```
"""
Mask R-CNN
The main Mask R-CNN model implemenetation.

Copyright (c) 2017 Matterport, Inc.
Licensed under the MIT License (see LICENSE for details)
Written by Waleed Abdulla
"""

import os
import sys
import glob
import random
import math
import datetime
import itertools
import json
import re
import logging
# from collections import OrderedDict
import numpy as np
# import scipy.misc
import tensorflow as tf
# import keras
# import keras.backend as K
# import keras.layers as KL
# import keras.initializers as KI
import keras.engine as KE
# import keras.models as KM
sys.path.append('..')
import mrcnn.utils as utils


############################################################
#  Detection Target Layer
############################################################

def overlaps_graph(boxes1, boxes2):
    """
    Computes IoU overlaps between two sets of boxes.
    boxes1, boxes2: [N, (y1, x1, y2, x2)].
    """
    # 1. Tile boxes2 and repeate boxes1. This allows us to compare
    # every boxes1 against every boxes2 without loops.
    # TF doesn't have an equivalent to np.repeate() so simulate it
    # using tf.tile() and tf.reshape.
    b1 = tf.reshape(tf.tile(tf.expand_dims(boxes1, 1),
                            [1, 1, tf.shape(boxes2)[0]]), [-1, 4])
    b2 = tf.tile(boxes2, [tf.shape(boxes1)[0], 1])
    # 2. Compute intersections
    b1_y1, b1_x1, b1_y2, b1_x2 = tf.split(b1, 4, axis=1)
    b2_y1, b2_x1, b2_y2, b2_x2 = tf.split(b2, 4, axis=1)
    y1 = tf.maximum(b1_y1, b2_y1)
    x1 = tf.maximum(b1_x1, b2_x1)
    y2 = tf.minimum(b1_y2, b2_y2)
    x2 = tf.minimum(b1_x2, b2_x2)
    intersection = tf.maximum(x2 - x1, 0) * tf.maximum(y2 - y1, 0)
    # 3. Compute unions
    b1_area = (b1_y2 - b1_y1) * (b1_x2 - b1_x1)
    b2_area = (b2_y2 - b2_y1) * (b2_x2 - b2_x1)
    union = b1_area + b2_area - intersection
    # 4. Compute IoU and reshape to [boxes1, boxes2]
    iou = intersection / union
    overlaps = tf.reshape(iou, [tf.shape(boxes1)[0], tf.shape(boxes2)[0]])
    return overlaps


def detection_targets_graph(proposals, gt_class_ids, gt_boxes, gt_masks, config):
    """
    Generates detection targets for one image. Subsamples proposals and
    generates target class IDs, bounding box deltas, and masks for each.

    Inputs:
    -------
    proposals:          [N, (y1, x1, y2, x2)] in normalized coordinates.
                        Might be zero padded if there are not enough proposals.
    gt_class_ids:       [MAX_GT_INSTANCES] int class IDs
    gt_boxes:           [MAX_GT_INSTANCES, (y1, x1, y2, x2)] in normalized coordinates.
    gt_masks:           [height, width, MAX_GT_INSTANCES] of boolean type.
```

```

    Returns:            Target ROIs and corresponding class IDs, bounding box shifts, and masks.
    --------
    rois:               [TRAIN_ROIS_PER_IMAGE, (y1, x1, y2, x2)] in normalized coordinates
    class_ids:          [TRAIN_ROIS_PER_IMAGE]. Integer class IDs. Zero padded.
    deltas:             [TRAIN_ROIS_PER_IMAGE, NUM_CLASSES, (dy, dx, log(dh), log(dw))]
                        Class-specific bbox refinments.
    masks:              [TRAIN_ROIS_PER_IMAGE, height, width). Masks cropped to bbox
                        boundaries and resized to neural network output size.

    Note: Returned arrays might be zero padded if not enough target ROIs.
    """
    # Assertions
    asserts = [
        tf.Assert(tf.greater(tf.shape(proposals)[0], 0), [proposals], name="roi_assertion"),
    ]

    with tf.control_dependencies(asserts):
        proposals = tf.identity(proposals)

    # Remove zero padding

    proposals, _        = utils.trim_zeros_graph(proposals, name="trim_proposals")
    gt_boxes, non_zeros = utils.trim_zeros_graph(gt_boxes, name="trim_gt_boxes")
    gt_class_ids        = tf.boolean_mask(gt_class_ids, non_zeros, name="trim_gt_class_ids")
    gt_masks            = tf.gather(gt_masks, tf.where(non_zeros)[:, 0], axis=2,name="trim_gt_masks")

    # Handle COCO crowds
    # A crowd box in COCO is a bounding box around several instances. Exclude
    # them from training. A crowd box is given a negative class ID.

    # tf.where : returns the coordinates of true elements of  the specified conditon.
    #            The coordinates are returned in a 2-D tensor where the first dimension (rows)
    #            represents the number of true elements, and the second dimension (columns)
    #            represents the coordinates of the true elements.
    #            Keep in mind, the shape of the output tensor can vary depending on how many
    #            true values there are in input. Indices are output in row-major order.

    # tf.gather: Gather slices from params axis (default = 0) according to indices.
    #            indices must be an integer tensor of any dimension (usually 0-D or 1-D).
    #            Produces an output tensor with shape params.shape[:axis] + indices.shape + params.shape[axis + 1:] where:
    crowd_ix     = tf.where(gt_class_ids < 0)[:, 0]
    non_crowd_ix = tf.where(gt_class_ids > 0)[:, 0]
    crowd_boxes  = tf.gather(gt_boxes, crowd_ix)
    crowd_masks  = tf.gather(gt_masks, crowd_ix, axis=2)
    gt_class_ids = tf.gather(gt_class_ids, non_crowd_ix)
    gt_boxes     = tf.gather(gt_boxes, non_crowd_ix)
    gt_masks     = tf.gather(gt_masks, non_crowd_ix, axis=2)

    # Compute overlaps matrix [proposals, gt_boxes]
    overlaps = overlaps_graph(proposals, gt_boxes)

    # Compute overlaps with crowd boxes [anchors, crowds]
    crowd_overlaps = overlaps_graph(proposals, crowd_boxes)
    crowd_iou_max  = tf.reduce_max(crowd_overlaps, axis=1)
    no_crowd_bool  = (crowd_iou_max < 0.001)

    # Determine postive and negative ROIs
    roi_iou_max = tf.reduce_max(overlaps, axis=1)

    # 1. Positive ROIs are those with >= 0.5 IoU with a GT box
    positive_roi_bool = (roi_iou_max >= 0.5)
    positive_indices = tf.where(positive_roi_bool)[:, 0]

    # 2. Negative ROIs are those with < 0.5 with every GT box. Skip crowds.
    negative_indices = tf.where(tf.logical_and(roi_iou_max < 0.5, no_crowd_bool))[:, 0]

    # Subsample ROIs. Aim for 33% positive
    # Positive ROIs
    positive_count = int(config.TRAIN_ROIS_PER_IMAGE * config.ROI_POSITIVE_RATIO)
    positive_indices = tf.random_shuffle(positive_indices)[:positive_count]
    positive_count = tf.shape(positive_indices)[0]

    # Negative ROIs. Add enough to maintain positive:negative ratio.

    # negative_count = int((positive_count / config.ROI_POSITIVE_RATIO) - positive_count)

    r = 1.0 / config.ROI_POSITIVE_RATIO
```

```python
    negative_count = tf.cast(r * tf.cast(positive_count, tf.float32), tf.int32) - positive_count

    negative_indices = tf.random_shuffle(negative_indices)[:negative_count]

    # Gather selected ROIs
    positive_rois = tf.gather(proposals, positive_indices)
    negative_rois = tf.gather(proposals, negative_indices)

    # Assign positive ROIs to GT boxes.
    positive_overlaps     = tf.gather(overlaps, positive_indices)
    roi_gt_box_assignment = tf.argmax(positive_overlaps, axis=1)
    roi_gt_boxes          = tf.gather(gt_boxes, roi_gt_box_assignment)
    roi_gt_class_ids      = tf.gather(gt_class_ids, roi_gt_box_assignment)

    # Compute bbox refinement for positive ROIs
    deltas = utils.box_refinement_graph(positive_rois, roi_gt_boxes)
    deltas /= config.BBOX_STD_DEV

    # Assign positive ROIs to GT masks
    # Permute masks to [N, height, width, 1]

    transposed_masks = tf.expand_dims(tf.transpose(gt_masks, [2, 0, 1]), -1)

    # Pick the right mask for each ROI
    roi_masks = tf.gather(transposed_masks, roi_gt_box_assignment)

    # Compute mask targets
    boxes = positive_rois
    if config.USE_MINI_MASK:
        # Transform ROI corrdinates from normalized image space
        # to normalized mini-mask space.
        y1, x1, y2, x2 = tf.split(positive_rois, 4, axis=1)
        gt_y1, gt_x1, gt_y2, gt_x2 = tf.split(roi_gt_boxes, 4, axis=1)
        gt_h = gt_y2 - gt_y1
        gt_w = gt_x2 - gt_x1
        y1 = (y1 - gt_y1) / gt_h
        x1 = (x1 - gt_x1) / gt_w
        y2 = (y2 - gt_y1) / gt_h
        x2 = (x2 - gt_x1) / gt_w
        boxes = tf.concat([y1, x1, y2, x2], 1)
    box_ids = tf.range(0, tf.shape(roi_masks)[0])
    masks = tf.image.crop_and_resize(tf.cast(roi_masks, tf.float32), boxes,
                                     box_ids,
                                     config.MASK_SHAPE)
    # Remove the extra dimension from masks.
    masks = tf.squeeze(masks, axis=3)

    # Threshold mask pixels at 0.5 to have GT masks be 0 or 1 to use with
    # binary cross entropy loss.
    masks = tf.round(masks)

    # Append negative ROIs and pad bbox deltas and masks that
    # are not used for negative ROIs with zeros.
    rois = tf.concat([positive_rois, negative_rois], axis=0)
    N    = tf.shape(negative_rois)[0]
    P    = tf.maximum(config.TRAIN_ROIS_PER_IMAGE - tf.shape(rois)[0], 0)
    rois = tf.pad(rois, [(0, P), (0, 0)])
    roi_gt_boxes = tf.pad(roi_gt_boxes, [(0, N + P), (0, 0)])
    roi_gt_class_ids = tf.pad(roi_gt_class_ids, [(0, N + P)])
    deltas = tf.pad(deltas, [(0, N + P), (0, 0)])
    masks  = tf.pad(masks, [[0, N + P], (0, 0), (0, 0)])

    return rois, roi_gt_class_ids, deltas, masks


class DetectionTargetLayer(KE.Layer):
    """Subsamples proposals and generates target box refinment, class_ids,
    and masks for each.

    Inputs:
    -------
    proposals:  [batch, N, (y1, x1, y2, x2)] in normalized coordinates. Might
                be zero padded if there are not enough proposals.
    gt_class_ids: [batch, MAX_GT_INSTANCES] Integer class IDs.
    gt_boxes:   [batch, MAX_GT_INSTANCES, (y1, x1, y2, x2)] in normalized
                coordinates.
    gt_masks:   [batch, height, width, MAX_GT_INSTANCES] of boolean type

    Returns:
```

```
    -------
            Target ROIs and corresponding class IDs, bounding box shifts, and masks.
    rois:               [batch, TRAIN_ROIS_PER_IMAGE, (y1, x1, y2, x2)] in normalized coordinates
    target_class_ids:   [batch, TRAIN_ROIS_PER_IMAGE]. Integer class IDs.
    target_deltas:      [batch, TRAIN_ROIS_PER_IMAGE, NUM_CLASSES,(dy, dx, log(dh), log(dw), class_id)]
                          Class-specific bbox refinments.
    target_mask:        [batch, TRAIN_ROIS_PER_IMAGE, height, width)
                        Masks cropped to bbox boundaries and resized to neural network output size.

    Note: Returned arrays might be zero padded if not enough target ROIs.
    """

    def __init__(self, config, **kwargs):
        # super(DetectionTargetLayer, self).__init__(**kwargs)
        super().__init__(**kwargs)
        self.config = config

    def call(self, inputs):
        proposals = inputs[0]
        gt_class_ids = inputs[1]
        gt_boxes = inputs[2]
        gt_masks = inputs[3]

        # Slice the batch and run a graph for each slice
        # TODO: Rename target_bbox to target_deltas for clarity

        names = ["rois", "target_class_ids", "target_bbox", "target_mask"]
        outputs = utils.batch_slice([proposals, gt_class_ids, gt_boxes, gt_masks],
                                    lambda w, x, y, z: detection_targets_graph(w, x, y, z, self.config),
                                    self.config.IMAGES_PER_GPU, names=names)
        return outputs

    def compute_output_shape(self, input_shape):
        return [
            (None, self.config.TRAIN_ROIS_PER_IMAGE, 4),  # rois
            (None, 1),  # class_ids
            (None, self.config.TRAIN_ROIS_PER_IMAGE, 4),  # deltas
            (None, self.config.TRAIN_ROIS_PER_IMAGE, self.config.MASK_SHAPE[0],
             self.config.MASK_SHAPE[1])  # masks
        ]

    def compute_mask(self, inputs, mask=None):
        return [None, None, None, None]


############################################################
#  Detection Layer
############################################################

def clip_to_window(window, boxes):
    """
    window: (y1, x1, y2, x2). The window in the image we want to clip to.
    boxes: [N, (y1, x1, y2, x2)]
    """
    boxes[:, 0] = np.maximum(np.minimum(boxes[:, 0], window[2]), window[0])
    boxes[:, 1] = np.maximum(np.minimum(boxes[:, 1], window[3]), window[1])
    boxes[:, 2] = np.maximum(np.minimum(boxes[:, 2], window[2]), window[0])
    boxes[:, 3] = np.maximum(np.minimum(boxes[:, 3], window[3]), window[1])
    return boxes


def refine_detections(rois, probs, deltas, window, config):
    """Refine classified proposals and filter overlaps and return final
    detections.

    Inputs:
        rois: [N, (y1, x1, y2, x2)] in normalized coordinates
        probs: [N, num_classes]. Class probabilities.
        deltas: [N, num_classes, (dy, dx, log(dh), log(dw))]. Class-specific
                bounding box deltas.
        window: (y1, x1, y2, x2) in image coordinates. The part of the image
            that contains the image excluding the padding.

    Returns detections shaped: [N, (y1, x1, y2, x2, class_id, score)]
    """
    # Class IDs per ROI
    class_ids = np.argmax(probs, axis=1)
    # Class probability of the top class of each ROI
    class_scores = probs[np.arange(class_ids.shape[0]), class_ids]
```

```python
    # Class-specific bounding box deltas
    deltas_specific = deltas[np.arange(deltas.shape[0]), class_ids]
    # Apply bounding box deltas
    # Shape: [boxes, (y1, x1, y2, x2)] in normalized coordinates
    refined_rois = utils.apply_box_deltas(
        rois, deltas_specific * config.BBOX_STD_DEV)
    # Convert coordiates to image domain
    # TODO: better to keep them normalized until later
    height, width = config.IMAGE_SHAPE[:2]
    refined_rois *= np.array([height, width, height, width])
    # Clip boxes to image window
    refined_rois = clip_to_window(window, refined_rois)
    # Round and cast to int since we're deadling with pixels now
    refined_rois = np.rint(refined_rois).astype(np.int32)

    # TODO: Filter out boxes with zero area

    # Filter out background boxes
    keep = np.where(class_ids > 0)[0]
    # Filter out low confidence boxes
    if config.DETECTION_MIN_CONFIDENCE:
        keep = np.intersect1d(
            keep, np.where(class_scores >= config.DETECTION_MIN_CONFIDENCE)[0])

    # Apply per-class NMS
    pre_nms_class_ids = class_ids[keep]
    pre_nms_scores = class_scores[keep]
    pre_nms_rois = refined_rois[keep]
    nms_keep = []
    for class_id in np.unique(pre_nms_class_ids):
        # Pick detections of this class
        ixs = np.where(pre_nms_class_ids == class_id)[0]
        # Apply NMS
        class_keep = utils.non_max_suppression(
            pre_nms_rois[ixs], pre_nms_scores[ixs],
            config.DETECTION_NMS_THRESHOLD)
        # Map indicies
        class_keep = keep[ixs[class_keep]]
        nms_keep = np.union1d(nms_keep, class_keep)
    keep = np.intersect1d(keep, nms_keep).astype(np.int32)

    # Keep top detections
    roi_count = config.DETECTION_MAX_INSTANCES
    top_ids = np.argsort(class_scores[keep])[::-1][:roi_count]
    keep = keep[top_ids]

    # Arrange output as [N, (y1, x1, y2, x2, class_id, score)]
    # Coordinates are in image domain.
    result = np.hstack((refined_rois[keep],
                        class_ids[keep][..., np.newaxis],
                        class_scores[keep][..., np.newaxis]))
    return result


class DetectionLayer(KE.Layer):
    """Takes classified proposal boxes and their bounding box deltas and
    returns the final detection boxes.

    Returns:
    [batch, num_detections, (y1, x1, y2, x2, class_score)] in pixels
    """

    def __init__(self, config=None, **kwargs):
        super(DetectionLayer, self).__init__(**kwargs)
        self.config = config

    def call(self, inputs):
        def wrapper(rois, mrcnn_class, mrcnn_bbox, image_meta):
            detections_batch = []
            for b in range(self.config.BATCH_SIZE):
                _, _, window, _ =  parse_image_meta(image_meta)
                detections = refine_detections(
                    rois[b], mrcnn_class[b], mrcnn_bbox[b], window[b], self.config)
                # Pad with zeros if detections < DETECTION_MAX_INSTANCES
                gap = self.config.DETECTION_MAX_INSTANCES - detections.shape[0]
                assert gap >= 0
                if gap > 0:
                    detections = np.pad(
                        detections, [(0, gap), (0, 0)], 'constant', constant_values=0)
```

```
                detections_batch.append(detections)

            # Stack detections and cast to float32
            # TODO: track where float64 is introduced
            detections_batch = np.array(detections_batch).astype(np.float32)
            # Reshape output
            # [batch, num_detections, (y1, x1, y2, x2, class_score)] in pixels
            return np.reshape(detections_batch, [self.config.BATCH_SIZE, self.config.DETECTION_MAX_INSTANCES, 6])

        # Return wrapped function
        return tf.py_func(wrapper, inputs, tf.float32)

    def compute_output_shape(self, input_shape):
        return (None, self.config.DETECTION_MAX_INSTANCES, 6)


```