Efficient and Distributed training with TensorFlow on Piz Daint

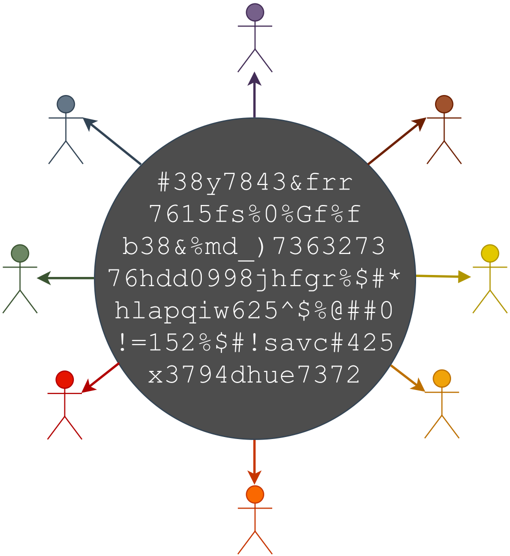
Input pipelines with TensorFlow's `tf.data` API

Rafael Sarmiento and Guilherme Peretti-Pezzi
ETHZürich / CSCS
Lugano, 14th-15th March 2019

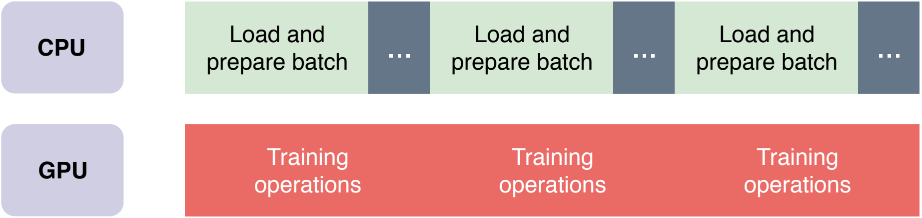
Outline

- Input pipelines
- [lab] Building input pipelines
- Read and write TFRecord files
- [lab] Reading and writing TFRecord files
- [lab] Decoding ImageNet data from a TFRecord file
- Optimizing pipelining
- Feeding datasets to models
- [lab] Feeding datasets to models

Consider large shared datasets



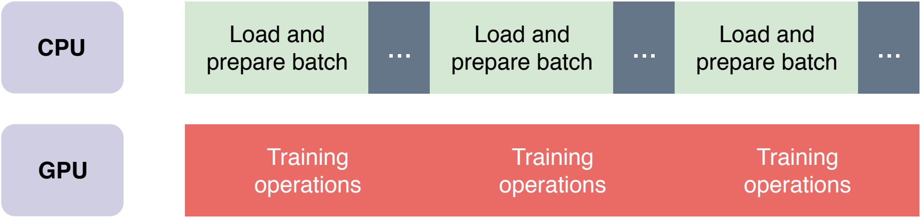
Pipelining


CPU
GPU
Load and prepare batch
...
Load and prepare batch
...
Load and prepare batch
...
Training operations
Training operations
Training operations

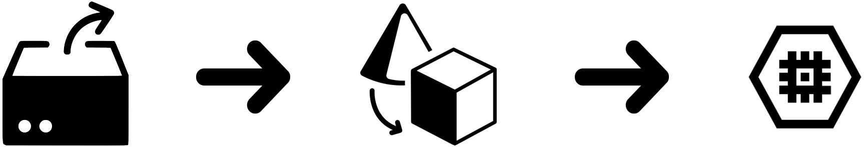
Pipelining


CPU
Load and prepare batch
Idle while training
Load and prepare batch
Idle while training
GPU
Waiting for data
Training operations
Waiting for data
Training operations

Pipelining


CPU
GPU
Load and prepare batch
...
Load and prepare batch
...
Load and prepare batch
...
Training operations
Training operations
Training operations

Input pipelines



Extract
Read data from persistent storage (HDD, SSD, GCS, HDFS, ...).

Transform
Use **CPU cores** to parse and perform preprocessing operations on the data, shuffling and batching.

Load
Load the transformed data onto the accelerator devices that execute the model.

TensorFlow's `tf.data` API

```
# Extract
dataset = tf.data.TFRecordDataset("./train.tfrecords")
# Transform
dataset = dataset.shuffle(1000)
dataset = dataset.batch(64)
dataset = dataset.repeat(100)
# Load
iterator = dataset.make_one_shot_iterator()
next_item = iterator.get_next()
```

TensorFlow's `tf.data` API: Extract

```
# Read data from TensorFlow's TFRecord file
dataset = tf.data.TFRecordDataset("./train.tfrecords")
```

TensorFlow's `tf.data` API: Extract

```
# Read data from cvs file:
#
# column1     column2     column3
#   0.0         4.1         dog
#   3.2         5.7         cat
dataset = tf.data.experimental.CsvDataset('train.csv',
                                          header=True,
                                          record_defaults=[tf.float32,
                                                           tf.float32,
                                                           tf.string])
```

TensorFlow's `tf.data` API: Extract

```python
# Read data from numpy arrays or `tf.Tensor`s in memory:
dataset = tf.data.Dataset.from_tensor_slices((x_numpy, y_numpy))
```

TensorFlow's `tf.data` API: Extract

```
# Read data with custom logic defined as a generator:
# def dataset_generator():
#     f = open('train.txt')
#     lines = f.readlines()
#     for l in lines:
#         x = np.array(l.split()[:4]).astype(np.float32)
#         y = np.array(l.split()[4]).astype(np.int32)
#         yield x, y
dataset = tf.data.Dataset.from_generator(dataset_generator,
                                         output_types=(tf.float32,
                                                       tf.int32))
```

TensorFlow's `tf.data` API: Transform

```python
# Reading data
dataset = tf.data.TFRecordDataset("./train.tfrecords")
```

TensorFlow's `tf.data` API: Transform

```python
# Shuffle groups of 4 components at the time
dataset = dataset.shuffle(4)
```

C D A B H E F G J L K I O M N P

TensorFlow's `tf.data` API: Transform

```
# Group data in batches of 4 components
dataset = dataset.batch(4)
```

TensorFlow's `tf.data` API: Transform

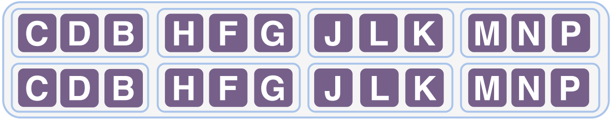
```python
# Filter out elements for which the user-defined function
# `filter_vowels` returns False.
dataset = dataset.filter(filter_vowels)
```

TensorFlow's `tf.data` API: Transform

```python
# Operations like map or filter can be applied batch by batch
# to take advantage of vectorial operations.
# Here we apply the user-defined function `change_color` to
# each of the components of the dataset batch by batch.
dataset = dataset.map(change_color)
```

TensorFlow's `tf.data` API: Transform

```
# Repeat twice the dataset to make 2 epochs
dataset = dataset.repeat(2)
```



TensorFlow's `tf.data` API: Transform

```python
# Reading data
dataset = tf.data.TFRecordDataset("./train.tfrecords")
```

A B C D E F G H I J K L M N O P

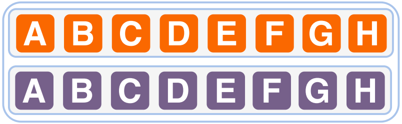
```python
# Split the input pipeline by shards
dataset = dataset.shard(num_ranks, rank)
```

A C E G I K M O (Rank 0)

B D F H J L N P (Rank 1)

TensorFlow's `tf.data` API: Transform

```python
# listing input files
dataset = tf.data.Dataset.list_files(['file_1', 'file_2'], <opts>)
```



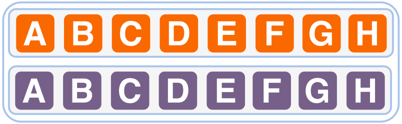
Content of file_1

Content of file_2

```python
# interleaving with block length 1
dataset = dataset.interleave(dataset_reader, block_length=1, <opts>)
```

TensorFlow's `tf.data` API: Transform

```
# listing input files
dataset = tf.data.Dataset.list_files(['file_1', 'file_2'], <opts>)
```



Content of file_1

Content of file_2

```
# interleaving with block length 2
dataset = dataset.interleave(dataset_reader, block_length=2, <opts>)
```

TensorFlow's `tf.data` API: Transform

... and much more!

TensorFlow's `tf.data` API: Load

```python
# Create a stream of data from the hard drive to the model
iterator = dataset.make_one_shot_iterator()
next_item = iterator.get_next()
```

TensorFlow's `tf.data` API: Load

```
# Create a stream of data from the hard drive to the model
iterator = dataset.make_one_shot_iterator()
features, label = iterator.get_next()
```

Iterating over data

```
with tf.Session() as sess:
    try:
        while True:
            features, label = sess.run(next_item)
            print('features: %s  |  label: %s' % (features, label))
    except tf.errors.OutOfRangeError:
        print('The dataset ran out of entries!')

# features: [[0.32762216 0.19132466]]  |  label: [2]
# features: [[0.40871843 0.02722579]]  |  label: [1]
# ...
# features: [[0.13172416 0.40897961]]  |  label: [1]
# The dataset ran out of entries!
```

[lab] Reading and writing TFRecord files

Let's open an empty notebook and create some simple input pipelines. A good starting point can be to generate random numpy data and create the pipeline including maps, filters, batching, shuffling and repeating operations. `tf.data.Dataset.from_tensor_slices` can be used for the *extract* phase.

Then we can run together the notebooks `getting_started_with_tensorflows_dataset_api_*.ipynb`, that are on the folder `input_pipelines/` to try other examples.

TFRecord file system

- TFRecord is a simple record-oriented binary format
- Data is stored as collections of meaningful units (records) in contrast to a byte-oriented filesystem, where the data is treated as an unformatted stream of bytes
- On the deep learning context each record would be an item of the dataset
- TFRecord is the format of data storage recommended for TensorFlow

TFRecord file system

```
dataset = tf.data.TFRecordDataset("./train.tfrecords")
```

Write data as a TFRecord

```python
# For representing data structures, for instance images.
def _bytes_feature(value):
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=[value]))

# For representing integer values, for instance integer labels.
def _int64_feature(value):
    return tf.train.Feature(int64_list=tf.train.Int64List(value=[value]))

# For representing float values. Useful for regression problems.
def _float_feature(value):
    return tf.train.Feature(float_list=tf.train.FloatList(value=[value]))
```

Write data as a TFRecord

```python
# 1. Open a TFRecord file for writing
with tf.io.TFRecordWriter(filename) as writer:
    ...
```

Write data as a TFRecord

```
# 2. Create a features dictionary entry with the column names (features)
# as keys and the data wrapped with the types defined before.
#
# Use your own logic and python utilities to encode as string or
# bytes the structured data before passing it to the feature wrappers.
features=tf.train.Features(
    feature={
        "height": _int64_feature(image.shape[0]),
        "width": _int64_feature(image.shape[1]),
        "image": _bytes_feature(image.tostring()),
        "label": _bytes_feature(label.encode(encoding="utf-8"))
    }
)
```

Write data as a TFRecord

```
# 3. Wrap the feature dictionary within an `Example` which will be
# a record on the TFRecord file.
# Here `example` is a protocol buffer message.
example = tf.train.Example(features)
```

[1] Protocol buffers are a mechanism for serializing structured data developed by Google

Write data as a TFRecord

```
# 4. Serialize and write the example on the TFRecord file.
writer.write(example.SerializeToString())
```

Write data as a TFRecord

```
with tf.io.TFRecordWriter(filename) as writer:
    for image, label in data:
        example = tf.train.Example(
            features=tf.train.Features(
                feature={
                    "height": _int64_feature(image.shape[0]),
                    "width": _int64_feature(image.shape[1]),
                    "image": _bytes_feature(image.tostring()),
                    "label": _bytes_feature(label.encode(encoding="utf-8"))
                }
            )
        )
        writer.write(example.SerializeToString())
```

Read data from a TFRecord file

```
# Data streamed from a TFRecord needs to be decoded back to numerical
# types. This is done as part of the input pipeline with the help of
# the `map` method of the `Dataset` objects.
dataset = tf.data.TFRecordDataset(filename)
dataset = dataset.map(decode)
dataset = ...
```

Read data from a TFRecord file

```
# 1. Define a parser with the features you whish to extract and
# specify their types. Pass the serialized example to the parser.
example = tf.parse_single_example(
    serialized_example,
    features={
        "image": tf.FixedLenFeature([], tf.string),
        "label": tf.FixedLenFeature([], tf.string),
    })
```

Read data from a TFRecord file

```
# 2. Cast/decode each feature to the proper types.
label = tf.cast(example["label"], tf.string)
image = tf.decode_raw(example["image"], tf.uint8)
```

Read data from a TFRecord file

```python
def decode(serialized_example):
    features = tf.parse_single_example(
        serialized_example,
        features={
            "height": tf.FixedLenFeature([], tf.int64),
            "width": tf.FixedLenFeature([], tf.int64),
            "image": tf.FixedLenFeature([], tf.string),
            "label": tf.FixedLenFeature([], tf.string),
        })
    label = tf.cast(features["label"], tf.string)
    width = tf.cast(features["width"], tf.int64)
    height = tf.cast(features["height"], tf.int64)
    image = tf.decode_raw(features["image"], tf.uint8)
    image = tf.reshape(image, (height, width, 3))
    return image, label
```

[lab] Reading and writing TFRecord files

Let's run the notebook `read_and_write_TFRecord_files.ipynb`. We are going to write two cat images to a TFRecord file and then we are going to read them to check that the they can be recovered correctly.

Following the same steps, write the MNIST dataset to a TFRecord file. We will use it later on a lab.

You can get the MNIST data as numpy arrays with

```
from tensorflow.keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Best practices for storing data

- Avoid storing data as multiple small files.
- Divide large datasets ($\gg$1 Gb) into TFRecord files of about 150 Mb.
- For smaller data sets (200MB-1GB), a single TFRecord is enough.
- Randomly divide the data into multiple files and then shuffle the filenames uniformly. This helps to decrease the size of the shuffle buffer.

[lab] Decoding an ImageNet TFRecord file

ImageNet is a large visual database designed for use in visual object recognition software research. It contains more than 14 million labeled images.

- Write an input pipeline reading one of the TFRecord files of the ImageNet dataset. You can find them at `/scratch/snx3000/stud71/imagenet`

  The identifiers for the images and the labels are `'image/class/label'` (`string`) and `'image/encoded'` (`int64`) respectively. These images are encoded from jpeg format. You can use the functions `tf.image.decode_jpeg()` and `tf.image.resize_images()` of the `tf.image` API to decode them and resize them to `(224,224)`.

- Visualize some images and check that they were recovered correctly.

- Loop over the dataset and find how many images the TFRecord that you chose has, and how many different classes

Finding bottlenecks on the input pipeline

- Monitor GPU utilization with `nvidia-smi -l2`. A constant GPU utilization of 80-100% is expected for large models.
- Compare the number of samples per second given by your model with the one given by a very simple one, using the same input pipeline.
- Generate a timeline and check if there are blank spaces

Optimizing pipelining: Parallelize file reading

```
dataset = tf.data.Dataset.TFRecordDataset(['train1.tfrecords',
                                           'train2.tfrecords',
                                           ...],
                                           num_parallel_reads=12)
```

Optimizing pipelining: Parallelize file reading

```
dataset = tf.data.Dataset.list_files(['train1.tfrecords',
                                      'train2.tfrecords', ...])
dataset = dataset.interleave(tf.data.TFRecordDataset,
                             cycle_length=120,
                             block_length=1,
                             num_parallel_calls=12)
```

Optimizing pipelining: Parallelize mapping

```
# write vectorized map functions and map after batching
# to take dvantage of vectorial operations
dataset = dataset.map(decode,
                      num_parallel_calls=12)
```

Optimizing pipelining: Parallelize mapping

```
# set the number of parallel calls dynamically at run time
dataset = dataset.map(decode,
                      num_parallel_calls=tf.data.experimental.AUTOTUNE)
```

Optimizing pipelining: Consider using fused transformations

```
# functionally, this is equivalent to a map followed by batch but
# may be more efficient than the two operations done individually
# tf.data.experimental.map_and_batch(
#     function,
#     batch_size,
#     num_parallel_batches=None,
#     drop_remainder=False,
#     num_parallel_calls=None)

dataset = dataset.apply(tf.data.experimental.map_and_batch(...))
```

Optimizing pipelining: Consider using fused transformations

```
# functionally, it is equivalent to a shuffle followed by repeat but
# may be more efficient than the two operations done individually
# tf.data.experimental.shuffle_and_repeat(
#     buffer_size,
#     count=None,
#     seed=None
# )

dataset = dataset.apply(tf.data.experimental.shuffle_and_repeat(...))
```

Optimizing pipelining: Consider `parallel_interleave`

```
# if a deterministic sequence on the iteration over the dataset
# is not necessary, sloppy interleave can enable additional
# performance optimizations
tf.data.experimental.parallel_interleave(
    map_func,
    cycle_length,
    block_length=1,
    sloppy=False,
    buffer_output_elements=None,
    prefetch_input_elements=None
)

# dataset = dataset.apply(tf.data.experimental.parallel_interleave(...))
```

Optimizing pipelining: Prefetching elements of the dataset

```
# prefetches elements of the dataset to the host memory.
#
# this can be set manually or dynamically at run time
# with tf.data.experimental.AUTOTUNE
dataset = dataset.prefetch(4)
```

Optimizing pipelining: Prefetching elements of the dataset

```
# prefetches elements of the dataset to the memory of the given device.
# if used, this must be the final transformation in the input pipeline.
dataset = dataset.apply(
    tf.data.experimental.prefetch_to_device('/gpu:0',
                                            buffer_size=4))
```

Feeding models with input pipelines (custom model)

```
# define the input pipeline
dataset = tf.data.Dataset.from_tensor_slices((x_numpy.astype(np.float32),
                                              y_numpy.astype(np.float32)))
dataset = dataset.batch(BATH_SIZE)
dataset = dataset.repeat(NUM_EPOCHS)
iterator = dataset.make_one_shot_iterator()
next_item = iterator.get_next()
```

Feeding models with input pipelines (custom model)

```
# define the model (y = m * x + n)
slope = tf.Variable(np.random.randn() , name='slope')
offset = tf.Variable(np.random.randn() , name='offset')
x, y = next_item
y_hat = slope * x + offset

# define the loss
loss = tf.losses.mean_squared_error(y_hat, y)

# choose an optimizer and create minimization and
# variable initialization operations
...
```

Feeding models with input pipelines (custom model)

```
# define the input pipeline
dataset = tf.data.Dataset.from_tensor_slices((x_numpy.astype(np.float32),
                                               y_numpy.astype(np.float32)))
dataset = dataset.batch(BATH_SIZE)
dataset = dataset.repeat(NUM_EPOCHS)
iterator = dataset.make_one_shot_iterator()
x, y = iterator.get_next()
```

Feeding models with input pipelines (custom model)

```
# define the model (y = m * x + n)
slope = tf.Variable(np.random.randn() , name='slope')
offset = tf.Variable(np.random.randn() , name='offset')
y_hat = slope * x + offset

# define the loss
loss = tf.losses.mean_squared_error(y_hat, y)

# choose an optimizer and create minimization and
# variable initialization operations
...
```

[lab] Feeding models with `tf.data` input pipelines

Let's run the notebooks in `input_pipelines/feeding_models` and check out how to feed models using Keras and the `tf.estimator` API.

TensorFlow Datasets

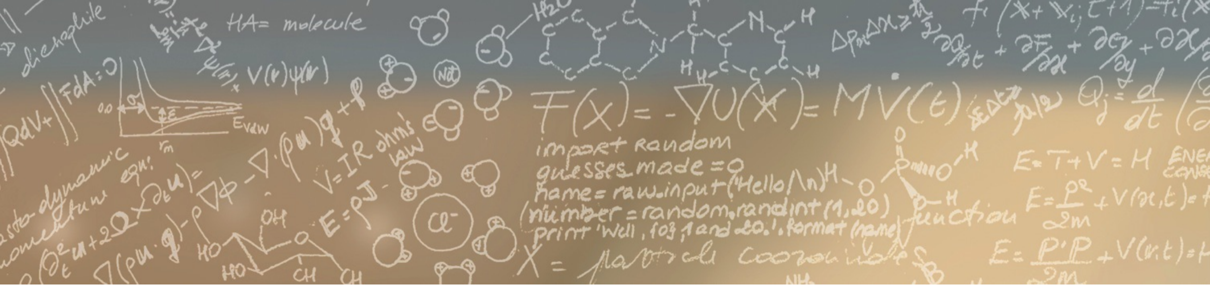
TensorFlow Datasets provides many public datasets as `tf.data.Datasets` and you can contribute back with your datasets.

https://github.com/tensorflow/datasets

[video] TensorFlow Datasets (TensorFlow Dev Summit 2019)

Useful info on input pipelines

- Importing data into TensorFlow
- Input Pipelines Performance
- General best practices for performance in TensorFlow
- [video] `tf.data`: Fast, flexible, and easy-to-use input pipelines (TensorFlow Dev Summit 2018)
- [video] Training Performance: A user's guide to converge faster (TensorFlow Dev Summit 2018)



Thank you for your attention!