Autonomous_driving_application_Car_detection

May 31, 2021

1 Autonomous Driving - Car Detection

Welcome to the Week 3 programming assignment! In this notebook, you'll implement object detection using the very powerful YOLO model. Many of the ideas in this notebook are described in the two YOLO papers: Redmon et al., 2016 and Redmon and Farhadi, 2016.

By the end of this assignment, you'll be able to:

- Detect objects in a car detection dataset
- Implement non-max suppression to increase accuracy
- Implement intersection over union
- Handle bounding boxes, a type of image annotation popular in deep learning

1.1 Table of Content

Packages

Run the following cell to load the packages and dependencies that will come in handy as you build the object detector!

[1]:
```
import argparse
import os
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow
import scipy.io
import scipy.misc
import numpy as np
import pandas as pd
import PIL
from PIL import ImageFont, ImageDraw, Image
import tensorflow as tf
from tensorflow.python.framework.ops import EagerTensor

from tensorflow.keras.models import load_model
from yad2k.models.keras_yolo import yolo_head
from yad2k.utils.utils import draw_boxes, get_colors_for_classes, scale_boxes,␣
↪read_classes, read_anchors, preprocess_image

%matplotlib inline
```

1 - Problem Statement

You are working on a self-driving car. Go you! As a critical component of this project, you'd like to first build a car detection system. To collect data, you've mounted a camera to the hood (meaning the front) of the car, which takes pictures of the road ahead every few seconds as you drive around.

Pictures taken from a car-mounted camera while driving around Silicon Valley. Dataset provided by drive.ai.

You've gathered all these images into a folder and labelled them by drawing bounding boxes around every car you found. Here's an example of what your bounding boxes look like:

Figure 1: Definition of a box

If there are 80 classes you want the object detector to recognize, you can represent the class label c either as an integer from 1 to 80, or as an 80-dimensional vector (with 80 numbers) one component of which is 1, and the rest of which are 0. The video lectures used the latter representation; in this notebook, you'll use both representations, depending on which is more convenient for a particular step.

In this exercise, you'll discover how YOLO ("You Only Look Once") performs object detection, and then apply it to car detection. Because the YOLO model is very computationally expensive to train, the pre-trained weights are already loaded for you to use.

2 - YOLO

"You Only Look Once" (YOLO) is a popular algorithm because it achieves high accuracy while also being able to run in real time. This algorithm "only looks once" at the image in the sense

that it requires only one forward propagation pass through the network to make predictions. After non-max suppression, it then outputs recognized objects together with the bounding boxes.

2.1 - Model Details

Inputs and outputs

- The **input** is a batch of images, and each image has the shape (m, 608, 608, 3)
- The **output** is a list of bounding boxes along with the recognized classes. Each bounding box is represented by 6 numbers $(p_c, b_x, b_y, b_h, b_w, c)$ as explained above. If you expand c into an 80-dimensional vector, each bounding box is then represented by 85 numbers.

Anchor Boxes

- Anchor boxes are chosen by exploring the training data to choose reasonable height/width ratios that represent the different classes. For this assignment, 5 anchor boxes were chosen for you (to cover the 80 classes), and stored in the file './model_data/yolo_anchors.txt'
- The dimension for anchor boxes is the second to last dimension in the encoding: $(m, n_H, n_W, anchors, classes)$.
- The YOLO architecture is: IMAGE (m, 608, 608, 3) -> DEEP CNN -> ENCODING (m, 19, 19, 5, 85).

Encoding Let's look in greater detail at what this encoding represents.

Figure 2 : Encoding architecture for YOLO

If the center/midpoint of an object falls into a grid cell, that grid cell is responsible for detecting that object.

Since you're using 5 anchor boxes, each of the 19 x19 cells thus encodes information about 5 boxes. Anchor boxes are defined only by their width and height.

For simplicity, you'll flatten the last two dimensions of the shape (19, 19, 5, 85) encoding, so the output of the Deep CNN is (19, 19, 425).

Figure 3 : Flattening the last two last dimensions

Class score Now, for each box (of each cell) you'll compute the following element-wise product and extract a probability that the box contains a certain class.
The class score is $score_{c,i} = p_c \times c_i$: the probability that there is an object p_c times the probability that the object is a certain class c_i.

Figure 4: Find the class detected by each box

Example of figure 4

- In figure 4, let's say for box 1 (cell 1), the probability that an object exists is $p_1 = 0.60$. So there's a 60% chance that an object exists in box 1 (cell 1).

- The probability that the object is the class "category 3 (a car)" is $c_3 = 0.73$.
- The score for box 1 and for category "3" is $score_{1,3} = 0.60 \times 0.73 = 0.44$.
- Let's say you calculate the score for all 80 classes in box 1, and find that the score for the car class (class 3) is the maximum. So you'll assign the score 0.44 and class "3" to this box "1".

Visualizing classes Here's one way to visualize what YOLO is predicting on an image:

- For each of the 19x19 grid cells, find the maximum of the probability scores (taking a max across the 80 classes, one maximum for each of the 5 anchor boxes).
- Color that grid cell according to what object that grid cell considers the most likely.

Doing this results in this picture:

Figure 5: Each one of the 19x19 grid cells is colored according to which class has the largest predicted probability in that cell.

Note that this visualization isn't a core part of the YOLO algorithm itself for making predictions; it's just a nice way of visualizing an intermediate result of the algorithm.

Visualizing bounding boxes Another way to visualize YOLO's output is to plot the bounding boxes that it outputs. Doing that results in a visualization like this:

Figure 6: Each cell gives you 5 boxes. In total, the model predicts: 19x19x5 = 1805 boxes just by looking once at the image (one forward pass through the network)! Different colors denote different classes.

Non-Max suppression In the figure above, the only boxes plotted are ones for which the model had assigned a high probability, but this is still too many boxes. You'd like to reduce the algorithm's output to a much smaller number of detected objects.

To do so, you'll use **non-max suppression**. Specifically, you'll carry out these steps: - Get rid of boxes with a low score. Meaning, the box is not very confident about detecting a class, either due to the low probability of any object, or low probability of this particular class. - Select only one box when several boxes overlap with each other and detect the same object.

[ ]:

2.2 - Filtering with a Threshold on Class Scores

You're going to first apply a filter by thresholding, meaning you'll get rid of any box for which the class "score" is less than a chosen threshold.

The model gives you a total of 19x19x5x85 numbers, with each box described by 85 numbers. It's convenient to rearrange the (19,19,5,85) (or (19,19,425)) dimensional tensor into the following variables:
- `box_confidence`: tensor of shape $(19, 19, 5, 1)$ containing p_c (confidence probability that there's some object) for each of the 5 boxes predicted in each of the 19x19 cells. - `boxes`: tensor of shape $(19, 19, 5, 4)$ containing the midpoint and dimensions (b_x, b_y, b_h, b_w) for each of the 5 boxes

in each cell. - `box_class_probs`: tensor of shape $(19, 19, 5, 80)$ containing the "class probabilities" $(c_1, c_2, ...c_{80})$ for each of the 80 classes for each of the 5 boxes per cell.

Exercise 1 - yolo_filter_boxes

Implement `yolo_filter_boxes()`. 1. Compute box scores by doing the elementwise product as described in Figure 4 ($p \times c$).
The following code may help you choose the right operator:

```
a = np.random.randn(19, 19, 5, 1)
b = np.random.randn(19, 19, 5, 80)
c = a * b # shape of c will be (19, 19, 5, 80)
```

This is an example of **broadcasting** (multiplying vectors of different sizes).

2. For each box, find:

   - the index of the class with the maximum box score
   - the corresponding box score

   Useful References * tf.math.argmax * tf.math.reduce_max

   Helpful Hints * For the `axis` parameter of `argmax` and `reduce_max`, if you want to select the **last** axis, one way to do so is to set `axis=-1`. This is similar to Python array indexing, where you can select the last position of an array using `arrayname[-1]`. * Applying `reduce_max` normally collapses the axis for which the maximum is applied. `keepdims=False` is the default option, and allows that dimension to be removed. You don't need to keep the last dimension after applying the maximum here.

3. Create a mask by using a threshold. As a reminder: `([0.9, 0.3, 0.4, 0.5, 0.1] < 0.4)` returns: `[False, True, False, False, True]`. The mask should be `True` for the boxes you want to keep.

4. Use TensorFlow to apply the mask to `box_class_scores`, `boxes` and `box_classes` to filter out the boxes you don't want. You should be left with just the subset of boxes you want to keep.

   One more useful reference:

   - tf.boolean mask

   And one more helpful hint: :)

   - For the `tf.boolean_mask`, you can keep the default `axis=None`.

[29]:
```
def yolo_filter_boxes(boxes, box_confidence, box_class_probs, threshold = .6):
    """Filters YOLO boxes by thresholding on object and class confidence.

    Arguments:
        boxes -- tensor of shape (19, 19, 5, 4)
        box_confidence -- tensor of shape (19, 19, 5, 1)
        box_class_probs -- tensor of shape (19, 19, 5, 80)
        threshold -- real value, if [ highest class probability score <␣
 ↪threshold],
```

```
                    then get rid of the corresponding box

    Returns:
        scores -- tensor of shape (None,), containing the class probability␣
↪score for selected boxes
        boxes -- tensor of shape (None, 4), containing (b_x, b_y, b_h, b_w)␣
↪coordinates of selected boxes
        classes -- tensor of shape (None,), containing the index of the class␣
↪detected by the selected boxes

    Note: "None" is here because you don't know the exact number of selected␣
↪boxes, as it depends on the threshold.
    For example, the actual output size of scores would be (10,) if there are␣
↪10 boxes.
    """

    x = 10
    y = tf.constant(100)


    # Step 1: Compute box scores
    box_scores = box_confidence * box_class_probs
    # Step 2: Find the box_classes using the max box_scores, keep track of the␣
↪corresponding score
    box_classes = tf.math.argmax(box_scores, axis = -1)
    box_class_scores = tf.math.reduce_max(box_scores, axis = -1, keepdims =␣
↪False)

    # Step 3: Create a filtering mask based on "box_class_scores" by using␣
↪"threshold". The mask should have the
    # same dimension as box_class_scores, and be True for the boxes you want to␣
↪keep (with probability >= threshold)
    filtering_mask = (box_class_scores >= threshold)

    # Step 4: Apply the mask to box_class_scores, boxes and box_classes
    scores = tf.boolean_mask(box_class_scores, filtering_mask)
    boxes = tf.boolean_mask(boxes, filtering_mask)
    classes = tf.boolean_mask(box_classes, filtering_mask)


    return scores, boxes, classes
```

[30]:
```
tf.random.set_seed(10)
box_confidence = tf.random.normal([19, 19, 5, 1], mean=1, stddev=4, seed = 1)
boxes = tf.random.normal([19, 19, 5, 4], mean=1, stddev=4, seed = 1)
box_class_probs = tf.random.normal([19, 19, 5, 80], mean=1, stddev=4, seed = 1)
```

```
scores, boxes, classes = yolo_filter_boxes(boxes, box_confidence,␣
 ↪box_class_probs, threshold = 0.5)
print("scores[2] = " + str(scores[2].numpy()))
print("boxes[2] = " + str(boxes[2].numpy()))
print("classes[2] = " + str(classes[2].numpy()))
print("scores.shape = " + str(scores.shape))
print("boxes.shape = " + str(boxes.shape))
print("classes.shape = " + str(classes.shape))

assert type(scores) == EagerTensor, "Use tensorflow functions"
assert type(boxes) == EagerTensor, "Use tensorflow functions"
assert type(classes) == EagerTensor, "Use tensorflow functions"

assert scores.shape == (1789,), "Wrong shape in scores"
assert boxes.shape == (1789, 4), "Wrong shape in boxes"
assert classes.shape == (1789,), "Wrong shape in classes"

assert np.isclose(scores[2].numpy(), 9.270486), "Values are wrong on scores"
assert np.allclose(boxes[2].numpy(), [4.6399336, 3.2303846, 4.431282, -2.
 ↪202031]), "Values are wrong on boxes"
assert classes[2].numpy() == 8, "Values are wrong on classes"

print("\033[92m All tests passed!")
```

```
scores[2] = 9.270486
boxes[2] = [ 4.6399336  3.2303846  4.431282  -2.202031 ]
classes[2] = 8
scores.shape = (1789,)
boxes.shape = (1789, 4)
classes.shape = (1789,)
 All tests passed!
```

Expected Output:

scores[2]

9.270486

boxes[2]

[4.6399336 3.2303846 4.431282 -2.202031]

classes[2]

8

```
<tr>
    <td>
        <b>scores.shape</b>
    </td>
    <td>
```

```
       (1789,)
    </td>
</tr>
<tr>
    <td>
        <b>boxes.shape</b>
    </td>
    <td>
       (1789, 4)
    </td>
</tr>
<tr>
    <td>
        <b>classes.shape</b>
    </td>
    <td>
       (1789,)
    </td>
</tr>
```

Note In the test for `yolo_filter_boxes`, you're using random numbers to test the function. In real data, the `box_class_probs` would contain non-zero values between 0 and 1 for the probabilities. The box coordinates in `boxes` would also be chosen so that lengths and heights are non-negative.

2.3 - Non-max Suppression

Even after filtering by thresholding over the class scores, you still end up with a lot of overlapping boxes. A second filter for selecting the right boxes is called non-maximum suppression (NMS).

Figure 7 : In this example, the model has predicted 3 cars, but it's actually 3 predictions of the same car. Running non-max suppression (NMS) will select only the most accurate (highest probability) of the 3 boxes.

Non-max suppression uses the very important function called **"Intersection over Union"**, or IoU.

Figure 8 : Definition of "Intersection over Union".

Exercise 2 - iou

Implement `iou()`

Some hints: - This code uses the convention that (0,0) is the top-left corner of an image, (1,0) is the upper-right corner, and (1,1) is the lower-right corner. In other words, the (0,0) origin starts at the top left corner of the image. As x increases, you move to the right. As y increases, you move down. - For this exercise, a box is defined using its two corners: upper left (x_1, y_1) and lower right (x_2, y_2), instead of using the midpoint, height and width. This makes it a bit easier to calculate the intersection. - To calculate the area of a rectangle, multiply its height $(y_2 - y_1)$ by its width $(x_2 - x_1)$. Since (x_1, y_1) is the top left and x_2, y_2 are the bottom right, these differences should be non-negative. - To find the **intersection** of the two boxes (xi_1, yi_1, xi_2, yi_2): - Feel free to draw some examples on paper to clarify this conceptually. - The top left corner of the intersection (xi_1, yi_1) is found by comparing the top left corners (x_1, y_1) of the two boxes and finding a vertex

that has an x-coordinate that is closer to the right, and y-coordinate that is closer to the bottom. - The bottom right corner of the intersection (xi_2, yi_2) is found by comparing the bottom right corners (x_2, y_2) of the two boxes and finding a vertex whose x-coordinate is closer to the left, and the y-coordinate that is closer to the top. - The two boxes **may have no intersection**. You can detect this if the intersection coordinates you calculate end up being the top right and/or bottom left corners of an intersection box. Another way to think of this is if you calculate the height $(y_2 - y_1)$ or width $(x_2 - x_1)$ and find that at least one of these lengths is negative, then there is no intersection (intersection area is zero).
- The two boxes may intersect at the **edges or vertices**, in which case the intersection area is still zero. This happens when either the height or width (or both) of the calculated intersection is zero.

Additional Hints

- `xi1` = **max**imum of the x1 coordinates of the two boxes
- `yi1` = **max**imum of the y1 coordinates of the two boxes
- `xi2` = **min**imum of the x2 coordinates of the two boxes
- `yi2` = **min**imum of the y2 coordinates of the two boxes
- `inter_area` = You can use `max(height, 0)` and `max(width, 0)`

[38]:
```
def iou(box1, box2):
    """Implement the intersection over union (IoU) between box1 and box2

    Arguments:
    box1 -- first box, list object with coordinates (box1_x1, box1_y1, box1_x2,␣
↪box_1_y2)
    box2 -- second box, list object with coordinates (box2_x1, box2_y1,␣
↪box2_x2, box2_y2)
    """


    (box1_x1, box1_y1, box1_x2, box1_y2) = box1
    (box2_x1, box2_y1, box2_x2, box2_y2) = box2


    # Calculate the (yi1, xi1, yi2, xi2) coordinates of the intersection of␣
↪box1 and box2. Calculate its Area.
    xi1 = np.maximum(box1[0], box2[0])
    yi1 = np.maximum(box1[1], box2[1])
    xi2 = np.minimum(box1[2], box2[2])
    yi2 = np.minimum(box1[3], box2[3])
    inter_width = xi2-xi1
    inter_height = yi2-yi1
    inter_area = max(inter_width, 0) * max(inter_height, 0)

    # Calculate the Union area by using Formula: Union(A,B) = A + B - Inter(A,B)
    box1_area = (box1[2]-box1[0])*(box1[3]-box1[1])
    box2_area = (box2[2]-box2[0])*(box2[3]-box2[1])
    union_area = box1_area + box2_area - inter_area
```

```
    # compute the IoU
    iou = float(inter_area)/float(union_area)
    print(iou)

    return iou
```

[39]:
```
## Test case 1: boxes intersect
box1 = (2, 1, 4, 3)
box2 = (1, 2, 3, 4)

print("iou for intersecting boxes = " + str(iou(box1, box2)))
assert iou(box1, box2) < 1, "The intersection area must be always smaller or␣
 ↪equal than the union area."

## Test case 2: boxes do not intersect
box1 = (1,2,3,4)
box2 = (5,6,7,8)
print("iou for non-intersecting boxes = " + str(iou(box1,box2)))
assert iou(box1, box2) == 0, "Intersection must be 0"

## Test case 3: boxes intersect at vertices only
box1 = (1,1,2,2)
box2 = (2,2,3,3)
print("iou for boxes that only touch at vertices = " + str(iou(box1,box2)))
assert iou(box1, box2) == 0, "Intersection at vertices must be 0"

## Test case 4: boxes intersect at edge only
box1 = (1,1,3,3)
box2 = (2,3,3,4)
print("iou for boxes that only touch at edges = " + str(iou(box1,box2)))
assert iou(box1, box2) == 0, "Intersection at edges must be 0"

print("\033[92m All tests passed!")
```

```
0.14285714285714285
iou for intersecting boxes = 0.14285714285714285
0.14285714285714285
0.0
iou for non-intersecting boxes = 0.0
0.0
0.0
iou for boxes that only touch at vertices = 0.0
0.0
0.0
iou for boxes that only touch at edges = 0.0
0.0
```

```
All tests passed!
```

Expected Output:

```
iou for intersecting boxes = 0.14285714285714285
iou for non-intersecting boxes = 0.0
iou for boxes that only touch at vertices = 0.0
iou for boxes that only touch at edges = 0.0
```

2.4 - YOLO Non-max Suppression

You are now ready to implement non-max suppression. The key steps are: 1. Select the box that has the highest score. 2. Compute the overlap of this box with all other boxes, and remove boxes that overlap significantly (iou >= `iou_threshold`). 3. Go back to step 1 and iterate until there are no more boxes with a lower score than the currently selected box.

This will remove all boxes that have a large overlap with the selected boxes. Only the "best" boxes remain.

Exercise 3 - yolo_non_max_suppression

Implement `yolo_non_max_suppression()` using TensorFlow. TensorFlow has two built-in functions that are used to implement non-max suppression (so you don't actually need to use your `iou()` implementation):

Reference documentation:

- tf.image.non_max_suppression()

```
tf.image.non_max_suppression(
    boxes,
    scores,
    max_output_size,
    iou_threshold=0.5,
    name=None
)
```

Note that in the version of TensorFlow used here, there is no parameter `score_threshold` (it's shown in the documentation for the latest version) so trying to set this value will result in an error message: *got an unexpected keyword argument* ***score_threshold****.*

- tf.gather()

```
keras.gather(
    reference,
    indices
)
```

[40]:
```
def yolo_non_max_suppression(scores, boxes, classes, max_boxes = 10,␣
↪iou_threshold = 0.5):
    """
    Applies Non-max suppression (NMS) to set of boxes

    Arguments:
```

```
    scores -- tensor of shape (None,), output of yolo_filter_boxes()
    boxes -- tensor of shape (None, 4), output of yolo_filter_boxes() that have␣
↪been scaled to the image size (see later)
    classes -- tensor of shape (None,), output of yolo_filter_boxes()
    max_boxes -- integer, maximum number of predicted boxes you'd like
    iou_threshold -- real value, "intersection over union" threshold used for␣
↪NMS filtering

    Returns:
    scores -- tensor of shape (, None), predicted score for each box
    boxes -- tensor of shape (4, None), predicted box coordinates
    classes -- tensor of shape (, None), predicted class for each box

    Note: The "None" dimension of the output tensors has obviously to be less␣
↪than max_boxes. Note also that this
    function will transpose the shapes of scores, boxes, classes. This is made␣
↪for convenience.
    """

    max_boxes_tensor = tf.Variable(max_boxes, dtype='int32')     # tensor to be␣
↪used in tf.image.non_max_suppression()


    # Use tf.image.non_max_suppression() to get the list of indices␣
↪corresponding to boxes you keep
    nms_indices = tf.image.non_max_suppression(boxes, scores, max_output_size =␣
↪max_boxes, iou_threshold = iou_threshold)

    # Use tf.gather() to select only nms_indices from scores, boxes and classes
    scores = tf.gather(scores, nms_indices)
    boxes = tf.gather(boxes, nms_indices)
    classes = tf.gather(classes, nms_indices)



    return scores, boxes, classes
```

[41]:
```
tf.random.set_seed(10)
scores = tf.random.normal([54,], mean=1, stddev=4, seed = 1)
boxes = tf.random.normal([54, 4], mean=1, stddev=4, seed = 1)
classes = tf.random.normal([54,], mean=1, stddev=4, seed = 1)
scores, boxes, classes = yolo_non_max_suppression(scores, boxes, classes)

assert type(scores) == EagerTensor, "Use tensoflow functions"
print("scores[2] = " + str(scores[2].numpy()))
print("boxes[2] = " + str(boxes[2].numpy()))
```

```
print("classes[2] = " + str(classes[2].numpy()))
print("scores.shape = " + str(scores.numpy().shape))
print("boxes.shape = " + str(boxes.numpy().shape))
print("classes.shape = " + str(classes.numpy().shape))

assert type(scores) == EagerTensor, "Use tensoflow functions"
assert type(boxes) == EagerTensor, "Use tensoflow functions"
assert type(classes) == EagerTensor, "Use tensoflow functions"

assert scores.shape == (10,), "Wrong shape"
assert boxes.shape == (10, 4), "Wrong shape"
assert classes.shape == (10,), "Wrong shape"

assert np.isclose(scores[2].numpy(), 8.147684), "Wrong value on scores"
assert np.allclose(boxes[2].numpy(), [ 6.0797963, 3.743308, 1.3914018, -0.
↪34089637]), "Wrong value on boxes"
assert np.isclose(classes[2].numpy(), 1.7079165), "Wrong value on classes"

print("\033[92m All tests passed!")
```

```
scores[2] = 8.147684
boxes[2] = [ 6.0797963   3.743308    1.3914018  -0.34089637]
classes[2] = 1.7079165
scores.shape = (10,)
boxes.shape = (10, 4)
classes.shape = (10,)
 All tests passed!
```

Expected Output:

scores[2]

8.147684

boxes[2]

[6.0797963 3.743308 1.3914018 -0.34089637]

classes[2]

1.7079165

```
<tr>
    <td>
        <b>scores.shape</b>
    </td>
    <td>
       (10,)
    </td>
</tr>
<tr>
```

```
    <td>
        <b>boxes.shape</b>
    </td>
    <td>
       (10, 4)
    </td>
</tr>
<tr>
    <td>
        <b>classes.shape</b>
    </td>
    <td>
       (10,)
    </td>
</tr>
```

2.5 - Wrapping Up the Filtering

It's time to implement a function taking the output of the deep CNN (the 19x19x5x85 dimensional encoding) and filtering through all the boxes using the functions you've just implemented.

Exercise 4 - yolo_eval

Implement `yolo_eval()` which takes the output of the YOLO encoding and filters the boxes using score threshold and NMS. There's just one last implementational detail you have to know. There're a few ways of representing boxes, such as via their corners or via their midpoint and height/width. YOLO converts between a few such formats at different times, using the following functions (which are provided):

```
boxes = yolo_boxes_to_corners(box_xy, box_wh)
```

which converts the yolo box coordinates (x,y,w,h) to box corners' coordinates (x1, y1, x2, y2) to fit the input of `yolo_filter_boxes`

```
boxes = scale_boxes(boxes, image_shape)
```

YOLO's network was trained to run on 608x608 images. If you are testing this data on a different size image – for example, the car detection dataset had 720x1280 images – this step rescales the boxes so that they can be plotted on top of the original 720x1280 image.

Don't worry about these two functions; you'll see where they need to be called below.

[57]:
```
def yolo_boxes_to_corners(box_xy, box_wh):
    """Convert YOLO box predictions to bounding box corners."""
    box_mins = box_xy - (box_wh / 2.)
    box_maxes = box_xy + (box_wh / 2.)

    return tf.keras.backend.concatenate([
        box_mins[..., 1:2],  # y_min
        box_mins[..., 0:1],  # x_min
        box_maxes[..., 1:2],  # y_max
        box_maxes[..., 0:1]  # x_max
```

```
    ])
```

[60]:
```
def yolo_eval(yolo_outputs, image_shape = (720, 1280), max_boxes=10,␣
↪score_threshold=.6, iou_threshold=.5):
    """
    Converts the output of YOLO encoding (a lot of boxes) to your predicted␣
↪boxes along with their scores, box coordinates and classes.

    Arguments:
    yolo_outputs -- output of the encoding model (for image_shape of (608, 608,␣
↪3)), contains 4 tensors:
                    box_xy: tensor of shape (None, 19, 19, 5, 2)
                    box_wh: tensor of shape (None, 19, 19, 5, 2)
                    box_confidence: tensor of shape (None, 19, 19, 5, 1)
                    box_class_probs: tensor of shape (None, 19, 19, 5, 80)
    image_shape -- tensor of shape (2,) containing the input shape, in this␣
↪notebook we use (608., 608.) (has to be float32 dtype)
    max_boxes -- integer, maximum number of predicted boxes you'd like
    score_threshold -- real value, if [ highest class probability score <␣
↪threshold], then get rid of the corresponding box
    iou_threshold -- real value, "intersection over union" threshold used for␣
↪NMS filtering

    Returns:
    scores -- tensor of shape (None, ), predicted score for each box
    boxes -- tensor of shape (None, 4), predicted box coordinates
    classes -- tensor of shape (None,), predicted class for each box
    """



    # Retrieve outputs of the YOLO model
    box_xy, box_wh, box_confidence, box_class_probs = yolo_outputs

    # Convert boxes to be ready for filtering functions (convert boxes box_xy␣
↪and box_wh to corner coordinates)
    boxes = yolo_boxes_to_corners(box_xy, box_wh)

    # Use one of the functions you've implemented to perform Score-filtering␣
↪with a threshold of score_threshold
    scores, boxes, classes = yolo_filter_boxes(boxes, box_confidence,␣
↪box_class_probs, score_threshold)

    # Scale boxes back to original image shape.
    boxes = scale_boxes(boxes, image_shape)
```

```
    # Use one of the functions you've implemented to perform Non-max␣
↪suppression with
    # maximum number of boxes set to max_boxes and a threshold of iou_threshold
    scores, boxes, classes = yolo_non_max_suppression(scores, boxes, classes,␣
↪max_boxes, iou_threshold)

    return scores, boxes, classes
```

[61]:
```
tf.random.set_seed(10)
yolo_outputs = (tf.random.normal([19, 19, 5, 2], mean=1, stddev=4, seed = 1),
                tf.random.normal([19, 19, 5, 2], mean=1, stddev=4, seed = 1),
                tf.random.normal([19, 19, 5, 1], mean=1, stddev=4, seed = 1),
                tf.random.normal([19, 19, 5, 80], mean=1, stddev=4, seed = 1))
scores, boxes, classes = yolo_eval(yolo_outputs)
print("scores[2] = " + str(scores[2].numpy()))
print("boxes[2] = " + str(boxes[2].numpy()))
print("classes[2] = " + str(classes[2].numpy()))
print("scores.shape = " + str(scores.numpy().shape))
print("boxes.shape = " + str(boxes.numpy().shape))
print("classes.shape = " + str(classes.numpy().shape))

assert type(scores) == EagerTensor, "Use tensoflow functions"
assert type(boxes) == EagerTensor, "Use tensoflow functions"
assert type(classes) == EagerTensor, "Use tensoflow functions"

assert scores.shape == (10,), "Wrong shape"
assert boxes.shape == (10, 4), "Wrong shape"
assert classes.shape == (10,), "Wrong shape"

assert np.isclose(scores[2].numpy(), 171.60194), "Wrong value on scores"
assert np.allclose(boxes[2].numpy(), [-1240.3483, -3212.5881, -645.78, 2024.
↪3052]), "Wrong value on boxes"
assert np.isclose(classes[2].numpy(), 16), "Wrong value on classes"

print("\033[92m All tests passed!")
```

```
scores[2] = 171.60194
boxes[2] = [-1240.3483 -3212.5881  -645.78    2024.3052]
classes[2] = 16
scores.shape = (10,)
boxes.shape = (10, 4)
classes.shape = (10,)
 All tests passed!
```

Expected Output:

scores[2]

171.60194

boxes[2]

[-1240.3483 -3212.5881 -645.78 2024.3052]

classes[2]

16

```
<tr>
    <td>
        <b>scores.shape</b>
    </td>
    <td>
       (10,)
    </td>
</tr>
<tr>
    <td>
        <b>boxes.shape</b>
    </td>
    <td>
       (10, 4)
    </td>
</tr>
<tr>
    <td>
        <b>classes.shape</b>
    </td>
    <td>
       (10,)
    </td>
</tr>
```

3 - Test YOLO Pre-trained Model on Images

In this section, you are going to use a pre-trained model and test it on the car detection dataset.

3.1 - Defining Classes, Anchors and Image Shape

You're trying to detect 80 classes, and are using 5 anchor boxes. The information on the 80 classes and 5 boxes is gathered in two files: "coco_classes.txt" and "yolo_anchors.txt". You'll read class names and anchors from text files. The car detection dataset has 720x1280 images, which are pre-processed into 608x608 images.

[62]:
```
class_names = read_classes("model_data/coco_classes.txt")
anchors = read_anchors("model_data/yolo_anchors.txt")
model_image_size = (608, 608) # Same as yolo_model input layer size
```

3.2 - Loading a Pre-trained Model

Training a YOLO model takes a very long time and requires a fairly large dataset of labelled bounding boxes for a large range of target classes. You are going to load an existing pre-trained

Keras YOLO model stored in "yolo.h5". These weights come from the official YOLO website, and were converted using a function written by Allan Zelener. References are at the end of this notebook. Technically, these are the parameters from the "YOLOv2" model, but are simply referred to as "YOLO" in this notebook.

Run the cell below to load the model from this file.

[63]:
```
yolo_model = load_model("model_data/", compile=False)
```

This loads the weights of a trained YOLO model. Here's a summary of the layers your model contains:

[64]:
```
yolo_model.summary()
```

```
Model: "functional_1"
__________________________________________________________________________________
__________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================
==================
input_1 (InputLayer)            [(None, 608, 608, 3) 0
__________________________________________________________________________________
__________________
conv2d (Conv2D)                 (None, 608, 608, 32) 864         input_1[0][0]
__________________________________________________________________________________
__________________
batch_normalization (BatchNorma (None, 608, 608, 32) 128         conv2d[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu (LeakyReLU)         (None, 608, 608, 32) 0
batch_normalization[0][0]
__________________________________________________________________________________
__________________
max_pooling2d (MaxPooling2D)    (None, 304, 304, 32) 0
leaky_re_lu[0][0]
__________________________________________________________________________________
__________________
conv2d_1 (Conv2D)               (None, 304, 304, 64) 18432
max_pooling2d[0][0]
__________________________________________________________________________________
__________________
batch_normalization_1 (BatchNor (None, 304, 304, 64) 256         conv2d_1[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_1 (LeakyReLU)       (None, 304, 304, 64) 0
batch_normalization_1[0][0]
__________________________________________________________________________________
__________________
max_pooling2d_1 (MaxPooling2D)  (None, 152, 152, 64) 0
```

```
leaky_re_lu_1[0][0]
__________________________________________________________________________________
__________________
conv2d_2 (Conv2D)               (None, 152, 152, 128 73728
max_pooling2d_1[0][0]
__________________________________________________________________________________
__________________
batch_normalization_2 (BatchNor (None, 152, 152, 128 512         conv2d_2[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_2 (LeakyReLU)       (None, 152, 152, 128 0
batch_normalization_2[0][0]
__________________________________________________________________________________
__________________
conv2d_3 (Conv2D)               (None, 152, 152, 64) 8192
leaky_re_lu_2[0][0]
__________________________________________________________________________________
__________________
batch_normalization_3 (BatchNor (None, 152, 152, 64) 256         conv2d_3[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_3 (LeakyReLU)       (None, 152, 152, 64) 0
batch_normalization_3[0][0]
__________________________________________________________________________________
__________________
conv2d_4 (Conv2D)               (None, 152, 152, 128 73728
leaky_re_lu_3[0][0]
__________________________________________________________________________________
__________________
batch_normalization_4 (BatchNor (None, 152, 152, 128 512         conv2d_4[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_4 (LeakyReLU)       (None, 152, 152, 128 0
batch_normalization_4[0][0]
__________________________________________________________________________________
__________________
max_pooling2d_2 (MaxPooling2D)  (None, 76, 76, 128)  0
leaky_re_lu_4[0][0]
__________________________________________________________________________________
__________________
conv2d_5 (Conv2D)               (None, 76, 76, 256)  294912
max_pooling2d_2[0][0]
__________________________________________________________________________________
__________________
batch_normalization_5 (BatchNor (None, 76, 76, 256)  1024        conv2d_5[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_5 (LeakyReLU)       (None, 76, 76, 256)  0
```

```
batch_normalization_5[0][0]
__________________________________________________________________________________
__________________
conv2d_6 (Conv2D)               (None, 76, 76, 128)  32768
leaky_re_lu_5[0][0]
__________________________________________________________________________________
__________________
batch_normalization_6 (BatchNor (None, 76, 76, 128)  512         conv2d_6[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_6 (LeakyReLU)       (None, 76, 76, 128)  0
batch_normalization_6[0][0]
__________________________________________________________________________________
__________________
conv2d_7 (Conv2D)               (None, 76, 76, 256)  294912
leaky_re_lu_6[0][0]
__________________________________________________________________________________
__________________
batch_normalization_7 (BatchNor (None, 76, 76, 256)  1024        conv2d_7[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_7 (LeakyReLU)       (None, 76, 76, 256)  0
batch_normalization_7[0][0]
__________________________________________________________________________________
__________________
max_pooling2d_3 (MaxPooling2D)  (None, 38, 38, 256)  0
leaky_re_lu_7[0][0]
__________________________________________________________________________________
__________________
conv2d_8 (Conv2D)               (None, 38, 38, 512)  1179648
max_pooling2d_3[0][0]
__________________________________________________________________________________
__________________
batch_normalization_8 (BatchNor (None, 38, 38, 512)  2048        conv2d_8[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_8 (LeakyReLU)       (None, 38, 38, 512)  0
batch_normalization_8[0][0]
__________________________________________________________________________________
__________________
conv2d_9 (Conv2D)               (None, 38, 38, 256)  131072
leaky_re_lu_8[0][0]
__________________________________________________________________________________
__________________
batch_normalization_9 (BatchNor (None, 38, 38, 256)  1024        conv2d_9[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_9 (LeakyReLU)       (None, 38, 38, 256)  0
```

```
batch_normalization_9[0][0]
__________________________________________________________________________________
__________________
conv2d_10 (Conv2D)              (None, 38, 38, 512)  1179648
leaky_re_lu_9[0][0]
__________________________________________________________________________________
__________________
batch_normalization_10 (BatchNo (None, 38, 38, 512)  2048        conv2d_10[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_10 (LeakyReLU)      (None, 38, 38, 512)  0
batch_normalization_10[0][0]
__________________________________________________________________________________
__________________
conv2d_11 (Conv2D)              (None, 38, 38, 256)  131072
leaky_re_lu_10[0][0]
__________________________________________________________________________________
__________________
batch_normalization_11 (BatchNo (None, 38, 38, 256)  1024        conv2d_11[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_11 (LeakyReLU)      (None, 38, 38, 256)  0
batch_normalization_11[0][0]
__________________________________________________________________________________
__________________
conv2d_12 (Conv2D)              (None, 38, 38, 512)  1179648
leaky_re_lu_11[0][0]
__________________________________________________________________________________
__________________
batch_normalization_12 (BatchNo (None, 38, 38, 512)  2048        conv2d_12[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_12 (LeakyReLU)      (None, 38, 38, 512)  0
batch_normalization_12[0][0]
__________________________________________________________________________________
__________________
max_pooling2d_4 (MaxPooling2D)  (None, 19, 19, 512)  0
leaky_re_lu_12[0][0]
__________________________________________________________________________________
__________________
conv2d_13 (Conv2D)              (None, 19, 19, 1024) 4718592
max_pooling2d_4[0][0]
__________________________________________________________________________________
__________________
batch_normalization_13 (BatchNo (None, 19, 19, 1024) 4096        conv2d_13[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_13 (LeakyReLU)      (None, 19, 19, 1024) 0
```

```
batch_normalization_13[0][0]
__________________________________________________________________________________
__________________
conv2d_14 (Conv2D)              (None, 19, 19, 512)  524288
leaky_re_lu_13[0][0]
__________________________________________________________________________________
__________________
batch_normalization_14 (BatchNo (None, 19, 19, 512)  2048        conv2d_14[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_14 (LeakyReLU)      (None, 19, 19, 512)  0
batch_normalization_14[0][0]
__________________________________________________________________________________
__________________
conv2d_15 (Conv2D)              (None, 19, 19, 1024) 4718592
leaky_re_lu_14[0][0]
__________________________________________________________________________________
__________________
batch_normalization_15 (BatchNo (None, 19, 19, 1024) 4096        conv2d_15[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_15 (LeakyReLU)      (None, 19, 19, 1024) 0
batch_normalization_15[0][0]
__________________________________________________________________________________
__________________
conv2d_16 (Conv2D)              (None, 19, 19, 512)  524288
leaky_re_lu_15[0][0]
__________________________________________________________________________________
__________________
batch_normalization_16 (BatchNo (None, 19, 19, 512)  2048        conv2d_16[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_16 (LeakyReLU)      (None, 19, 19, 512)  0
batch_normalization_16[0][0]
__________________________________________________________________________________
__________________
conv2d_17 (Conv2D)              (None, 19, 19, 1024) 4718592
leaky_re_lu_16[0][0]
__________________________________________________________________________________
__________________
batch_normalization_17 (BatchNo (None, 19, 19, 1024) 4096        conv2d_17[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_17 (LeakyReLU)      (None, 19, 19, 1024) 0
batch_normalization_17[0][0]
__________________________________________________________________________________
__________________
conv2d_18 (Conv2D)              (None, 19, 19, 1024) 9437184
```

```
leaky_re_lu_17[0][0]
__________________________________________________________________________________
__________________
batch_normalization_18 (BatchNo (None, 19, 19, 1024) 4096        conv2d_18[0][0]
__________________________________________________________________________________
__________________
conv2d_20 (Conv2D)              (None, 38, 38, 64)   32768
leaky_re_lu_12[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_18 (LeakyReLU)      (None, 19, 19, 1024) 0
batch_normalization_18[0][0]
__________________________________________________________________________________
__________________
batch_normalization_20 (BatchNo (None, 38, 38, 64)   256         conv2d_20[0][0]
__________________________________________________________________________________
__________________
conv2d_19 (Conv2D)              (None, 19, 19, 1024) 9437184
leaky_re_lu_18[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_20 (LeakyReLU)      (None, 38, 38, 64)   0
batch_normalization_20[0][0]
__________________________________________________________________________________
__________________
batch_normalization_19 (BatchNo (None, 19, 19, 1024) 4096        conv2d_19[0][0]
__________________________________________________________________________________
__________________
space_to_depth_x2 (Lambda)      (None, 19, 19, 256)  0
leaky_re_lu_20[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_19 (LeakyReLU)      (None, 19, 19, 1024) 0
batch_normalization_19[0][0]
__________________________________________________________________________________
__________________
concatenate (Concatenate)       (None, 19, 19, 1280) 0
space_to_depth_x2[0][0]
leaky_re_lu_19[0][0]
__________________________________________________________________________________
__________________
conv2d_21 (Conv2D)              (None, 19, 19, 1024) 11796480
concatenate[0][0]
__________________________________________________________________________________
__________________
batch_normalization_21 (BatchNo (None, 19, 19, 1024) 4096        conv2d_21[0][0]
__________________________________________________________________________________
__________________
```

```
leaky_re_lu_21 (LeakyReLU)      (None, 19, 19, 1024) 0
batch_normalization_21[0][0]
__________________________________________________________________________________
__________________
conv2d_22 (Conv2D)              (None, 19, 19, 425)  435625
leaky_re_lu_21[0][0]
==================================================================================
==================
Total params: 50,983,561
Trainable params: 50,962,889
Non-trainable params: 20,672
__________________________________________________________________________________
__________________
```

Note: On some computers, you may see a warning message from Keras. Don't worry about it if you do – this is fine!

Reminder: This model converts a preprocessed batch of input images (shape: (m, 608, 608, 3)) into a tensor of shape (m, 19, 19, 5, 85) as explained in Figure (2).

3.3 - Convert Output of the Model to Usable Bounding Box Tensors

The output of `yolo_model` is a (m, 19, 19, 5, 85) tensor that needs to pass through non-trivial processing and conversion. You will need to call `yolo_head` to format the encoding of the model you got from `yolo_model` into something decipherable:

yolo_model_outputs = yolo_model(image_data) yolo_outputs = yolo_head(yolo_model_outputs, anchors, len(class_names)) The variable `yolo_outputs` will be defined as a set of 4 tensors that you can then use as input by your yolo_eval function. If you are curious about how yolo_head is implemented, you can find the function definition in the file `keras_yolo.py`. The file is also located in your workspace in this path: `yad2k/models/keras_yolo.py`.

3.4 - Filtering Boxes

`yolo_outputs` gave you all the predicted boxes of `yolo_model` in the correct format. To perform filtering and select only the best boxes, you will call `yolo_eval`, which you had previously implemented, to do so:

```
out_scores, out_boxes, out_classes = yolo_eval(yolo_outputs, [image.size[1],  image.size[0]],
```

3.5 - Run the YOLO on an Image

Let the fun begin! You will create a graph that can be summarized as follows:

`yolo_model.input` is given to `yolo_model`. The model is used to compute the output `yolo_model.output` `yolo_model.output` is processed by `yolo_head`. It gives you `yolo_outputs` `yolo_outputs` goes through a filtering function, `yolo_eval`. It outputs your predictions: `out_scores`, `out_boxes`, `out_classes`.

Now, we have implemented for you the `predict(image_file)` function, which runs the graph to test YOLO on an image to compute `out_scores`, `out_boxes`, `out_classes`.

The code below also uses the following function:

```
image, image_data = preprocess_image("images/" + image_file, model_image_size = (608, 608))
```

which opens the image file and scales, reshapes and normalizes the image. It returns the outputs:

```
image: a python (PIL) representation of your image used for drawing boxes. You won't need to us
image_data: a numpy-array representing the image. This will be the input to the CNN.
```

[ ]:
```
def predict(image_file):
    """
    Runs the graph to predict boxes for "image_file". Prints and plots the␣
↪predictions.

    Arguments:
    image_file -- name of an image stored in the "images" folder.

    Returns:
    out_scores -- tensor of shape (None, ), scores of the predicted boxes
    out_boxes -- tensor of shape (None, 4), coordinates of the predicted boxes
    out_classes -- tensor of shape (None, ), class index of the predicted boxes

    Note: "None" actually represents the number of predicted boxes, it varies␣
↪between 0 and max_boxes.
    """

    # Preprocess your image
    image, image_data = preprocess_image("images/" + image_file,␣
↪model_image_size = (608, 608))

    yolo_model_outputs = yolo_model(image_data)
    yolo_outputs = yolo_head(yolo_model_outputs, anchors, len(class_names))

    out_scores, out_boxes, out_classes = yolo_eval(yolo_outputs, [image.
↪size[1],  image.size[0]], 10, 0.3, 0.5)

    # Print predictions info
    print('Found {} boxes for {}'.format(len(out_boxes), "images/" +␣
↪image_file))
    # Generate colors for drawing bounding boxes.
    colors = get_colors_for_classes(len(class_names))
    # Draw bounding boxes on the image file
    #draw_boxes2(image, out_scores, out_boxes, out_classes, class_names,␣
↪colors, image_shape)
    draw_boxes(image, out_boxes, out_classes, class_names, out_scores)
    # Save the predicted bounding box on the image
    image.save(os.path.join("out", image_file), quality=100)
    # Display the results in the notebook
    output_image = Image.open(os.path.join("out", image_file))
    imshow(output_image)
```

```
    return out_scores, out_boxes, out_classes
```

Run the following cell on the “test.jpg” image to verify that your function is correct.

[ ]:
```
out_scores, out_boxes, out_classes = predict("test.jpg")
```

Expected Output:

Found 10 boxes for images/test.jpg

car

0.89 (367, 300) (745, 648)

car

0.80 (761, 282) (942, 412)

car

0.74 (159, 303) (346, 440)

car

0.70 (947, 324) (1280, 705)

bus

0.67 (5, 266) (220, 407)

car

0.66 (706, 279) (786, 350)

car

0.60 (925, 285) (1045, 374)

```
<tr>
    <td>
        <b>car</b>
    </td>
    <td>
       0.44 (336, 296) (378, 335)
    </td>
</tr>
<tr>
    <td>
        <b>car</b>
    </td>
    <td>
       0.37 (965, 273) (1022, 292)
    </td>
</tr>
<tr>
```

```
    <td>
        <b>traffic light</b>
    </td>
    <td>
       00.36 (681, 195) (692, 214)
    </td>
</tr>
```

The model you've just run is actually able to detect 80 different classes listed in "coco_classes.txt". To test the model on your own images: 1. Click on "File" in the upper bar of this notebook, then click "Open" to go on your Coursera Hub. 2. Add your image to this Jupyter Notebook's directory, in the "images" folder 3. Write your image's name in the cell above code 4. Run the code and see the output of the algorithm!

If you were to run your session in a for loop over all your images. Here's what you would get:

Predictions of the YOLO model on pictures taken from a camera while driving around the Silicon Valley Thanks to drive.ai for providing this dataset!

4 - Summary for YOLO

- Input image (608, 608, 3)
- The input image goes through a CNN, resulting in a (19,19,5,85) dimensional output.
- After flattening the last two dimensions, the output is a volume of shape (19, 19, 425):
  - Each cell in a 19x19 grid over the input image gives 425 numbers.
  - 425 = 5 x 85 because each cell contains predictions for 5 boxes, corresponding to 5 anchor boxes, as seen in lecture.
  - 85 = 5 + 80 where 5 is because $(p_c, b_x, b_y, b_h, b_w)$ has 5 numbers, and 80 is the number of classes we'd like to detect
- You then select only few boxes based on:
  - Score-thresholding: throw away boxes that have detected a class with a score less than the threshold
  - Non-max suppression: Compute the Intersection over Union and avoid selecting overlapping boxes
- This gives you YOLO's final output.

What you should remember:

- YOLO is a state-of-the-art object detection model that is fast and accurate
- It runs an input image through a CNN, which outputs a 19x19x5x85 dimensional volume.
- The encoding can be seen as a grid where each of the 19x19 cells contains information about 5 boxes.
- You filter through all the boxes using non-max suppression. Specifically:
  - Score thresholding on the probability of detecting a class to keep only accurate (high probability) boxes
  - Intersection over Union (IoU) thresholding to eliminate overlapping boxes
- Because training a YOLO model from randomly initialized weights is non-trivial and requires a large dataset as well as lot of computation, previously trained model parameters were used in this exercise. If you wish, you can also try fine-tuning the YOLO model with your own dataset, though this would be a fairly non-trivial exercise.

Congratulations! You've come to the end of this assignment.

Here's a quick recap of all you've accomplished.

You've:

- Detected objects in a car detection dataset
- Implemented non-max suppression to achieve better accuracy
- Implemented intersection over union as a function of NMS
- Created usable bounding box tensors from the model's predictions

Amazing work! If you'd like to know more about the origins of these ideas, spend some time on the papers referenced below.

5 - References

The ideas presented in this notebook came primarily from the two YOLO papers. The implementation here also took significant inspiration and used many components from Allan Zelener's GitHub repository. The pre-trained weights used in this exercise came from the official YOLO website. - Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi - You Only Look Once: Unified, Real-Time Object Detection (2015) - Joseph Redmon, Ali Farhadi - YOLO9000: Better, Faster, Stronger (2016) - Allan Zelener - YAD2K: Yet Another Darknet 2 Keras - The official YOLO website (https://pjreddie.com/darknet/yolo/)

1.1.1 Car detection dataset

The Drive.ai Sample Dataset (provided by drive.ai) is licensed under a Creative Commons Attribution 4.0 International License. Thanks to Brody Huval, Chih Hu and Rahul Patel for providing this data.

[ ]: