Configuring the Object Detection Training Pipeline

Overview

The TensorFlow Object Detection API uses protobuf files to configure the training and evaluation process. The schema for the training pipeline can be found in object_detection/protos/pipeline.proto. At a high level, the config file is split into 5 parts:

1. The `model` configuration. This defines what type of model will be trained (ie. meta-architecture, feature extractor).
2. The `train_config`, which decides what parameters should be used to train model parameters (ie. SGD parameters, input preprocessing and feature extractor initialization values).
3. The `eval_config`, which determines what set of metrics will be reported for evaluation.
4. The `train_input_config`, which defines what dataset the model should be trained on.
5. The `eval_input_config`, which defines what dataset the model will be evaluated on. Typically this should be different than the training input dataset.

A skeleton configuration file is shown below:

```
model {
  (... Add model config here...)
}

train_config : {
  (... Add train_config here...)
}

train_input_reader: {
  (... Add train_input configuration here...)
}

eval_config: {
  (... Add eval_config here...)
}

eval_input_reader: {
  (... Add eval_input configuration here...)
}
```

Picking Model Parameters

There are a large number of model parameters to configure. The best settings will depend on your given application. Faster R-CNN models are better suited to cases where high accuracy is desired and latency is of lower priority. Conversely, if processing time is the most important factor, SSD models are recommended. Read our paper for a more detailed discussion on the speed vs accuracy tradeoff.

To help new users get started, sample model configurations have been provided in the object_detection/samples/configs folder. The contents of these configuration files can be pasted into `model` field of the skeleton configuration. Users should note that the `num_classes` field should be changed to a value suited for the dataset the user is training on.

Anchor box parameters

Many object detection models use an anchor generator as a region-sampling strategy, which generates a large number of anchor boxes in a range of shapes and sizes, in many locations of the image. The detection algorithm then incrementally offsets the anchor box closest to the ground truth until it (closely) matches. You can specify the variety of and position of these anchor boxes in the `anchor_generator` config.

Usually, the anchor configs provided with pre-trained checkpoints are designed for large/versatile datasets (COCO, ImageNet), in which the goal is to improve accuracy for a wide range of object sizes and positions. But in most real-world applications, objects are confined to a limited number of sizes. So adjusting the anchors to be specific to your dataset and environment can both improve model accuracy and reduce training time.

The format for these anchor box parameters differ depending on your model architecture. For details about all fields, see the `anchor_generator` definition. On this page, we'll focus on parameters used in a traditional single shot detector (SSD) model and SSD models with a feature pyramid network (FPN) head.

Regardless of the model architecture, you'll need to understand the following anchor box concepts:

- **Scale**: This defines the variety of anchor box sizes. Each box size is defined as a proportion of the original image size (for SSD models) or as a factor of the filter's stride length (for FPN). The number of different sizes is defined using a range of "scales" (relative to image size) or "levels" (the level on the feature pyramid). For example, to detect small objects with the configurations below, the `min_scale` and `min_level` are set to a small value, while `max_scale` and `max_level` specify the largest objects to detect.
- **Aspect ratio**: This is the height/width ratio for the anchor boxes. For example, the `aspect_ratio` value of `1.0` creates a square, and `2.0` creates

a 1:2 rectangle (landscape orientation). You can define as many aspects as you want and each one is repeated at all anchor box scales.

Beware that increasing the total number of anchor boxes will exponentially increase computation costs. Whereas generating fewer anchors that have a higher chance to overlap with ground truth will both improve accuracy and reduce computation costs.

And although you can manually select values for both scale and aspect ratios that work well for your dataset, there are programmatic techniques you can use instead. One such strategy to determine the ideal aspect ratios is to perform k-means clustering of all the ground-truth bounding-box ratios, as shown in this Colab notebook to Generate SSD anchor box aspect ratios using k-means clustering.

Single Shot Detector (SSD) full model:

Setting `num_layers` to 6 means the model generates each box aspect at 6 different sizes. The exact sizes are not specified but they're evenly spaced out between the `min_scale` and `max_scale` values, which specify the smallest box size is 20% of the input image size and the largest is 95% that size.

```
model {
  ssd {
    anchor_generator {
      ssd_anchor_generator {
        num_layers: 6
        min_scale: 0.2
        max_scale: 0.95
        aspect_ratios: 1.0
        aspect_ratios: 2.0
        aspect_ratios: 0.5
      }
    }
  }
}
```

For more details, see `ssd_anchor_generator.proto`.

SSD with Feature Pyramid Network (FPN) head:

When using an FPN head, you must specify the anchor box size relative to the convolutional filter's stride length at a given pyramid level, using `anchor_scale`. So in this example, the box size is 4.0 multiplied by the layer's stride length. The number of sizes you get for each aspect simply depends on how many levels there are between the `min_level` and `max_level`.

```
model {
  ssd {
    anchor_generator {
```

```
      multiscale_anchor_generator {
        anchor_scale: 4.0
        min_level: 3
        max_level: 7
        aspect_ratios: 1.0
        aspect_ratios: 2.0
        aspect_ratios: 0.5
      }
    }
  }
}
```

For more details, see `multiscale_anchor_generator.proto`.

Defining Inputs

The TensorFlow Object Detection API accepts inputs in the TFRecord file format. Users must specify the locations of both the training and evaluation files. Additionally, users should also specify a label map, which define the mapping between a class id and class name. The label map should be identical between training and evaluation datasets.

An example training input configuration looks as follows:

```
train_input_reader: {
  tf_record_input_reader {
    input_path: "/usr/home/username/data/train.record-?????-of-00010"
  }
  label_map_path: "/usr/home/username/data/label_map.pbtxt"
}
```

The `eval_input_reader` follows the same format. Users should substitute the `input_path` and `label_map_path` arguments. Note that the paths can also point to Google Cloud Storage buckets (ie. "gs://project_bucket/train.record") to pull datasets hosted on Google Cloud.

Configuring the Trainer

The `train_config` defines parts of the training process:

1. Model parameter initialization.
2. Input preprocessing.
3. SGD parameters.

A sample `train_config` is below:

```
train_config: {
  batch_size: 1
  optimizer {
```

```
    momentum_optimizer: {
      learning_rate: {
        manual_step_learning_rate {
          initial_learning_rate: 0.0002
          schedule {
            step: 0
            learning_rate: .0002
          }
          schedule {
            step: 900000
            learning_rate: .00002
          }
          schedule {
            step: 1200000
            learning_rate: .000002
          }
        }
      }
      momentum_optimizer_value: 0.9
    }
    use_moving_average: false
  }
  fine_tune_checkpoint: "/usr/home/username/tmp/model.ckpt-#####"
  from_detection_checkpoint: true
  load_all_detection_checkpoint_vars: true
  gradient_clipping_by_norm: 10.0
  data_augmentation_options {
    random_horizontal_flip {
    }
  }
}
```

Input Preprocessing

The `data_augmentation_options` in `train_config` can be used to specify how training data can be modified. This field is optional.

SGD Parameters

The remainings parameters in `train_config` are hyperparameters for gradient descent. Please note that the optimal learning rates provided in these configuration files may depend on the specifics of the training setup (e.g. number of workers, gpu type).

Configuring the Evaluator

The main components to set in `eval_config` are `num_examples` and `metrics_set`. The parameter `num_examples` indicates the number of batches (currently of batch size 1) used for an evaluation cycle, and often is the total size of the evaluation dataset. The parameter `metrics_set` indicates which metrics to run during evaluation (i.e. `"coco_detection_metrics"`).