TensorFlow and Keras Lab Exercise#2

Palacode Narayana Iyer Anantharaman

21 Aug 2018

References

- TensorFlow References
  - https://www.tensorflow.org/performance/datasets_performance
  - https://www.tensorflow.org/api_docs/python/tf/metrics
  - https://www.tensorflow.org/api_docs/python/tf/estimator/DNNClassifier
- Dataset (EMNIST) References
  - https://www.kaggle.com/crawford/emnist/home
  - https://www.nist.gov/itl/iad/image-group/emnist-dataset

Figure, Code Credits

- Some of the figures and code illustrations in this presentation are directly copy/pasted from different parts of TensorFlow website: https://www.tensorflow.org/

- They are reproduced here for the purpose of classroom explanations

Goal of this lab session

- Perform classification tasks using Data API and Estimators in a pipelined parallel training procedure

- Today: Work with Data API, Estimators, use pipelining to improve training speed. You are provided with the EMIST data that has handwritten letters, digits similar to MNIST

Key Topics in TensorFlow

- Data API used with prefetching
- Pre-made Estimators: DNNClassifier
- TensorFlow metrics module: Precision, Recall, F1 Score, Accuracy
- TensorBoard to view the compute graph and loss behavior

Using Datasets API for Pipeline Performance

- Mini batches can be pre-fetched to run them in parallel with training
- Within a minibatch we can parallelize data transformation (such as augmentation) by running several map functions in parallel
- If the datasets are set up in remote servers, we can interleave data from many sources by parallelizing I/O
- Data API supports all the above use cases
- In today's lab we will implement the first use case as above

TensorFlow ETL model

Input Pipeline Structure

A typical TensorFlow training input pipeline can be framed as an ETL process:

1. **Extract**: Read data from persistent storage -- either local (e.g. HDD or SSD) or remote (e.g. GCS or HDFS).
2. **Transform**: Use CPU cores to parse and perform preprocessing operations on the data such as image decompression, data augmentation transformations (such as random crop, flips, and color distortions), shuffling, and batching.
3. **Load**: Load the transformed data onto the accelerator device(s) (for example, GPU(s) or TPU(s)) that execute the machine learning model.

This pattern effectively utilizes the CPU, while reserving the accelerator for the heavy lifting of training your model. In addition, viewing input pipelines as an ETL process provides structure that facilitates the application of performance optimizations.

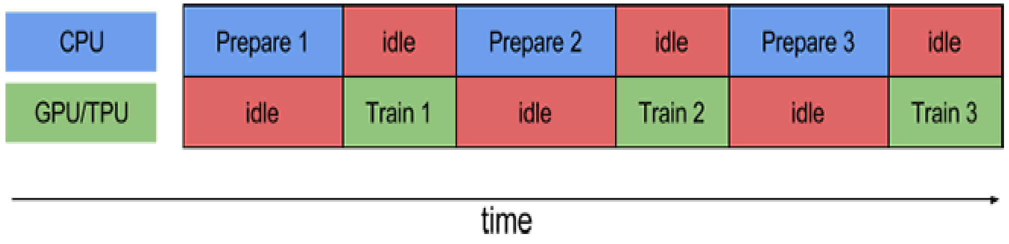
When using the `tf.estimator.Estimator` API, the first two phases (Extract and Transform) are captured in the `input_fn` passed to `tf.estimator.Estimator.train`. In code, this might look like the following (naive, sequential) implementation:

Naïve Implementation Example

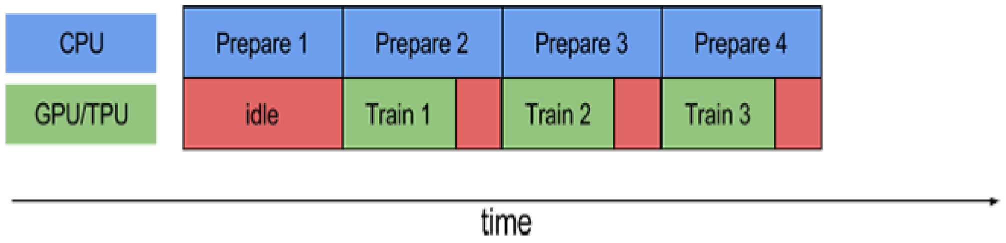
```
def parse_fn(example):
  "Parse TFExample records and perform simple data augmentation."
  example_fmt = {
    "image": tf.FixedLengthFeature((), tf.string, ""),
    "label": tf.FixedLengthFeature((), tf.int64, -1)
  }
  parsed = tf.parse_single_example(example, example_fmt)
  image = tf.image.decode_image(parsed["image"])
  image = _augment_helper(image)  # augments image using slice, reshape, resize_bilinear
  return image, parsed["label"]
def input_fn():
  files = tf.data.Dataset.list_files("/path/to/dataset/train-*.tfrecord")
  dataset = files.interleave(tf.data.TFRecordDataset)
  dataset = dataset.shuffle(buffer_size=FLAGS.shuffle_buffer_size)
  dataset = dataset.map(map_func=parse_fn)
  dataset = dataset.batch(batch_size=FLAGS.batch_size)
  return dataset
```

Pipelining : Preparing the batch of examples

Without pipelining, the CPU and the GPU/TPU sit idle much of the time:



With pipelining, idle time diminishes significantly:



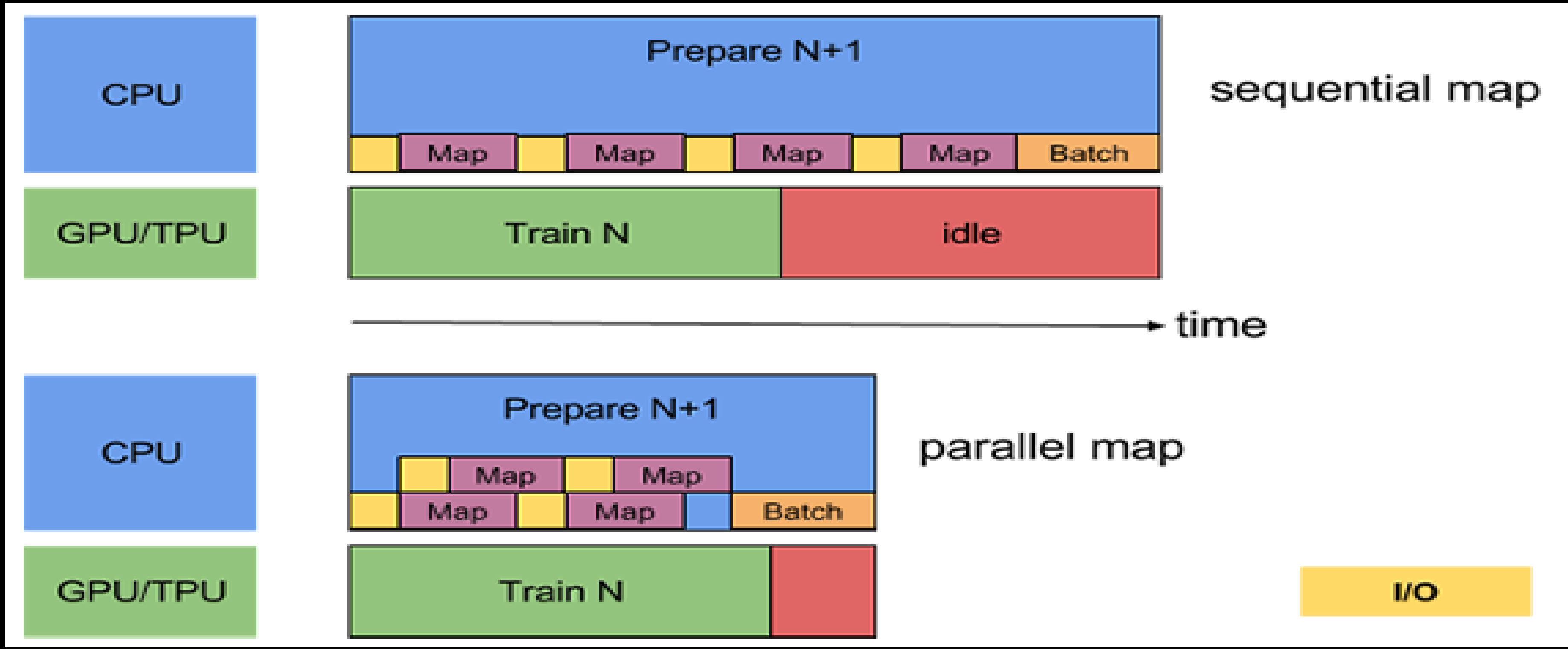
To apply this change to our running example, change:

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

to:

```
dataset = dataset.batch(batch_size=FLAGS.batch_size)
dataset = dataset.prefetch(buffer_size=FLAGS.prefetch_buffer_size)
return dataset
```

Pipelining – Preprocess the data



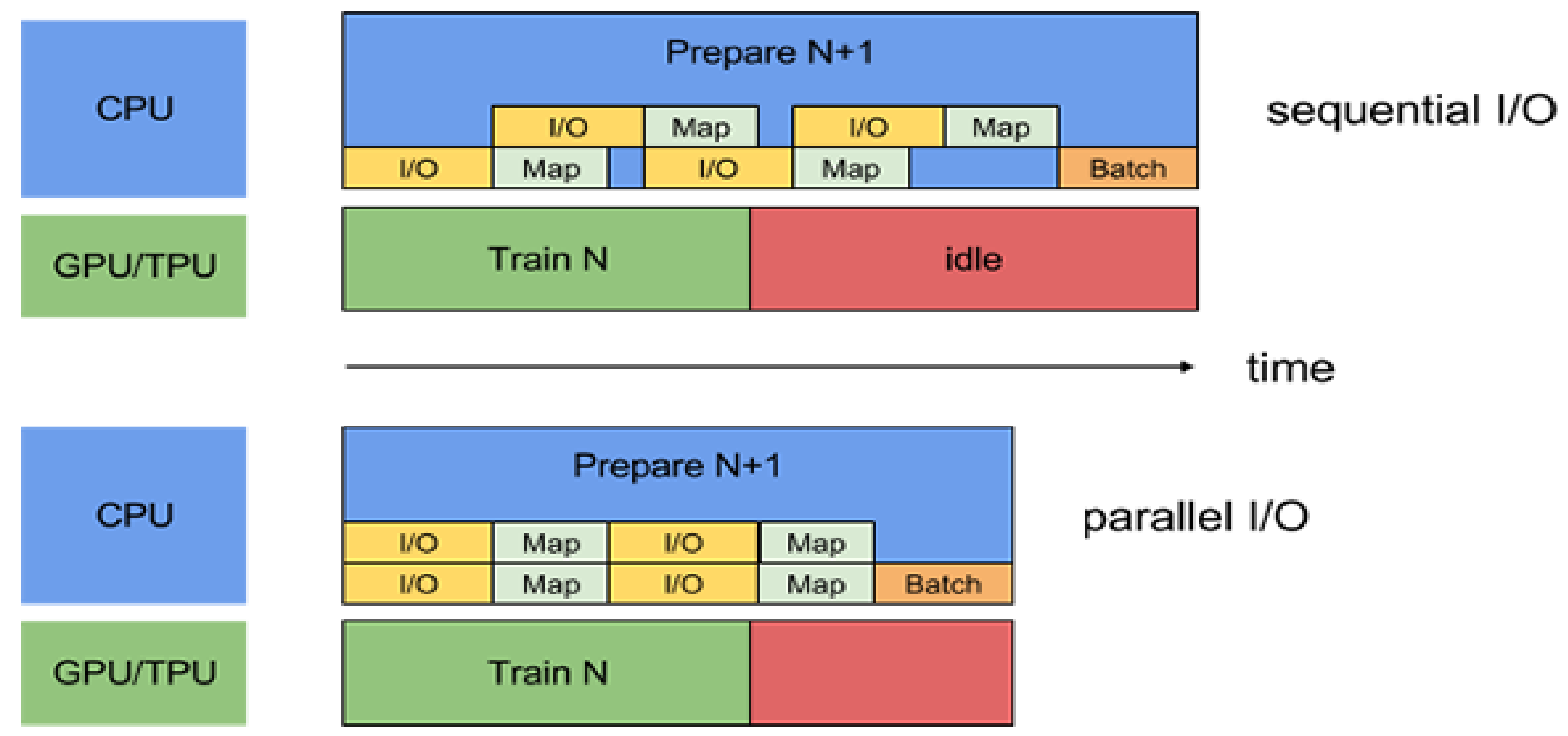
To apply this change to our running example, change:

```
dataset = dataset.map(map_func=parse_fn)
```

to:

```
dataset = dataset.map(map_func=parse_fn, num_parallel_calls=FLAGS.num_parallel_calls)
```

Pipelining – Interleaving with parallel I/O



To apply this change to our running example, change:

```
dataset = files.interleave(tf.data.TFRecordDataset)
```

to:

```
dataset = files.apply(tf.contrib.data.parallel_interleave(
    tf.data.TFRecordDataset, cycle_length=FLAGS.num_parallel_readers))
```

Lab Assignment#2, Task#1

- You are provided with the dataset for classification of EMNIST images. These are 28x28 images, each image belongs to a letter or digit (26 + 26 + 10 = 62 classes)
  - Link: https://drive.google.com/open?id=1QoqEsdO2mxJy7tGIWAn4tkcFLQVJj94

- In Task#1, you are required to use the Data API of TensorFlow and write a data generator input function compatible with the DNNClassifier Estimator. Use EMNIST ByClass: 814,255 characters. 62 unbalanced classes.

- Perform necessary steps to parse the binary dataset (MNIST format). Data augmentation is not necessary for this exercise

- Implement the support for pipelining data preparation with training

Task#2

- Implement the DNNClassifier based model to classify across 62 classes
- Empirically decide on size of dataset to use
- Initially start with about 70000 samples (similar to MNIST) and scale to over 300K and above
- Report the metrics : See the metrics page of TensorFlow – You are required to compute Precision, Recall, F1 Score, Confusion Matrix for each class