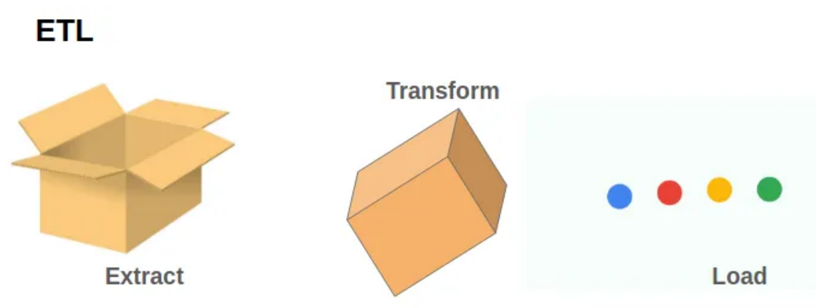
TensorFlow 2.0: tf.data API



ETL process

If you remember the time when only Queues were available in TensorFlow 1.x as the data structure for **ETL** pipeline (Extract / Transform / Load), and how it was sometimes difficult to manage some constraints and implicit pitfalls. But NOW there is very successful module that organize whole **ETL** pipeline: **tf.data**.

I want to give brief introduction and usage of tf.data API.

tf.data works with different types of input data:

1. CSV
2. NumPy
3. Text
4. Images
5. pandas.DataFrame
6. TF.Text
7. Unicode
8. TFRecord and tf.Example

For each type there are own classes and methods that work with them.

Two important phases in tf.data:

1. Create **Data Storage** that stores input data. All data elements become to be a Tensor object:

```
dataset = tf.data.Dataset.from_tensor_slices()
dataset = tf.data.Dataset.from_tensors()
```

2. Define **Transformations** and apply chain **Operations** to the dataset:

```
dataset.map()
dataset.batch()
    ...
```

For **NumPy** example:

```
dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
dataset = dataset.map(map_func=preprocess, num_parallel_calls=tf.data.experimental.AUTOTUNE)
dataset = dataset.batch(batch_size=batch_size).prefetch(buffer_size=prefetch_buffer_size)
```

As you see there is an interesting moment: from what kind of parts this chain operations consists?

Let’s consider the most usable functions.

1. **map:** apply the given transformation function to the input data. Allows to parallelize this process.

```
dataset.map(map_func=preprocess,
                      num_parallel_calls=tf.data.experimental.AUTOTUNE)
```

num_parallel_calls should be equal the number of processes that can be used for transformation.

tf.data.experimental.AUTOTUNE defines appropriate number of processes that are free for working.

2. **batch:** split dataset into subset of the given size.

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

3. **repeat:** repeat dataset several times. It’s useful when data ends up and the training process should be continued, then *repeat* function starts from the very beginning and training is continue *count* times.

```
dataset = dataset.repeat(count=NROF_REPETITIONS)
```

4. **shuffle:** very important function for the training data input pipeline. But this shuffle requires **buffer siz**e that is responsible for the number of elements that will be shuffled.

```
dataset = dataset.shuffle(buffer_size=len(IMAGE_PATHS))
```

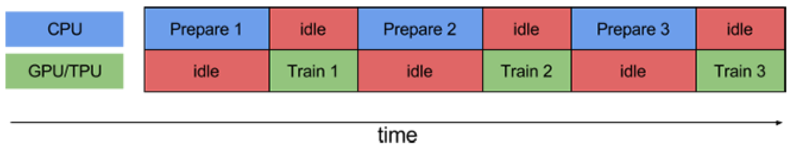
Every time when data was needed, it takes from the buffer. After that buffer is filled up with newest elements to the given buffer size.

5. **cache:** allows to cache elements of the dataset for future reusing. Cached data will be store in memory (by default) or in file.

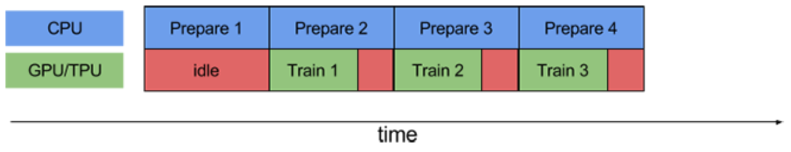
```
dataset = dataset.cache(filename=CACHE_PATH)
```

6. **prefetch:** TensorFlow showed very clear picture how to obtain train pipeline in terms of time and memory efficiency.

It was before:



Now:



prefetch doesn’t allow CPU stand idle. When model is training *prefetch* continue prepare data while GPU is busy.

```
dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
```

7. **interleave:** read data from different files and parallelize this process.

```
dataset = dataset.interleave(map_func=parse_files, cycle_length=NROF_READERS, block_length=1,
num_parallel_calls=tf.data.experimental.AUTOTUNE)
```

parse_files — read input data from the given files;

cycle_length — the number of input elements will be processed concurrently;

block_length — the number of consecutive elements to produce from each input element before cycling to another input element.

Best practices

1. The order of **map** and **batch** makes sense in terms of performance.

- use **map** and then **batch** when map is expensive function. Every batch will be constructed from elements that have the order that were presented in the dataset. But if *num_parallel_calls* used in map the order of the elements as presented in the given dataset will not be gurantied.
- use **batch** and then **map** when map is cheap function. In such case it makes sense to **vectorize** map function and process whole batch of elements at the same time. *num_parallel_calls* will not effect on the elements order.

2. Use **prefetch** at the end of the data input pipeline to prevent CPU stands idle. But if *map_func* increases the number of output data **prefetch**, **shuffle** and **repeat** should be used at the very beginning for the purpose of saving memory.

3. Use **cache** for caching data for the purpose to not spend time for data preprocessing.

4. Don’t forget to use *num_parallel_calls* in methods if it has such property.