How to train a model in Tensorflow Obj Detection API

Take a look at GitHub: TensorFlow Object Detection API Tutorial to set up the environment and to have more insights.

Look also to the git repo README file to have more examples and tutorials.

OpenImagesV4 Dataset

Open Images is a dataset of ~9M images annotated with image-level labels, object bounding boxes, object segmentation masks, visual relationships, and localized narratives. It contains a total of 16M bounding boxes for 600 object classes on 1.9M images, making it the largest existing dataset with object location annotations.

Download a subset of OpenImagesV4 Dataset

There is a GitHub Repo that allows to download all the images of this this dataset containing a specific class (and only that class). The annotations of the images will include just labels and boxes for that class (so (for example) I downloaded just images of license plates).

Example of a command for executing OIDv4 Toolkit:

```
python main.py downloader --classes Vehicle_registration_plate --type_csv all
```

--classes : you specify what classes you want to download (write the corresponding label). If the class name has a space in it, like "polar bear", you must write it with **underscores** "polar_bear". To download multiple classes, you can create a classes.txt (and give this to the --classes option) in which each line corresponds to a class name.

--type_csv : you can select "train", "test", "validation", "all". Selecting "all" you will download 3 folders with images divided into train, valid and test sets (so you are downloading all the images available for your class).

TFRecord generation

There is a GitHub Repo that gives an easy script for generating the TFRecords of the OIDv4 subset downloaded.

```
python generate_tf_record.py --classes 12c-classes.txt --class_descriptions 12c-class-descriptions-boxable.csv
--annotations_file=train-annotations-bbox.csv --images_dir=train_images_folder/ --output_file=train.tfrecord
```

IMPORTANT: Here the classes.txt file doesn't want underscores instead of white spaces!! (unlike the dataset downloader OIDv4 Toolkit!)
For example, you have to write again "polar bear" instead of "polar_bear".

Prepare Workspace

```
workspace_api
└── models
    ├── CTC101
    └── research
        ├── object_detection
        │   ├── ...
        │   ├── ...
        │   └── ...
        └── ...
```

```
pre-trained-model
├── checkpoint (other files of checkpoint for fine-tuning)
├── frozen_inference_graph.pb (exported frozen graph)
│   └── saved_model
│       ├── variables
│       │   ├── variables.data-00000-of-00001
│       │   └── variables.index
│       └── saved_model.pb (exported frozen graph)
├── model_config (dict with for training and evaluation)
└── pipeline_training_job.sh
```

- **pre-trained-model**: The folder will contain the pre-trained model of our choice, which shall be used as a starting checkpoint for our training job.
- **training**: This folder will contain the training pipeline configuration file *.config, as well as a *.pbtxt label map file and all files generated during the training of our model by "model_main.py" script.
- **trained-inference-graphs**: this folder will contain the frozen graph exported by "export_inference_graph.py" script.

Annotating your own images

You can use the Labeling package.

- Use xml_to_csv.py script to convert xml files generated by Labeling to a unique csv file with all annotations.
- Use from_csv_to_tfrecord.py (inside labeling_utilities folder) script to convert the csv file to a tfrecord

Label Map

TensorFlow requires a label map, which namely maps each of the used labels to an integer values. This label map is used both by the training and detection processes.

The Label Maps for standard datasets can be found in the tensorflow/models repository at
`models/research/object_detection/data`

The classes included in the label map should be exactly the ones that you are training on. If you set to train on just 1 class, then save only that class in the label_map.pbtxt file with `id: 1`.

Configuration

How to setup the training/config files/config file:

You can find all the default config files in `models/research/object_detection/samples/configs` folder. Make sure to set correctly all the paths (search for "PATH_TO_BE_CONFIGURED" to find the fields).

More details on the essential fields to set can be found here and at the "Useful stuff" paragraph.

Metrics

All the metrics available are declared in `models/research/object_detection/eval_util.py`.

By default, COCO metrics are used. Follows the definition:

The following 12 metrics are used for characterizing the performance of an object detector on COCO:

```
Average Precision (AP):
  AP                  % AP at IoU=.50:.05:.95 (primary challenge metric)
  AP^IoU=.50          % AP at IoU=.50 (PASCAL VOC metric)
  AP^IoU=.75          % AP at IoU=.75 (strict metric)
AP Across Scales:
  AP^small            % AP for small objects: area < 32²
  AP^medium           % AP for medium objects: 32² < area < 96²
  AP^large            % AP for large objects: area > 96²
Average Recall (AR):
  AR^max=1            % AR given 1 detection per image
  AR^max=10           % AR given 10 detections per image
  AR^max=100          % AR given 100 detections per image
AR Across Scales:
  AR^small            % AR for small objects: area < 32²
  AR^medium           % AR for medium objects: 32² < area < 96²
  AR^large            % AR for large objects: area > 96²
```

1. Unless otherwise specified, AP and AR are averaged over multiple intersection over Union (IoU) values. Specifically we use 10 IoU thresholds of .50:.05:.95. This is a slight change from the PASCAL metric, which only uses a single IoU of .50 (which corresponds to our metric $AP^{IoU=.50}$). Averaging over IoUs rewards detectors with better localization.
2. AP is averaged over all categories. Traditionally, this is called "mean average precision" (mAP). We make no distinction between AP and mAP (and similarly AR and mAR) and assume the difference is clear from context.
3. AP (averaged across all 10 IoU thresholds and all 80 categories) will determine the challenge winner. This should be considered the single most important metric when considering performance on COCO.
4. In COCO, there are more small objects than large objects. Specifically: approximately 41% of objects are small (area < 32²), 34% are medium (32² < area < 96²), and 24% are large (area > 96²). Area is measured as the number of pixels in the segmentation mask.
5. AR is the maximum recall given a fixed number of detections per image, averaged over categories and IoUs. AR is related to the metric of the same name used in proposal evaluation but is computed on a per-category basis.
6. All measures evaluate detection performance for 100 categories and 20,000 test images that are annotated at categories
7. The evaluation metrics for detection with bounding boxes and segmentation masks are identical in all respects except for the IoU computation (which is performed over boxes or masks, respectively).

Training

Use the `model_main.py` script to train your model. It will save checkpoints and tensorflow events that will keep trace of the training process.

Example training command:

```
python model_main.py --alsologtostderr --model_dir=path/to/model/directory --
pipeline_config_path=path/to/pipeline/config
```

--model_dir : where checkpoints an tensorboard logs will be saved

TensorBoard

To visualize the training history (mAP scores, loss, learning rate, predictions) use TensorBoard. Look here to install it.

First, lunch a session on the training folder with this command:

`tensorboard --logdir=training/`

Then open firefox and copy the http address that pops up. You will see:

- **DetectionBoxes_Precision**: the mAP score is the main reference score, with IoU averaged on [0.5 : 0.95]
- **DetectionBoxes_Recall** : Average Recall scores
- **Loss**: validation loss, divided in "classification" and "localization" losses
- **learning_rate**: how the learning rate decays with steps
- **learning_rate_1**: same as before
- **loss**: same as "loss", is the validation loss
- **loss_0**: is the training loss!
- **loss_1**: same as "loss_1", is the training loss

You can switch to the IMAGES folder inside Tensorboard to see how predictions evolve with training steps (it's fun!)

Export frozen graph

Before testing a model you need to export a frozen graph with the `export_inference_graph.py`.

Example command:

```
python export_inference_graph.py --input_type image_tensor --pipeline_config_path
training/ssd_mobilnet_v2_dir_v4.config --trained_checkpoint_prefix training/model.ckpt-XXXXX --output_directory
trained-inference-graphs/output_inference_graph_XXXXX.pb
```

where:

--pipeline_config_path: config file
--trained_checkpoint_prefix: inside saved checkpoint (replace XXXX with the corresponding step number)
--output_directory: where to save the frozen model

Testing

Example testing command:

```
python model_main.py --pipeline_config_path=training/ssd_mobilnet_v2_dir_v4.config --checkpoint_dir=trained
-inference-graphs/output_inference_graph_22000.pb --run_once
```

As output you will get the metrics selected, COCO metrics by default.

Useful stuff

1. Set image size for trainig and testing

In `training/config_file.config` set these (self explanatory) parameters:

```
model {
  ssd {
    image_resizer {
      fixed_shape_resizer {
        height: 300
        width: 300
      }
    }
    ...
}
}
```

2. Freeze backbone at training time

In `training/config_file.config` set this parameter:

```
ssd.config {
  ...
  freeze_variables: "feature_extractor"
  ...
}
```

3. how to quantize

In `training/config_file.config` set these parameters:

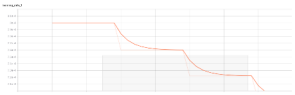
```
graph_rewriter {
  quantization {
    delay: 48000    # after this step stop quantization (start to
    weight_bits: 8  # bits for weights
    activation_bits: 8 # bits for activations
  }
}
```

4. Set learing rate exponential decay

In `training/config_file.config` set these parameters:

```
train_config {
  optimizer {
    rms_prop_optimizer {
      learning_rate {
        exponential_decay_learning_rate {
          initial_learning_rate: 0.0004
          decay_steps: 10000
          decay_factor: 0.95
        }
      }
      momentum_optimizer_value: 0.9
      decay: 0.9
      epsilon: 1.0
    }
  }
}
```

decay_steps: how many steps before decreasing the learning rate
decay_factor: 0.95: the final learning rate will be the 95% of the original



5. Print trainable variables

Add here this line:

`print('trainable variables: %s', tf.trainable_variables, sess = "")`

6. Count the number of network's parameters (Tensorflow)

Two ways:

1. The original Tensorflow repo provides a script called `inspect_checkpoint` to analyze `.ckpt` files.
   You can use this to count the number of the model's parameter.
   Note: this counts also all the training parameters that wont be part of the inference model. So it is a pessimistic estimation.
2. convert the frozen graph `graph.pb` to tf file with the Tensorflow command `tflite_convert` with "tensorflow 8-bit quantization". Look at the size of the `.tflite` file, that is the number of parameters

Example command:

```
tflite_convert --graph_def_file=graph.pb --output_file=graph.tflite --inference_type=QUANTIZED_UINT8 --
default_ranges_stats --default_ranges_stats --input_array=input --output_array=out_predictions --mean_value=128
--std_dev_values=127
```

Note: find the names of `--input_array` and `--output_array` opening the `graph.pb` with netron web