How to Construct a New Dataset for Retraining

One can use the existing scripts supplied with this model to build a new dataset for training or fine-tuning. The main script to employ is build_image_data.py. Briefly, this script takes a structured directory of images and converts it to a sharded TFRecord that can be read by the Inception model.

In particular, you will need to create a directory of training images that reside within $TRAIN_DIR and $VALIDATION_DIRarranged as such:

```
$TRAIN_DIR/dog/image0.jpeg
$TRAIN_DIR/dog/image1.jpg
$TRAIN_DIR/dog/image2.png
...
$TRAIN_DIR/cat/weird-image.jpeg
$TRAIN_DIR/cat/my-image.jpeg
$TRAIN_DIR/cat/my-image.JPG
...
$VALIDATION_DIR/dog/imageA.jpeg
$VALIDATION_DIR/dog/imageB.jpg
$VALIDATION_DIR/dog/imageC.png
...
$VALIDATION_DIR/cat/weird-image.PNG
$VALIDATION_DIR/cat/that-image.jpg
$VALIDATION_DIR/cat/cat.JPG
...
```

NOTE: This script will append an extra background class indexed at 0, so your class labels will range from 0 to num_labels. Using the example above, the corresponding class labels generated from `build_image_data.py` will be as follows:

```
0
1 dog
2 cat
```

Each sub-directory in `$TRAIN_DIR` and `$VALIDATION_DIR` corresponds to a unique label for the images that reside within that sub-directory. The images may be JPEG or PNG images. We do not support other images types currently.

Once the data is arranged in this directory structure, we can run `build_image_data.py` on the data to generate the sharded `TFRecord` dataset. Each entry of the `TFRecord` is a serialized `tf.Example` protocol buffer. A complete list of information contained in the `tf.Example` is described in the comments of `build_image_data.py`.

To run `build_image_data.py`, you can run the following command line:

```
# location to where to save the TFRecord data.
OUTPUT_DIRECTORY=$HOME/my-custom-data/
# build the preprocessing script.
bazel build inception/build_image_data
# convert the data.
bazel-bin/inception/build_image_data \
  --train_directory="${TRAIN_DIR}" \
  --validation_directory="${VALIDATION_DIR}" \
  --output_directory="${OUTPUT_DIRECTORY}" \
```

```
  --labels_file="${LABELS_FILE}" \
  --train_shards=128 \
  --validation_shards=24 \
  --num_threads=8
```

where the $OUTPUT_DIRECTORY is the location of the sharded TFRecords.

The $LABELS_FILE will be a text file that is read by the script that provides a list of all of the labels. For instance, in the case flowers data set, the $LABELS_FILE contained the following data:

```
daisy
dandelion
roses
sunflowers
tulips
```

Note that each row of each label corresponds with the entry in the final classifier in the model. That is, the daisy corresponds to the classifier for entry 1; dandelion is entry 2, etc. We skip label 0 as a background class.

After running this script produces files that look like the following:

```
  $TRAIN_DIR/train-00000-of-00128
  $TRAIN_DIR/train-00001-of-00128
  ...
  $TRAIN_DIR/train-00127-of-00128
```

and

```
  $VALIDATION_DIR/validation-00000-of-00024
```

```
$VALIDATION_DIR/validation-00001-of-00024
...
$VALIDATION_DIR/validation-00023-of-00024
```

where 128 and 24 are the number of shards specified for each dataset, respectively. Generally speaking, we aim for selecting the number of shards such that roughly 1024 images reside in each shard. Once this data set is built, you are ready to train or fine-tune an Inception model on this data set.

Note, if you are piggy backing on the flowers retraining scripts, be sure to update `num_classes()` and `num_examples_per_epoch()` in `flowers_data.py` to correspond with your data.

Practical Considerations for Training a Model

The model architecture and training procedure is heavily dependent on the hardware used to train the model. If you wish to train or fine-tune this model on your machine **you will need to adjust and empirically determine a good set of training hyper-parameters for your setup**. What follows are some general considerations for novices.

Finding Good Hyperparameters

Roughly 5-10 hyper-parameters govern the speed at which a network is trained. In addition to `--batch_size` and `--num_gpus`, there are several constants defined in inception_train.py which dictate the learning schedule.

```
RMSPROP_DECAY = 0.9                # Decay term for RMSProp.
MOMENTUM = 0.9                     # Momentum in RMSProp.
```

```
RMSPROP_EPSILON = 1.0                 # Epsilon term for RMSProp.
INITIAL_LEARNING_RATE = 0.1           # Initial learning rate.
NUM_EPOCHS_PER_DECAY = 30.0           # Epochs after which learning rate decays.
LEARNING_RATE_DECAY_FACTOR = 0.16  # Learning rate decay factor.
```

There are many papers that discuss the various tricks and trade-offs associated with training a model with stochastic gradient descent. For those new to the field, some great references are:

- Y Bengio, Practical recommendations for gradient-based training of deep architectures
- I Goodfellow, Y Bengio and A Courville, [Deep Learning] (http://www.deeplearningbook.org/)

What follows is a summary of some general advice for identifying appropriate model hyper-parameters in the context of this particular model training setup. Namely, this library provides synchronous updates to model parameters based on batch-splitting the model across multiple GPUs.

-

  Higher learning rates leads to faster training. Too high of learning rate leads to instability and will cause model parameters to diverge to infinity or NaN.

-
-

  Larger batch sizes lead to higher quality estimates of the gradient and permit training the model with higher learning rates.

-
-

    Often the GPU memory is a bottleneck that prevents employing larger batch sizes. Employing more GPUs allows one to use larger batch sizes because this model splits the batch across the GPUs.

-

NOTE If one wishes to train this model with asynchronous gradient updates, one will need to substantially alter this model and new considerations need to be factored into hyperparameter tuning. See Large Scale Distributed Deep Networks for a discussion in this domain.

Adjusting Memory Demands

Training this model has large memory demands in terms of the CPU and GPU. Let's discuss each item in turn.

GPU memory is relatively small compared to CPU memory. Two items dictate the amount of GPU memory employed -- model architecture and batch size. Assuming that you keep the model architecture fixed, the sole parameter governing the GPU demand is the batch size. A good rule of thumb is to try employ as large of batch size as will fit on the GPU.

If you run out of GPU memory, either lower the `--batch_size` or employ more GPUs on your desktop. The model performs batch-splitting across GPUs, thus N GPUs can handle N times the batch size of 1 GPU.

The model requires a large amount of CPU memory as well. We have tuned the model to employ about ~20GB of CPU memory. Thus, having access to about 40 GB of CPU memory would be ideal.

If that is not possible, you can tune down the memory demands of the model via lowering `--input_queue_memory_factor`. Images are preprocessed asynchronously with respect to the main training across `--num_preprocess_threads` threads. The preprocessed images are stored in shuffling queue in which each GPU performs a dequeue operation in order to receive a `batch_size` worth of images.

In order to guarantee good shuffling across the data, we maintain a large shuffling queue of 1024 x `input_queue_memory_factor` images. For the current model architecture, this corresponds to about 4GB of CPU memory. You may lower `input_queue_memory_factor` in order to decrease the memory footprint. Keep in mind though that lowering this value drastically may result in a model with slightly lower predictive accuracy when training from scratch. Please see comments in `image_processing.py` for more details.