Copyright Notice

Boosting tf.data pipeline

- Rise of accelerators and parallelism
- High-performant tf.data input pipelines
- Adopt pipelines to different scenarios
- Learn better ways of using tf.data operations

ETL Revisited

Extract → Transform → Load

Local (HDD/SSD)

Remote (GCS/HDFS)

Shuffling & Batching

Decompression

Augmentation

Vectorization

. . .

Load transformed data to an accelerator

ETL Revisited

Extract → Transform → Load

Extract

Local (HDD/SSD)

Remote (GCS/HDFS)

Transform

Shuffling & Batching

Decompression

Augmentation

Vectorization

. . .

Load

Load transformed data to an accelerator

ETL Revisited

Extract → Transform → Load

Extract

Local (HDD/SSD)

Remote (GCS/HDFS)

Transform

Shuffling & Batching

Decompression

Augmentation

Vectorization

. . .

Load

Load transformed data to an accelerator

ETL Revisited

Extract

Transform

Load

Local (HDD/SSD)

Remote (GCS/HDFS)

Shuffling & Batching

Decompression

Augmentation

Vectorization

. . .

Load transformed data to an accelerator

What happens when you train a model?

Preprocessing + Training → Model training

CPU Utilization

GPU/TPU Utilization

Maximize

Data and its problems

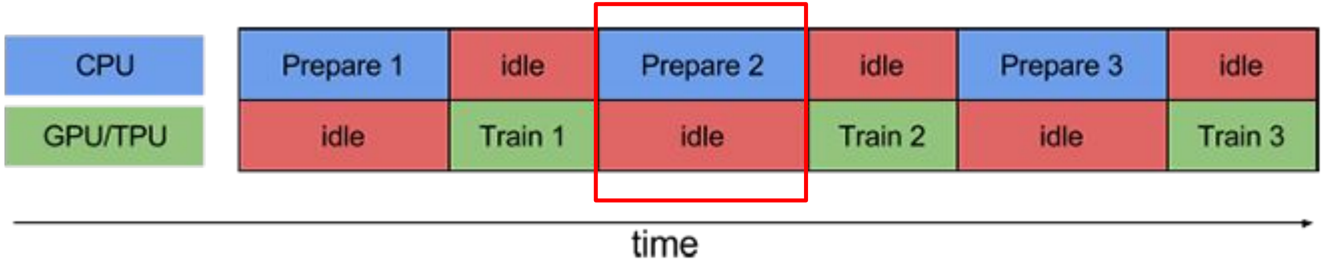
- Bound to come across fitting input data locally
- When **distributed training** expects **distributed data**
- Avoid having the same data on every machine

Data and models

Data and models

Data and models

Data and models



Data and models

Pipelining

Without pipelining

With pipelining

Improve training time with caching

- In-memory

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

- Disk

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

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=...)
```

Parallelism with tf.data

`tf.data.Dataset`

map

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

prefetch

```
prefetch(buffer_size)
```

interleave

```
interleave(func,
           cycle_length,
           num_parallel_calls=...)
```

Dataset

Data transformations

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

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

Consider the following transformation

```python
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']
```

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=...)
```

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

Parallelizing data transformation

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

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

Maximizing the utilization of CPU cores

```
# 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)

Autotune in practice

```
from tensorflow.data.experimental import AUTOTUNE
```

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

Maximizing utilization

With `prefetch`

Parallelizing data loading

`prefetch(buffer_size)`

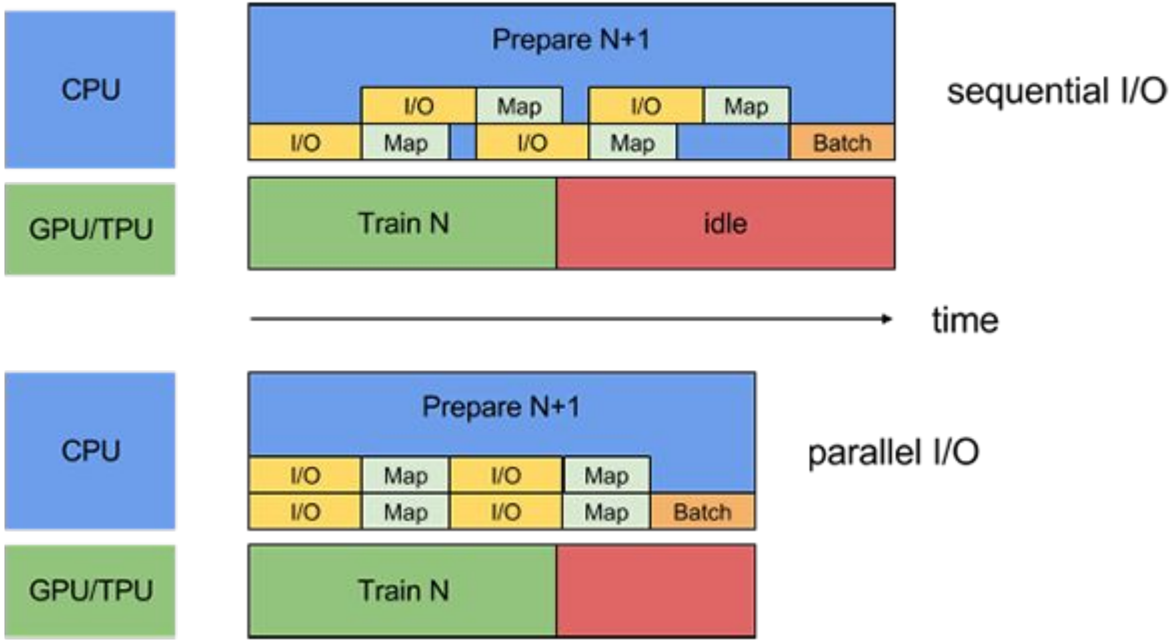
```
dataset = tfds.load('cats_vs_dogs', split=tfds.Split.TRAIN)

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

Parallelizing data loading

```
prefetch(buffer_size)
```

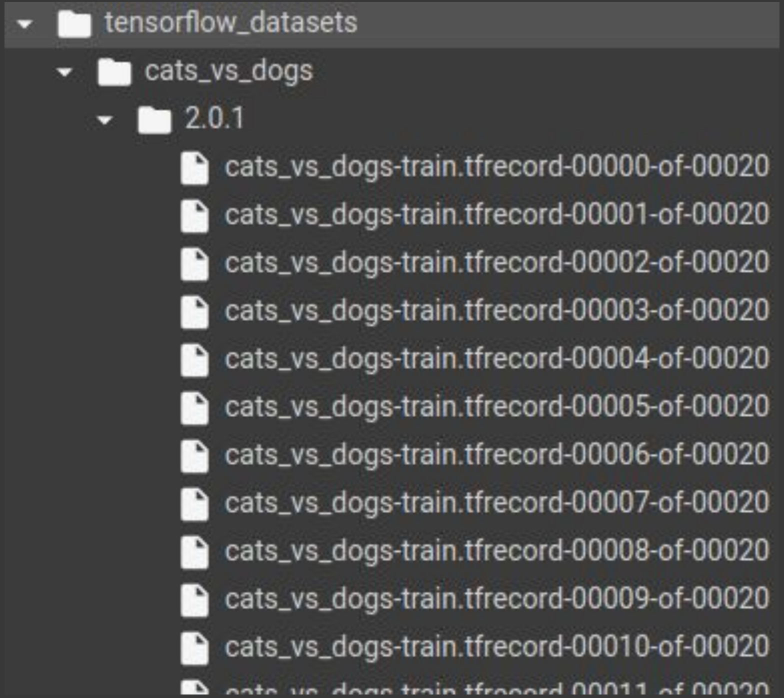
```
dataset = tfds.load('cats_vs_dogs', split=tfds.Split.TRAIN)

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

Maximizing I/O utilization



Let’s inspect TFRecords of a TFDS

```
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) # ...
```

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) # ...
```

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.,`

Map and Batch

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)
```

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)
```

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)
```

Map and Cache

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

Shuffle and Repeat

- 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)

- 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

Exercise

Question: Learn to classify the cats vs dogs dataset by creating an efficient training pipeline. At the end of the exercise, you would have learnt how to:

1. Parallelize extraction of stored TFRecords using interleave operation.
2. Parallelize transformation of extracted dataset using map operation.
3. Cache the processed dataset in memory for faster retrieval.
4. Parallelize the loading of cached dataset during training cycle.

Answer: **Colab**