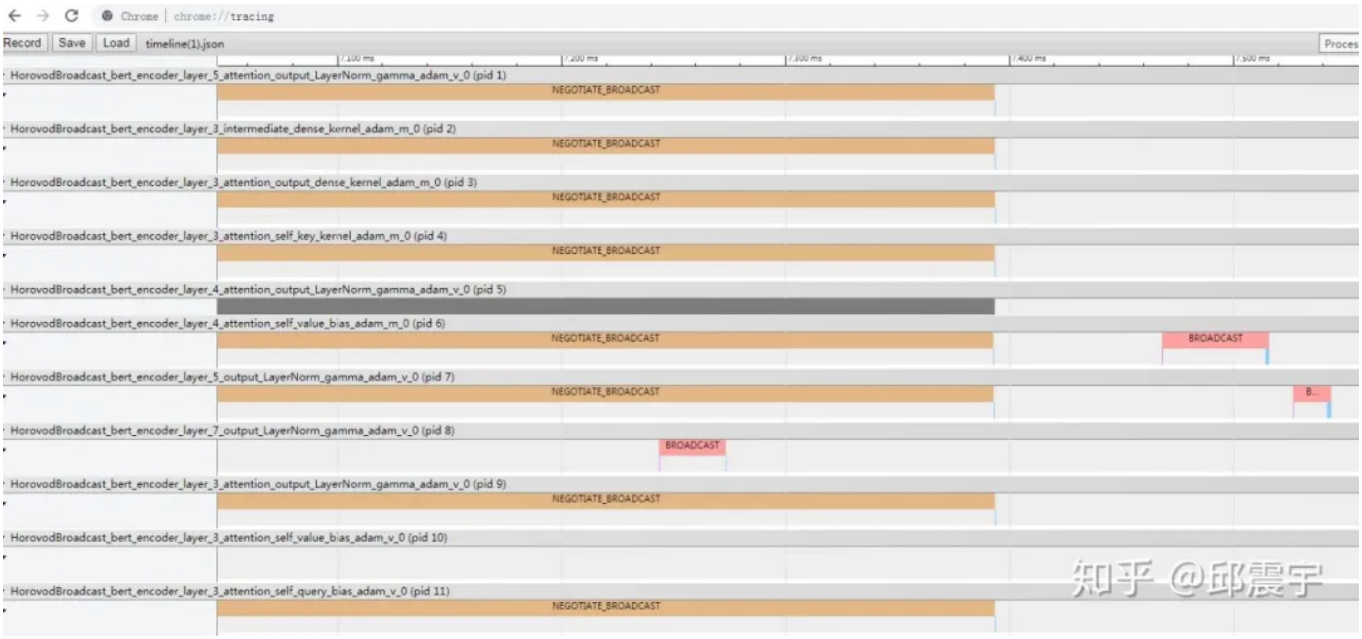
2、如果是安装最新版本的horovod（0.20以上版本），那么可以使用github上的pip install 来安装，在安装的过程中，会出现链接NCCL的日志。然而，horovod-0.20以上版本要求GCC的版本比较高，而我们平台的GCC版本默认是4.8，因此编译的时候会报错。而在内网离线的情况下，升级GCC的代价是很高的。因此我选择了安装horovod-0.19版本，然而这个版本如果要安装链接NCCL的horovod，需要显示的指明安装的依赖条件：

```
HOROVOD_GPU_OPERATIONS=NCCL
HOROVOD_NCCL_INCLUDE=/usr/local/nccl_2.8.3-1+cuda10.0_x86_64/include
HOROVOD_NCCL_LIB=/usr/local/nccl_2.8.3-1+cuda10.0_x86_64/lib
HOROVOD_GPU_ALLREDUCE=NCCL HOROVOD_GPU_BROADCAST=NCCL pip install --no-cache-dir horovod-0.19.0.tar.gz
```

比较重要的是**HOROVOD_GPU_ALLREDUCE=NCCL**和**HOROVOD_GPU_BROADCAST=NCCL**，用于指定AllReduce和Broadcast操作都是用NCCL来实现，否则默认是用MPI来实现。

当然，我们还可以通过horovod的timeline来定位哪些环境是性能的瓶颈。在执行mpirun的时候使用-x指定timeline的路径-x

HOROVOD_TIMELINE=timeline.json。生成的json可以使用chrome的tracing来可视化展示，如下所示：



上述图中，有些空白的地方并非是没有进程在操作，而是有些操作不会在horovod的timeline中显示（例如tensorflow本身的一些tensor操作），如果想看的话，可以使用tensorflow中的timeline来实现，同样其生成的日志文件也可以用Chrome的tracing来可视化。

另外，简单介绍一下上述图中的一些操作阶段，参考：

Analyze Performance horovod.readthedocs.io

1、Negotiation阶段，此时所有工作节点会向工作节点0发送信号，表示已经准备好进行tensor的整合。

2、Processing阶段，此阶段是tensor整合阶段，可以分为几个子流程，下面介绍几个常见的子进程。 WAIT_FOR_DATA 表明当前等待每个GPU设备结束计算，准备好allreduce、allgather、或者boardcast操作的输入数据。NCCL_ALLREDUCE 表示使用NCCL做allreduce，如果没有装nccl，则会使用MPI_ALLREDUCE。

了解你的GPU底层

这里并不是说要对GPU底层硬件进行很深入的研究，而是要知道GPU设备间的拓扑是什么样的。在某些拓扑下，多卡训练反而会降低训练的效率。我就是吃了这个亏。开头也说了，我使用的是虚拟化后的镜像环境，虽说逻辑是单机模式，但是GPU物理上不一定是在一个服务器。一开始，我六张卡全部用上的时候，反而还不如用两张卡的性能。后来我使用**nvidia-smi topo --matrix** 命令，得到我的拓扑如下：

```
        GPU0    GPU1    GPU2    GPU3    GPU4    GPU5    CPU Affinity
GPU0     X      PIX     NODE    NODE    NODE    NODE    12-23,36-47
GPU1    PIX      X      NODE    NODE    NODE    NODE    12-23,36-47
GPU2    NODE    NODE     X      PIX     PIX     PIX     12-23,36-47
GPU3    NODE    NODE    PIX      X      PIX     PIX     12-23,36-47
GPU4    NODE    NODE    PIX     PIX      X      PIX     12-23,36-47
GPU5    NODE    NODE    PIX     PIX     PIX      X      12-23,36-47

Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB  = Connection traversing multiple PCIe switches (without traversing the PCIe Host Bridge)
  PIX  = Connection traversing a single PCIe switch
  NV#  = Connection traversing a bonded set of # NVLinks
```


上图下方显示了不同标记表示的具体GPU连接形式。列表越靠下的，数据通信效率越高。其中，GPU0和GPU1以及GPU2-5分别通过PCI总线连接，说明0-1两张卡在物理上是在一台机器上，而2-5是在另一台机器上。我使用了nccl-test分别对{0,1}以及{2,3,4,5}两组设备进行带宽测试，发现基本都能达到15-20gps。然而，当我使用六张卡测的时候，发现只有1gps左右。说明分属两台服务器的卡间进行通信的时候，存在巨大的时延。在没办法修改底层拓扑的情况下，我最后只能选择使用{2,3,4,5}4卡的组合来做最终训练。

进一步优化数据传输的性能损耗

如果你觉得你的性能损耗在数据传输通信方面还是太高，那么可以尝试以下改造：

1、如果你的timeline中的Allgather操作阶段时间花费较长，那么可以检查一下你传输的tensor中的non-zero元素是否占大部分，如果是，那么你可以在DistributedOptimizer中设置参数sparse_as_dense=True，这样可以提升数据传输的效率。原因在于Allgather表示数据收集到一个主woker上，而allreduce包含Allgather的操作，然而allreduce并不支持sparse_tensor。假使我们的tensor中大部分元素都是非零的，此时将其转化为dense tensor的成本会非常小，此时就可以使用allreduce操作直接作用于allgather。这种改造大概能提升30%左右的性能。详见

https://github.com/horovod/horovod/issues/679 github.com

2、可以尝试对传输的tensor进行fp16的精度压缩。具体的操作方式为在DistributedOptimizer中设置参数compression=hvd.Compression.fp16。这种方式能够大幅提升GPU间通信效率。不过，要明确fp16 compress只针对allreduce操作阶段。它能够将整体的训练性能提升50%。

对于horovod，除了上述优化改造外，还有很多改造的尝试可能会有用，但最后由于不适用我的场景，因此没有去深究，下面简单列一下，方便感兴趣的同学自行研究。

1、如果你的Allreduce的操作时延比较长，可以参考

Why NEGOTIATE_ALLREDUCE is much longer in TensorFlow comparing to PyTorch? · Issue #1454 · horovod/horovod github.com

看看是否是参数HOROVOD_CYCLE_TIME设置的问题，原因在于horovod的RunLoopOnce()方法中，有个sleep方法，其设置为5ms - A，其中A为一个loop的执行时间，如果的A比较短，可以尝试在mpirun的时候使用-x将HOROVOD_CYCLE_TIME设为短一点。

2、horovod还支持tensor fusion。NCCL中，对于大tensor的allreduce操作比对小tensor的allreduce操作效率更高。因此如果通信中，存在大量的小tensor，通常建议先进性tensor fusion，再做allreduce。由于该内容我并没有使用，所以感兴趣的同学可以看一下horovod的论文研究：

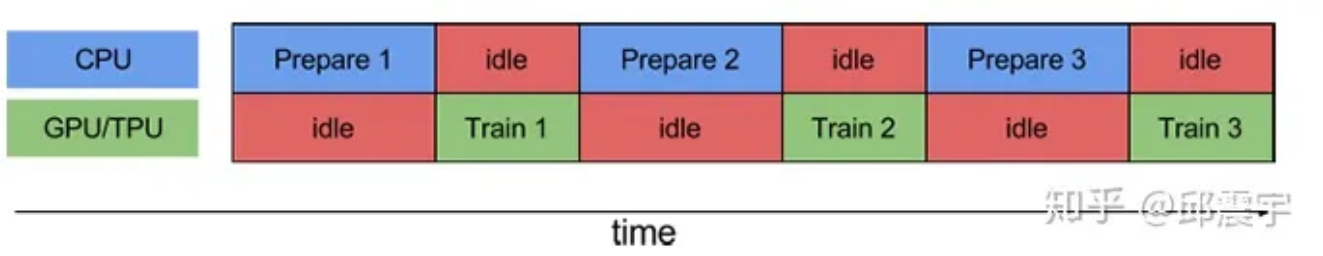
https://arxiv.org/abs/1802.05799 arxiv.org

3、其实horovod还有很多优化的参数可以提升性能，它支持使用类似于自动超参数寻优的功能自动调参，具体为mpirun时，使用-x指定HOROVOD_AUTOTUNE=1。当

然，这种方式会减慢训练速度。可以先用小部分的数据先搜寻一组较优的参数组合，然后再用这个组合进行全量训练。

优化改造input_fn

在深度学习训练过程中，有一部分性能瓶颈来自于数据的读取和准备，原因在于它包含了从硬盘上读取文件、加载内存、由CPU进行一些简单的预处理等比较耗时的工作，传统的数据预处理操作通常会等待GPU完成上一次迭代的数据计算，才会进行下一迭代的数据读取。综上，整个训练效率将因此大打折扣，其示意图如下所示。



我参考了dell.com/support/articl 中的数据处理pipeline，对原始的bert数据处理模块做了部分改造。原始的bert数据读取处理流程代码如下：

```
if is_training:
      d = tf.data.Dataset.from_tensor_slices(tf.constant(input_files))
      d = d.repeat()
      d = d.shuffle(buffer_size=len(input_files))

      # `cycle_length` is the number of parallel files that get read.
      cycle_length = min(num_cpu_threads, len(input_files))

      # `sloppy` mode means that the interleaving is not exact. This adds
      # even more randomness to the training pipeline.
      d = d.apply(
          tf.contrib.data.parallel_interleave(
              tf.data.TFRecordDataset,
              sloppy=is_training,
              cycle_length=cycle_length))
      d = d.shuffle(buffer_size=100)
      d = d.apply(
            tf.contrib.data.map_and_batch(
                lambda record: _decode_record(record, name_to_features),
                batch_size=batch_size,
```

```
        num_parallel_batches=num_cpu_threads,
        drop_remainder=True))
```

上述代码只列了训练时的数据处理作为参考，其流程总结下来可以分为以下几步：

1、使用Dataset类型构造数据文件列表。由于我预先将文件分块了，所以原始数据文件是一个列表。

2、对文件列表进行repeat和预shuffle操作。

3、使用TFRecordDataset从2得到的文件中构造Dataset类型的训练数据，其中，cycle_length表示并发处理的文件数量，而parallel_interleave使得构造Dataset数据的过程能够并行化，sloppy能够控制是否使得数据生成的顺序随机。

4、对Dataset数据进行shuffle。

5、使用map_and_batch方法，并发处理，将tfrecord文件中的数据反序列化出来，生成下游model_fn接收的数据格式。_decode_record会放一些简单的数据转化逻辑。

上述过程，其实已经包含了对数据预处理的优化思想，它利用tensorflow的tf.data的一些机制，使得数据处理流水线不再是等待GPU处理完上一批数据才执行，而是可以在处理完一批数据后立刻处理下一批数据，如下图所示：

其实，上面的数据处理流程已经优化的不错了，很多同学对使用tf.Dataset肯定也比较熟悉。不过本着精益求精的精神，还是需要继续研究优化上述流程，比如引入shard。

这里的shard是什么意思呢？其实很简单，对于多卡训练的场景下，每个GPU设备可以看做是一个处理进程，shard可以预先将数据块进行分配，这样每个进程都能分配到的近似规模的数据量，以后每个进程只要处理自己的数据就可以了，与全局分配数据到不同节点的流程相比，节省了不少时间。

之前我在数据预处理的时候，已经将数据进行了预分块，每个文件的规模近似，此时我们可以在使用Dataset构造数据文件列表后，加入shard操作，将所有数据文件近似平均分配给所有进程。具体代码如下：

```
if is_training:
    d = tf.data.Dataset.from_tensor_slices(tf.constant(input_files
    d = d.shard(hvd.size(), hvd.rank())
    # d = d.repeat()
    d = d.shuffle(buffer_size=len(input_files))

    # `cycle_length` is the number of parallel files that get read
    cycle_length = min(num_cpu_threads, len(input_files))

    # `sloppy` mode means that the interleaving is not exact. This
    # even more randomness to the training pipeline.
    d = d.apply(
        tf.contrib.data.parallel_interleave(
            tf.data.TFRecordDataset,
            sloppy=is_training,
            cycle_length=cycle_length))
    d = d.prefetch(buffer_size=batch_size)
    d = d.shuffle(buffer_size=256)
    d = d.repeat()
    d = d.apply(
        tf.contrib.data.map_and_batch(
        lambda record: _decode_record(record, name_to_features),
        batch_size=batch_size,
        num_parallel_batches=num_cpu_threads,
        drop_remainder=True))
```

跟旧代码相比，其实改动并不是很多，除了增加了shard操作外，还增加了prefetch，每个step会预先将一些数据预加载，使得整个流程中，设备之间等待的时间尽可能缩短。当然，这个buffer_size也是一个需要调参的点，设置太大的话，反而会影响正常的数据流处理，设置太小的话，对性能的提升不会太明显，一般可以先试试设置成batch_size。值得注意的是，这里用到了两次shuffle，个人觉得这里可以只保留一次shuffle，因为我在之前数据预处理的时候已经做过两次全量的shuffle了，因此这里的随机性在一定程度上是可以保证的。

> 有关tensorflow的shard操作还有一个注意点需要备注一下，就是输入的数据文件列表数量最好是能够整除GPU的设备数（M），如果小于M，或者不能整除M，会在训练过程中产生stalled情

> 况，即部分设备会循环等待没有数据输入的设备，导致训练停滞，这个问题在于horovod集成时尤为常见。

尝试DALI

之前从oneflow的测评文章中，偶然得知了DALI框架（NVIDIA/DALI）。作为NVIDIA开源的另一款框架，它主要关注的是利用GPU来做数据读取和预处理，从而间接提升深度学习训练的整体性能。它跟tensorflow中的计算图类似，也是把数据数据相关的操作视为一个计算图。网上很多DALI的例子都是图像相关的，其内部的example也大多是图像和语音视频相关，专门针对NLP的例子几乎没有。因此，为了将DALI应用到BERT训练中着实费了我不少功夫。不过，当我将DALI和上一小节优化后的tf-datapipline进行对比后，发现两者的性能几乎差不多，并没有提升太多，也有可能是我实现方面有一些问题。下面我对DALI的主要核心操作进行简要说明，有兴趣的同学可以自行实现对比。

1、构建一个dali的pipeline。dali处理数据的基本链路称为一个dali的pipeline，在pipeline的类定义中，你可以定义dali读取数据的适配接口，DALI支持多种接口包括tfrecord，因此我很自然得选用了dali.ops.TFRecordReader，代码如下：

```
self.input = dali.ops.TFRecordReader(path=tfrec_filenames, index_path=tfre
                                     random_shuffle=True, shard_id
                                     initial_fill=10240, features=
                                     prefetch_queue_depth=128)
```

这里比较重要的index_path以及shard_id这个参数。上一小节说过，利用shard技术，可以提升整体的数据处理性能，而DALI也用到了shard，但是它的shard比tensorflow的shard粒度更小。DALI的开源代码中有个工具脚本叫**tfrecord2idx**，它主要用于为tfrecord数据文件生成对应的tfrec_idx_file。根据代码内容，可以很清楚知道这个idx到底是什么涵义：

```
f = open(sys.argv[1], 'rb')
idx = open(sys.argv[2], 'w')

while True:
    current = f.tell()
    try:
        # length
        byte_len = f.read(8)
        if len(byte_len) == 0:
```

```
                break
            # crc
            f.read(4)
            proto_len = struct.unpack('q', byte_len)[0]
            # proto
            f.read(proto_len)
            # crc
            f.read(4)
            idx.write(str(current) + ' ' + str(f.tell() - current) + '\n')
        except Exception:
            print("Not a valid TFRecord file")
            break

    f.close()
    idx.close()
```

该脚本先把tfrecord文件的头部信息读取出来，包括byte_len，crc，proto，crc信息。剩下的就是每个tf的Example序列化字节形态后在文件中的起始位置。tfrec_idx_file即是存储每个样本在tfrecord中的位置信息。将该信息传入到DALI后，DALI会根据这些信息以Example的粒度将所有样本shard到不同的设备进程。相对于将数据文件进行shard，这种实现方式不需要预先将数据进行shard，同时也能保证每个进程都能分到差不多规模的数据。shard_id则指定当前进程的标识（单机多卡的情况下，就是使用的GPU卡的index）。

2、之后我们实现pipeline类中的define_graph方法，这个方法主要是定义一些数据的预处理操作，对于图像和视频来说会有诸如裁切等操作，可以在这里进行。但对于我们的bert场景，并不需要做很多预处理，一般只需要两个操作，reshape（防止输入数据与预定义的tensor placeholder不一致）和cast（INT64转INT32，因为tfrecord只有INT64）。而DALI的ops库里面有很多与tensorflow的op类似的方法，**dali.ops.Cast**和**dali.ops.Reshape**。你可以对这些ops指定设备，例如gpu，使其可以在GPU上进行操作。

3、最后我们定义一个processor类，初始化DALI的pipeline， 利用DALI的tf插件nvidia.dali.plugin.tf，定义一个DALI数据流的迭代器来消费数据：

```
daliop = dali_tf.DALIIterator()
with tf.device("/gpu:0"):
            self.output = daliop(pipeline=pipe,
                                 shapes=[(batch_size, max_seq_length), (ba
                                         (batch_size, max_predictions_per_
```

```
                                    (batch_size, max_predictions_per_
                                dtypes=[tf.int32, tf.int32, tf.int32, tf.
                                device_id=device_id)
```

其中，pipe即我们之前定义的dali.pipeline.Pipeline 。这个output如果需要适配我们之前的model_fn流程，还需要将每个input feature取出来，重命名输出：

```
    def get_device_minibatches(self):
        # must be (features,label) to model_fn
        input_ids = tf.identity(self.output[0], name="input_ids")
        input_len = tf.identity(self.output[1], name="input_len")
        segment_ids = tf.identity(self.output[2], name="segment_ids")
        masked_lm_positions = tf.identity(self.output[3], name="masked_lm_
        masked_lm_ids = tf.identity(self.output[4], name="masked_lm_ids")
        masked_lm_weights = tf.identity(self.output[5], name="masked_lm_we

        return ((input_ids, input_len, segment_ids, masked_lm_positions, m
```

> 有关DALI的使用，鉴于篇幅原因就不多展开了，里面还有很多超参数没有提及，但是确实很重要的优化点。有兴趣的同学可以自行研究其开源代码。

混合精度训练

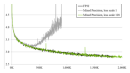
mixed precision training相信很多人都比较了解了，这里简单提一下。它是在模型训练时，将FP32和FP16两种数据精度混合使用，因为FP16能够为模型带来很多性能上的提升，包括提升batch size的规模，加快训练速度等。

> 混合精度训练队性能的提升是有条件的，只有支持Tensor cores的GPU设备才会有明显效果，如V100，或者A系列显卡。Tensor cores能够为GPU计算带来高倍的计算效率提升。每个tensor core实施 $D = A \times B + C$ ，其中，A和B是FP16精度的4*4矩阵，而D和C是同维度的矩阵，可以是FP16也可以是FP32。通过这个运算，tensor core能够将FP16精度的tensor融入到FP32中（或者FP16）。鉴于此，我们的batch size，序列长度以及其他tensor相关的维度最好是设置为8的倍数，此时能带来比较好的性能提升。

然而，直接将所有tensor的精度都降为FP16是不行的，会带来梯度计算溢出的问题，使得模型的训练精度有所损失。因此需要模型在合适的地方使用FP16。tensorflow提供了auto mixed precision功能，自动在合适的地方转FP16,这块原理我暂时还没有深入研究，以后可能会专门开个专题来详细说说。

除此以外，使用混合精度训练还可以使用loss scaling来避免梯度在更新过程中变为0。这个方法我并没有在我的训练中使用，因为在小数量训练的时候，整个模型的梯度更新还是在正常范围内。如果有兴趣尝试的话，可以参考

https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html#scalefactor
docs.nvidia.com



其中，有个细节需要关注，就是在前向计算的时候进行loss的scaling后，需要在权重更新的时候做unscaling，保证梯度更新时候的量级与FP32的时候是一致的。

使用tensorflow实现混合精度训练其实很简单，只要添加几行代码就可以。一个是在训练主代码中添加：

```
os.environ['TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_IGNORE_PERFORMANCE'] = '
```

另一个是在优化器代码中，添加：

```
optimizer = tf.train.experimental.enable_mixed_precision_graph_rewrite(opt
```

这里如果不用loss scaling，可以不用传loss_scaler。

开启XLA

为tensorflow开启XLA加速，也是提升训练整体性能的一个比较快捷的方式。XLA 使用 JIT 编译技术来分析用户在运行时（runtime）创建的计算图，它能够将一些计算图中的operations进行融合，并形成高效运行的本地机器代码。在tensorflow中开启XLA很简单，只需要在训练主流程代码中添加一行代码就可以实现：

```
def train():
    ....
    config = tf.ConfigProto()
    ...
    config.graph_options.optimizer_options.global_jit_level = tf.Optimizer
```

通过开启XLA，可以让整个训练的性能提升30%左右。有些平台声明开启XLA一般可以提升到1.5倍。我推测应该是使用了tensorflow的预编译版本，并不能很好得适配硬件上的指令集和，如果能够从头开始在训练的环境上编译一个tensorflow的gpu版本，可能XLA的提升效率会更好。

oneflow的尝试

为了能够尽可能地提升训练的效率，我还去尝试了下oneflow框架。这个框架可以说是专门为大模型高性能分布式训练而准备的，虽然目前它的一些生态还不是很完善，但是它的性能确是实打实的，我通过它项目空间中的转换代码将tfrecord转化为了ofrecord格式（期间还顺带学了点pyspark的东西），然后用它教程中的bert训练代码进行了尝试，相关的性能对比如下：

oneflow不带xla，16batch size单卡的训练速度约为29sentence/sec:

```
iter 49, total_loss: 9.385, mlm_loss: 9.385, speed:30.838(sec), 0.617(sec/batch), 25.942(sentences/sec)
iter 99, total_loss: 9.086, mlm_loss: 9.086, speed:26.811(sec), 0.536(sec/batch), 29.838(sentences/sec)
iter 149, total_loss: 8.458, mlm_loss: 8.458, speed:26.949(sec), 0.539(sec/batch), 29.686(sentences/sec)
iter 199, total_loss: 8.008, mlm_loss: 8.008, speed:27.075(sec), 0.541(sec/batch), 29.548(sentences/sec)
iter 249, total_loss: 7.628, mlm_loss: 7.628, speed:27.247(sec), 0.545(sec/batch), 29.361(sentences/sec)
iter 299, total_loss: 7.236, mlm_loss: 7.236, speed:27.201(sec), 0.544(sec/batch), 29.411(sentences/sec)
iter 349, total_loss: 6.869, mlm_loss: 6.869, speed:27.041(sec), 0.541(sec/batch), 29.585(sentences/sec)
iter 399, total_loss: 6.837, mlm_loss: 6.837, speed:26.998(sec), 0.540(sec/batch), 29.631(sentences/sec)
iter 449, total_loss: 6.637, mlm_loss: 6.637, speed:26.945(sec), 0.539(sec/batch), 29.691(sentences/sec)
iter 499, total_loss: 6.523, mlm_loss: 6.523, speed:26.973(sec), 0.539(sec/batch), 29.659(sentences/sec)
iter 549, total_loss: 6.385, mlm_loss: 6.385, speed:26.993(sec), 0.540(sec/batch), 29.637(sentences/sec)
iter 599, total_loss: 6.395, mlm_loss: 6.395, speed:27.032(sec), 0.541(sec/batch), 29.594(sentences/sec)
iter 649, total_loss: 6.357, mlm_loss: 6.357, speed:27.031(sec), 0.541(sec/batch), 29.595(sentences/sec)
iter 699, total_loss: 6.329, mlm_loss: 6.329, speed:27.080(sec), 0.542(sec/batch), 29.542(sentences/sec)
```

oneflow不带xla，16batch size四卡的训练速度约为110sentence/sec:

```
iter 49, total_loss: 9.290, mlm_loss: 9.290, speed:33.341(sec), 0.667(sec/batch), 95.978(sentences/sec)
iter 99, total_loss: 8.849, mlm_loss: 8.849, speed:29.079(sec), 0.582(sec/batch), 110.045(sentences/sec)
iter 149, total_loss: 8.262, mlm_loss: 8.262, speed:28.990(sec), 0.580(sec/batch), 110.383(sentences/sec)
iter 199, total_loss: 7.691, mlm_loss: 7.691, speed:29.147(sec), 0.583(sec/batch), 109.787(sentences/sec)
iter 249, total_loss: 7.212, mlm_loss: 7.212, speed:29.386(sec), 0.588(sec/batch), 108.894(sentences/sec)
iter 299, total_loss: 6.849, mlm_loss: 6.849, speed:29.296(sec), 0.586(sec/batch), 109.231(sentences/sec)
iter 349, total_loss: 6.670, mlm_loss: 6.670, speed:29.164(sec), 0.583(sec/batch), 109.725(sentences/sec)
iter 399, total_loss: 6.362, mlm_loss: 6.362, speed:29.119(sec), 0.582(sec/batch), 109.894(sentences/sec)
iter 449, total_loss: 6.293, mlm_loss: 6.293, speed:29.045(sec), 0.581(sec/batch), 110.173(sentences/sec)
```

可以看到，随着GPU设备的提升，其性能提升也是很明显的。

对于tensorflow来说，开混合精度和xla，同样是16batch size，且单卡的训练速度为41sentence/sec.

```
: 22:53:13.253331 139769874331456 tpu_estimator.py:2160] examples/sec: 41.2441
: 22:53:13.646231 139769874331456 tpu_estimator.py:2159] global_step/sec: 2.5427
: 22:53:13.646772 139769874331456 tpu_estimator.py:2160] examples/sec: 40.6833
: 22:53:14.072072 139769874331456 tpu_estimator.py:2159] global_step/sec: 2.34755
: 22:53:14.072412 139769874331456 tpu_estimator.py:2160] examples/sec: 37.5608
: 22:53:14.462625 139769874331456 tpu_estimator.py:2159] global_step/sec: 2.56144
: 22:53:14.463190 139769874331456 tpu_estimator.py:2160] examples/sec: 40.9831
: 22:53:14.850066 139769874331456 tpu_estimator.py:2159] global_step/sec: 2.58111
: 22:53:14.850522 139769874331456 tpu_estimator.py:2160] examples/sec: 41.2988
```

tensorflow来说，开混合精度和xla，同样是16batch size，四卡的训练速度为50*4*16/15.6=205sentence/sec.(因为四张卡，因此相当于一次性跑了4个50steps)

```
I1207 16:27:01.494995 139890416850752 basic_session_run_hooks.py:260] global_steps = 800, loss = 1.4893067 (15.583 sec)
I1207 16:27:01.498085 139724205868864 basic_session_run_hooks.py:692] global_step/sec: 3.20286
I1207 16:27:01.499613 139724205868864 basic_session_run_hooks.py:260] loss = 1.9292368, step = 800 (15.612 sec)
I1207 16:27:01.499138 140407583295296 basic_session_run_hooks.py:692] global_step/sec: 3.20447
I1207 16:27:01.500111 139724205868864 basic_session_run_hooks.py:260] global_steps = 800, loss = 1.9292368 (15.612 sec)
I1207 16:27:01.500556 140407583295296 basic_session_run_hooks.py:260] loss = 1.4573699, step = 800 (15.603 sec)
I1207 16:27:01.500835 140407583295296 basic_session_run_hooks.py:260] global_steps = 800, loss = 1.4573699 (15.603 sec)
I1207 16:27:01.516369 140361674520384 basic_session_run_hooks.py:692] global_step/sec: 3.20206
I1207 16:27:01.517720 140361674520384 basic_session_run_hooks.py:260] loss = 1.292848, step = 800 (15.615 sec)
I1207 16:27:01.517930 140361674520384 basic_session_run_hooks.py:260] global_steps = 800, loss = 1.292848 (15.615 sec)
I1207 16:27:17.053910 139890416850752 basic_session_run_hooks.py:692] global_step/sec: 3.21349
I1207 16:27:17.054807 139890416850752 basic_session_run_hooks.py:260] loss = 1.636846, step = 850 (15.560 sec)
I1207 16:27:17.054951 139890416850752 basic_session_run_hooks.py:260] global_steps = 850, loss = 1.636846 (15.560 sec)
I1207 16:27:17.057888 139724205868864 basic_session_run_hooks.py:692] global_step/sec: 3.21354
I1207 16:27:17.059933 139724205868864 basic_session_run_hooks.py:260] loss = 1.5122862, step = 851 (15.560 sec)
I1207 16:27:17.060373 139724205868864 basic_session_run_hooks.py:260] global_steps = 851, loss = 1.5122862 (15.560 sec)
```


从上面的性能对比来看，oneflow似乎并没有性能上的优势，具体原因目前还还未知。不过，综合了上面所有的性能优化改造后，多卡训练的性能相比于单卡来说，已经有了很大的提升，就结果来说我是满意的。

训练效果调优

优化完性能后，我就开始着眼于提升模型在训练中的收敛速度以及最终的效果。不过截止当前，该工作还在进行中，当前还只是尝试了一下梯度累加这个方法。

梯度累加

梯度累加也是大部分同学都比较了解的方法了，其机制简单来说就是每次原始step训练时，并不急于将当前step的梯度进行更新，而是通过某种方式保存累积下来，当训练步数达到一定的阈值时，才执行梯度更新的操作。这种做法在某种程度上增大了训练的batch size（并不划等号），使得我们能够用相对来说稍微大一点学习率来进行训练，每次模型的训练也能更加稳定。tensorflow中实现梯度累加的改造相对来说比较复杂，需要构造一个变量用来存储所有参与梯度累加的variable，同时构造局部的训练step计数用于判断当前是否累积到指定的步数，当还没到指定的阈值步数时，就不更新global_step以及不执行apply_gradient；当累积到指定阈值步数时，执行apply_gradient，同时更新global_step。具体可参考

https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/LanguageModeling/BERT github.com

后续计划尝试的方法

除了梯度累加外，后续可能还会去尝试如下方法：LAMBOptimization方法，这个优化器适合batch size很大的情况，能够加速训练收敛；另外还会参考论文Improved Pretraining for Domain-specific Contextual Embedding Models 中的方法，针对domain-pretraining任务中的灾难遗忘问题，进行改造尝试。

> 最后关于多卡训练中的学习率有一点需要说明一下，horovod的很多bert训练代码中会把learning rate乘以GPU的卡数，意为随着卡的增多，相当于batch size成倍增加，因此也可以使用相应的倍数增大学习率。关于这一点，我在尝试的过程中，发现这个并不会得到理想中的效果，训练的震荡情况有时候反而会比较严重。我推测应该是我们预加载了通用bert模型的权重，此时本身就需要用比较小的学习率，因此可以尝试去掉乘以倍数，或者乘以一个较小的倍数。

阶段性结果

之前，熵简科技发布的金融领域的Fin-bert模型，是金融领域里第一个中文的预训练模型，是一个非常棒的工作，我也和李博士就训练技术问题交流了一番，深有感触。本来是想将他们的模型和我们的模型以及通用的预训练模型做一番对比，不过金融领域这一块的公开评测数据很少，因此也只能拿一些我们自己实际场景中的任务做一下测评。

当前我使用了四分之一的数据量先训练了30万step的中间模型，简称huatai-bert，然后分别在我们的句子级别的实体情感分析任务以及篇章级别的情感分析任务上使用huatai-bert、Fin-bert以及chinese-wwm-roberta做了finetune，训练步数和超参数都一样，最后看测试集的F1分数，结果如下：

1、对于句子级别的实体情感分析任务，该任务本身比较难，使用chinese-wwm-roberta进行finetune的F1分数大概为0.85；而Fin-bert和huatai-bert的F1分数也是0.85。就这个任务上来说，基本没有提升。

2、对于篇章级别的情感分析任务，使用chinese-wwm-roberta进行finetune的F1分数大概为0.95；而Fin-bert的F1分数仍然为0.95，不过这次我们的huatai-bert得到了0.96的分数，提升了1个百分点。就这个任务上来说我们的huatai-bert是有提升的。

当然，上述对比测验只是一个简单的预测试，使用的测试集数量不是很多，所以结果相对来说并没有很大的参考价值。不过，至少可以说明我们做domain-pretrianing是有

效果的，因此我也可以安心使用全量数据去做训练，同时研究更多方法去改造优化。

总结

本文主要结合最近一个月在bert预训练上的各种研究工作，从数据准备、训练性能调优到训练效果调优三个维度，总结了其中的各种注意事项，以及优化的建议。读完本篇文章，你可以将本文的一些技术运用其他分布式大规模深度学习场景上。另外，本文中的一些性能优化技术原理我研究得也不是很深入，如果里面有些表述错误，欢迎来讨论。

推荐阅读

这个NLP工具，玩得根本停不下来

金融NLP需求落地实践总结——热门话题生成

专注于金融领域任务，首个金融领域的开源中文预训练语言模型 FinBERT 了解下

bert性能优化之——用另一种方式整合多头注意力