Tensorflow训练优化方法调研 （TF1.X）

摘要：

本文主要介绍了多种最佳实践技巧，通过这些技巧优化各个性能瓶颈，进而实现加速的目的。

优化方向：优化训练流程，充分合理利用训练硬件资源

优化主要方法：1、使用性能分析工具`timeline`分析训练过程中的性能瓶颈；2、根据性能瓶颈位置优化计算流程（如input pipeline优化）或采用更高效的操作（如 Dataset API、Fused Ops等）替换瓶颈操作；

注：直到2020/09/17， tensorflow已经跟新至 TF2.3版本，其中2.X相比于1.X有了质的变化，有关2.X的模型训练优化可参考官方文档指南 https://tensorflow.google.cn/guide/function

一、优化方向

训练模型加速主要有两个方向：1、使用模型技巧加速；2、优化训练流程，充分合理利用训练硬件资源。本文主要调研的方向是通过优化训练流程，充分合理利用训练硬件资源来进行TF模型训练的加速。

二、优化技巧

2.1 模型训练耗时分析——使用`timeline`

`timeline`可以分析整个模型在forward和backward的时候,每个操作消耗的时间，由此可以针对性的优化耗时的操作。

实例：

尝试使用tensorflow多卡来加速训练的时候， 发现多卡速度还不如单卡快，改用`tf.data`来 加速读图片还是很慢，最后使用`timeline`分析出了速度慢的原因，`timeline`的使用如下：

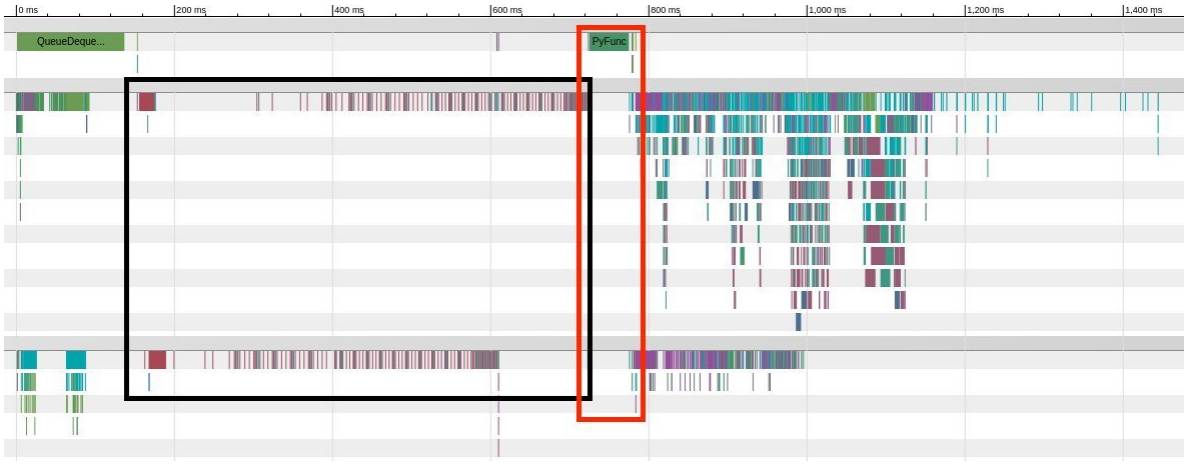
1. 获取timeline.json`

```
run_metadata = tf.RunMetadata()
run_options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
config = tf.ConfigProto(graph_options=tf.GraphOptions(

 optimizer_options=tf.OptimizerOptions(opt_level=tf.OptimizerOptions.L0)))
with tf.Session(config=config) as sess:
    c_np = sess.run(c,options=run_options,run_metadata=run_metadata)
    tl = timeline.Timeline(run_metadata.step_stats)
    ctf = tl.generate_chrome_trace_format()
with open('timeline.json','w') as wd:
    wd.write(ctf)
```

2. 分析`timeline.json`

到谷歌浏览器中打卡`chrome://tracing`并导入`timeline.json`，最后可以看得如下图所示的每个操作消耗的时间。



3. 根据分析结果优化流程或方法

上图中，横坐标为时间，从左到右依次为模型一次完整的forward and backward过程中，每个操作分别在cpu,gpu 0, gpu 1上消耗的时间，这些操作可以放大，非常方便观察具体每个操作在哪一个设备上消耗多少时间。

这里我们cpu上主要有QueueDequeue操作，这是进行图片预期过程，这个时候gpu在并行计算的所以gpu没有空等；另外我的模型还有一个PyFunc在cpu上运行，如红框所示，此时gpu在等这个结果，没有任何操作运行，这个操作应该要优化的。

另外就是如黑框所示，gpu上执行的时候有很大空隙，如黑框所示，这个导致gpu上的性能没有很好的利用起来，最后分析发现是bn在多卡环境下没有使用正确。bn有一个参数updates_collections设置为None这时bn的参数mean，var是立即更新的，也是计算完当前layer的mean，var就更新，然后进行下一个layer的操作，这在单卡下没有问题的， 但是多卡情况下就会**写等读**的冲突，因为可能存在gpu0更新（写）mean但此时gpu1还没有计算到该层，所以gpu0就要等gpu1读完mean才能写，这样导致了 如黑框所示的空隙，这时只需将参数设置成updates_collections=tf.GraphKeys.UPDATE_OPS` 即可，表示所有的bn参数更新由用户来显示指定更新，如

```
update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
  with tf.control_dependencies(update_ops):
    train_op = optimizer.minimize(loss)
```

这样可以在每个卡forward完后，再更新`bn`参数，此时的**读写**不存在冲突。优化后，2卡训练获得了接近2倍的加速比。

2.2 快速训练方法

针对通过分析`timeline.json`可以分析出模型训练过程中主要的耗时点，我们可以针对耗时点采用更加快速的实现方式加速模型训练，以下通过三个部分介绍tf训练优化时各个部分的最佳实现方式。

2.2.1通用最佳实践

覆盖多种模型类型和硬件的主题

1. input pipeline优化

常见模型会从磁盘中抽取数据，进行预处理，然后通过网络发送数据。例如，处理JPEG图片的模型会有下面的流程：从磁盘加载图片，将JPEG解码成一个tensor，进行裁减（crop）和补齐（pad），可能还会进行翻转（flip）和扭曲（distort），然后再batch。该流程被称为input pipeline。随着GPUs和其它硬件加速器越来越快，数据预处理可能是个瓶劲。

判断input pipeline是否是瓶颈可能很复杂。一个最简单的方法是，在pipeline之后，将模型减至单个操作（trivial model），并测量每秒处理的样本数。如果对于完整模型（full model）和简单模型（trivial model）每秒处理样本的差距很小，那么输入的pipeline很可能是瓶颈。下面还有其它方法来验证该问题：

- 通过nvidia-smi -l 2确认是否一个GPU未被充分利用。如果GPU使用率（GPU utilization）没有达到80-100%，那么输入的pipeline可能是个瓶颈。
- 生成一个timeline，然后观察那些空白（等待状态）的大块(large blocks)。生成timeline的示例详见XLA JTI
- 预估所需要的吞吐量，并验证所使用的磁盘是否能支撑该吞吐量。一些云解决方案（cloud solutions）具有云盘（network attached disks），可能低至50M/sec，它比spinning disk（150MB/sec）、SATA SSDs (500 MB/sec)、 和PCIe SSDs (2,000+ MB/sec)还低

2. CPU预处理

在CPU上放置input pipeline操作，可以极大提升性能。对于input pipeline，使用CPU可以充分释放GPU，让它更聚焦于训练上。为了确保预处理过程发生在CPU上，需要按以下方式包装预处理操作：

```
with tf.device('/cpu:0'):
  # function to get and process images or data.
  distorted_inputs = load_and_distort_images()
```

如果使用tf.estimator.Estimator，input function会被自动放置在CPU上。

3. 使用Dataset API

对于构建input pipeline，推荐使用Dataset API来替代queue_runner。该API在tensorFlow 1.2中作为contrib的一部分被添加进去，并在后续版本会移至core包中。ResNet examplearXiv:1512.03385会训练CIFAR-10,它展示了如何使用Dataset API以及tf.estimator.Estimator。Dataset API会使用C++的多线程机制，会比基于python的queue_runner（受限于python的多线程低性能）的开销更低。

当使用一个feed_dict来feeding数据时，会提供更高的灵活性，使用feed_dict的大多数实例不可以进行合适的扩展。然而，在只有一个GPU的实例中，这种差异是微不足道的。我们仍推荐使用Dataset API。避免使用以下的情况：

```
# feed_dict often results in suboptimal performance when using large inputs.
sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

4. 融合解码和剪裁（fused decode & crop）

如果输入是JPEG图片，也需要裁减（cropping），使用fused tf.image.decode_and_crop_jpeg可以加速预处理。tf.image.decode_and_crop_jpeg只会解码（decode）在图片裁减窗口（crop window）上的部分。如果裁减窗口比起整个图片更小，就会极大地加速处理。对于ImageNet数据，该方法可以极大加速input pipeline，可提升30%。

示例：

```
def _image_preprocess_fn(image_buffer):
    # image_buffer 1-D string Tensor representing the raw JPEG image buffer.

    # Extract image shape from raw JPEG image buffer.
    image_shape = tf.image.extract_jpeg_shape(image_buffer)

    # Get a crop window with distorted bounding box.
    sample_distorted_bounding_box = tf.image.sample_distorted_bounding_box(
      image_shape, ...)
```

```
    bbox_begin, bbox_size, distort_bbox = sample_distorted_bounding_box

    # Decode and crop image.
    offset_y, offset_x, _ = tf.unstack(bbox_begin)
    target_height, target_width, _ = tf.unstack(bbox_size)
    crop_window = tf.stack([offset_y, offset_x, target_height, target_width])
    cropped_image = tf.image.decode_and_crop_jpeg(image, crop_window)
```

tf.image.decode_and_crop_jpeg在所有平台上都有提供。在Windows平台上不会有加速，因为它使用libjpeg，而在其它平台上则使用libjpeg-turbo。

5. 使用大文件

读取大量小文件可以极大影响I/O性能。获取最大的I/O吞吐量的其中一种方法是，将数据预处理成更大的（~100MB） TFRecord文件。对于更小的数据集（200MB-1GB），最好的方法通常是加载整个数据集到内存中。文档下载和转化成TFRecord格式包含了相关的信息和脚本来创建TFRecords，该脚本会将CIFAR-10数据集转化成TFRecords。

6. 数据格式

数据格式指的是传递给一个给定Op的Tensor结构。下面的讨论关于表示图片的4D Tensors。在TensorFlow中，4D tensor的部分通常指的是：

- N：表示在一个batch中的图片数
- H：表示在垂直维度(height)上的像素数
- W: 在水平维度（width）上的像素数
- C: 表示通道（channels）。例如，1表示黑和白，或者灰度和3表示RGB。

在TensorFlow中，存在两个命名约定，来表示两种最常用的数据格式：

- NCHW 或 channels_first
- NHWC 或 channels_last

NHWC是Tensorflow的缺省方式，NCHW是当在NVIDIA GPU上使用cuDNN训练时的最优格式。

最好的实践是，构建支持两种数据格式的模型。这可以简化在GPU上的训练，接着在CPU上进行inference。如果TensorFlow在Intel MKL优化器上进行编译，许多op，特别是那些与基于CNN相关的模型，会被优化并支持NCHW。如果不使用MKL，当使用NCHW时，一些op不会支持在CPU上运行。

这两种格式的历史是，TensorFlow刚开始使用NHWC是因为它在CPU上更快一些。在很长一段时间内，我们致力于开发工具来自动化重写graphs来在格式间进行透明切换，并充分利用微优化：比起常用有效的NCHW，使用NHWC的一个GPU Op会更快。

7、使用Fused Ops

Fused Ops会将多个Op结合成单个kernel来提升性能。在Tensorflow中有许多fused Ops，当可能时XLA会创建fused Ops来自动提升性能。下面的示例会使用fused Ops，可以极大提升性能。

示例：Fused batch norm

Fused batch norm 会结合多种op，来将batch归一化（normalization）到单个kernel中。对于一些模型，Batch norm是一个开销昂贵的处理，会占据大量的操作时间。使用fused batch norm可以进行12%-30%的加速。

有两个常用的batch norm，它们同时支持fusing。核心的tf.layers.batch_normalization在TensorFlow 1.3中被添加进去。

```
bn = tf.layers.batch_normalization(
    input_layer, fused=True, data_format='NCHW')
```

自TensorFlow 1.0后，tf.contrib.layers.batch_norm方法具有fused操作。

```
bn = tf.contrib.layers.batch_norm(input_layer, fused=True, data_format='NCHW')
```

8、使用混合精度进行训练（tensorflow 1.14 后）

混合精度是指在训练期间在模型中同时使用16位和32位浮点类型，以使其运行更快并使用更少的内存。通过将模型的某些部分保持在32位类型中以保持数值稳定性，该模型将具有更短的步长时间，并且在评估指标（如准确性）方面同样得到训练。本指南介绍了如何使用实验性Keras混合精度API来加速模型。使用此API可以在现代GPU上将性能提高三倍以上，在TPU上可以提高60%。

9、从源码进行构建和安装

缺省的TensorFLow二进制包面向大多数的硬件，以便TensorFlow能为所有人所使用。如果使用CPU进行training或inference，推荐使用CPU的所有优化来编译TensorFlow。在CPU中的training和inference加速在Comparing compiler optimizations有文档说明。

为了安装最优化版的TensorFlow，可以从源码进行编译和安装。如果有必要在一个平台上构建TensorFlow，该平台具有不同的硬件，那么可以对目标平台进行最高级优化的交叉编译。下面的命令是使用bazel来编译一个特定的平台：

```
# This command optimizes for Intel’s Broadwell processor
bazel build -c opt --copt=-march="broadwell" --config=cuda
//tensorflow/tools/pip_package:build_pip_package
```

环境，构建，安装tips：

- ./configure 会计算在build中的容量。它不会影响整个性能，但会影响初始的startup。在运行TensorFlow一次后，编译的kernels会被CUDA缓存。如果使用一个docker container，数据不会被缓存，每次TensorFlow启动时，会花费penalty。最好的实践是，包含所使用GPU的计算能力，例如：P100: 6.0, Titan X (Pascal): 6.1, Titan X (Maxwell): 5.2, K80: 3.7.
- 使用gcc版本，它支持目标CPU的所有优化。推荐gcc最低版本为：4.8.3. 在OS X上，更新最新的Xcode版本，使用xcode带的clang版本。

2.2.3 GPU优化最佳实践

该部分包含了GPU相关的tops，它在通用最佳实践中没有被涵盖。

在multi-GPU上获取最优性化是个挑战，常用的方法是使用数据并行化。通过使用数据并行化进行scaling涉及到生成多个模型拷贝，这被称为是“塔（towers）”，接着在每个GPU上放置一个tower。每个tower在一个不同的mini-batch数据上进行操作，接着更新变量（也称为参数），这些变量需要在每个towers间进行共享。然而，每个tower如何去获取更新后的变量，以及梯度如何被应用，会对性能、可扩展性、模型收敛都有影响。该节其它部分会提供了关于变量放置的总览、以及如何在多GPU上对模型进行towering。高性能模型有进一步的介绍，它介绍了用于在tower间进行共享和更新变量的复杂方法。

处理变量更新的最好方法依赖于模型、硬件、以及硬件如何配置。有个示例，两个系统可以使用NVIDIA Tesla P100s进行构建，但其中一个使用PCIe，另一个使用NVLink。在这种情况下，每个系统的可选解决方案可以不一样。对于真实的示例，读取benchmark页，它有关于多种平台设置的详情。下面是一些平台的bechmarking和配置：

- Tesla K80: 如果GPUs在相同的PCI Express root complex，并且能端到端(peer to peer)地使用NVIDIA GPUDirect，那么在训练时跨GPU间放置相同的变量是最好的方法。如果GPU不能使用GPUDirect，那么在CPU上放置变量是最好的方法。
- Titan X (Maxwell and Pascal), M40, P100, and similar：对于像ResNet和InceptionV3这样的模型，在CPU上放置变量是最优设置，但对于像AlexNet和VGG这样有许多变量的模型，使用带

NCCL的GPU更好。

对放置变量进行管理的最常用方法是，创建一个方法来决定每个Op放置的地方，并在一个指定设备上（通过调用with tf.device()）使用该方法：考虑到这样的场景：一个模型在2个GPU上进行训练，变量被放置在CPU上。在每个GPU上存在一个loop来创建和放置"towers"。一个定制的设备放置方法可以被创建，该方法会监视类型为Variable、VariableV2、以及VarHandleOp的Ops，以及表明了它们被放置在CPU上。所有其它的Ops可以被放置在目标GPU上。graph的构建可以如下进行处理：

- 在第一个loop上，模型的一个"tower"在gpu:0上被创建。在Op的放置（placement）期间，定制的设置放置方法将表明那些变量被放置在cpu:0上，而其它的Ops则放置在gpu:0上。
- 在第二个loop上，reuse被设置成True表明那些变量可以被利用，接着该"tower"在gpu:1上被创建。在该"tower"相关的Ops的放置期间，放置在cpu:0上的变量可以被复用，所有其它的Ops在gpu:1上被创建和放置。

最终结果是，在CPU上放置的所有变量，每个GPU都具有一份关于所有与该模型相关的可计算Ops的拷贝。

下面的代码片段展示了两种不同的方法进行变量放置：一个是在CPU上放置变量；另一种是跨GPU放置相同的变量。

```
class GpuParamServerDeviceSetter(object):
  """Used with tf.device() to place variables on the least loaded GPU.

    A common use for this class is to pass a list of GPU devices, e.g.
['gpu:0',
    'gpu:1','gpu:2'], as ps_devices.  When each variable is placed, it will be
    placed on the least loaded gpu. All other Ops, which will be the
computation
    Ops, will be placed on the worker_device.
  """

  def __init__(self, worker_device, ps_devices):
    """Initializer for GpuParamServerDeviceSetter.
    Args:
      worker_device: the device to use for computation Ops.
      ps_devices: a list of devices to use for Variable Ops. Each variable is
      assigned to the least loaded device.
    """
    self.ps_devices = ps_devices
    self.worker_device = worker_device
    self.ps_sizes = [0] * len(self.ps_devices)

  def __call__(self, op):
    if op.device:
      return op.device
    if op.type not in ['Variable', 'VariableV2', 'VarHandleOp']:
      return self.worker_device

    # Gets the least loaded ps_device
    device_index, _ = min(enumerate(self.ps_sizes),
key=operator.itemgetter(1))
    device_name = self.ps_devices[device_index]
    var_size = op.outputs[0].get_shape().num_elements()
    self.ps_sizes[device_index] += var_size

    return device_name
```

```
def _create_device_setter(is_cpu_ps, worker, num_gpus):
  """Create device setter object."""
  if is_cpu_ps:
    # tf.train.replica_device_setter supports placing variables on the CPU,
all
    # on one GPU, or on ps_servers defined in a cluster_spec.
    return tf.train.replica_device_setter(
        worker_device=worker, ps_device='/cpu:0', ps_tasks=1)
  else:
    gpus = ['/gpu:%d' % i for i in range(num_gpus)]
    return ParamServerDeviceSetter(worker, gpus)

# The method below is a modified snippet from the full example.
def _resnet_model_fn():
    # When set to False, variables are placed on the least loaded GPU. If set
    # to True, the variables will be placed on the CPU.
    is_cpu_ps = False

    # Loops over the number of GPUs and creates a copy ("tower") of the model
on
    # each GPU.
    for i in range(num_gpus):
      worker = '/gpu:%d' % i
      # Creates a device setter used to determine where Ops are to be placed.
      device_setter = _create_device_setter(is_cpu_ps, worker, FLAGS.num_gpus)
      # Creates variables on the first loop.  On subsequent loops reuse is set
      # to True, which results in the "towers" sharing variables.
      with tf.variable_scope('resnet', reuse=bool(i != 0)):
        with tf.name_scope('tower_%d' % i) as name_scope:
          # tf.device calls the device_setter for each Op that is created.
          # device_setter returns the device the Op is to be placed on.
          with tf.device(device_setter):
            # Creates the "tower".
            _tower_fn(is_training, weight_decay, tower_features[i],
                      tower_labels[i], tower_losses, tower_gradvars,
                      tower_preds, False)
```

上面的代码用于说明，使用高级方法可以更简单地支持一系列流行的方法。该示例可以随着API进行更新，并可以扩展到放置在多GPU的场景。

2.2.4 CPU优化最佳实践

当Tensorflow使用源码、根据目标CPU的说明书进行构建时，CPUs（包括Intel® Xeon Phi™,）可以达到最优性能。

使用最新的指令集，Intel® 已经添加了Intel® Math Kernel Library for Deep Neural Networks (Intel® MKL-DNN) 的支持到TensorFlow中。其中，该名字并不完全准确，这些优化经常被称为'MKL'或者'TensorFlow with MKL'。TensorFlow with Intel® MKL-DNN包含了在MKL上优化的详情。

下面列出了两种配置，通过调整线程池来用于优化CPU性能：

- intra_op_parallelism_threads： 注意可以使用多线程来并发执行（parallelize），这可以在该池中调用单个pieces。
- inter_op_parallelism_threads：所有准备节点（ready nodes）都在该池中被调度。

这些配置通过tf.ConfigProto进行设置，并传递给tf.Session的config属性项中。对于这两种配置选项，如果都未设置或者置为0, 将缺省为逻辑CPU cores的个数。实验说明，对于4核的单CPU，以及70+组合的逻辑cores的CPU，缺省是有效的。一个常见的可选优化是，在池中线程数设置成物理核的数目，而非逻辑核的数目。

```
config = tf.ConfigProto()
config.intra_op_parallelism_threads = 44
config.inter_op_parallelism_threads = 44
tf.session(config=config)
```

三、总结

以上调研的TensorFlow训练优化手段均是在tensorflow框架内的，大部分为高阶的Tensorflow的使用技巧，但从目前的资料显示，在充分使用这些高阶技巧后，能够有效的加速绝大部分模型训练场景。