



Building a data pipeline

Using Tensorflow `tf.data` for text and images

← Previous page

Next page →

Motivation

Building the input pipeline in a machine learning project is always long and painful, and can take more time than building the actual model. In this tutorial we will learn how to use TensorFlow's Dataset module `tf.data` to build efficient pipelines for images and text.

This tutorial is among a series explaining how to structure a deep learning project: This tutorial is among a series explaining how to structure a deep learning project:

- installation, get started with the code for the projects
- (TensorFlow) explain the global structure of the code
- **this post: how to build the data pipeline**
- (Tensorflow) how to build the model and train it

Goals of this tutorial

- learn how to use `tf.data` and the best practices
- build an efficient pipeline for loading images and preprocessing them
- build an efficient pipeline for text, including how to build a vocabulary

An overview of `tf.data`


The `Dataset` API allows you to build an asynchronous, highly optimized data pipeline to prevent your GPU from data starvation. It loads data from the disk (images or text), applies optimized transformations, creates batches and sends it to the GPU. Former data pipelines made the GPU wait for the CPU to load the data, leading to performance issues.

Before explaining how `tf.data` works with a simple example, we'll share some great official resources:

- API docs for `tf.data`
- API docs for `tf.contrib.data` : new features still in beta mode. Contains useful functions that will soon be added to the main `tf.data`
- Datasets Quick Start: gentle introduction to `tf.data`
- Programmer's guide: more advanced and detailed guide to the best practices when using Datasets in TensorFlow
- Performance guide: advanced guide to improve performance of the data pipeline
- Official blog post introducing Datasets and Estimators. We don't use Estimators in our code examples so you can safely ignore them for now.
- Slides from the creator of `tf.data` explaining the API, best practices (don't forget to read the speaker notes below the slides)
- Origin github issue for Datasets: a bit of history on the origin of `tf.data`
- Stackoverflow tag for the Datasets API

Introduction to `tf.data` with a Text Example

Let's go over a quick example. Let's say we have a `file.txt` file containing sentences



```
I use Tensorflow
You use PyTorch
Both are great
```

Let's read this file with the `tf.data` API:

```
dataset = tf.data.TextLineDataset("file.txt")
```

Let's try to iterate over it

```
for line in dataset:  
    print(line)
```

We get an error

```
> TypeError: 'TextLineDataset' object is not iterable
```

Wait... What just happened ? I thought it was supposed to read the data.

Iterators and transformations

What's really happening is that `dataset` is a node of the Tensorflow `Graph` that contains instructions to read the file. We need to initialize the graph and evaluate this node in a Session if we want to read it. While this may sound awfully complicated, this is quite the opposite : now, even the dataset object is a part of the graph, so you don't need to worry about how to feed the data into your model !

We need to add a few things to make it work. First, let's create an `iterator` object over the dataset

```
iterator = dataset.make_one_shot_iterator()  
next_element = iterator.get_next()
```

The `one_shot_iterator` method creates an iterator that will be able to iterate once over the dataset. In other words, once we reach the end of the dataset, it will stop yielding elements and raise an Exception.

Now, `next_element` is a graph's node that will contain the next element of iterator over the Dataset at each execution. Now, let's run it

```
with tf.Session() as sess:
```

```
for i in range(3):  
    print(sess.run(next_element))
```

```
>'I use Tensorflow'  
>'You use PyTorch'  
>'Both are great'
```

Now that you understand the idea behind the `tf.data` API, let's quickly review some more advanced tricks. First, you can easily apply transformations to your dataset. For instance, splitting words by space is as easy as adding one line

```
dataset = dataset.map(lambda string: tf.string_split([string]).values)
```

Shuffling the dataset is also straightforward

```
dataset = dataset.shuffle(buffer_size=3)
```

It will load elements 3 by 3 and shuffle them at each iteration.

You can also create batches

```
dataset = dataset.batch(2)
```

and pre-fetch the data (in other words, it will always have one batch ready to be loaded).

```
dataset = dataset.prefetch(1)
```

Now, let's see what our iterator has become

```
iterator = dataset.make_one_shot_iterator()  
next_element = iterator.get_next()  
with tf.Session() as sess:  
    print(sess.run(next_element))
```

```
>[['Both' 'are' 'great']  
  ['You' 'use' 'PyTorch']]
```

and as you can see, we now have a batch created from the shuffled Dataset !

All the nodes in the Graph are assumed to be batched: every Tensor will have `shape = [None, ...]` where None corresponds to the (unspecified) batch dimension

Why we use initializable iterators

As you'll see in the `input_fn.py` files, we decided to use an initializable iterator.

```
dataset = tf.data.TextLineDataset("file.txt")  
iterator = dataset.make_initializable_iterator()  
next_element = iterator.get_next()  
init_op = iterator.initializer
```

Its behavior is similar to the one above, but thanks to the `init_op` we can chose to “restart” from the beginning. This will become quite handy when we want to perform multiple epochs !

```
with tf.Session() as sess:  
    #Initialize the iterator  
    sess.run(init_op)  
    print(sess.run(next_element))  
    print(sess.run(next_element))  
    #Move the iterator back to the beginning  
    sess.run(init_op)  
    print(sess.run(next_element))  
  
> 'I use Tensorflow'  
'You use PyTorch'  
'I use Tensorflow' # Iterator moved back at the beginning
```

As we use only one session over the different epochs, we need to be able to restart the iterator. Some other approaches (like `tf.Estimator`) alleviate the need of using `initializable` iterators by creating a new session at each epoch. But this comes at a cost: the weights and the graph must be re-loaded and re-initialized with each call to `estimator.train()` or `estimator.evaluate()` .

Where do I find the data pipeline in the code examples ?

The `model/input_fn.py` defines a function `input_fn` that returns a dictionary that looks like

```
images, labels = iterator.get_next()
iterator_init_op = iterator.initializer

inputs = {'images': images, 'labels': labels, 'iterator_init_op': iterator_i
```

This dictionary of inputs will be passed to the model function, which we will detail in the [next post](#).

Building an image data pipeline

Here is what a Dataset for images might look like. Here we already have a list of `filenames` to jpeg images and a corresponding list of `labels`. We apply the following steps for training:

1. Create the dataset from slices of the filenames and labels
2. Shuffle the data with a buffer size equal to the length of the dataset. This ensures good shuffling (cf. [this answer](#))
3. Parse the images from filename to the pixel values. Use multiple threads to improve the speed of preprocessing
4. (Optional for training) Data augmentation for the images. Use multiple threads to improve the speed of preprocessing
5. Batch the images
6. Prefetch one batch to make sure that a batch is ready to be served at all time

```
dataset = tf.data.Dataset.from_tensor_slices((filenames, labels))
dataset = dataset.shuffle(len(filenames))
dataset = dataset.map(parse_function, num_parallel_calls=4)
dataset = dataset.map(train_preprocess, num_parallel_calls=4)
dataset = dataset.batch(batch_size)
dataset = dataset.prefetch(1)
```

The `parse_function` will do the following:

- read the content of the file
- decode using jpeg format

- convert to float values in `[0, 1]`
- resize to size `(64, 64)`

```
def parse_function(filename, label):
    image_string = tf.read_file(filename)

    #Don't use tf.image.decode_image, or the output shape will be undefined
    image = tf.image.decode_jpeg(image_string, channels=3)

    #This will convert to float values in [0, 1]
    image = tf.image.convert_image_dtype(image, tf.float32)

    image = tf.image.resize_images(image, [64, 64])
    return image, label
```

And finally the `train_preprocess` can be optionally used during training to perform data augmentation:

- Horizontally flip the image with probability 1/2
- Apply random brightness and saturation

```
def train_preprocess(image, label):
    image = tf.image.random_flip_left_right(image)

    image = tf.image.random_brightness(image, max_delta=32.0 / 255.0)
    image = tf.image.random_saturation(image, lower=0.5, upper=1.5)

    #Make sure the image is still in [0, 1]
    image = tf.clip_by_value(image, 0.0, 1.0)

    return image, label
```

Building a text data pipeline

Have a look at the Tensorflow seq2seq tutorial using the tf.data pipeline

- [documentation](#)
- [github](#)

Files format

We've covered a simple example in the **Overview of tf.data** section. Now, let's cover a more advanced example. Let's assume that our task is Named Entity Recognition. In other words, our input is a sentence, and our output is a label for each word, like in

```
John   lives in New   York
B-PER  0      0  B-LOC I-LOC
```

Our dataset will thus need to load both the sentences and the labels. We will store those in 2 different files, a `sentence.txt` file containing the sentences (one per line) and a `labels.txt` containing the labels. For example

```
#sentences.txt
John lives in New York
Where is John ?
```

```
#labels.txt
B-PER 0 0 B-LOC I-LOC
0 0 B-PER 0
```

Constructing `tf.data` objects that iterate over these files is easy

```
#Load txt file, one example per line
sentences = tf.data.TextLineDataset("sentences.txt")
labels = tf.data.TextLineDataset("labels.txt")
```

Zip datasets together

At this stage, we might want to iterate over these 2 files at the same time. This operation is usually known as a "zip". Luckily, the `tf.data` comes with such a function


```

#Zip the sentence and the labels together
dataset = tf.data.Dataset.zip((sentences, labels))

#Create a one shot iterator over the zipped dataset
iterator = dataset.make_one_shot_iterator()
next_element = iterator.get_next()

#Actually run in a session
with tf.Session() as sess:
    for i in range(2):
        print(sess.run(dataset))

> ('John lives in New York', 'B-PER 0 0 B-LOC I-LOC')
('Where is John ?', '0 0 B-PER 0')

```

Creating the vocabulary

Great, now we can get the sentence and the labels as we iterate. Let's see how we can transform this string into a sequence of words and then in a sequence of ids.

Most NLP systems rely on ids as input for the words, meaning that you'll eventually have to convert your sentence into a sequence of ids.

Here we assume that we ran some script, like `build_vocab.py` that created some vocabulary files in our `/data` directory. We'll need one file for the words and one file for the labels. They will contain one token per line. For instance

```

#words.txt
John
lives
in
...

```

and

```

#tags.txt
B-PER

```

B-LOC

...

Tensorflow has a cool built-in tool to take care of the mapping. We simply define 2 lookup tables

```
words = tf.contrib.lookup.index_table_from_file("data/words.txt", num_oov_buckets=1)
tags = tf.contrib.lookup.index_table_from_file("data/tags.txt")
```

The parameter `num_oov_buckets` specifies the number of buckets created for unknown words. The id will be determined by Tensorflow and we don't have to worry about it. As in most of the cases, we just want to have one id reserved for the out-of-vocabulary words, we just use

```
num_oov_buckets=1 .
```

Now that we initialized this lookup table, we are going to transform the way we read the files, by adding these extra lines

```
#Convert line into list of tokens, splitting by white space
sentences = sentences.map(lambda string: tf.string_split([string]).values)

#Lookup tokens to return their ids
sentences = sentences.map(lambda tokens: (words.lookup(tokens), tf.size(tokens)))
```

Be careful that `tf.string_split` returns a `tf.SparseTensor`, that's why we need to extract the values.

Creating padded batches

Great! Now we can iterate and get a list of ids of words and labels for each sentence. We just need to take care of one final thing: **batches**! But here comes a problem: sentences have different length. Thus, we need to perform an extra **padding** operation that will add special token to shorter sentences so that our final batch Tensor is a tensor of shape `[batch_size, max_len_of_sentence_in_the_batch]`.

We first need to specify the padding shapes and values

```
#Create batches and pad the sentences of different length
padded_shapes = (tf.TensorShape([None]), # sentence of unknown size)
```

```
tf.TensorShape([None])) # labels of unknown size

padding_values = (params.id_pad_word, # sentence padded on the right with
                  params.id_pad_tag)  # labels padded on the right with id_
```

Note that the padding_values must be in the vocabulary (otherwise we might have a problem later on). That's why we get the id of the special "" token in *tra* ∈ .py with `id_pad_word = words.lookup(tf.constant(''))`.

Then, we can just use the `tf.data padded_batch` method, that takes care of the padding !

```
#Shuffle the dataset and then create the padded batches
dataset = (dataset
           .shuffle(buffer_size=buffer_size)
           .padded_batch(32, padded_shapes=padded_shapes, padding_values=paddin
           )
```

Computing the sentence's size

Is that all that we need in general ? Not quite. As we mentionned padding, we have to make sure that our model does not take the extra padded-tokens into account when computing its prediction. A common way of solving this issue is to add extra information to our data iterator and give the length of the input sentence as input. Later on, we will be able to give this argument to the `dynamic_rnn` function or create binary masks with `tf.sequence_mask`.

Look at the `model/input_fn.py` file for more details. But basically, it boils down to adding one line, using `tf.size`

```
sentences = sentences.map(lambda tokens: (vocab.lookup(tokens), tf.size(token
```

Advanced use - extracting characters

Now, let's try to perform a more complicated operation. We want to extract characters from each word, maybe because our NLP system relies on characters. Our input is a file that looks like

```
1 22
3333 4 55
```

We first create a dataset that yields the words for each sentence, as usual

```
dataset = tf.data.TextLineDataset("file.txt")
dataset = dataset.map(lambda token: tf.string_split([token]).values)
```

Now, we are going to reuse the `tf.string_split` function . However, it outputs a sparse tensor, a convenient data representation in general but which doesn't seem to be supported (yet) by `tf.data` . Thus, we need to convert this `SparseTensor` to a regular `Tensor` .

```
def extract_char(token, default_value="<pad_char>"):
    #Split characters
    out = tf.string_split(token, delimiter='')
    #Convert to Dense tensor, filling with default value
    out = tf.sparse_tensor_to_dense(out, default_value=default_value)
    return out

#Dataset yields word and characters
dataset = dataset.map(lambda token: (token, extract_char(token)))
```

Notice how we specified a `default_value` to the `tf.sparse_tensor_to_dense` function: words have different lengths, thus the `SparseTensor` that we need to convert has some unspecified entries !

Creating the padded batches is still as easy as above

```
#Creating the padded batch
padded_shapes = (tf.TensorShape([None]),          # padding the words
                 tf.TensorShape([None, None])) # padding the characters for e
padding_values = ('<pad_word>', # sentences padded on the right with <pad>
                 '<pad_char>') # arrays of characters padded on the right wi

dataset = dataset.padded_batch(2, padded_shapes=padded_shapes, padding_value
```

and you can test that the output matches your expectations

```
iterator = dataset.make_one_shot_iterator()
```

```

next_element = iterator.get_next()

with tf.Session() as sess:
    for i in range(1):
        sentences, characters = sess.run(next_element)
        print(sentences[0])
        print(characters[0][1])

> ['1', '22', '<pad_word>'] # sentence 1 (words)
   ['2', '2', '<pad_char>', '<pad_char>'] # sentence 1 word 2 (chars)

```

Can you explain why we have 2 `<pad_char>` and 1 `<pad_word>` in the first batch ?

Best Practices

One general tip mentioned in [the performance guide](#) is to put all the data processing pipeline on the CPU to make sure that the GPU is only used for training the deep neural network model:

```

with tf.device('/cpu:0'):
    dataset = ...

```

Shuffle and repeat

When training on a dataset, we often need to repeat it for multiple epochs and we need to shuffle it.

One big caveat when shuffling is to make sure that the `buffer_size` argument is big enough. The bigger it is, the longer it is going to take to load the data at the beginning. However a low buffer size can be disastrous for training. Here is a good [answer](#) on stackoverflow detailing an example of why.

The best way to avoid this kind of error might be to split the dataset into train / dev / test in advance and already shuffle the data there (see our other [post](#)).

In general, it is good to have the shuffling and repeat at the beginning of the pipeline. For instance if the input to the dataset is a list of filenames, if we directly shuffle after that the buffer of `tf.data.Dataset.shuffle()` will only contain filenames, which is very light on memory.

When choosing the ordering between shuffle and repeat, you may consider two options:

- **shuffle then repeat:** we shuffle the dataset in a certain way, and repeat this shuffling for

multiple epochs (ex: `[1, 3, 2, 1, 3, 2]` for 2 epochs with 3 elements in the dataset)

- **repeat then shuffle**: we repeat the dataset for multiple epochs and then shuffle (ex: `[1, 2, 1, 3, 3, 2]` for 2 epochs with 3 elements in the dataset)

The second method provides a better shuffling, but you might wait multiple epochs without seeing an example. The first method makes sure that you always see every element in the dataset at each epoch. You can also use `tf.contrib.data.shuffle_and_repeat()` to perform shuffle and repeat.

Parallelization: using multiple threads

Parallelization of the data processing pipeline using multiple threads is almost transparent when using the `tf.data` module. We only need to add a `num_parallel_calls` argument to every `dataset.map()` call.

```
num_threads = 4
dataset = dataset.map(parse_function, num_parallel_calls=num_threads)
```

Prefetch data

When the GPU is working on forward / backward propagation on the current batch, we want the CPU to process the next batch of data so that it is immediately ready. As the most expensive part of the computer, we want the GPU to be fully used all the time during training. We call this consumer / producer overlap, where the consumer is the GPU and the producer is the CPU.

With `tf.data`, you can do this with a simple call to `dataset.prefetch(1)` at the end of the pipeline (after batching). This will always prefetch one batch of data and make sure that there is always one ready.

```
dataset = dataset.batch(64)
dataset = dataset.prefetch(1)
```

In some cases, it can be useful to prefetch more than one batch. For instance if the duration of the preprocessing varies a lot, prefetching 10 batches would average out the processing time over 10 batches, instead of sometimes waiting for longer batches.

To give a concrete example, suppose that 10% of the batches take 10s to compute, and 90% take 1s. If the GPU takes 2s to train on one batch, by prefetching multiple batches you make sure that we never wait for these rare longer batches.

Order of the operations

To summarize, one good order for the different transformations is:

1. create the dataset
2. shuffle (with a big enough buffer size) 3, repeat
3. map with the actual work (preprocessing, augmentation...) using multiple parallel calls
4. batch
5. prefetch

[← Previous page](#)

[Next page →](#)

[← BACK TO BLOG](#)