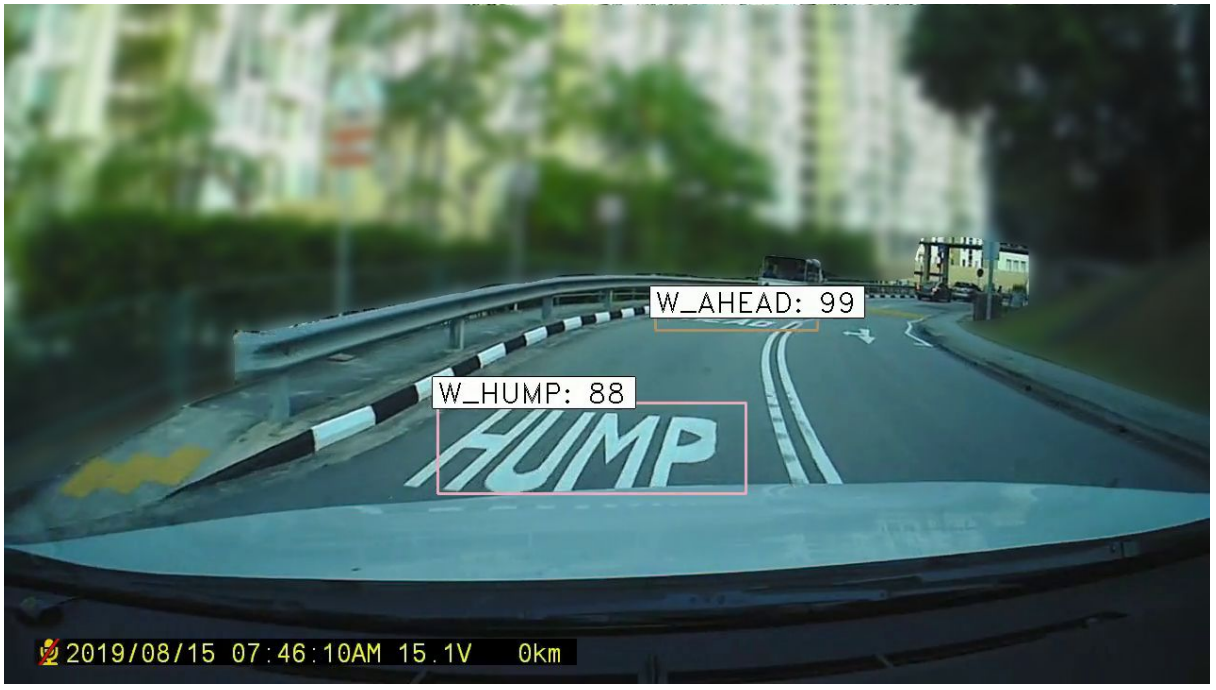
Intelligent Sensing Systems - RealTime Audio Visual Sensing And Sense Making (RTAVS) CA1

Faster R-CNN Object Detection



Authors: team **ATY**

Ang Boon Yew (A0096966E)

Tea Lee Seng (A0198538J)

Yang Xiaoyan (A0056720L)

Table of contents

1. Introduction

Object detection and recognition is a trivial task for humans, but is a challenging problem for digital processing methods using computers. In this task, the same object may appear with different scales and orientations, depending on its location in the image, for which conventional image recognition techniques might not be able to handle. In addition, an iterative grid search across the entire image might be computationally expensive and may not be suitable for real-time application. To address these issues, methods such as Region-based Convolutional Neural Networks (R-CNN) and its variants have been introduced and have been successful for this task [1,2,3].

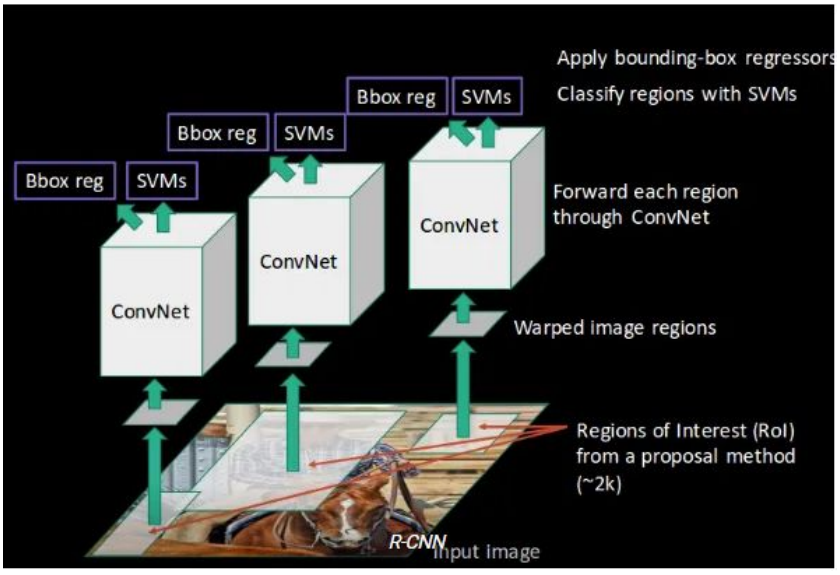
There are different series of R-CNN object detection algorithms. The first R-CNN methodology was introduced in Oct 2014 by Ross Girshick, which performs an object classification model on such selected region on an image [1]. An improved version over this R-CNN model was introduced in Apr 2015 as the Fast R-CNN model, which reduced the computational time and improved the detection accuracy [2] . The Faster R-CNN model build upons this method and further introduced the region-proposal network (RPNs) to further reduce the processing time for object detection [3].

For the operation of autonomous vehicles, object detection and recognition is a crucial task that is performed continuously in order to inform the vehicle controller of potential obstacles and changes in the road environment. As a quick response may be required to such obstacles and changes, the employed method needs to be fast and accurate in order to provide vital information about the vehicle's environment. Hence, state-of-the-art methods such as Faster R-CNN and YOLOv3 are viable methods that can be used for this task and will be explored in this project.

In this project, we provide a background of the R-CNN family of object detection models and introduce the methodology of the Faster R-CNN model in detail. We also detail the API design of the code used to train and test the model for a road object detection task and evaluate the performance of the model.

1.1. Region-Based CNN (R-CNN)

The R-CNN method first extracts different regions from the input image using selective search and passes each region into a CNN for specific feature extraction for that region. These features maps are then used to detect objects and bounding box regression for each selected region. As every region is passed to CNN for feature extraction, the R-CNN method takes a much longer time to process and detect objects in each image, typically 40-50 seconds for each image.

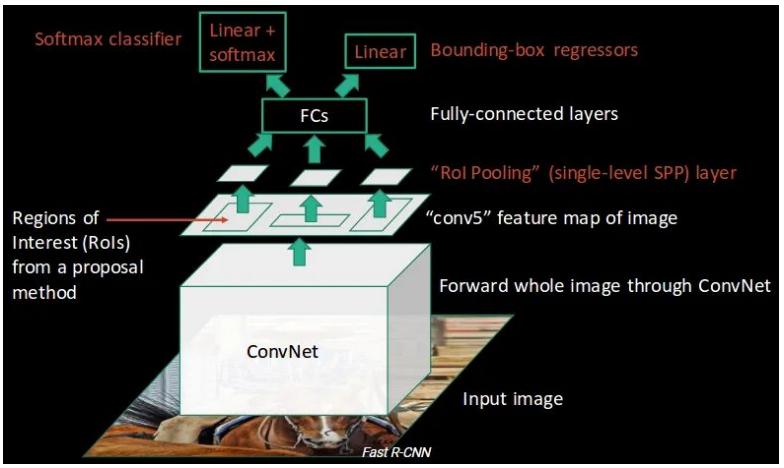


Source: https://www.analyticsvidhya.com/blog/2018/11/implementation-faster-r-cnn-python-object-detection/ [5]

1.2. Fast R-CNN

In the Fast R-CNN method, the images are passed to a base CNN model in order to extract feature maps. A selective search to generate region of interests (ROI) is performed using the feature maps, which undergo max pooling for each region to get fix-length of feature vector outputs. These feature vectors are then passed to a fully connected layer branch into two output layers: one for object class, another one for bounding box prediction in order to classify the object as well as to determine its location on the image. As only a single CNN model is used for feature extraction, Fast R-CNN executed faster as compared to R-CNN,

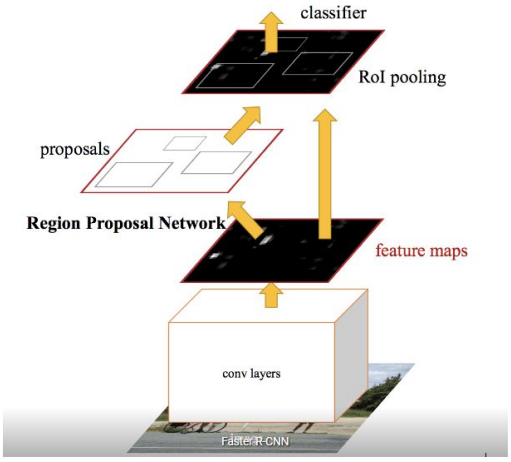
but still faced challenges in training when a large dataset was used due to the large number of selected regions generated at the region proposal stage.



Source: https://www.analyticsvidhya.com/blog/2018/11/implementation-faster-r-cnn-python-object-detection/ [5]

1.3. Faster R-CNN

The Faster R-CNN improves the region proposal process over the Fast R-CNN architecture by introducing a Region Proposal Network (RPN) layer instead of a selective search process. The base network and classifier layers are similar to those of the Fast R-CNN model. The extracted feature maps from the base networks are passed through the RPN and return object proposals (regions and anchor boxes). These region proposals under ROI pooling in order to get the same feature vector size, which are then passed through a fully connected layer to do the object classification and bounding box regression. As a result of this architecture, Faster R-CNN provides an improvement in object detection and processing time over Fast R-CNN.



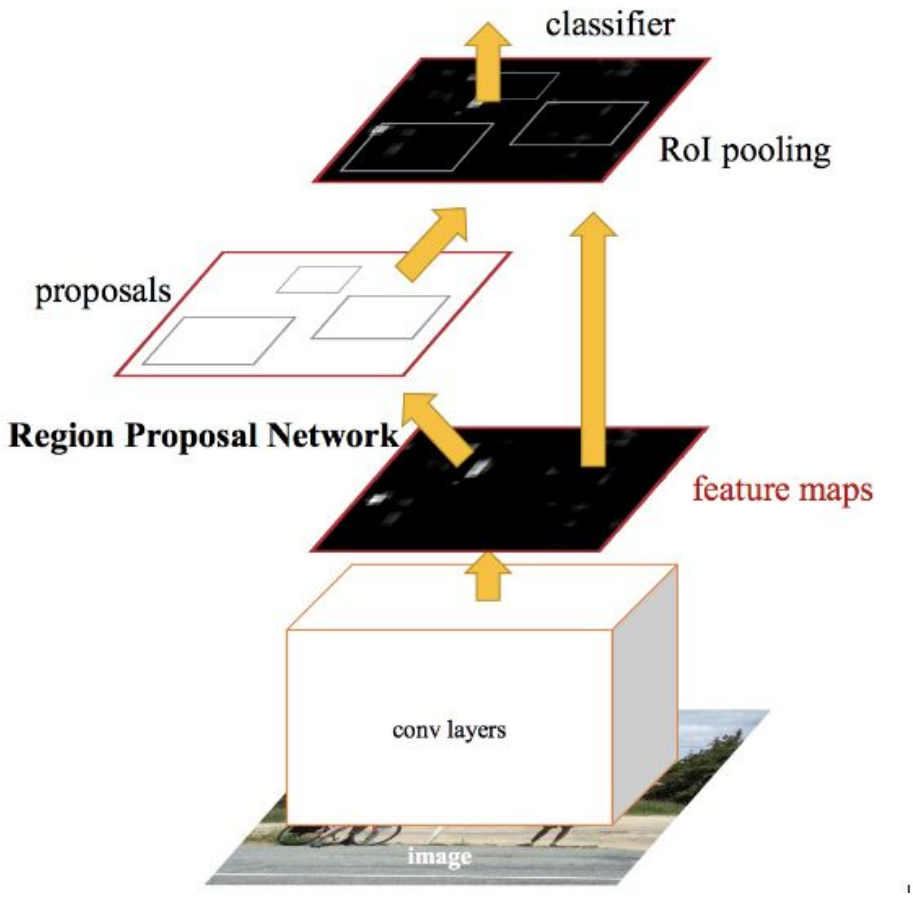
Source:
https://www.analyticsvidhya.com/blog/2018/11/implementation-faster-r-cnn-python-object-detection/ [4]

1.4. Comparison of different R-CNNs

Below table shows the features, process time and limitation of the three versions of R-CNNs.

Algorithm	**Features**	**Process time per image**	**Limitation**
R-CNN Oct, 2014	Extract regions by using random selective search	40~50 sec	High computation time as each region is passed to the CNN separately.
Fast R-CNN Apr, 2015	Pass entire image to CNN to extract feature maps, then generate the regions.	2 sec	Selective search is slow.
Faster R-CNN Jun, 2016	Region Proposal Network (RPN) is added.	0.2 sec 0.8 sec on our GPU. 20 sec on cpu.	Object proposal takes time and the performance of systems depends on how the previous system has performed.

2. Faster RCNN understanding



Source: https://www.analyticsvidhya.com/blog/2018/11/implementation-faster-r-cnn-python-object-detection/ [5]

Faster RCNN is the 3rd improvement of RCNN object detection series. Its major improvement is replacing manual designed algorithm of selective search to learnable Region Proposal Network (RPN). It also replaces heavy SVMs for classification and regression tasks with neural network dense layer. This enables end to end network training.

2.1. Base Network: Residual Neural Network (ResNet)

In Fast R-CNN and Faster R-CNN, input images are passed into a CNN first in order to extract feature maps for further processing tasks such as object detection and bounding box regression. This network can leverage off popular network architectures such as VGG-16 or ResNet, which are known to be able to produce high quality feature maps for image recognition tasks [3]. These feature

maps can then be used by downstream networks for other tasks. In our implementation of Faster R-CNN model, we use the ResNet50 architecture as our base network, which serves as a common trunk to the Region Proposal Network and the Classifier Network.

API Design: The base networks are implemented in our code as a class constructor method, which can be used to create a base network object. The base network model is initiated by calling the ResNet.nn_base() method, while the RPN networks and classifier networks are initiated using ResNet.rpn() and ResNet.classifer() methods.

```
class ResNet:

    # def __init__(self):

    @staticmethod
    def get_weight_path(K):...

    @staticmethod
    def get_img_output_length(width, height):...

    @staticmethod
    def identity_block(input_tensor, kernel_size, filters, stage, block, trainable=True):...

    @staticmethod
    def identity_block_td(input_tensor, kernel_size, filters, stage, block, trainable=True):...

    @staticmethod
    def conv_block(input_tensor, kernel_size, filters, stage, block, strides=(2, 2), trainable=True):...

    @staticmethod
    def conv_block_td(input_tensor, kernel_size, filters, stage, block, input_shape, strides=(2, 2),...

    @staticmethod
    def nn_base(input_tensor=None, trainable=False):...

    @staticmethod
    def rpn(base_layers, num_anchors):...

    @staticmethod
    def classifier(base_layers, input_rois, num_rois, nb_classes=21, trainable=False):...

    @staticmethod
    def classifier_layers(x, input_shape, trainable=False):...

'''Simple ResNet50 model for Keras.
# Reference:
- [Deep Residual Learning for Image Recognition](https://arxiv.org/abs/1512.03385)
Adapted from code contributed by BigMoyan.
'''
```

2.2. Region Proposal Network (RPN)

Region Proposal Network created to resolve selective search slowness problem found in RCNN and Fast RCNN. RCNN requires 2000 selective search of ROI (region of interest) on original image and passing ROI smaller images through ConvNet e.g. ResNet50 or VGG16, for feature extraction. Fast RCNN improved the situation for passing the whole images though ConvNet once to generate the feature map, and only performs 2000 selective search on feature map. Instead of

performing selective search, Faster RCNN replaces that with Region Proposal Network (RPN) and let the network performs learning to estimate where ROI are in feature map.

The objective of the RPN is hence to propose anchor boxes that potentially contain objects, as well as fine tuning the exact location of the bounding box over the object with an estimated coordinate translation of the anchor boxes.

Anchor Boxes: In Faster R-CNN, the concept of anchor boxes, which serve as reference bounding boxes on input images is introduced. Anchor boxes are bounding boxes of varying sizes and aspect ratios (width to height ratios), and are fixed with respect to a specified point on the image or feature map. This point serves as the center point of the box. In this approach, anchor boxes are generated based on the feature maps generated from the base network, which are the last convolutional layer in the network before the classifier layers. In VGG-16, this layer has a dimension of 7 x 7 x 512.

In order to generate such anchor boxes, a sliding window e.g. size of 3 x 3 is iterated over the given feature map space. The center of this sliding window serves as the anchor point and its position is projected onto the original image.

Anchor boxes of the chosen combinations and aspect ratios are generated with this point as the center point e.g. 9 anchor boxes per point. For such anchor boxes that exceed the boundaries of the image, they are removed and not considered in the model.

For each of these anchor boxes, the Intersection-Over-Union (IOU) is calculated with respect to all the ground-truth boxes in the image in order to determine if the anchor box contains a labelled object and its box.

If the IOU value exceeds a pre-determined threshold such as 0.7, this anchor box is labelled as ‘positive’ with respect to the compared ground-truth box and it’s coordinate offset (x,y, width and height) to the ground-truth box is calculated as a regression target. If the IOU value falls below a lower limit such as 0.3, it is classified as a ‘negative’ anchor box with respect to the ground-truth box. This is done in order to help the model learn to differentiate between background (non-labelled parts of the image) and the actual ‘positive’ boxes. For IOU values that fall between these two limits, they are labelled as ‘neutral’ and not

considered in training as there is too much ambiguity in the region to be useful in training.

Hence, in this step, every ground truth box is assigned potential anchor boxes that it can take reference from in terms of classification and location. Using this method, representations that map the extracted feature maps to translation of fixed anchor boxes in order to locate objects in the image.

API Design: The step is implemented in the data_generators class in the data_generators.calc_rpn() method. For every anchor box, the method iterates over all the ground truth bounding boxes annotated in the image, and calculates the IoU value as well as the bounding box regression targets.

```
# Initialise array variables to hold empty output objectives

# Target training labels for anchor boxes:
# y_is_box_valid: 1 if the anchor should be considered in training to classify if
# the anchor box contains an object or background
# y_rpn_valid: 1 if the anchor box overlap exceeds the threshold value and is classified as an object
# y_rpn_regr: 1 if the anchor box regression targets should be used
y_rpn_overlap = np.zeros((output_height, output_width, num_anchors))
y_is_box_valid = np.zeros((output_height, output_width, num_anchors))
y_rpn_regr = np.zeros((output_height, output_width, num_anchors * 4))

num_bboxes = len(img_data['bboxes'])

# For each annotated bounding box in the image, the number of valid anchor boxes to use for this bounding box,
# the best anchor box and its attributes are recorded
num_anchors_for_bbox = np.zeros(num_bboxes).astype(int)
best_anchor_for_bbox = -1 * np.ones((num_bboxes, 4)).astype(int)
best_iou_for_bbox = np.zeros(num_bboxes).astype(np.float32)
best_x_for_bbox = np.zeros((num_bboxes, 4)).astype(int)
best_dx_for_bbox = np.zeros((num_bboxes, 4)).astype(np.float32)
```

```
    if img_data['bboxes'][bbox_num]['class'] != 'bg':
        # Compares the IoU calculated for this anchor box to the current list of IoUs and
        # keeps track of the best anchor box and its attributes
        if curr_iou > best_iou_for_bbox[bbox_num]:
            best_anchor_for_bbox[bbox_num] = [jy, ix, anchor_ratio_idx, anchor_size_idx]
            best_iou_for_bbox[bbox_num] = curr_iou
            best_x_for_bbox[bbox_num, :] = [x1_anc, x2_anc, y1_anc, y2_anc]
            best_dx_for_bbox[bbox_num, :] = [tx, ty, tw, th]

        # Anchor box is set to positive if its IoU is more than the threshold value e.g. 0.7
        # The box is also added to a counter box for the number of potential anchors for this object box
        if curr_iou > C.rpn_max_overlap:
            bbox_type = 'pos'
            num_anchors_for_bbox[bbox_num] += 1

        # Regression targets are updated is this anchor box has the best IoU so far
            if curr_iou > best_iou_for_loc:
                best_iou_for_loc = curr_iou
                best_regr = (tx, ty, tw, th)

        # if the IOU is >0.3 and <0.7, it is ambiguous and not included in the objective
        if C.rpn_min_overlap < curr_iou < C.rpn_max_overlap:
            # gray zone between neg and pos
            if bbox_type != 'pos':
                bbox_type = 'neutral'

# The 3 target labels are updated for each box
if bbox_type == 'neg':
    y_is_box_valid[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] = 1
    y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] = 0
elif bbox_type == 'neutral':
    y_is_box_valid[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] = 0
    y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] = 0
elif bbox_type == 'pos':
    y_is_box_valid[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] = 1
    y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] = 1
    start = 4 * (anchor_ratio_idx + n_anchratios * anchor_size_idx)
    y_rpn_regr[jy, ix, start:start + 4] = best_regr
```

RPN Proposal Reduction: As there are many anchor boxes generated over an image, most of the anchor boxes may be tagged as ‘negative’ or as a background as the image will contain a much lesser number of labelled objects of interest.

Hence, the top number of 'negative' anchor boxes can be chosen instead of all the boxes to balance the proportion of training classes.

API Design: This is also included in the data_generators.calc_rpn() method:

```
# As the RPN also labels negative anchors boxes as valid boxes to train it to recognize background regions,
# there are much more negative than positive regions. Hence some of the negative regions will be excluded
# and the total number is reduced to 256 regions.

num_regions = 256

if len(pos_locs[0]) > num_regions / 2:
    val_locs = random.sample(range(len(pos_locs[0])), len(pos_locs[0]) - num_regions / 2)
    y_is_box_valid[0, pos_locs[0][val_locs], pos_locs[1][val_locs], pos_locs[2][val_locs]] = 0
    num_pos = num_regions / 2

if len(neg_locs[0]) + num_pos > num_regions:
    val_locs = random.sample(range(len(neg_locs[0])), len(neg_locs[0]) - num_pos)
    y_is_box_valid[0, neg_locs[0][val_locs], neg_locs[1][val_locs], neg_locs[2][val_locs]] = 0

y_rpn_cls = np.concatenate([y_is_box_valid, y_rpn_overlap], axis=1)
y_rpn_regr = np.concatenate([np.repeat(y_rpn_overlap, 4, axis=1), y_rpn_regr], axis=1)

return np.copy(y_rpn_cls), np.copy(y_rpn_regr)
```

From the reference code of rpn(), we can see the ResNet50 become the base layer for RPN to training. rpn_out_class layer to be trained to decide if those 9 anchor boxes having object or not. Rpn_out_regress layer will be trained to estimate the shifting of 2 pairs x,y values, e.g. top left and bottom right.

```
1725
1726    @staticmethod
1727    def rpn(base_layers, num_anchors):
1728        ## base_layers: Tensor("activation_39/Relu:0", shape=(?, ?, ?, 1024), dtype=float32)
1729        ## num_anchors: 9
1730
1731        x = Convolution2D(512, (3, 3), padding='same', activation='relu', kernel_initializer='normal',
1732                          name='rpn_conv1')(base_layers)
1733        ## x : Tensor("rpn_conv1/Relu:0", shape=(?, ?, ?, 512), dtype=float32)
1734
1735        x_class = Convolution2D(num_anchors, (1, 1), activation='sigmoid', kernel_initializer='uniform',
1736                                name='rpn_out_class')(x)
1737        ## x_class: Tensor("rpn_out_class/Sigmoid:0", shape=(?, ?, ?, 9), dtype=float32)
1738
1739        x_regr = Convolution2D(num_anchors * 4, (1, 1), activation='linear', kernel_initializer='zero',
1740                               name='rpn_out_regress')(x)
1741        ## x_regr: Tensor("rpn_out_regress/BiasAdd:0", shape=(?, ?, ?, 36), dtype=float32)
1742
1743
1744        return [x_class, x_regr, base_layers]
1745
```

Training Loss Functions: In order to guide the training of the model to recognize objects and the respective anchor box transformations, custom loss functions for each objective were created under the lossers class constructor. The lossers.rpn_loss_cls() method determines the binary cross entropy between the predicted values for each anchor boxes (whether the box contains an object or is background) and the ground truth labels calculated using data_generators.calc_rpn() and aims to minimize this loss.

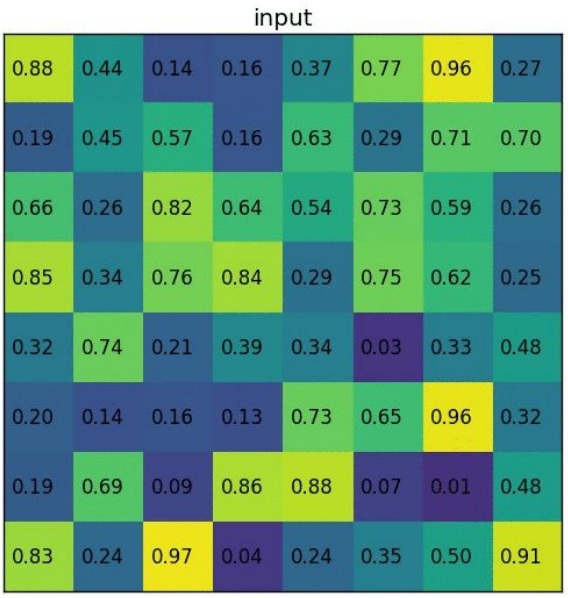
The lossers.rpn_loss_regr() method determines the regression loss between the predicted anchor box transformations and the calculated regression targets.

```
class lossers:

    @staticmethod
    def rpn_loss_regr(num_anchors):
        def rpn_loss_regr_fixed_num(y_true, y_pred):
            if image_dim_ordering(K) == 'th':
                x = y_true[:, 4 * num_anchors:, :, :] - y_pred
                x_abs = K.abs(x)
                x_bool = K.less_equal(x_abs, 1.0)
                return lambda_rpn_regr * K.sum(
                    y_true[:, :4 * num_anchors, :, :] * (
                            x_bool * (0.5 * x * x) + (1 - x_bool) * (x_abs - 0.5))) / K.sum(
                    epsilon + y_true[:, :4 * num_anchors, :, :])
            else:
                x = y_true[:, :, :, 4 * num_anchors:] - y_pred
                x_abs = K.abs(x)
                x_bool = K.cast(K.less_equal(x_abs, 1.0), tf.float32)

                return lambda_rpn_regr * K.sum(
                    y_true[:, :, :, :4 * num_anchors] * (
                            x_bool * (0.5 * x * x) + (1 - x_bool) * (x_abs - 0.5))) / K.sum(
                    epsilon + y_true[:, :, :, :4 * num_anchors])

        return rpn_loss_regr_fixed_num

    @staticmethod
    def rpn_loss_cls(num_anchors):
        def rpn_loss_cls_fixed_num(y_true, y_pred):
            if image_dim_ordering(K) == 'tf':
                return lambda_rpn_class * K.sum(
                    y_true[:, :, :, :num_anchors] * K.binary_crossentropy(y_pred[:, :, :, :],
                                                                          y_true[:, :, :,
                                                                          num_anchors:])) / K.sum(
                    epsilon + y_true[:, :, :, :num_anchors])
            else:
                return lambda_rpn_class * K.sum(
                    y_true[:, :num_anchors, :, :] * K.binary_crossentropy(y_pred[:, :, :, :],
                                                                          y_true[:,
                                                                          num_anchors:, :,
                                                                          :])) / K.sum(
                    epsilon + y_true[:, :num_anchors, :, :])

        return rpn_loss_cls_fixed_num
```

2.3. Region of Interest Pooling (ROI Pooling)

After estimated region proposals have been generated from the RPN layer, these regions have to be passed into a classifier in order to determine the class of the object. However, these regions are likely to be of different sizes according to the estimated size and aspect ratio of the bounding boxes. This presents a challenge in classification as classifiers such as a fully-connected neural network or support vector machine (SVM) requires a fixed input size.

In order to transform the varying proposed regions into fixed input sizes, the ROI pooling method introduced by Girschick et al is used. In ROI pooling a cropped region is first divided into a fixed, specified number of sub-regions. A pooling operation such as max or average pooling is performed in each divided sub-region in order to obtain a single input. Thus, a fixed number of input values can be obtained from any input image size which can then be used with a classifier.



Animated gif: https://cdn-sv1.deepsense.ai/wp-content/uploads/2017/02/roi_pooling-1.gif

Source: https://deepsense.ai/region-of-interest-pooling-explained/ [8]

The ROI pooling is a custom layer in tensorflow keras. More details of custom layer creation can be found at https://www.tensorflow.org/guide/keras/custom_layers_and_models

The input/output shape of ROI Pooling layer shown as below:

```
1762
1763        # base_layers: Tensor("activation_39/Relu:0", shape=(?, ?, ?, 1024), dtype=float32)
1764        # input_rois: Tensor("input_2:0", shape=(?, ?, 4), dtype=float32)
1765        # num_rois: 64
1766        out_roi_pool = RoiPoolingConv(pooling_regions, num_rois)([base_layers, input_rois])
1767        # out_roi_pool: Tensor("roi_pooling_conv/transpose:0", shape=(1, 64, 14, 14, 1024), dtype=float32)
1768
```

2.4. Classification Layer

Once fixed input values have been obtained from the ROI pooling layer, these inputs can be used with a classifier layer, such as a fully connected layer or SVM in order to determine the class of object in the predicted bounding box. In this stage, the classifier acts like a regular CNN performing an image classification task. At the same time, the loss and error of the coordinates of this predicted bounding box versus the actual ground-truth bounding box is also recorded. In our implementation, we pass the 2D output from the ROI pooling layer into a

ResNet convolution block and an average pooling layer before using it in a fully connected layer.

Training Loss Functions: In the classifier layer, the model is guided using the object classification loss and the bounding box loss targets, implemented with the lossers.class_loss_cls() and lossers.class_loss_regr() methods respectively.

```
@staticmethod
def class_loss_regr(num_classes):
    def class_loss_regr_fixed_num(y_true, y_pred):
        x = y_true[:, :, 4 * num_classes:] - y_pred
        x_abs = K.abs(x)
        x_bool = K.cast(K.less_equal(x_abs, 1.0), 'float32')
        return lambda_cls_regr * K.sum(
            y_true[:, :, :4 * num_classes] * (x_bool * (0.5 * x * x) + (1 - x_bool) * (x_abs - 0.5))) / K.sum(
            epsilon + y_true[:, :, :4 * num_classes])

    return class_loss_regr_fixed_num

@staticmethod
def class_loss_cls(y_true, y_pred):
    return lambda_cls_class * K.mean(categorical_crossentropy(y_true[0, :, :], y_pred[0, :, :]))
```

2.5. Network Training

The Faster R-CNN model is first instantiated as an object using the FasterRCNNModel, which includes train() and test() methods for training the model and testing on images separately.

In the training phase, the images are loaded and transformed using a custom data generator that generates the ground truth anchor boxes and their attributes. The resized images and their respective anchor box labels are first used to train the RPN sub-model, which is then used to propose potential regions on the image in the form of anchor boxes and their bounding box offsets. These regions are converted into a useable ROI format and their IoU values are calculated. A sample of positive and negative (background) ROIs are selected as training examples for the classifier sub-model in order to determine the object class as well as the refined location of the bounding boxes.

```
X, Y, img_data = next(data_gen_train)
# print(X.shape, Y[0].shape)

loss_rpn = model_rpn.train_on_batch(X, Y)

P_rpn = model_rpn.predict_on_batch(X)

roi_helpers = roi_helper()
R = roi_helpers.rpn_to_roi(P_rpn[0], P_rpn[1], C, image_dim_ordering(K), use_regr=True,
                           overlap_thresh=0.7, max_boxes=300)
# note: calc_iou converts from (x1,y1,x2,y2) to (x,y,w,h) format
X2, Y1, Y2, IouS = roi_helpers.calc_iou(R, img_data, C, class_mapping)
```

```
loss_class = model_classifier.train_on_batch([X, X2[:, sel_samples, :]],
                                             [Y1[:, sel_samples, :], Y2[:, sel_samples, :]])

losses[iter_num, 0] = loss_rpn[1]
losses[iter_num, 1] = loss_rpn[2]

losses[iter_num, 2] = loss_class[1]
losses[iter_num, 3] = loss_class[2]
losses[iter_num, 4] = loss_class[3]
```

3. Implementation, Train, and Test Strategy

3.1. Search and Verification of Reference Implementation

In order to further understand Faster R-CNN implementation methodology, we retrieved two Github repositories to examine the design of the Faster R-CNN methods using the Keras API. Due to tensorflow version 1.13.1, we backport is_keras_tensor() and image_dim_ordering() from TensorFlow 1.14 source code. In addition, we also took reference from the research papers by the original authors of the method as well as online articles.

Github Repositories:

1. *RockyXu66*: This repository consisted of a sample implementation of Faster R-CNN. However upon using the code to train a model based on the stated datasets, the model did not provide good results.

   Link:

   https://github.com/RockyXu66/Faster_RCNN_for_Open_Images_Dataset_Keras

2. *kbardool*: This repository was trained with the VOCDevKit introduced in 2012. The sample code ran successfully and had good results when the model was tested on the testing images. In this project, we used this repository as reference code for training our model.

   Link:

   https://github.com/kbardool/keras-frcnn

3.2. Annotation and Integration

OpenCV's CVAT https://github.com/opencv/cvat was used for annotating our dataset. The main advantage of CVAT over other annotation tools is that it is able to handle video file and we can perform interpolation annotation on the video file. A demo of the interpolation process can be found here: https://www.youtube.com/watch?v=U3MYDhESHo4&feature=youtu.be

The output of the video annotation is in an XML format, for which we developed a parser to convert it into a useable structure that is compatible with the Faster R-CNN implementation.

We first annotated 1 video and ran the cvatParser for simple csv format for object detection training in kbardool reference implementation, in order to verify the end-to-end process of the code. After 3 hours of training, we got 1 object detection on training image.

3.3. Experiment with our own naive implementation without TimeDistributed() method, class SimpleResnet

Understand that changing network structure enabled us to train model for other applications, e.g. human posture detector, we try to modify reference implementation for learning. As the reference implementation is quite modular, we created SimpleResNet out of ResNet class in a more naive implementation. We discovered that TimeDistributed() constructs appear after RoiPoolingConv() layer.

```
1765
1766    # base_layers: Tensor("activation_39/Relu:0", shape=(?, ?, ?, 1024), dtype=float32)
1767    # input_rois: Tensor("input_2:0", shape=(?, ?, 4), dtype=float32)
1768    # num_rois: 64
1769    out_roi_pool = RoiPoolingConv(pooling_regions, num_rois)([base_layers, input_rois])
1770    # out_roi_pool: Tensor("roi_pooling_conv/transpose:0", shape=(1, 64, 14, 14, 1024), dtype=float32)
1771
1772    out = SimpleResNet.classifier_layers(out_roi_pool, input_shape=input_shape, trainable=True)
1773    ## out: Tensor("avg_pool/Reshape_1:0", shape=(1, 64, 1, 1, 2048), dtype=float32)
1774
1775    out = TimeDistributed(Flatten())(out)
1776    ## out: Tensor("time_distributed/Reshape_2:0", shape=(1, 64, 2048), dtype=float32)
1777
1778    # out = Flatten()(out)
1779    ## out: Tensor("flatten/Reshape:0", shape=(1, 131072), dtype=float32)
1780    ## due to non TimeDistributed(Flatten())
1781    ## ValueError: `TimeDistributed` Layer should be passed an `input_shape ` with at least 3 dimensions, received: [1, 131072]
1782
1783    ## ValueError: Index out of range using input dim 2; input has only 2 dims for 'loss_1/dense_class_21_loss/strided_slice' (op: 'StridedSlice') w
1784
```

Typically, Conv2D() accepts 4 dimensions tensor. E.g. base_layers. As RoiPoolingConv() layer output a 5 dimensional tensor, Conv2D() throwing error of

"ValueError: Input 0 of layer res5a_branch2a is incompatible with the layer: expected ndim=4, found ndim=5. Full shape received: [1, 64, 14, 14, 1024]"

TimeDistributed() is effectively unwrapping 5 dimensions tensor into 4 dimensions one, e.g. from [1,64,14,14,1024] into 64 x [1,14,14,1024] and passing each [1,14,14,1024] to Conv2D() as per following code snippet in conv_block_td().

```
1364
1365        x = TimeDistributed(
1366            Convolution2D(nb_filter1, (1, 1), strides=strides, trainable=trainable, kernel_initializer='normal'),
1367            input_shape=input_shape, name=conv_name_base + '2a')(input_tensor)
1368        x = TimeDistributed(FixedBatchNormalization(axis=bn_axis), name=bn_name_base + '2a')(x)
1369        x = Activation('relu')(x)
1370
```

More info about TimeDistributed() can be found at https://www.tensorflow.org/api_docs/python/tf/keras/layers/TimeDistributed

3.4. First round of Training

After the verification process, we annotated another 9 videos and processed all annotations for training the model. During the training process, the loss values does going down. However after 8 hours of training, we got ZERO detection from all training images even with 0.1 threshold. Based on past experience about deep neural network classification, neural network easily overfit to training data. We reviewed the annotations of the images and concluded that annotation with small size regions might be the source of the issue.

3.5. Second round of Training

In order to address the issue of object detections, we changed anchor box scales from [128, 256, 512] to [32, 64,128,256] and filtered away annotation boxes that were smaller than 32x32, and perform training for another 3 hours. The loss values does going down during training, yet we still got ZERO detection from all training images again, on threshold of 0.1. Based on our previous experience in the previous iteration, we believe that 3 hours training shall gives us some detection, yet we got ZERO.

3.6. Third round of training

Based on the training result from the second iteration, we did further research on methods to address this issue of dealing with small objects such as changing the anchor size [9] and through online discussions

(https://www.researchgate.net/post/How_to_improve_the_quality_of_generating_region_proposals_in_Faster-Rcnn_that_can_not_deal_with_small_object). From these resources, we concluded that small objects require further network customization. We decided to filter away training data smaller than 64x64, and change anchor box scale from [32, 64,128,256] to [64,128,256, 512], which allowed the model to detect objects after a certain number of training iterations. We decided to let the training go further, and monitor loss changes, performs simple testing to see how detection performs (this requires another GPU card as the model is 6.5GB in GPU memory).

4. Results and Discussion

Following is the testing result after 95 epochs and 20.6 hours of training:

Total Frames, 22 seconds for 28 fps	Positive Detection	False Detection
616	bg 405, detection: 168	43

Based on (405+168)/616, the accuracy from all frames is about 93.02%. Non background detection rate is about 168/(168+43) = 79.62%.

By metric of mAP,

Object Class	mAP
HUMP	0.371
LINE_ZIGZAG	0.834
W_AHEAD	0.876
W_HUMP	0.673
HTL_GREEN, VTL_GREEN, LINE_DOUBLE_YELLOW, W_X-ING	nan (background truth, but detected as class on the left) Model predicted such objects in images, but were not present in image (ground-truth is background)

The detection video can be found at https://github.com/XiaoyanYang2008/ISS-VSE-ObjectDetection/blob/master/data/test/1-test-video.avi-detected-loss-0.285.mp4?raw=true

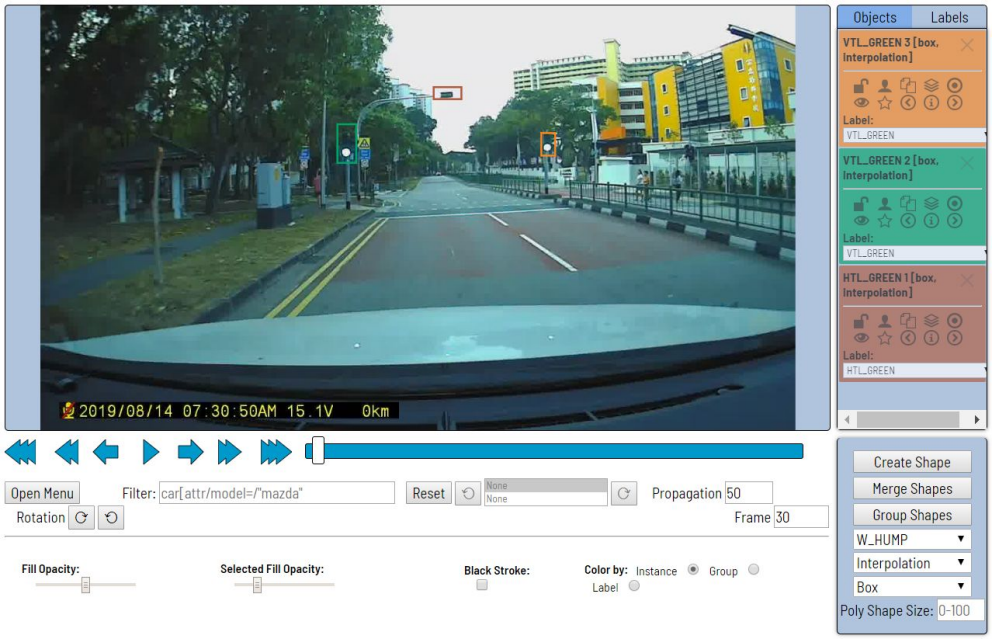
We noticed following false detections and we propose corrections for our future training:

Truth	False detection	Proposed correction
SLOW wording	W_HUMP	Annotate new class, W_SLOW
Reversed SLOW wording	W_AHEAD	Annotate as bg
Reversed ARROW	W_AHEAD	Annotate as bg

5. Dataset preparation

5.1. Annotation for classes

In this implementation, we used car dashcam videos as our dataset source and extracted frames from these videos and annotated the object labels using Computer Vision Annotation Tool (CVAT). CVAT is an open source tool for annotating digital images and videos. The main function of the application is to provide users with convenient annotation instruments. CVAT is web based, easy to deploy in Ubuntu and open source. The following figure is a screenshot during our annotation.



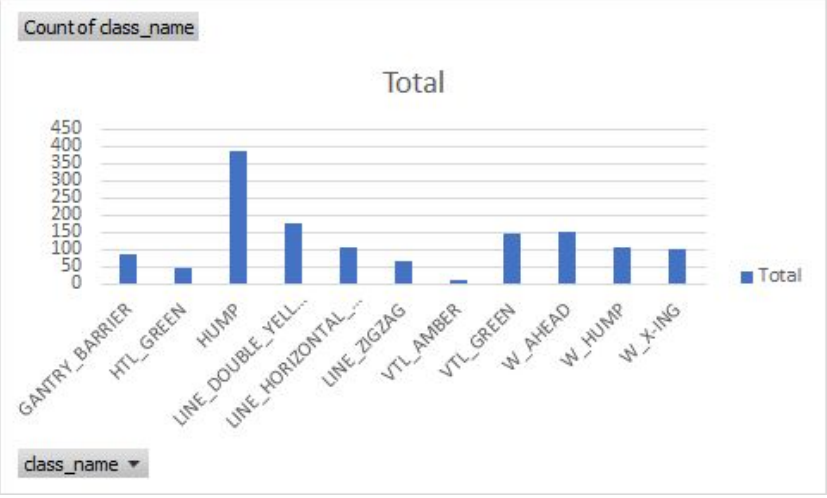
Source: https://software.intel.com/en-us/articles/computer-vision-annotation-tool-a-universal-approach-to-data-annotation [7]

5.2. Object Detection Classes

In our dataset, we have below classes for our object detection:

VTL_GREEN, VTL_RED, VTL_AMBER, HTL_GREEN, HTL_RED, HTL_AMBER, LINE_DOUBLE_YELLOW, LINE_ZIGZAG, LINE_HORIZONTAL_PEDESTRIAN, GANTRY_BARRIER, HUMP, W_HUMP, W_AHEAD, W_X-ING.

The data set is not balance due to different areas have different road signs, also some classes are very hard to capture (eg. VTL_AMBER, HTL_AMBER) when driving because yellow traffic light blinking very fast.



Row Labels	Count of class_name	%
GANTRY_BARRIER	87	6.26
HTL_GREEN	47	3.38
HUMP	388	27.91
LINE_DOUBLE_YELLOW	177	12.73
LINE_HORIZONTAL_PEDESTRIAN	108	7.77
LINE_ZIGZAG	65	4.68
VTL_AMBER	10	0.72
VTL_GREEN	148	10.65
W_AHEAD	153	11.01
W_HUMP	105	7.55
W_X-ING	102	7.34
Grand Total	**1390**	100.00

5.3. No Data Augmentation

Data augmentation is a technique to increase potential truth data, which may be rare or difficult to obtain in reality. As the objective of our model is to detect and recognize road markings and objects from the view of an autonomous vehicle, we did not perform data augmentation on our dataset. For road markings which are in the form of words such as W_HUMP, and W_X-ING, horizontal flipping was not used as such orientations of the augmented object image are not encountered from the vehicle's viewpoint. In addition, the inclusion of such augmented images may introduce more noise in the model and hinder the model's ability to recognize images. For traffic lights, augmentation was also not used as the orientation and position of the lighting area relative to the housing will be an important characteristic to differentiate between different lights. As for augmentation of objects for different scales and aspect ratios, this was also addressed by images of the same objects at different viewpoints when the vehicle is moving, providing such relevant examples for training without additional augmentation.

6. Challenges and Limitations

6.1. Low Detection of Small Objects

During the first attempt at training the verified model, the model failed to detect any objects in the training images despite being trained over a full set of training epochs. We examined different parameters that were potentially the cause of the model's poor performance, such as the anchor box scale, aspect ratio, and IoU thresholds for each anchor box relative to the ground truth boxes. Lowering

the IoU threshold did not improve the performance and still did not result in the model detecting any objects in the images.

However, we recognized that some of the annotated objects had a relatively small bounding box size e.g. less than 64px for each side. This may have affected the IoU calculations, as they will have a much smaller intersection area with the large anchor boxes compared to the total area size of both boxes and leading to smaller IoU values. The appropriate activations of such valid anchor boxes may have been deactivated, leading to poor performance when training the model with these targets.

In order to address this issue, we reduced the anchor box scales from [128,256,512] to [64,128,256,512] in order to accommodate smaller annotated objects in the training images and also excluded such objects smaller than 64px by 64px. The model was re-trained using the updated parameters and resulted in a significant improvement in the model's performance.

From this use case, we experienced the challenge of dealing with training a model to detect small objects in an image and also the importance of the anchor box parameters for the Faster R-CNN method.

6.2. Decreasing Loss Function Without Object Detection

Another challenge that we faced was the interpretation of the loss function metrics to determine the training performance for the model. The loss functions for RPN network and classifier measured the performance of the model's ability to detect objects and their bounding box locations and to classify the object within the bounding box respectively. However, the training labels provided to the RPN network consisted of object (foreground) and background labels in order to train the network to differentiate between these two classifications. Hence, although the training loss based on the targets were decreasing, it could have indicated an improvement in the model's ability to recognize a background from the anchor box and not necessarily of the objects and their boxes. In our implementation of the model, we addressed this by printing and monitoring the prediction labels in the console during the training phase in order to ensure that the model is able to predict objects in the image as well,

6.3. Slow Model Training Speed

The implemented Faster R-CNN model also required a long training time of at least 8 hours, which limited our ability to introduce changes to the model and re-train the model. This was partly due to the nature of the Faster R-CNN architecture, which generates multiple anchor boxes along with their attributes and stores them in the GPU memory during training. This step limits the number of images that can be used for training in one batch. In our implementation, we were limited to a batch size of 1 and used a stochastic gradient descent approach, which resulted in a much longer processing time for each training epoch.

7. Conclusion

In this project, we explored the R-CNN family of object detection techniques and examined the working details of the Faster R-CNN model in particular. We implemented the model architecture and customized for use as a road object detection model. Using this technique, we trained an object detection model that was able to detect warning signs, traffic lights and road markings, which are potential features that would be useful for autonomous driving use cases in Singapore. Through this project, we have gained knowledge on the process of collecting data for faster RCNN network architect, and annotating objects using open source tools such as CVAT, which is an important and beneficial skill for future related work. We have also gained experience in building and customizing an object detection model architecture and training it to address a use case.

References:

[1] R. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2014.

[2] R. Girshick, “Fast R-CNN,” in *Proceedings of the IEEE International Conference on Computer Vision*, 2015.

[3] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” *IEEE Trans. Pattern Anal. Mach. Intell.*, 2017.

[4] K. He, X. Zhang, S. Ren, and J. Sun, “Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition,” *IEEE Trans. Pattern Anal. Mach. Intell.*, 2015.

[5] S. Pulkit, “A Practical Implementation of the Faster R-CNN Algorithm for Object Detection (Part 2 – with Python codes),” 2018. [Online]. Available: https://www.analyticsvidhya.com/blog/2018/11/implementation-faster-r-cnn-python-object-detection/.

[6] J. Rey, “Faster R-CNN: Down the rabbit hole of modern object detection,” 2018. [Online]. Available: https://tryolabs.com/blog/2018/01/18/faster-r-cnn-down-the-rabbit-hole-of-modern-object-detection/.

[7] B. Sekachev, N. M., and A. Z., “Computer Vision Annotation Tool: A Universal Approach to Data Annotation,” 2019. [Online]. Available: https://software.intel.com/en-us/articles/computer-vision-annotation-tool-a-universal-approach-to-data-annotation.

[8] T. Grel, “Region of interest pooling explained,” 2017. [Online]. Available: https://deepsense.ai/region-of-interest-pooling-explained/.

[9] Y. Ren, C. Zhu, and S. Xiao, “Small object detection in optical remote sensing images via modified Faster R-CNN,” *Applied Sciences (Switzerland)*. 2018.