Improve training time with caching

- In-memory

```
tf.data.Dataset.cache()
```

- Disk

```
tf.data.Dataset.cache(filename=...)
```

Before we go into detail on parallelism, a handy tool for performance is caching. The tf.data.Dataset.cache transformation can cache dataset either in memory or on local storage. If you're data can fit into memory, use the cache transformation to cache it in memory during the first epoch.

This is done so that subsequent epochs can avoid the overhead associated with reading, parsing and transforming it. This transformation can cache on local storage by specifying a file name

Caching with tf.data

```
dataset = tfds.load('cats_vs_dogs',split=tfds.Split.TRAIN)


# In-memory caching

train_dataset = dataset.cache()

model.fit(train_dataset, epochs=...)


# Disk caching

train_dataset = dataset.cache(filename='cache')

model.fit(train_dataset, epochs=...)
```

When using disk caching, the filename is passed to the API, which represents the name of the directory on the file system to use for caching tensors from this dataset. If a filename is not provided, the dataset will be cached in memory.

Parallelism with tf.data

Data transformations

- Transformations can be expensive
- Time-consuming as CPU is not fully utilized

e.g., Resizing, preprocessing, augmentation in images

A quick recap on transformations and consider the impact of that they can have on performance. These transformations generally deal with preprocessing tasks that tend to add overhead expense to your training pipeline.

Moreover, when augmenting image classification datasets, many of these transformations are usually applied element-wise by the tf.data's map operation. As a result, this can lead to the CPU being underutilized.

Consider the following transformation

```
def augment(features):
  X = tf.image.random_flip_left_right(features['image'])
  X = tf.image.random_flip_up_down(X)
  X = tf.image.random_brightness(X, max_delta=0.1)
  X = tf.image.random_saturation(X, lower=0.75, upper=1.5)
  X = tf.image.random_hue(X, max_delta=0.15)
  X = tf.image.random_contrast(X, lower=0.75, upper=1.5)
  X = tf.image.resize(X, (224, 224))
  image = X / 255.0
  return image, features['label']
```

let's consider image augmentation transformations. You can see a variety of them here.

Each operation here is relatively inexpensive by themselves, but when combined they can become super expensive computationally. This is an operation that should not lock up the overall training

Whats happens when you map that transformation?

```
dataset = tfds.load('cats_vs_dogs',
                    split=tfds.Split.TRAIN)
```

```
augmented_dataset = dataset.map(augment)
```

Parallelizing data transformation

```
map(func, num_parallel_calls=...)
```

The map function takes a parameter called num parallel calls, which defines the number of cores in your CPU that it can use. This can greatly reduce the latency by parallelizing the mapping function across a multi-core CPU.

```
augmented_dataset = dataset.map(augment, num_parallel_calls=1)
```

Parallelizing data transformation

```
map(func, num_parallel_calls=...)
```

To use it is as easy as this, by setting the number of cores that you want to use, by setting it's to one like this, we're only using one core. So of course, the next question is, how do we figure out how many cores we can use?

```
augmented_dataset = dataset.map(augment, num_parallel_calls=1)
```

Maximizing the utilization of CPU cores

There's no hard and fast rule to set num parallel calls, but the easy and obvious answer would be to use the number of cores available in the CPU of your machine like this.

But with complex environmental scenarios were hardware was virtualized and keeps changing, this might not be a good idea. So next, we'll see how this can be solved with one of the more exciting features of tf.data.

```
# Get the number of available cpu cores
num_cores = multiprocessing.cpu_count()

# Set num_parallel_calls with 'num_cores'
augmented_dataset = dataset.map(augment, num_parallel_calls=num_cores)
```

Autotuning

- `tf.data.experimental.AUTOTUNE`
- Tunes the value dynamically at runtime
- Decides on the level of parallelism
- Tweaks values of parameters in transformations (tf.data)
  - Buffer size (`map, prefetch, shuffle, ...`)
  - CPU budget (`num_parallel_calls`)
  - I/O (num_parallel_reads)

While manually setting optimization parameters like num_parallel_calls of map can indeed help you get an improvement that can only get you so far. Setting these parameters by hand might not necessarily help you reach a CPU's peak usage.

This is what auto-tune comes in to automatically tune such values dynamically at runtime. Auto-tune delegates the decision about what level of parallelism to use for the tf.data runtime. Under the hood, it identifies the algorithm to use for performing the autotune optimization. This can be in terms of tweaking the buffer size of datasets or determining how much CPU budget to utilize.

Also, concerning parallelizing the calls by default, the behavior is to use the number of schedulable CPU cores. In general, this can be applied to most transformation operations in the tf.data data set API.

Autotune in practice

Instead of hard coding the number of parallel calls, we can simply just tell it to autotune like this

```
from tensorflow.data.experimental import AUTOTUNE

augmented_dataset = dataset.map(
                        augment,
                        num_parallel_calls=AUTOTUNE)
```

Maximizing utilization

Once we're training the first batch of data, we want to be preparing the next one and so on. One way to achieve this reducing idle time is to use the prefetch operation of the data set API.

With `prefetch`

Parallelizing data loading

prefetch(buffer_size)

The prefetch transformation prefetches elements from the input dataset ahead of time.

The number of items to prefetch should be equal to (or possibly higher than) the number of batches consumed by a single training step.

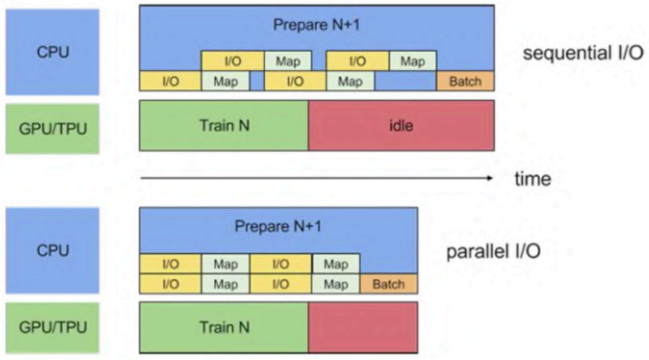
```
dataset = tfds.load('cats_vs_dogs', split=tfds.Split.TRAIN)

# With prefetch
train_dataset = dataset.map(format_image).prefetch(tf.data.experimental.AUTOTUNE)
```

So let's consider our typical code to load data from a dataset and then call a mapping function. Here we can add a prefetch method until its autotuned. This can be used to decouple the time from when data is produced to the time when data is consumed. Mostly,you can relate this to software pipelining.

Internally, this prefetched transformation runs a background thread and also makes use of a buffer to prefetch elements from the input data set ahead of time. The number of items to prefetch should be equal to or possibly a little higher than the number of batches consumed by a single training step.

You could either manually tune this value or set it to tf.data.experimental.AUTOTUNE which will instruct the tf.data runtime to handle this automatically.

Maximizing I/O utilization



Up until now we've seen it some transformation operations like map can be parallelized. But what about data extraction? Can we parallelize that as well?

At the top the CPU is striving to achieve parallelization in transformation, but the extraction of data from the disk is causing overhead and IO. Besides once the raw bytes are read into memory, it may also be necessary to deserialize and/or decrypt the data, which of course, requires additional computation.

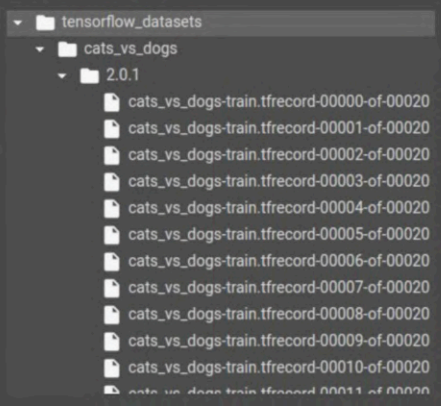
This overhead is present regardless of whether the data is stored locally or remotely, but it can be worse than the remote case if the data is not retrieve effectively. To mitigate the impact of the various data extraction overheads extraction needs to be parallelized interleaving the contents of each dataset.

The diagram at the bottom illustrates the data extraction parallelism through a process called Interleaving. Here the map and IO operations are interleaved whether executed in parallel in cycles of two. Let's take a look at how to implement this.

Let's inspect TFRecords of a TFDS

These are all the TFRecords generated when a Tensor flow data set is set up. You can see that all of them can be found in your local storage under the root directory. So let's read the data from these TFRecords by making use of parallelized data extraction and the TF data pipeline.

```
dataset = tfds.load(name='cats_vs_dogs',
                    split=tfds.Split.TRAIN)
```



Parallelizing data extraction

```
TFRECORDS_DIR = '/root/tensorflow_datasets/cats_vs_dogs/<dataset-version>/'
files = tf.data.Dataset.list_files(TFRECORDS_DIR +
                                   "cats_vs_dogs-train.tfrecord-*")

num_parallel_reads = 4

dataset = files.interleave(
                tf.data.TFRecordDataset, # map function
                cycle_length=num_parallel_reads, # ...
                num_parallel_calls=tf.data.experimental.AUTOTUNE) # ...
```

To enable interleaving we simply set up a files list to point to the list of files of TFRecord and specify the number of parallel reads that we want to use. Then we can use the interleave method to parallelize the data extraction step. The number of datasets to overlap can be specified by the cycle length argument while the num_parallel_calls argument can determine the level of parallelism. Similar to the prefetch and map transformations the interleave transformation supports tf.data.experimental.AUTOTUNE. And this will delegate the decision about the buffer size to use to the tf.data run time.

Performance considerations

- The Dataset APIs are designed to be flexible
- Most operations are commutative
- Order transformations accordingly

e.g., map, batch, shuffle, repeat, interleave, prefetch, etc.,

One of the biggest negative impacts. If you aren't careful with your coding is in the ordering of the functions we've been looking at

Map and Batch

First of all is the map transformation and this can have significant. Had with regards to scheduling and executing the user-defined mapping function

The map transformation has overhead in terms of

- Scheduling
- Executing the user-defined function

Map and Batch

Solution: Vectorize the user-defined function

```
dataset = dataset.batch(BATCH_SIZE).map(func)
```

or

```
options = tf.data.Options()
options.experimental_optimization.map_vectorization.enabled = True
dataset= dataset.with_options(options)
```

One way to reduce the impact of this overhead is to vectorize the function. When vectorizing, we'll have map operate over a batch of inputs at the same time instead of one by one. There are two ways that you can achieve this. First of course is just to define a batch by specifying a batch with a DOT batch method and then mapping that.

Map and Batch

Solution: Vectorize the user-defined function

```
dataset = dataset.batch(BATCH_SIZE).map(func)
```

or

```
options = tf.data.Options()
options.experimental_optimization.map_vectorization.enabled = True
dataset= dataset.with_options(options)
```

The second is to use an options object and on set map vectorization to be enabled. Then when creating the data set you can call it with options method and use these options.

Map and Cache

Next consider a scenario where your data needs a lot of transformations in that case you'll end up with a user-defined function that's quite expensive in operation.

So it's recommended that you apply the cash transform after the map transformation. This is done to avoid going through performing the computationally costly transformation repeatedly.

You can do this as long as the resulting data set can still fit into memory or local storage. If the user defined function increases the space required to store the data set beyond the cache capacity, consider pre-processing your data before your training to reduce resource usage.

```
# Use map before cache when the transformation is expensive
transformed_dataset = dataset.map(transforms).cache()
```

Shuffle and Repeat

As you might already know the repeat transformation repeats the input data and number of times. Whereas Shuffle randomizes the order of the datasets examples when the repeat transformation is applied. Before the shuffle transformation the epoch boundaries can be blurred that is some elements can be repeated before others even appear once. On the other hand if the shuffle transformation is applied before the repeat transformation, then performance might slow down at the beginning of each Epoch, related to the initialization of the internal state of the shuffle transformation.

From this you can probably come to realize that it's better to have Shuffle before repeat for stronger ordering. Guarantees while chaining repeat then Shuffle gives you better performance.

- Shuffling the dataset before applying repeat can cause slow downs
- `shuffle.repeat` for ordering guarantees
- `repeat.shuffle` for better performance

Map and (Interleave / Prefetch / Shuffle)

A number of Transformations including interleave prefetch and Shuffle maintain an internal buffer of elements. If the user defined-function passed into the map transformation changes the size of the elements, then the ordering of the map transformation and the Transformations that buffer elements affects the memory usage in general. We recommend choosing the order that results in a lower memory footprint. Unless different ordering is desirable for performance for example to enable fusing of the map and batch transformations.

- All transformations maintain an internal buffer
- Memory footprint is affected if map affects the size of elements
- Generally, have order that affects the memory usage the least