Продолжение темы “Data Input Pipeline”.

Dataset API

Не так давно TensorFlow выпустил новую библиотеку, с более удобным интерфейсом. tf.data позволяет сделать обработку данных более эффективно, помимо распараллеливания предобработки данных, как в TF Queue, оптимизирован итерационный процесс подготовки батчей во время обучения, тем самым позволило увеличить скорость обучения.

tf.data работает с двумя основными этапами:

1. Создание **хранилища**, состоящие из тензоров, которые могут быть разных типов. Это элементы исходных данных для работы с заданной моделью. `tf.data.Dataset.from_tensor_slices()`, `tf.data.Dataset.from_tensors()`
2. **Преобразования** над данными, лежащие в хранилище, для получения окончательного вида входных данных. `tf.data.Dataset.map()`, `tf.data.Dataset.batch() etc.`

Загрузка данных с помощью tf.data можно сделать для разного вида данных, точнее в каком они виде и где хранятся.

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

Как указано в списке, поддерживается различный формат данных. Для каждого подхода есть свои классы и методы в tf.data, которые позволяют делать загрузку данных для дальнейшей обработки и настройки процесса работы с данными.

Пример того, как это может выглядеть для загрузки NumPy массивов:

```
dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
dataset = dataset.shuffle(buffer_size=len(x_train))
dataset = dataset.map(map_func=preprocess)
dataset = dataset.batch(batch_size=BATCH_SIZE)
```

Метод *preprocess* -- для преобразования данных или та функция, которая подается в качестве аргумента *map_func*. Метод работает с типом tf.Tensor.

Методы shuffle и batch -- и есть перемешивание выборки и деление на батчи, соответственно.

tf.data.TFRecordDataset

Класс для работы с TFRecord файлами.

Что такое TFRecord файл?
TensorFlow создал свой формат файла для хранения сериализованных данных.
Сериализация данных позволяет легко оперировать ими.
TFRecord -- хороший способ представления данных, когда файлы нужно считывать по сети.

tf.Example -- это то, что непосредственно создает маппинг (соответствие) между названием характеристики и ее значением для дальнейшей сериализации. Такой подход позволяет всегда хранить описание структуры данных. С помощью tf.Example создается TFRecord файл.

```
feature = {
    'feature0': _int64_feature(feature0),
    'feature1': _int64_feature(feature1),
    'feature2': _bytes_feature(feature2),
    'feature3': _float_feature(feature3),
}
```

```
example_proto = tf.train.Example(features=tf.train.Features(feature=feature))
return example_proto.SerializeToString()
```

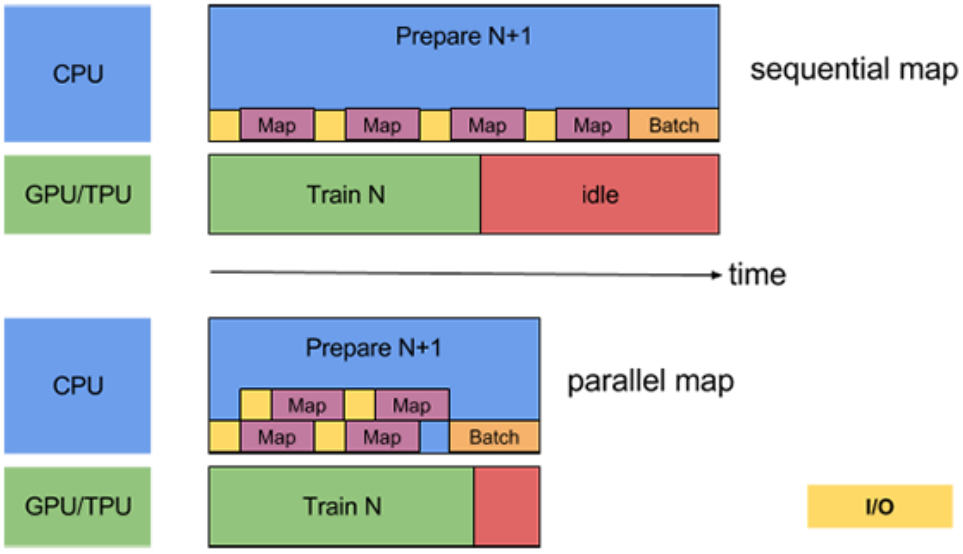
Файл записывается следующим образом:

```
filename = 'test.tfrecord'
writer = tf.data.experimental.TFRecordWriter(filename)
writer.write(serialized_features_dataset)
```

Сериализованные данные записываются с помощью **TFRecordWriter** в файл с расширением .tfrecord

Оптимизация предобработки и подготовки данных для обучения:

map
Функция map позволяет распараллелить процесс обработки при помощи аргумента num_parallel_calls, его значение определяется тем, сколько процессов свободно для распараллеливания. Но в Tensorflow есть tf.data.experimental.AUTOTUNE, позволяющая определить в момент запуска подходящее значение.




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

Также имеет значение в каком порядке вызывать функцию **map(...)** и **batch(...)**. От этого зависит, как будет выглядеть функция предобработки данных.
Если **map(..)** вызывается перед **batch(...)**, то в функцию предобработки приходит **один** элементы из набора данных, а если функция **map(...)** вызывается после **batch(...)**, то в функцию предобработки приходит уже **батч** элементов.
Ниже указано, как это может влиять на производительность.

Repeat and Shuffle

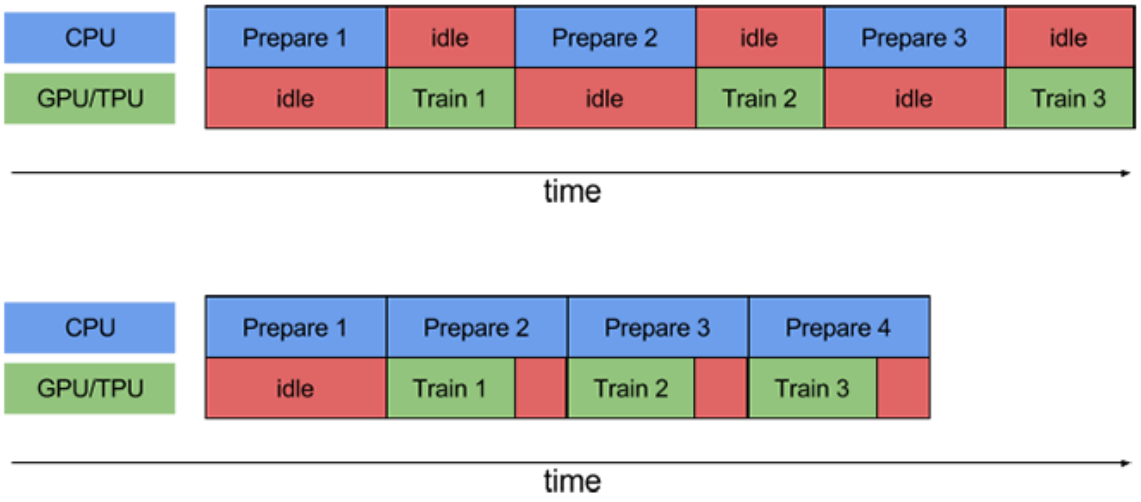
Датасеты по размеру могут быть небольшими, и чтобы можно было размер эпохи (количество итераций в одной эпохи) делать независимо от размера датасета, то функция *repeat* позволяет заново проходить по датасету, если все элементы были просмотрены, модель на них уже обучалась.
Параметр *count* позволяет задать, какое количество раз повторно проходить по датасету, по умолчанию этот процесс будет неопределен, сколько необходимо будет во время обучения, столько раз и будет повторов.

Функция *shuffle* позволяет перемешивать данные, но нужно указать *buffer_size*. Данный параметр отвечает за то, сколько элементов будет храниться в буфере, которые будут перемешиваются. Если один из элементов или батч элементов были взяты из буфера, то следующее количество элементов будут браться из датасета и заполнять буфер до заданного значения.

В то же время надо понимать, как это будет работать в связке с функцией repeat: если **repeat** стоит перед **shuffle**, то некоторые элементы могут появляться намного

чаще, в то время как другие могут быть использованы всего лишь один раз. Но такой порядок дает более лучшую производительность.
prefetch

При обучении, пока выполняется следующий шаг, CPU простаивает, пока не закончится выполнение шага. Для того чтобы CPU не простаивала, метод prefetch позволяет продолжать CPU обрабатывать данные для следующего шага, пока идет обучение:




```
dataset = dataset.prefetch(buffer_size=PREFETCH_BUFFER_SIZE)
```

Параметр buffer_size указывает, какое количество элементов будет обрабатываться во время обучения.

```
dataset = dataset.map(map_func=preprocess)
dataset = dataset.batch(batch_size=BATCH_SIZE)
dataset = dataset.prefetch(buffer_size=PREFETCH_BUFFER_SIZE)
```

interleave

Данный метод позволяет распараллелить процесс извлечения данных. Если необходимо считать данные с нескольких файлов, то данный метод очень подходит, так как позволяет одновременно считывать данные с нескольких ресурсов.
tf.data.Dataset.interleave имеет несколько важных параметров:
cycle_length -- сколько элементов (файлов и т.п.) будет обрабатываться одновременно.
num_parallel_calls -- кол-во потоков для распараллеливания.
block_length -- кол-во элементов, которое будет браться из датасета, когда происходит считывание.

Best Practices:

1. Порядок вызовов методов map и batch имеет значение:

- если сначала идет вызов *map*, а затем *batch*, то объект получает доступ ко всем элементам выборки и функция предобработки работает с **одним** элементом выборки. Каждый батч будет формироваться согласно тому порядку, который изначально представлен в данных при создании объекта *dataset*. При распараллеливании (num_parallel_calls) каждый следующий батч будет формироваться уже из тех элементов, которые прошли предобработку и готовы к обучению. В таком случае порядок элементов будет уже другой. Важно учесть, если порядок в данных имеет значение. Если функция по предобработки вычислительно сложная, то именно в такой последовательности нужно вызывать методы *map* и *batch*.
- Функция, предназначенная для предобработки данных, подаваемая в метод map, может содержать вычислительно недорогие операции. В таком случае стоит изменить порядок вызово методов *map* и *batch*: сначала *batch*, затем *map*. При таком вызове объект *dataset* будет видеть только данные с размером BATCH_SIZE. Но функция предобработки будет уже работать не с одним элементом, а с батчем (т.е. необходимо векторизовать). Это оправдано, если функция преобработки вычислительно несложная, и тогда оптимальнее обрабатывать батчем элементы, а не по одному, поскольку на выстраивание процесса методом map тоже уходят ресурсы, и порой это выходит гораздо дороже, если предобработка идет по одному элементу. Стоит учесть, что при такой последовательности вызовов методов порядок элементов поступающих в батч не меняется. А указание num_parallel_calls в map позволяет распараллелить предобработку, но порядок элементов не изменится.

2. *prefetch* лучше использовать в самом конце цепочке вызовов методов для объекта dataset. Поскольку это позволит, как можно меньше по времени простаивать CPU, пока идет обучение. Но в то же время, если функция обработки данных на выходе увеличивает количество выдаваемых параметров, то *prefetch, repeat, shuffle* лучше ставить в начале, чтобы уменьшить размер используемой памяти.
3. *tf.data.Dataset.cache* -- позволяет закешировать предобработанные элементы, чтобы в дальнейшем не тратить время на их предобработку. Позволяет хранить в памяти, либо в файле не диске, указав путь в качестве аргумента.
4. Используйте num_parallel_calls в тех методах, где это возможно. Но важно правильно подобрать кол-во выделяемых поток, так как если их будет задано больше, чем возможно, то это может замедлить работу, и при не распараллеливании было бы даже быстрее. Поэтому можно использовать параметр *tf.data.experimental.AUTOTUNE*, чтобы делегировать решение на кол-во выделяемых ресурсов на tf.data.runtime.