Parallelization with TFDS

In this week's exercise, we'll go back to the classic cats versus dogs example, but instead of just naively loading the data to train a model, you will be parallelizing various stages of the Extract, Transform and Load processes. In particular, you will be performing following tasks:

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

Setup

In [1]:

```
import multiprocessing

import tensorflow as tf
import tensorflow_datasets as tfds

from os import getcwd
```

Create and Compile the Model

In [2]:

```
def create_model():
    input_layer = tf.keras.layers.Input(shape=(224, 224, 3))
    base_model = tf.keras.applications.MobileNetV2(input_tensor=input_layer,
                                                   weights='imagenet',
                                                   include_top=False)
    base_model.trainable = False
    x = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)
    x = tf.keras.layers.Dense(2, activation='softmax')(x)

    model = tf.keras.models.Model(inputs=input_layer, outputs=x)
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['acc'])
    return model
```

Naive Approach

Just for comparison, let's start by using the naive approach to Extract, Transform, and Load the data to train the model defined above. By naive approach we mean that we won't apply any of the new concepts of parallelization that we learned about in this module.

In [3]:

```
dataset_name = 'cats_vs_dogs'
filePath = f"{getcwd()}/../tmp2"
dataset, info = tfds.load(name=dataset_name, split=tfds.Split.TRAIN, with_info=True,
data_dir=filePath)
```

```
WARNING:absl:Found a different version 4.0.0 of dataset cats_vs_dogs in data_dir
/tf/week3/../tmp2. Using currently defined version 2.0.1.
```

In [4]:

```
print(info.version)
```

```
2.0.1
```

In [5]:

```
def preprocess(features):
    image = features['image']
    image = tf.image.resize(image, (224, 224))
    image = image / 255.0
    return image, features['label']
```

In [6]:

```
train_dataset = dataset.map(preprocess).batch(32)
```

The next step will be to train the model using the following code:

```
model = create_model()
model.fit(train_dataset, epochs=5)
```

Since we want to focus on the parallelization techniques, we won't go through the training process here, as this can take some time.

Parallelize Various Stages of the ETL Processes

The following exercises are about parallelizing various stages of Extract, Transform and Load processes. In particular, you will be tasked with performing following tasks:

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

We start by creating a dataset of strings corresponding to the `file_pattern` of the TFRecords of the cats_vs_dogs dataset.

In [7]:

```
file_pattern = f'{getcwd()}/../tmp2/{dataset_name}/{info.version}/{dataset_name}-train.tfrecord*'
files = tf.data.Dataset.list_files(file_pattern)
```

Let's recall that the TFRecord format is a simple format for storing a sequence of binary records. This is very useful because by serializing the data and storing it in a set of files (100-200MB each) that can each be read linearly greatly increases the efficiency when reading the data.

Since we will use it later, we should also recall that a `tf.Example` message (or protobuf) is a flexible message type that represents a `{"string": tf.train.Feature}` mapping.

Parallelize Extraction

In the cell below you will use the interleave operation with certain arguments to parallelize the extraction of the stored TFRecords of the cats_vs_dogs dataset.

Recall that `tf.data.experimental.AUTOTUNE` will delegate the decision about what level of parallelism to use to the `tf.data` runtime.

In [8]:

```
# EXERCISE: Parallelize the extraction of the stored TFRecords of
# the cats_vs_dogs dataset by using the interleave operation with
# cycle_length = 4 and the number of parallel calls set to tf.data.experimental.AUTOTUNE.
train_dataset = files.interleave(
    tf.data.TFRecordDataset,
    cycle_length=5,
    num_parallel_calls=tf.data.experimental.AUTOTUNE)
```

Parse and Decode

At this point the `train_dataset` contains serialized `tf.train.Example` messages. When iterated over, it returns these as scalar string tensors. The sample output for one record is given below:

```
<tf.Tensor: id=189, shape=(), dtype=string,
numpy=b'\n\x8f\xc4\x01\n\x0e\n\x05label\x12\x05\x1a\x03\n\x01\x00\n,\n\x0eimage/filename\x12
\n\x18\n\x16PetImages/Cat/4159.jpg\n\xcd\xc3\x01\n\x05image\x12...\xff\xd9'>
```

In order to be able to use these tensors to train our model, we must first parse them and decode them. We can parse and decode these string tensors by using a function. In the cell below you will create a `read_tfrecord` function that will read the serialized `tf.train.Example` messages and decode them. The function will also normalize and resize the images after they have been decoded.

In order to parse the `tf.train.Example` messages we need to create a `feature_description` dictionary. We need the `feature_description` dictionary because TFDS uses graph-execution and therefore, needs this description to build their shape and type signature. The basic structure of the `feature_description` dictionary looks like this:

```
feature_description = {'feature': tf.io.FixedLenFeature([], tf.Dtype, default_value)}
```

The number of features in your `feature_description` dictionary will vary depending on your dataset. In our particular case, the features are `'image'` and `'label'` and can be seen in the sample output of the string tensor above. Therefore, our `feature_description` dictionary will look like this:

```
feature_description = {
    'image': tf.io.FixedLenFeature((), tf.string, ""),
    'label': tf.io.FixedLenFeature((), tf.int64, -1),
}
```

where we have given the default values of `""` and `-1` to the `'image'` and `'label'` respectively.

The next step will be to parse the serialized `tf.train.Example` message using the `feature_description` dictionary given above. This can be done with the following code:

```
example = tf.io.parse_single_example(serialized_example, feature_description)
```

Finally, we can decode the image by using:

```
image = tf.io.decode_jpeg(example['image'], channels=3)
```

Use the code given above to complete the exercise below.

In [9]:

```
# EXERCISE: Fill in the missing code below.

def read_tfrecord(serialized_example):

    # Create the feature description dictionary
    feature_description = {
        'image': tf.io.FixedLenFeature((), tf.string, ""),
        'label': tf.io.FixedLenFeature((), tf.int64, -1),
    }
    # Parse the serialized_example and decode the image
    example = tf.io.parse_single_example(serialized_example, feature_description)
    image = tf.io.decode_jpeg(example['image'], channels=3)

    image = tf.cast(image, tf.float32)

    # Normalize the pixels in the image
    image = image * 1. / 255.

    # Resize the image to (224, 224) using tf.image.resize
    image = tf.image.resize(image, (244, 244))

    return image, example['label']
```

Parallelize Transformation

You can now apply the `read_tfrecord` function to each item in the `train_dataset` by using the `map` method. You can parallelize the transformation of the `train_dataset` by using the `map` method with the `num_parallel_calls` set to the number of CPU cores.

In [10]:

```
# EXERCISE: Fill in the missing code below.

# Get the number of CPU cores.
cores = multiprocessing.cpu_count()

print(cores)

# Parallelize the transformation of the train_dataset by using
# the map operation with the number of parallel calls set to
# the number of CPU cores.
train_dataset = train_dataset.map(read_tfrecord,
                                  num_parallel_calls=cores)
```

```
96
```

Cache the Dataset

In [11]:

```
# EXERCISE: Cache the train_dataset in-memory.
train_dataset = train_dataset.cache()
```

Parallelize Loading

In [12]:

```
# EXERCISE: Fill in the missing code below.

# Shuffle and batch the train_dataset. Use a buffer size of 1024
# for shuffling and a batch size 32 for batching.
train_dataset = train_dataset.shuffle(1024).batch(32)

# Parallelize the loading by prefetching the train_dataset.
# Set the prefetching buffer size to tf.data.experimental.AUTOTUNE.
train_dataset = train_dataset.prefetch(tf.data.experimental.AUTOTUNE)
```

The next step will be to train your model using the following code:

```
model = create_model()
model.fit(train_dataset, epochs=5)
```

We won't go through the training process here as this can take some time. However, due to the parallelization of the various stages of the ETL processes, you should see a decrease in training time as compared to the naive approach depicted at beginning of the notebook.

Submission Instructions

In [ ]:

```
# Now click the 'Submit Assignment' button above.
```

When you're done or would like to take a break, please run the two cells below to save your work and close the Notebook. This frees up resources for your fellow learners.

In [ ]:

```
%%javascript
<!-- Save the notebook -->
IPython.notebook.save_checkpoint();
```

In [ ]:

```
%%javascript
<!-- Shutdown and close the notebook -->
window.onbeforeunload = null
window.close();
IPython.notebook.session.delete();
```