tensorflow:input pipeline性能指南

Reading time ~1 minute

介绍

GPU和TPU可以从根本上减少执行单个training step所需的时间。**为了达到高性能，我们需要一个高效的input pipeline，它可以在当前step完成后为下一个step高效分发数据。**tf.data API可以帮助构建灵活和高效的input pipeline。该文档解释了tf.data API的特性以及最佳实践，来跨多种模型和加速器构建高性能tensorflow input pipelines。

该指南包括：

- 介绍了tensorflow input pipelines本质上是一个ETL流程
- 描述了在tf.data API上下文中的常见性能优化
- 讨论了转换（transformations）顺序的性能影响
- 总结了设计高性能tensorflow input pipeline的最佳实践

1.Input Pipeline结构

一个典型的tensorflow training input pipeline可以看成是一个ETL process问题：

- 1.**(E)Extract**: 从持久化存储中读取数据——可以是本地（比如：HDD或SSD）或远程（比如：GCS或HDFS）
- 2.**(T)Transform**：利用CPU cores来解析和执行在数据上的预处理操作：比如：图片解压，数据转换（比如：随机裁减、翻转、颜色扭曲），重排（shuffling），以及batching。
- 3.**(L)Load**：在加速设备上（例如：GPU或TPU）加载转换后的数据，执行机器学习模型

这种模式（pattern）可以有效地利用CPU，同时保留加速器让它去处理模型训练部分的重任。另外，将input pipeline看成是一个ETL process，这种结构更有利于性能优化。

当使用tf.estimator.Estimator API时，前两个phases（Extract和Transform）可以在input_fn中被捕获，然后传给tf.estimator.Estimator.train。在代码中，看起来实现如下：

```
def parse_fn(example):
  "Parse TFExample records and perform simple data augmentation."

  example_fmt = {
```

```
    "image": tf.FixedLengthFeature((), tf.string, ""),
    "label": tf.FixedLengthFeature((), tf.int64, -1)
  }
  parsed = tf.parse_single_example(example, example_fmt)
  image = tf.image.decode_image(parsed["image"])
  image = _augment_helper(image)  # augments image using slice, reshape, resize_bilinear
  return image, parsed["label"]

def input_fn():
  files = tf.data.Dataset.list_files("/path/to/dataset/train-*.tfrecord")
  dataset = files.interleave(tf.data.TFRecordDataset)
  dataset = dataset.shuffle(buffer_size=FLAGS.shuffle_buffer_size)
  dataset = dataset.map(map_func=parse_fn)
  dataset = dataset.batch(batch_size=FLAGS.batch_size)
  return dataset
```

下一节会在该input pipeline上构建，并添加性能优化。

2.性能优化

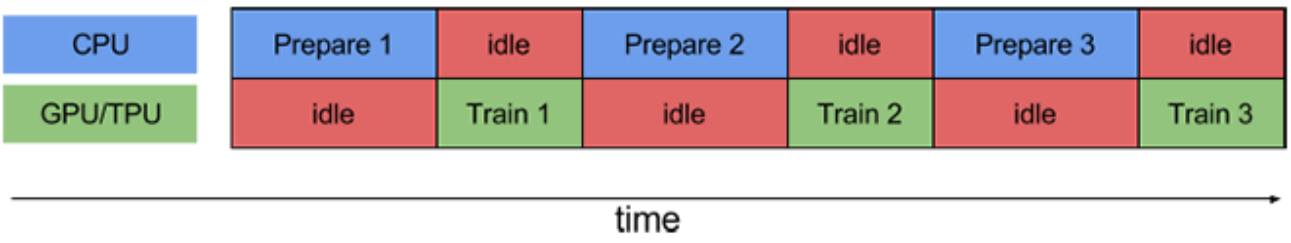
新计算设备（比如：CPU或TPU），可以以一个越来越快的速率来训练神经网络，而CPU处理速度被证明是个瓶劲。tf.data API提供给用户相应的构建块来设计input pipeline，可以有效利用CPU，优化在ETL过程中的每一步。

2.1 Pipeling

为了执行一个training step，你必须首先extract和transform训练数据，接着将它feed给一个运行在加速器上的模型。然而，在一个原始的同步实现（naive synchronous implementation）中，CPU会准备数据，加速器会保持空闲状态。相反地，当加速器在训练模型时，CPU会保持空闲状态。训练过程的时间会是CPU处理时间和加速器训练时间的总和。

Pipelining会将一个training step的预处理和模型执行在时序上重叠。当加速器执行了N个training step时，CPU会为第N+1个step准备数据。这样做是为了减少总时间。

如果没有做pipeling，CPU和GPU/TPU的空闲时间会有很多：



而有了pipeling，空闲时间会急剧减少：

tf.data API通过tf.data.Dataset.prefetch转换，提供了一个软件实现的管道机制(software pipeling），该转换可以被用于将数据生产时间和数据消费时间相解耦。特别的，该转换会使用一个后台线程，以及一个内部缓存（internal buffer），来prefetch来自input dataset的元素（在它们被请求之前）。这样，为了达到上述的pipeling效果，你可以添加prefetch(1) 作为最终转换到你的dataset pipeline中（或者prefetch(n)，如果单个training step消费n个元素的话）

为了在我们的示例中达到该变化，可以做如下更改，将：

```
dataset = dataset.batch(batch_size=FLAGS.batch_size)
return dataset
```

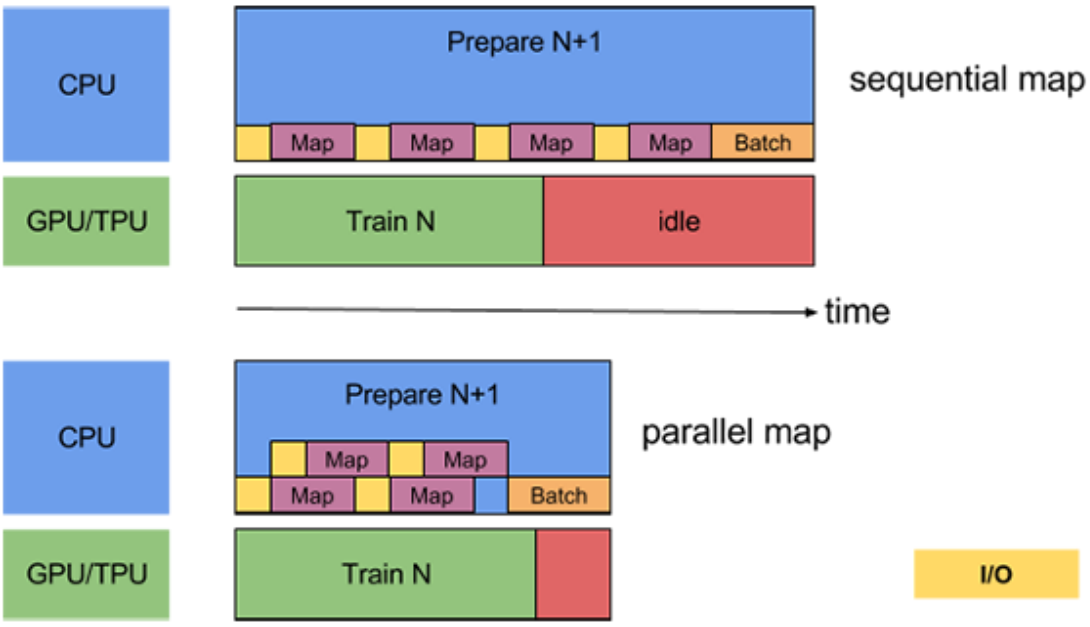
变更成：

```
dataset = dataset.batch(batch_size=FLAGS.batch_size)
dataset = dataset.prefetch(buffer_size=FLAGS.prefetch_buffer_size)
return dataset
```

注意，prefetch转换在任何时间都会有好处，有机会将一个producer的工作量和一个consumer的工作量在时序上重合。之前的推荐被简化成最常用的应用。

2.2 并行化数据转换

当准备一个batch时，input elements需要被预处理。为了这个目的，tf.data API提供了tf.data.Dataset.map转换，它会将一个用户定义的函数（例如：运行示例中的parse_fn）应用到input dataset中的每个元素上。由于input elements相互独立，预处理可以跨多个CPU core并行进行。为了达到该目的，map转换提供了num_parallel_calls参数来指定并行度。例如，下图展示了在map转换中设置num_parallel_calls=2的效果：



选择num_parallel_calls的最佳值取决于你的硬件、训练数据的特性（比如：size和shape）、map函数的开销、以及在CPU上同时发生的其它处理过程；一个简单的启发法（heuristic）是使用可提供的CPU cores的数目。例如：如果机器具有4个cores，那么设置num_parallel_calls=4更有效。在另一方面，将num_parallel_calls设置为一个比可提供的CPU数更大的值，会导致无效调度，反而会减慢速度。

可以将代码：

```
dataset = dataset.map(map_func=parse_fn)
```

更改为：

```
dataset = dataset.map(map_func=parse_fn, num_parallel_calls=FLAGS.num_parallel_calls)
```

更进一步，如果你的batch_size是几百或几千，你的pipeline将可能从batch创建并行化中获得额外收益。为了达到该目的，tf.data API提供了tf.contrib.data.map_and_batch转换，它可以用效地将map转换和batch转换相“混合（fuse）”。

可以将下面代码：

```
dataset = dataset.map(map_func=parse_fn, num_parallel_calls=FLAGS.num_parallel_calls)
dataset = dataset.batch(batch_size=FLAGS.batch_size)
```

更改为：

```
dataset = dataset.apply(tf.contrib.data.map_and_batch(
    map_func=parse_fn, batch_size=FLAGS.batch_size))
```

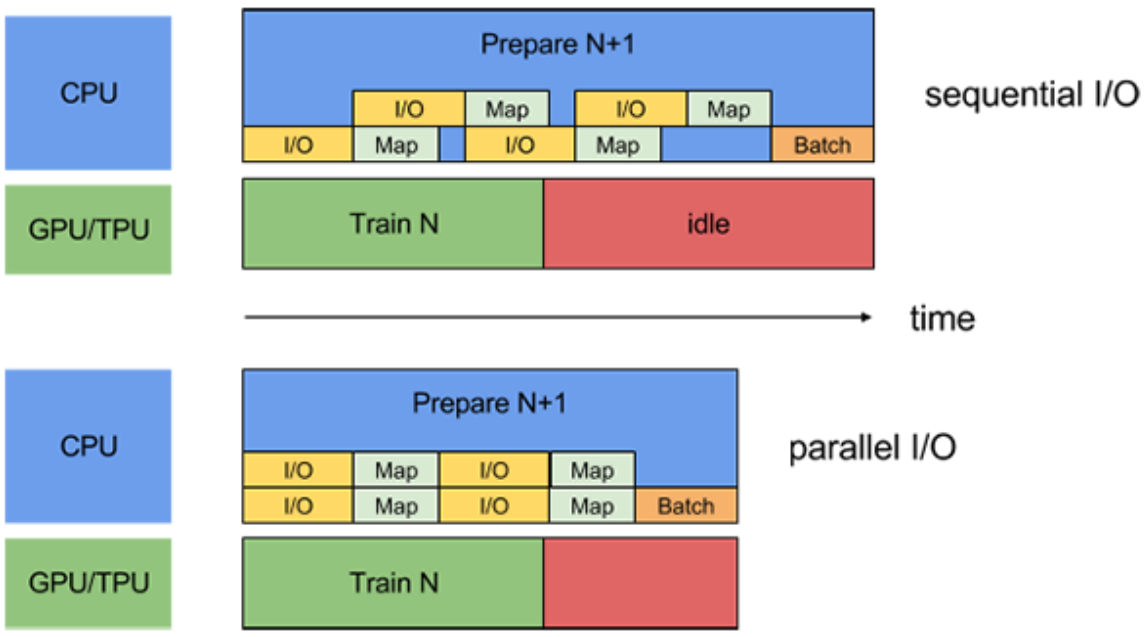
2.3 数据抽取并行化

在现实中，input data可以在远端存储（例如：GCS或HDFS），即可能因为input data太大在本地存不下，也可能因为训练是分布式的，不希望在每台机器上复制input data。当在本地读取数据时一个dataset pipeline可以很好地工作，而当远端读取时可能会有I/O瓶劲。这是由于以下在本地存储和远端存储的不同之处：

- **首字节的时间（Time-to-first-byte）**：从远端存储上读取一个文件的首字节，可以会比本地存储花费多个数量级。
- **读取吞吐量（Read throughput）**：远端存储通常提供了更大的聚集带宽，读取单个文件可能只利用了该带宽的一小部分。

另一方面，一旦原始字节被读到内存中，有必要做并行化和数据压缩（比如: protobuf），这会增加额外开销。这种开销不用管数据是否在本地或远端存储，但在远端存储的情况下，如果数据没有有效地做prefetch开销会很大。

为了减轻多种数据抽取开销的影响，tf.data提供了tf.contrib.data.parallel_interleave转换。使用该转换来并行化其它datasets内容（比如：data file readers）交错执行。 datasets的重合数目可以通过cycle_length参数进行指定。



为了在运行示例中应用该变化：

```
dataset = files.interleave(tf.data.TFRecordDataset)
```

变更为：

```
dataset = files.apply(tf.contrib.data.parallel_interleave(
    tf.data.TFRecordDataset, cycle_length=FLAGS.num_parallel_readers))
```

远端存储系统的吞吐量可以随时间变化，这是由于负载和网络事件。为了说明该差异，parallel_interleave转换可以可选地使用prefetching。（详见：tf.contrib.data.parallel_interleave）

缺省的，parallel_interleave转换提供了一个元素的确定顺序使可重现。作为prefetching的一种替代方法，parallel_interleave转换也提供了一个选项，可以在以顺序保障为代价上增强性能。如果sloppy参数设置为true，该转换会与它的确定顺序相背离，通过临时跳过那些当下个元素被需要时它的元素不可提供的文件来实现。

3.性能注意事项

tf.data API是围绕着转换进行设计的，提供给用户灵活性。尽管许多这种转换是交替的，特定转换的顺序对性能有影响。

3.1 Map和Batch

调用传给map转换的用户自定义函数有开销(overhead)，与用户自定义函数的调度（scheduling）和执行（executing）有效。通常，对比起该函数执行的计算量，该开销很小。然而，如果map做很少的工作量，该开销可能会成为主要开销。在这种情况下，我们推荐将用户自定义函数向量化（vectorizing：也就是说，一次一个batch上操作），并在map转换之前应用batch转换。

3.2 Map和Catch

tf.data.Dataset.cache转换可以cache一个dataset，即可以以内存方式，也可以以本地存储方式。如果传给map转换的用户自定义函数很昂贵，只要产生的dataset仍以满足内存或本地存储，可以在map转换后应用cache转换。如果用户自定义函数增加了超过cache容量存储dataset所需的空间，可以考虑在你的training job之前对你的数据预处理来减小资源使用。

3.3 Map和Interleave / Prefetch / Shuffle

一些转换（包括：interleave, prefetch和shuffle），维护着一个关于elements的内部缓存（internal buffer）。如果传给map的用户自定义函数更改了elements的size，那么map转换、以

及这些带buffer元素的转换的顺序会影响内存使用。总之，我们推荐选择能产生低内存footprint的顺序，除非为性能考虑采用不同的顺序。（例如，开启map和batch的fusing模式）

3.4 Repeat和Shuffle

tf.data.Dataset.repeat转换会将input数据以一个有限次数进行重复；数据的每次重复通常被称为一个epoch。tf.data.Dataset.shuffle则会对dataset的样本进行随机转换。

如果在shuffle转换之前应用repeat转换，那么epoch的边界是模糊的。也就是说，特定元素即使在其它元素只出现一次之前可以被重复。另一方面，如果在repeat转换之前使用shuffle转换，那么每个epoch的开始阶段性能会降低，这与shuffle转换的内部状态有关。换句话说，前者（repeat -> shuffle）提供更好的性能，后者(shuffle -> repeat)提供更强的顺序保障。

当可能时，我们推荐使用fused tf.contrib.data.shuffle_and_repeat转换，它会结合两者的最佳（性能好、顺序有保障）。否则，我们推荐在repeating之前shuffling。

3.5 最佳实践总结

这里有设计input pipeline的最佳实践总结：

- 使用**prefetch**转换来将一个producer和consumer进行时序重合。特别的，我们推荐添加prefetch(n)（其中，**n = 元素个数 /单个training step消费的batches**）到你的input pipeline的结尾处，在cpu上执行重合转换，在加速器上进行训练。
- 通过设置**num_parallel_calls**参数将map转换并行化。我们推荐使用可提供的**CPU cores**来作为它的值。
- 如果你使用batch转换将预处理的元素组装成一个batch，我们推荐使用fused **map_and_batch**转换；特别是，当你正使用大的batch_size时。
- 如果你的数据存储在远端，以及（或者）你的数据需要还原序列化（deserialization），我们推荐使用**parallel_interleave**转换来将从不同文件读取（以及deserialization）进行时序重合（overlap）。
- 将传给map转换的用户自定义函数进行**向量化（Vectorize）**，分摊调度和执行该函数的开销。
- 如果你的数据刚好可以装进内存，在第一个epoch中使用**cache**转换来缓存它到内存中，从而让后续的epochs可以避免对它进行读取、解析、转换的开销。
- 如果你的预处理增加了数据的size，我们推荐首先应用interleave, prefetch, 和 shuffle（如果可能）来减小内存使用量
- 我们推荐**在repeat转换之前使用shuffle转换**，理想情况下使用fused shuffle_and_repeat转换。