So you want to create a new model!

In this section, we discuss some of the abstractions that we use for defining detection models. If you would like to define a new model architecture for detection and use it in the TensorFlow Detection API, then this section should also serve as a high level guide to the files that you will need to edit to get your new model working.

DetectionModels (`object_detection/core/model.py`)

In order to be trained, evaluated, and exported for serving using our provided binaries, all models under the TensorFlow Object Detection API must implement the `DetectionModel` interface (see the full definition in `object_detection/core/model.py`). In particular, each of these models are responsible for implementing 5 functions:

- `preprocess` : Run any preprocessing (e.g., scaling/shifting/reshaping) of input values that is necessary prior to running the detector on an input image.
- `predict` : Produce "raw" prediction tensors that can be passed to loss or postprocess functions.
- `postprocess` : Convert predicted output tensors to final detections.
- `loss` : Compute scalar loss tensors with respect to provided groundtruth.
- `restore` : Load a checkpoint into the TensorFlow graph.

Given a `DetectionModel` at training time, we pass each image batch through the following sequence of functions to compute a loss which can be optimized via SGD:

```
inputs (images tensor) -> preprocess -> predict -> loss -> outputs (loss tensor)
```

And at eval time, we pass each image batch through the following sequence of functions to produce a set of detections:

```
inputs (images tensor) -> preprocess -> predict -> postprocess ->
  outputs (boxes tensor, scores tensor, classes tensor, num_detections tensor)
```

Some conventions to be aware of:

- `DetectionModel` s should make no assumptions about the input size or aspect ratio --- they are responsible for doing any resize/reshaping necessary (see docstring for the `preprocess` function).
- Output classes are always integers in the range `[0, num_classes)` . Any mapping of these integers to semantic labels is to be handled outside of this class. We never explicitly emit a "background class" --- thus 0 is the first non-background class and any logic of predicting and removing implicit background classes must be handled internally by the implementation.
- Detected boxes are to be interpreted as being in `[y_min, x_min, y_max, x_max]` format and normalized relative to the image window.
- We do not specifically assume any kind of probabilistic interpretation of the scores --- the only important thing is their relative ordering. Thus implementations of the postprocess function are free to output logits, probabilities, calibrated probabilities, or anything else.

Defining a new Faster R-CNN or SSD Feature Extractor

In most cases, you probably will not implement a `DetectionModel` from scratch --- instead you might create a new feature extractor to be used by one of the SSD or Faster R-CNN meta-architectures. (We think of meta-architectures as classes that define entire families of models using the `DetectionModel` abstraction).

Note: For the following discussion to make sense, we recommend first becoming familiar with the Faster R-CNN paper.

Let's now imagine that you have invented a brand new network architecture (say, "InceptionV100") for classification and want to see how InceptionV100 would behave as a feature extractor for detection (say, with Faster R-CNN). A similar procedure would hold for SSD models, but we'll discuss Faster R-CNN.

To use InceptionV100, we will have to define a new `FasterRCNNFeatureExtractor` and pass it to our `FasterRCNNMetaArch` constructor as input. See `object_detection/meta_architectures/faster_rcnn_meta_arch.py` for definitions of `FasterRCNNFeatureExtractor` and `FasterRCNNMetaArch`, respectively. A `FasterRCNNFeatureExtractor` must define a few functions:

- `preprocess`: Run any preprocessing of input values that is necessary prior to running the detector on an input image.
- `_extract_proposal_features`: Extract first stage Region Proposal Network (RPN) features.
- `_extract_box_classifier_features`: Extract second stage Box Classifier features.
- `restore_from_classification_checkpoint_fn`: Load a checkpoint into the TensorFlow graph.

See the `object_detection/models/faster_rcnn_resnet_v1_feature_extractor.py` definition as one example. Some remarks:

- We typically initialize the weights of this feature extractor using those from the Slim Resnet-101 classification checkpoint, and we know that images were preprocessed when training this checkpoint by subtracting a channel mean from each input image. Thus, we implement the preprocess function to replicate the same channel mean subtraction behavior.
- The "full" resnet classification network defined in slim is cut into two parts --- all but the last "resnet block" is put into the `_extract_proposal_features` function and the final block is separately defined in the `_extract_box_classifier_features function`. In general, some experimentation may be required to decide on an optimal layer at which to "cut" your feature extractor into these two pieces for Faster R-CNN.

Register your model for configuration

Assuming that your new feature extractor does not require nonstandard configuration, you will want to ideally be able to simply change the "feature_extractor.type" fields in your configuration protos to point to a new feature extractor. In order for our API to know how to understand this new type though, you will first have to register your new feature extractor with the model builder (`object_detection/builders/model_builder.py`), whose job is to create models from config protos..

Registration is simple --- just add a pointer to the new Feature Extractor class that you have defined in one of the SSD or Faster R-CNN Feature Extractor Class maps at the top of the `object_detection/builders/model_builder.py` file. We recommend adding a test in `object_detection/builders/model_builder_test.py` to make sure that parsing your proto will work as expected.

Taking your new model for a spin

After registration you are ready to go with your model! Some final tips:

- To save time debugging, try running your configuration file locally first (both training and evaluation).
- Do a sweep of learning rates to figure out which learning rate is best for your model.

- A small but often important detail: you may find it necessary to disable batchnorm training (that is, load the batch norm parameters from the classification checkpoint, but do not update them during gradient descent).