|  |  |
| --- | --- |
|  |  |
|  | import logging |
|  | import tensorflow as tf |
|  |  |
|  | from object\_detection import eval\_util |
|  | from object\_detection.core import prefetcher |
|  | from object\_detection.core import standard\_fields as fields |
|  | from object\_detection.metrics import coco\_evaluation |
|  | from object\_detection.utils import object\_detection\_evaluation |
|  |  |
|  | # A dictionary of metric names to classes that implement the metric. The classes |
|  | # in the dictionary must implement |
|  | # utils.object\_detection\_evaluation.DetectionEvaluator interface. |
|  | EVAL\_METRICS\_CLASS\_DICT = { |
|  | 'pascal\_voc\_detection\_metrics': |
|  | object\_detection\_evaluation.PascalDetectionEvaluator, |
|  | 'weighted\_pascal\_voc\_detection\_metrics': |
|  | object\_detection\_evaluation.WeightedPascalDetectionEvaluator, |
|  | 'pascal\_voc\_instance\_segmentation\_metrics': |
|  | object\_detection\_evaluation.PascalInstanceSegmentationEvaluator, |
|  | 'weighted\_pascal\_voc\_instance\_segmentation\_metrics': |
|  | object\_detection\_evaluation.WeightedPascalInstanceSegmentationEvaluator, |
|  | 'open\_images\_detection\_metrics': |
|  | object\_detection\_evaluation.OpenImagesDetectionEvaluator, |
|  | 'coco\_detection\_metrics': |
|  | coco\_evaluation.CocoDetectionEvaluator, |
|  | 'coco\_mask\_metrics': |
|  | coco\_evaluation.CocoMaskEvaluator, |
|  | } |
|  |  |
|  | EVAL\_DEFAULT\_METRIC = 'pascal\_voc\_detection\_metrics' |
|  |  |
|  |  |
|  | def \_extract\_prediction\_tensors(model, |
|  | create\_input\_dict\_fn, |
|  | ignore\_groundtruth=False): |
|  | """Restores the model in a tensorflow session. |
|  |  |
|  | Args: |
|  | model: model to perform predictions with. |
|  | create\_input\_dict\_fn: function to create input tensor dictionaries. |
|  | ignore\_groundtruth: whether groundtruth should be ignored. |
|  |  |
|  | Returns: |
|  | tensor\_dict: A tensor dictionary with evaluations. |
|  | """ |
|  | input\_dict = create\_input\_dict\_fn() |
|  | prefetch\_queue = prefetcher.prefetch(input\_dict, capacity=500) |
|  | input\_dict = prefetch\_queue.dequeue() |
|  | original\_image = tf.expand\_dims(input\_dict[fields.InputDataFields.image], 0) |
|  | preprocessed\_image, true\_image\_shapes = model.preprocess( |
|  | tf.to\_float(original\_image)) |
|  | prediction\_dict = model.predict(preprocessed\_image, true\_image\_shapes) |
|  | detections = model.postprocess(prediction\_dict, true\_image\_shapes) |
|  |  |
|  | groundtruth = None |
|  | if not ignore\_groundtruth: |
|  | groundtruth = { |
|  | fields.InputDataFields.groundtruth\_boxes: |
|  | input\_dict[fields.InputDataFields.groundtruth\_boxes], |
|  | fields.InputDataFields.groundtruth\_classes: |
|  | input\_dict[fields.InputDataFields.groundtruth\_classes], |
|  | fields.InputDataFields.groundtruth\_area: |
|  | input\_dict[fields.InputDataFields.groundtruth\_area], |
|  | fields.InputDataFields.groundtruth\_is\_crowd: |
|  | input\_dict[fields.InputDataFields.groundtruth\_is\_crowd], |
|  | fields.InputDataFields.groundtruth\_difficult: |
|  | input\_dict[fields.InputDataFields.groundtruth\_difficult] |
|  | } |
|  | if fields.InputDataFields.groundtruth\_group\_of in input\_dict: |
|  | groundtruth[fields.InputDataFields.groundtruth\_group\_of] = ( |
|  | input\_dict[fields.InputDataFields.groundtruth\_group\_of]) |
|  | if fields.DetectionResultFields.detection\_masks in detections: |
|  | groundtruth[fields.InputDataFields.groundtruth\_instance\_masks] = ( |
|  | input\_dict[fields.InputDataFields.groundtruth\_instance\_masks]) |
|  |  |
|  | return eval\_util.result\_dict\_for\_single\_example( |
|  | original\_image, |
|  | input\_dict[fields.InputDataFields.source\_id], |
|  | detections, |
|  | groundtruth, |
|  | class\_agnostic=( |
|  | fields.DetectionResultFields.detection\_classes not in detections), |
|  | scale\_to\_absolute=True) |
|  |  |
|  |  |
|  | def get\_evaluators(eval\_config, categories): |
|  | """Returns the evaluator class according to eval\_config, valid for categories. |
|  |  |
|  | Args: |
|  | eval\_config: evaluation configurations. |
|  | categories: a list of categories to evaluate. |
|  | Returns: |
|  | An list of instances of DetectionEvaluator. |
|  |  |
|  | Raises: |
|  | ValueError: if metric is not in the metric class dictionary. |
|  | """ |
|  | eval\_metric\_fn\_keys = eval\_config.metrics\_set |
|  | if not eval\_metric\_fn\_keys: |
|  | eval\_metric\_fn\_keys = [EVAL\_DEFAULT\_METRIC] |
|  | evaluators\_list = [] |
|  | for eval\_metric\_fn\_key in eval\_metric\_fn\_keys: |
|  | if eval\_metric\_fn\_key not in EVAL\_METRICS\_CLASS\_DICT: |
|  | raise ValueError('Metric not found: {}'.format(eval\_metric\_fn\_key)) |
|  | evaluators\_list.append( |
|  | EVAL\_METRICS\_CLASS\_DICT[eval\_metric\_fn\_key](categories=categories)) |
|  | return evaluators\_list |
|  |  |
|  |  |
|  | def evaluate(create\_input\_dict\_fn, create\_model\_fn, eval\_config, categories, |
|  | checkpoint\_dir, eval\_dir, graph\_hook\_fn=None): |
|  | """Evaluation function for detection models. |
|  |  |
|  | Args: |
|  | create\_input\_dict\_fn: a function to create a tensor input dictionary. |
|  | create\_model\_fn: a function that creates a DetectionModel. |
|  | eval\_config: a eval\_pb2.EvalConfig protobuf. |
|  | categories: a list of category dictionaries. Each dict in the list should |
|  | have an integer 'id' field and string 'name' field. |
|  | checkpoint\_dir: directory to load the checkpoints to evaluate from. |
|  | eval\_dir: directory to write evaluation metrics summary to. |
|  | graph\_hook\_fn: Optional function that is called after the training graph is |
|  | completely built. This is helpful to perform additional changes to the |
|  | training graph such as optimizing batchnorm. The function should modify |
|  | the default graph. |
|  |  |
|  | Returns: |
|  | metrics: A dictionary containing metric names and values from the latest |
|  | run. |
|  | """ |
|  |  |
|  | model = create\_model\_fn() |
|  |  |
|  | if eval\_config.ignore\_groundtruth and not eval\_config.export\_path: |
|  | logging.fatal('If ignore\_groundtruth=True then an export\_path is ' |
|  | 'required. Aborting!!!') |
|  |  |
|  | tensor\_dict = \_extract\_prediction\_tensors( |
|  | model=model, |
|  | create\_input\_dict\_fn=create\_input\_dict\_fn, |
|  | ignore\_groundtruth=eval\_config.ignore\_groundtruth) |
|  |  |
|  | def \_process\_batch(tensor\_dict, sess, batch\_index, counters): |
|  | """Evaluates tensors in tensor\_dict, visualizing the first K examples. |
|  |  |
|  | This function calls sess.run on tensor\_dict, evaluating the original\_image |
|  | tensor only on the first K examples and visualizing detections overlaid |
|  | on this original\_image. |
|  |  |
|  | Args: |
|  | tensor\_dict: a dictionary of tensors |
|  | sess: tensorflow session |
|  | batch\_index: the index of the batch amongst all batches in the run. |
|  | counters: a dictionary holding 'success' and 'skipped' fields which can |
|  | be updated to keep track of number of successful and failed runs, |
|  | respectively. If these fields are not updated, then the success/skipped |
|  | counter values shown at the end of evaluation will be incorrect. |
|  |  |
|  | Returns: |
|  | result\_dict: a dictionary of numpy arrays |
|  | """ |
|  | try: |
|  | result\_dict = sess.run(tensor\_dict) |
|  | counters['success'] += 1 |
|  | except tf.errors.InvalidArgumentError: |
|  | logging.info('Skipping image') |
|  | counters['skipped'] += 1 |
|  | return {} |
|  | global\_step = tf.train.global\_step(sess, tf.train.get\_global\_step()) |
|  | if batch\_index < eval\_config.num\_visualizations: |
|  | tag = 'image-{}'.format(batch\_index) |
|  | eval\_util.visualize\_detection\_results( |
|  | result\_dict, |
|  | tag, |
|  | global\_step, |
|  | categories=categories, |
|  | summary\_dir=eval\_dir, |
|  | export\_dir=eval\_config.visualization\_export\_dir, |
|  | show\_groundtruth=eval\_config.visualize\_groundtruth\_boxes, |
|  | groundtruth\_box\_visualization\_color=eval\_config. |
|  | groundtruth\_box\_visualization\_color, |
|  | min\_score\_thresh=eval\_config.min\_score\_threshold, |
|  | max\_num\_predictions=eval\_config.max\_num\_boxes\_to\_visualize, |
|  | skip\_scores=eval\_config.skip\_scores, |
|  | skip\_labels=eval\_config.skip\_labels, |
|  | keep\_image\_id\_for\_visualization\_export=eval\_config. |
|  | keep\_image\_id\_for\_visualization\_export) |
|  | return result\_dict |
|  |  |
|  | variables\_to\_restore = tf.global\_variables() |
|  | global\_step = tf.train.get\_or\_create\_global\_step() |
|  | variables\_to\_restore.append(global\_step) |
|  |  |
|  | if graph\_hook\_fn: graph\_hook\_fn() |
|  |  |
|  | if eval\_config.use\_moving\_averages: |
|  | variable\_averages = tf.train.ExponentialMovingAverage(0.0) |
|  | variables\_to\_restore = variable\_averages.variables\_to\_restore() |
|  | saver = tf.train.Saver(variables\_to\_restore) |
|  |  |
|  | def \_restore\_latest\_checkpoint(sess): |
|  | latest\_checkpoint = tf.train.latest\_checkpoint(checkpoint\_dir) |
|  | saver.restore(sess, latest\_checkpoint) |
|  |  |
|  | metrics = eval\_util.repeated\_checkpoint\_run( |
|  | tensor\_dict=tensor\_dict, |
|  | summary\_dir=eval\_dir, |
|  | evaluators=get\_evaluators(eval\_config, categories), |
|  | batch\_processor=\_process\_batch, |
|  | checkpoint\_dirs=[checkpoint\_dir], |
|  | variables\_to\_restore=None, |
|  | restore\_fn=\_restore\_latest\_checkpoint, |
|  | num\_batches=eval\_config.num\_examples, |
|  | eval\_interval\_secs=eval\_config.eval\_interval\_secs, |
|  | max\_number\_of\_evaluations=(1 if eval\_config.ignore\_groundtruth else |
|  | eval\_config.max\_evals |
|  | if eval\_config.max\_evals else None), |
|  | master=eval\_config.eval\_master, |
|  | save\_graph=eval\_config.save\_graph, |
|  | save\_graph\_dir=(eval\_dir if eval\_config.save\_graph else '')) |
|  |  |
|  | return metrics |