PREPARING A LARGE-SCALE IMAGE DATASET WITH TENSORFLOW'S TFRECORD FILES

29 JAN 2017 . CATEGORY: TECH (/CATEGORIES/TECH.HTML). COMMENTS (/TECH/2017/01/29/TFRECORDS.HTML#DISQUS_THREAD)
#TENSORFLOW (/TAGS/TENSORFLOW.HTML) #TFRECORDS (/TAGS/TFRECORDS.HTML)

There are several methods of reading image data in TensorFlow as mentioned in its documentation:

From disk: Using the typical `feed_dict` argument when running a session for the `train_op`. However, this is not always possible if your dataset is too large to be held in your GPU memory for it to be trained.

From CSV Files: Not as relevant for dealing with images.

From TFRecord files: This is done by first converting images that are already properly arranged in sub-directories according to their classes into a readable format for TensorFlow, so that you don't have to read in raw images in real-time as you train. This is much faster than reading images from disk.

While the creation of TFRecord files may not be intuitive, and indeed, less straightforward than simply reading data in HDF5 format (as used in Keras), using this supported native format for TensorFlow gives you greater access to the data pipeline tools you can use to train your images in batches - think of queue runners, coordinators and supervisors that can help you manage your data flow. In this guide, I will focus on a less painful way of writing and reading TFRecord files using TensorFlow-slim instead of pure TensorFlow. To put the guide into concrete practice, we will use the standard Flowers dataset from TensorFlow.

Note: A side benefit of using TensorFlow-slim is that is you could use the official pre-trained models - including the inception-resnet-v2 model - from Google for performing transfer learning. I find this as the main advantage TF-slim has over Keras.

DOWNLOADING THE DATASET

First download the dataset from the link below.

> **Flowers Dataset:** http://download.tensorflow.org/example_images/flower_photos.tgz (http://download.tensorflow.org/example_images/flower_photos.tgz)

ARRANGING YOUR DATA ACCORDING TO CLASSES

Write a simple script that arranges your examples into subdirectories, where each subdirectory represents the class of the examples. For the Flowers dataset, this has already been done for you. you should be able to see that your data is arranged into the following structure:

```
flowers\
    flower_photos\
        tulips\
            ....jpg
            ....jpg
            ....jpg
        sunflowers\
            ....jpg
        roses\
            ....jpg
        dandelion\
            ....jpg
        daisy\
            ....jpg
```

Note: Your dataset directory will be `path/to/flowers` and not `path/to/flowers/flower_photos`. Make sure you do not have any other folders beyond `flower_photos` in the `flowers` root directory!

WRITING A TFRECORD FILE

Due to the low-level nature of TensorFlow, it is hard to write all the code from scratch in order to just prepare a dataset. In fact, we do not have to reinvent the wheel as TF-Slim has already most of the code available for writing TFRecord files. I have compiled all the necessary functions into a `dataset_utils.py` file that we will import our dataset functions from to create the TFRecord file. The functions in `dataset_utils.py` are quite general and could be used for other datasets with JPEG images.

Note: If your images are in PNG format, then you would have to go to `dataset_utils.py` to change all the JPEG decoding to PNG decoding. The change is as straightforward as changing the 'jpeg' **and** 'jpg' characters to 'png'.

First import the required modules you'll need in a new script.

```
import random
import tensorflow as tf
from dataset_utils import _dataset_exists, _get_filenames_and_classes, write_label_file,
_convert_dataset
```

And some required arguments:

```
flags = tf.app.flags

#State your dataset directory
flags.DEFINE_string('dataset_dir', None, 'String: Your dataset directory')

# Proportion of dataset to be used for evaluation
flags.DEFINE_float('validation_size', 0.3, 'Float: The proportion of examples in the dat
aset to be used for validation')

# The number of shards to split the dataset into.
flags.DEFINE_integer('num_shards', 2, 'Int: Number of shards to split the TFRecord files
into')

# Seed for repeatability.
flags.DEFINE_integer('random_seed', 0, 'Int: Random seed to use for repeatability.')

#Output filename for the naming the TFRecord file
flags.DEFINE_string('tfrecord_filename', None, 'String: The output filename to name your
TFRecord file')

FLAGS = flags.FLAGS
```

Now we can proceed with writing the TFRecord file. We will need to use `_get_filename_and_classes` to return us a list of `photo_filenames` that contains strings of individual filenames and a list of sorted `class_names` that contains actual class names like 'daisy' and 'roses'. But we can't actually use strings as the class labels, so we proceed to create a dictionary to refers each class name to a label in `class_names_to_ids`. Here, we will realize how important it is to actually have your class_names **sorted**, otherwise you will always end up with different labels for each class.

```
photo_filenames, class_names = _get_filenames_and_classes(FLAGS.dataset_dir)
class_names_to_ids = dict(zip(class_names, range(len(class_names))))
```

Next, we find the proportion of the dataset to use for validation and slice the `photo_filenames` list accordingly to obtain the filenames for the training

and validation datasets.

```
#Find the number of validation examples we need
num_validation = int(FLAGS.validation_size * len(photo_filenames))

# Divide the training datasets into train and test:
random.seed(FLAGS.random_seed)
random.shuffle(photo_filenames)
training_filenames = photo_filenames[num_validation:]
validation_filenames = photo_filenames[:num_validation]
```

Now we let TensorFlow locate the images and their corresponding labels to write into TFRecord files using `_convert_dataset`.

```
# First, convert the training and validation sets.
_convert_dataset('train', training_filenames, class_names_to_ids,
                 dataset_dir = FLAGS.dataset_dir,
                 tfrecord_filename = FLAGS.tfrecord_filename,
                 _NUM_SHARDS = FLAGS.num_shards)
_convert_dataset('validation', validation_filenames, class_names_to_ids,
                 dataset_dir = FLAGS.dataset_dir,
                 tfrecord_filename = FLAGS.tfrecord_filename,
                 _NUM_SHARDS = FLAGS.num_shards)
```

In essence, what happens in the function `_convert_dataset` is that it searches the images one by one in the `training_filenames` or `validation_filenames`, read the image in byte form, find the height, width and class label of this image, before converting all the image data and its information (height, width, label) into a TFexample that could be encoded into the TFRecord file. This TFexample is then written down officially into a TFRecord. Most of the magic happens in the following code of `_convert_dataset`:

```
# Read the filename:
image_data = tf.gfile.FastGFile(filenames[i], 'r').read()
height, width = image_reader.read_image_dims(sess, image_data)

class_name = os.path.basename(os.path.dirname(filenames[i]))
class_id = class_names_to_ids[class_name]

example = image_to_tfexample(
    image_data, 'jpg', height, width, class_id)
tfrecord_writer.write(example.SerializeToString())
```

Finally, we write a labels file that will be useful as a reference later on:

```
# Finally, write the labels file:
labels_to_class_names = dict(zip(range(len(class_names)), class_names))
write_label_file(labels_to_class_names, FLAGS.dataset_dir)
```

And we are done! Here is the entire code file we need to write, including some additional checks, to successfully convert an image dataset into TFRecord files:

```
import random
import tensorflow as tf
from dataset_utils import _dataset_exists, _get_filenames_and_classes, write_label_file,
_convert_dataset

#==============DEFINE YOUR ARGUMENTS==============
flags = tf.app.flags

#State your dataset directory
flags.DEFINE_string('dataset_dir', None, 'String: Your dataset directory')

# The number of images in the validation set. You would have to know the total number of
examples in advance. This is essentially your evaluation dataset.
flags.DEFINE_float('validation_size', 0.3, 'Float: The proportion of examples in the dat
aset to be used for validation')

# The number of shards per dataset split.
flags.DEFINE_integer('num_shards', 2, 'Int: Number of shards to split the TFRecord files
')

# Seed for repeatability.
flags.DEFINE_integer('random_seed', 0, 'Int: Random seed to use for repeatability.')

#Output filename for the naming the TFRecord file
flags.DEFINE_string('tfrecord_filename', None, 'String: The output filename to name your
TFRecord file')

FLAGS = flags.FLAGS

def main():

    #=============CHECKS==============
    #Check if there is a tfrecord_filename entered
    if not FLAGS.tfrecord_filename:
        raise ValueError('tfrecord_filename is empty. Please state a tfrecord_filename a
rgument.')

    #Check if there is a dataset directory entered
    if not FLAGS.dataset_dir:
        raise ValueError('dataset_dir is empty. Please state a dataset_dir argument.')

    #If the TFRecord files already exist in the directory, then exit without creating th
e files again
    if _dataset_exists(dataset_dir = FLAGS.dataset_dir, _NUM_SHARDS = FLAGS.num_shards,
output_filename = FLAGS.tfrecord_filename):
        print 'Dataset files already exist. Exiting without re-creating them.'
        return None
    #==========END OF CHECKS============

    #Get a list of photo_filenames like ['123.jpg', '456.jpg'...] and a list of sorted c
lass names from parsing the subdirectories.
    photo_filenames, class_names = _get_filenames_and_classes(FLAGS.dataset_dir)

    #Refer each of the class name to a specific integer number for predictions later
    class_names_to_ids = dict(zip(class_names, range(len(class_names))))

    #Find the number of validation examples we need
    num_validation = int(FLAGS.validation_size * len(photo_filenames))

    # Divide the training datasets into train and test:
    random.seed(FLAGS.random_seed)
    random.shuffle(photo_filenames)
    training_filenames = photo_filenames[num_validation:]
    validation_filenames = photo_filenames[:num_validation]

    # First, convert the training and validation sets.
    _convert_dataset('train', training_filenames, class_names_to_ids,
                     dataset_dir = FLAGS.dataset_dir,
                     tfrecord_filename = FLAGS.tfrecord_filename,
                     _NUM_SHARDS = FLAGS.num_shards)
    _convert_dataset('validation', validation_filenames, class_names_to_ids,
                     dataset_dir = FLAGS.dataset_dir,
                     tfrecord_filename = FLAGS.tfrecord_filename,
                     _NUM_SHARDS = FLAGS.num_shards)

    # Finally, write the labels file:
    labels_to_class_names = dict(zip(range(len(class_names)), class_names))
    write_label_file(labels_to_class_names, FLAGS.dataset_dir)

    print '\nFinished converting the %s dataset!' % (FLAGS.tfrecord_filename)

if __name__ == "__main__":
    main()
```

OUTPUT

Here is what you should see after you successfully run the code:

```
$ python create_tfrecord.py --tfrecord_filename=flowers --dataset_dir=/home/kwotsin/Data
sets/flowers/

I tensorflow/stream_executor/dso_loader.cc:128] successfully opened CUDA library libcubl
as.so locally
I tensorflow/stream_executor/dso_loader.cc:128] successfully opened CUDA library libcudn
n.so locally
I tensorflow/stream_executor/dso_loader.cc:128] successfully opened CUDA library libcuff
t.so locally
I tensorflow/stream_executor/dso_loader.cc:128] successfully opened CUDA library libcuda
.so.1 locally
I tensorflow/stream_executor/dso_loader.cc:128] successfully opened CUDA library libcura
nd.so locally
tfrecord_filename: /home/kwotsin/Datasets/flowers/flowers_train_00000-of-00002.tfrecord
I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read fr
om SysFS had negative value (-1), but there must be at least one NUMA node, so returning
NUMA node zero
I tensorflow/core/common_runtime/gpu/gpu_device.cc:885] Found device 0 with properties:
name: GeForce GTX 860M
major: 5 minor: 0 memoryClockRate (GHz) 1.0195
pciBusID 0000:01:00.0
Total memory: 3.95GiB
Free memory: 3.13GiB
I tensorflow/core/common_runtime/gpu/gpu_device.cc:906] DMA: 0
I tensorflow/core/common_runtime/gpu/gpu_device.cc:916] 0:   Y
I tensorflow/core/common_runtime/gpu/gpu_device.cc:975] Creating TensorFlow device (/gpu
:0) -> (device: 0, name: GeForce GTX 860M, pci bus id: 0000:01:00.0)
>> Converting image 2569/2569 shard 1
I tensorflow/core/common_runtime/gpu/gpu_device.cc:975] Creating TensorFlow device (/gpu
:0) -> (device: 0, name: GeForce GTX 860M, pci bus id: 0000:01:00.0)
>> Converting image 1101/1101 shard 1

Finished converting the flowers dataset!
```

I'm not a big fan of flowers but I happen to have another flowers dataset as well. Here is what the output will look like for the Oxford VGG 17 Flowers dataset:

```
$ python create_tfrecord.py --tfrecord_filename=ox17 --dataset_dir=/home/kwotsin/Dataset
s/17flowers/
I tensorflow/stream_executor/dso_loader.cc:128] successfully opened CUDA library libcubl
as.so locally
I tensorflow/stream_executor/dso_loader.cc:128] successfully opened CUDA library libcudn
n.so locally
I tensorflow/stream_executor/dso_loader.cc:128] successfully opened CUDA library libcuff
t.so locally
I tensorflow/stream_executor/dso_loader.cc:128] successfully opened CUDA library libcuda
.so.1 locally
I tensorflow/stream_executor/dso_loader.cc:128] successfully opened CUDA library libcura
nd.so locally
I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read fr
om SysFS had negative value (-1), but there must be at least one NUMA node, so returning
NUMA node zero
I tensorflow/core/common_runtime/gpu/gpu_device.cc:885] Found device 0 with properties:
name: GeForce GTX 860M
major: 5 minor: 0 memoryClockRate (GHz) 1.0195
pciBusID 0000:01:00.0
Total memory: 3.95GiB
Free memory: 3.09GiB
I tensorflow/core/common_runtime/gpu/gpu_device.cc:906] DMA: 0
I tensorflow/core/common_runtime/gpu/gpu_device.cc:916] 0:   Y
I tensorflow/core/common_runtime/gpu/gpu_device.cc:975] Creating TensorFlow device (/gpu
:0) -> (device: 0, name: GeForce GTX 860M, pci bus id: 0000:01:00.0)
>> Converting image 952/952 shard 1
I tensorflow/core/common_runtime/gpu/gpu_device.cc:975] Creating TensorFlow device (/gpu
:0) -> (device: 0, name: GeForce GTX 860M, pci bus id: 0000:01:00.0)
>> Converting image 408/408 shard 1

Finished converting the ox17 dataset!
```

Note: To quickly verify whether your dataset is prepared correctly, you should see that the total size of your TFRecord files should have a similar (or slightly larger) size than the size of your entire dataset.

CONCLUSION

To sum it up, we've packaged all the required (and ugly) functions that we need to create TFRecord files into `dataset_utils.py` before actually starting to write one through a main script. This two files are what I usually use to prepare TFRecord files in a faster way, since I would only have to change certain arguments to prepare a dataset quickly. Certainly, for greater customization, you should look into the source code I provided in GitHub (see below). Unfortunately, I have not seen a fast way to create TFRecord files in a few lines or so, hence for now I'll just stick to using these two files!

I hope this guide has been useful to you! 😃

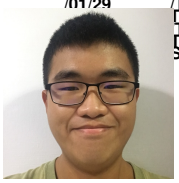
SOURCE CODE

Download the zip file from the official GitHub repository (https://github.com/kwotsin/create_tfrecords) or by cloning:

```
$ git clone https://github.com/kwotsin/create_tfrecords.git
```

CREDITS

Credits to authors of TF-slim who have contributed this wonderful library (https://github.com/tensorflow/models/tree/master/slim) for use by many people! A large part of this post was inspired by their source code, and I highly recommend you to carefully study related material offered by them - after all, they are few of the best engineers in the world!



machine learning and plays fingerstyle guitar.