

Images and TFRecords

Monday April 3, 2017

There are a number of ways to work with images in TensorFlow and, if you wish, with TFRecords. These methods aren't so mysterious if you **understand TFRecords** and a little bit about how digital images work.

Representations for Images

For example, this image is 600 pixels tall and 400 pixels wide. Every pixel has some intensity in red, green, and blue: three values, or channels, for every pixel. This image has shape [600, 400, 3]. (The order of the dimensions doesn't matter as long as everybody agrees on the convention.) The display on your screen is like a dense matrix; it is a **raster** graphic.



Neural networks that work with images typically work with this kind of dense matrix representation. For this image, the matrix will have $600 * 400 * 3 = 720,000$ values. If each value is an unsigned 8-bit integer, that's 720,000 bytes, or about three quarters of a megabyte.

Images sometimes also have an alpha transparency channel, which is a fourth channel in a color image, but not necessary if there's nothing else "underneath" the image. And not all images are color; greyscale (black and white) images can have just one channel.

Using unsigned 8-bit integers (256 possible values) for each value in the image array is enough for displaying images to humans. But when working with image data, it isn't uncommon to switch to 32-bit floats, for example. This quadruples the size of the data in memory. As always, remain aware of your data types.

Dense Array

One way to store complete raster image data is by serializing a NumPy array to disk with

```
numpy.save.
```

```
720,080 bytes  freedom.npy
```

The file `freedom.npy` has 80 more bytes than the ones required to store the array values. Those extra bytes specify things like the dimensions of the array. If we save raw array values in TFRecords, we'll also have to keep track of this additional information.

Because storing one or more value for every pixel takes so many bytes, file formats for images typically do something cleverer.

JPEG

JPEG is lossy. When you save an array as JPG and then read from the JPG, it will generally look about the same, but you won't necessarily get back exactly the same values for your array. JPEG is like MP3 for images. JPEG is good for photographs.

```
25,136 bytes  freedom.jpg
```

`freedom.jpg` is less than 4% the size of the NumPy array, and it still looks pretty good. It does have some ringing artifacts around the letter edges. JPEG file size depends on compression parameters when saving, and on the encoder used. For example, Google released their Guetzli JPEG encoder, which can produce smaller files but takes more computation to do so.

```
46,567 bytes  freedom2.jpg
```

`freedom2.jpg` is another JPEG file, saved with higher quality. Ringing artifacts are pretty much gone. The file is still under 7% the size of the NumPy array.

PNG

PNG is lossless. If you save as PNG and then read from the PNG, you can get back exactly what you had originally. PNG is like ZIP for images, but image viewers do the "un-zipping" automatically so they can put a raster image on the screen. PNG is like GIF without animation.

```
33,192 bytes  freedom.png
```

`freedom.png` is under 5% the size of the NumPy array, and it preserves the image perfectly. It's less often used, but compression parameters can also affect PNG size, and the encoder can make a difference too. PNG uses deflate (zlib) compression, so Google's Zopfli algorithm can be used, for example, while Brotli cannot.

SVG

SVG doesn't represent pixels directly at all, but represents in a text format the geometry of shapes in the image. SVG images can look good at any zoom level; they don't suffer from pixelization. They can also be edited with different tools than raster images. And for simple images, an SVG can be quite small. SVG is like HTML; you can look at it as text, but to see it as intended requires a program like a web browser.

```
42,721 bytes  freedom.svg
```

`freedom.svg` turns out not be a super efficient encoding of the image; it represents all the letter outlines instead of using plain text in some font, for example. But it represents the image fundamentally differently from just recreating pixels, and it's still under 6% the size of the NumPy array file.

TFRecords?

TFRecords don't know anything about image formats. You just put bytes in them. So you have your choice of whether that means you store dense arrays of values or a well-known image format. TensorFlow provides [image format support](#) for JPEG, PNG, and GIF in the computation graph.

Images without TensorFlow

As always, [you have a choice about whether you need to do everything with TensorFlow](#). If you're loading data batches with a `feed_dict`, in particular, feel free to use whatever Python functionality you're comfortable with. Even if you're not, you'll probably want to do some work with your data outside of TensorFlow before you move all your computation into the graph.

The Matplotlib [image tutorial](#) recommends using `matplotlib.image.imread` to read image formats from disk. This function will automatically change image array values to floats between zero and one, and it doesn't give you options about how to read the image.

The `scipy.misc.imread` function uses the Python Imaging Library (PIL) to support many image formats, including PNG and JPEG. This function also has some useful options. The original `freedom.png` has an alpha channel which we don't need. Passing `mode='RGB'` tells the reader to give us just three color channels.

```
>>> import scipy.misc
>>> image = scipy.misc.imread('freedom.png', mode='RGB')
>>> type(image)
## numpy.ndarray
>>> image.shape
## (600, 400, 3)
>>> image.dtype
## dtype('uint8')
```

Matplotlib's `imshow` is good for checking out what image arrays look like. (Also specify `%matplotlib inline` if you're in a notebook.)

```
>>> import matplotlib.pyplot as plt
>>> plt.imshow(image)
```

NumPy gives us a way to save and load its arrays.

```
>>> import numpy as np
>>> np.save('freedom.npy', image)
>>> same_image = np.load('freedom.npy')
```

As long as we have `scipy.misc` imported, we can use `scipy.misc.imsave` to write various image formats as well. For saving, `matplotlib.image.imsave` actually has more options. Neither of these offer control over compression level, for example.

```
>>> scipy.misc.imsave('freedom.jpg', image)
```

PNG and JPEG with TensorFlow

TensorFlow has ops for decoding and encoding PNGs and JPEGs, so we can put these operations into the computation graph.

Above, `imread` both read a file from disk and decoded it, and `imsave` both encoded an image and wrote it to disk. The TensorFlow functionality decouples the decoding and encoding from file reading and writing. This example will avoid using TensorFlow's file reading and writing, to focus just on the encoding and decoding.

The `channels=3` passed to `tf.image.decode_image()` is the equivalent of `mode='RGB'` above. We don't want to get the alpha channel from the PNG file.

```
>>> import tensorflow as tf
>>> with open('freedom.png', 'rb') as f:
...     png_bytes = f.read()
>>> bytes = tf.placeholder(tf.string)
>>> decode_png = tf.image.decode_image(bytes, channels=3)
>>> session = tf.Session()
>>> image = session.run(decode_png, feed_dict={bytes: png_bytes})
>>> type(image)
## numpy.ndarray
>>> image.shape
## (600, 400, 3)
>>> image.dtype
## dtype('uint8')
```

The `tf.image.decode_image` here intelligently uses `tf.image.decode_png`, `tf.image.decode_jpeg`, or `tf.image.decode_gif`, similar to how above `imread` can handle a variety of formats. You can also use the appropriate function directly.

Above, `imsave` supported writing various formats, choosing the one appropriate for the filename specified. In TensorFlow, you have to use `tf.image.encode_jpeg` or `tf.image.encode_png` directly, and both provide extra arguments controlling compression and more.

```
>>> tensor = tf.placeholder(tf.uint8)
>>> encode_jpeg = tf.image.encode_jpeg(tensor)
>>> jpeg_bytes = session.run(encode_jpeg, feed_dict={tensor: image})
>>> with open('freedom.jpg', 'wb') as f:
...     f.write(jpeg_bytes)
```

With the default settings, TensorFlow encodes a larger JPEG than `imsave`, coming in at 46,567 bytes. It looks pretty good.

PNG and JPEG in TFRecords

We can put plain bytes into Example TFRecords without too much trouble. So PNG or JPEG images are easily handled.

```
my_example = tf.train.Example(features=tf.train.Features(feature={
    'png_bytes': tf.train.Feature(bytes_list=tf.train.BytesList(value=[png_bytes]))
}))
```

```

my_example_str = my_example.SerializeToString()
with tf.python_io.TFRecordWriter('my_example.tfrecords') as writer:
    writer.write(my_example_str)

reader = tf.python_io.tf_record_iterator('my_example.tfrecords')
those_examples = [tf.train.Example().FromString(example_str)
                   for example_str in reader]
same_example = those_examples[0]

same_png_bytes = same_example.features.feature['png_bytes'].bytes_list.value[0]

```

When the `same_png_bytes` is decoded by `tf.image.decode_image`, as above, or `tf.image.decode_png` directly, you'll get back a tensor with the correct dimensions, because PNG (and JPEG) include that information in their encodings.

Image Arrays in TFRecords

If you want to save dense matrix representations in TFRecords, there's a little bit of bookkeeping to do, but it isn't too bad.

```

image_bytes = image.tostring()
image_shape = image.shape

my_example = tf.train.Example(features=tf.train.Features(feature={
    'image_bytes': tf.train.Feature(bytes_list=tf.train.BytesList(value=[image_bytes])),
    'image_shape': tf.train.Feature(int64_list=tf.train.Int64List(value=image_shape))
}))

my_example_str = my_example.SerializeToString()
with tf.python_io.TFRecordWriter('my_example.tfrecords') as writer:
    writer.write(my_example_str)

reader = tf.python_io.tf_record_iterator('my_example.tfrecords')
those_examples = [tf.train.Example().FromString(example_str)
                  for example_str in reader]
same_example = those_examples[0]

same_image_bytes = same_example.features.feature['image_bytes'].bytes_list.value[0]
same_image_shape = list(
    same_example.features.feature['image_shape'].int64_list.value)

```

With the information recovered from TFRecord form, it's easy to use NumPy to put the image back together.

```

same_image = np.fromstring(same_image_bytes, dtype=np.uint8)
same_image.shape = same_image_shape

```

You can do the same using TensorFlow.

```

shape = tf.placeholder(tf.int32)
new_image = tf.reshape(tf.decode_raw(bytes, tf.uint8), shape)
same_image = session.run(encode_jpeg, feed_dict={bytes: same_image_bytes,
                                                shape: same_image_shape})

```

In this example, however, the parsing of the `Example` was already done outside the TensorFlow graph, so there isn't a strong reason to stay inside the graph here.

Comparison to Caffe and LMDB

Caffe is an older deep learning framework that can work with data stored in LMDB on-disk databases, similar to how TensorFlow can work with data stored in TFRecords files.

Like TensorFlow, Caffe defines a protocol buffer message for training examples. In Caffe, it's called `Datum`. These are saved just like in TensorFlow, by serializing them and putting them in a file on disk, but instead of a TFRecords file, which just puts records in a row and reads them out in that order, here Caffe will work with LMDB, which has the semantics of a key-value store.

TensorFlow's `Example` format is super flexible, but the trade-off is that it doesn't automatically do things for you. Caffe's `Datum`, on the other hand, expects you to put in a dense image array, and an integer class label. So here there isn't any fiddling with array size, for example, but the trade-off is that it won't work easily for arbitrary data structures that we might eventually want to store.

In the TFRecords examples above, we stored only image data, and said nothing about a class label or anything else. This is because TFRecords lets you decide what you want to save, rather than defining a format in advance. You could save an integer label, or a float regression label, or a string of text, or an image mask, and on and on. Here, we're just using the built-in Caffe integer label.

```
import caffe
import lmdb

# `image` was read above
label = 9 # for a classification problem

datum = caffe.io.array_to_datum(arr=image, label=label)
datum_str = datum.SerializeToString()

env = lmdb.open('lmdb_data')
txn = env.begin(write=True)
txn.put(key='my datum', value=datum_str)

cur = txn.cursor()
same_datum_str = cur.get('my datum')

same_datum = caffe.proto.caffe_pb2.Datum().FromString(same_datum_str)
same_image = caffe.io.datum_to_array(same_datum)
same_label = datum.label
```

To actually work with the Caffe training process, there are some other conditions for the data to satisfy. Caffe expects [channels, height, width] instead of [height, width, channels], for example.

What should you use?

Prefer doing fewer separate manipulation steps to whatever your original data is, if you can help it: try not to have lots of different versions of your data in different places on disk. This will make your life easier.


If you are choosing a format, JPEG is good for photos.

Think about whether you need to put everything into the TensorFlow computation graph. Think about whether you need to use TFRecords. Try to spend your time on things that help solve your problems.

For two more complete *in situ* examples of converting images to TFRecords, check out [code for MNIST images](#) and [code for ImageNet images](#). The ImageNet code can be run on

the command-line.

I'm working on Building TensorFlow systems from components, a workshop at OSCON 2017.

- [Plan](#)  [Space](#)
 - [Edit this page](#)
 - Find [Aaron](#) on
 - [Twitter](#)
 - [LinkedIn](#)
 - [Google+](#)
 - [GitHub](#)
 - [email](#)
 - [Comment below](#)
-