Autonomous driving application - Car detection - v3

April 20, 2019

1 Autonomous driving - Car detection

Welcome to your week 3 programming assignment. You will learn about 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 (https://arxiv.org/abs/1506.02640) and Redmon and Farhadi, 2016 (https://arxiv.org/abs/1612.08242).

You will learn to: - Use object detection on a car detection dataset - Deal with bounding boxes

Run the following cell to load the packages and dependencies that are going to be useful for your journey!

```
In [110]: 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
          import tensorflow as tf
          from keras import backend as K
          from keras.layers import Input, Lambda, Conv2D
          from keras.models import load_model, Model
          from yolo_utils import read_classes, read_anchors, generate_colors, prepr
          from yad2k.models.keras_yolo import yolo_head, yolo_boxes_to_corners, pre

          %matplotlib inline
```

Important Note: As you can see, we import Keras's backend as K. This means that to use a Keras function in this notebook, you will need to write: `K.function(...)`.

1.1 1 - Problem Statement

You are working on a self-driving car. 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 while you drive around.

Pictures taken from a car-mounted camera while driving around Silicon Valley. We would like to especially thank drive.ai for providing this dataset! Drive.ai is a company building the brains of self-driving vehicles.

You've gathered all these images into a folder and have 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 you have 80 classes that you want YOLO 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 had used the latter representation; in this notebook, we will use both representations, depending on which is more convenient for a particular step.

In this exercise, you will learn how YOLO works, then apply it to car detection. Because the YOLO model is very computationally expensive to train, we will load pre-trained weights for you to use.

1.2 2 - YOLO

YOLO ("you only look once") is a popular algoritm 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.

1.2.1 2.1 - Model details

First things to know: - The **input** is a batch of images of 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.

We will use 5 anchor boxes. So you can think of the YOLO architecture as the following: IMAGE (m, 608, 608, 3) -> DEEP CNN -> ENCODING (m, 19, 19, 5, 85).

Lets 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 we are 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, we will flatten the last two last 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**

Now, for each box (of each cell) we will compute the following elementwise product and extract a probability that the box contains a certain class.

Figure 4 : **Find the class detected by each box**

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 both the 5 anchor boxes and across different classes). - Color that grid cell according to what object that grid cell considers the most likely.

Doing this results in this picture:

Figure 5 : Each of the 19x19 grid cells 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.

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.

In the figure above, we plotted only boxes that the model had assigned a high probability to, but this is still too many boxes. You'd like to filter the algorithm's output down 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) - Select only one box when several boxes overlap with each other and detect the same object.

1.2.2 2.2 - Filtering with a threshold on class scores

You are going to apply a first filter by thresholding. You would like to 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'll be convenient to rearrange the (19,19,5,85) (or (19,19,425)) dimensional tensor into the following variables:
- `box_confidence`: tensor of shape $(19 \times 19, 5, 1)$ 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 $(19 \times 19, 5, 4)$ containing (b_x, b_y, b_h, b_w) for each of the 5 boxes per cell. - `box_class_probs`: tensor of shape $(19 \times 19, 5, 80)$ containing the detection probabilities $(c_1, c_2, ...c_{80})$ for each of the 80 classes for each of the 5 boxes per cell.

Exercise: Implement `yolo_filter_boxes()`. 1. Compute box scores by doing the elementwise product as described in Figure 4. 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)
```

2. For each box, find:
   - the index of the class with the maximum box score (Hint) (Be careful with what axis you choose; consider using axis=-1)
   - the corresponding box score (Hint) (Be careful with what axis you choose; consider using axis=-1)
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 we don't want. You should be left with just the subset of boxes you want to keep. (Hint)

Reminder: to call a Keras function, you should use `K.function(...)`.

In [111]: # GRADED FUNCTION: yolo_filter_boxes

```
def yolo_filter_boxes(box_confidence, boxes, box_class_probs, threshold =
    """Filters YOLO boxes by thresholding on object and class confidence.

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

    Returns:
    scores -- tensor of shape (None,), containing the class probability s
    boxes -- tensor of shape (None, 4), containing (b_x, b_y, b_h, b_w) c
    classes -- tensor of shape (None,), containing the index of the class

    Note: "None" is here because you don't know the exact number of selec
    For example, the actual output size of scores would be (10,) if there
    """

    # Step 1: Compute box scores
    ### START CODE HERE ### (≈ 1 line)
    box_scores = box_confidence * box_class_probs
    ### END CODE HERE ###

    # Step 2: Find the box_classes thanks to the max box_scores, keep tra
    ### START CODE HERE ### (≈ 2 lines)
    box_classes = K.argmax(box_scores, axis = -1)
    box_class_scores = K.max(box_scores, axis = -1)
    ### END CODE HERE ###

    # Step 3: Create a filtering mask based on "box_class_scores" by usin
    # same dimension as box_class_scores, and be True for the boxes you w
    ### START CODE HERE ### (≈ 1 line)
    filtering_mask = box_class_scores >= threshold
    #filtering_mask = box_class_scores[box_class_scores >= threshold]
    #filtering_mask = boolean_mask(box_class_scores, threshold)
    ### END CODE HERE ###

    # Step 4: Apply the mask to scores, boxes and classes
    ### START CODE HERE ### (≈ 3 lines)
    #scores =  box_class_scores * filtering_mask
    #boxes = boxes * filtering_mask
    #classes = box_classes * filtering_mask
    scores =  tf.boolean_mask(box_class_scores , filtering_mask)
    boxes = tf.boolean_mask(boxes , filtering_mask)
    classes = tf.boolean_mask(box_classes , filtering_mask)
    #scores =  tf.boolean_mask(box_class_scores , filtering_mask , axis =
```

```
    #boxes = tf.boolean_mask(boxes , filtering_mask , axis = -1)
    #classes = tf.boolean_mask(box_classes , filtering_mask , axis = -1)
    ### END CODE HERE ###

    return scores, boxes, classes
```

In [112]:
```
with tf.Session() as test_a:
    box_confidence = tf.random_normal([19, 19, 5, 1], mean=1, stddev=4, s
    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,
    scores, boxes, classes = yolo_filter_boxes(box_confidence, boxes, box
    print("scores[2] = " + str(scores[2].eval()))
    print("boxes[2] = " + str(boxes[2].eval()))
    print("classes[2] = " + str(classes[2].eval()))
    print("scores.shape = " + str(scores.shape))
    print("boxes.shape = " + str(boxes.shape))
    print("classes.shape = " + str(classes.shape))
```

```
scores[2] = 10.7506
boxes[2] = [ 8.42653275  3.27136683 -0.5313437  -4.94137383]
classes[2] = 7
scores.shape = (?,)
boxes.shape = (?, 4)
classes.shape = (?,)
```

Expected Output:
scores[2]
10.7506
boxes[2]
[8.42653275 3.27136683 -0.5313437 -4.94137383]

```
<tr>
    <td>
        **classes[2]**
    </td>
    <td>
       7
    </td>
</tr>
    <tr>
    <td>
        **scores.shape**
    </td>
    <td>
       (?,)
    </td>
</tr>
<tr>
```

```
    <td>
        **boxes.shape**
    </td>
    <td>
       (?, 4)
    </td>
</tr>

<tr>
    <td>
        **classes.shape**
    </td>
    <td>
       (?,)
    </td>
</tr>
```

1.2.3 2.3 - Non-max suppression

Even after filtering by thresholding over the classes scores, you still end up 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 probabiliy) one 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: Implement iou(). Some hints: - In this exercise only, we define a box using its two corners (upper left and lower right): `(x1, y1, x2, y2)` rather than the midpoint and height/width. - To calculate the area of a rectangle you need to multiply its height `(y2 - y1)` by its width `(x2 - x1)`. - You'll also need to find the coordinates `(xi1, yi1, xi2, yi2)` of the intersection of two boxes. Remember that: - xi1 = maximum of the x1 coordinates of the two boxes - yi1 = maximum of the y1 coordinates of the two boxes - xi2 = minimum of the x2 coordinates of the two boxes - yi2 = minimum of the y2 coordinates of the two boxes - In order to compute the intersection area, you need to make sure the height and width of the intersection are positive, otherwise the intersection area should be zero. Use `max(height, 0)` and `max(width, 0)`.

In this code, we use the convention that (0,0) is the top-left corner of an image, (1,0) is the upper-right corner, and (1,1) the lower-right corner.

In [113]:
```
# GRADED FUNCTION: iou

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

    Arguments:
    box1 -- first box, list object with coordinates (x1, y1, x2, y2)
    box2 -- second box, list object with coordinates (x1, y1, x2, y2)
    """
```

```
    # Calculate the (y1, x1, y2, x2) coordinates of the intersection of b
    ### START CODE HERE ### (≈ 5 lines)
    xi1 = max(box1[0], box2[0])
    yi1 = max(box1[1], box2[1])
    xi2 = min(box1[2], box2[2])
    yi2 = min(box1[3], box2[3])
    inter_area = max( (xi2-xi1) , 0) * max( (yi2-yi1) , 0)
    ### END CODE HERE ###

    # Calculate the Union area by using Formula: Union(A,B) = A + B - Int
    ### START CODE HERE ### (≈ 3 lines)
    box1_area = (box1[2]-box1[0]) * (box1[1]-box1[3])
    box2_area = (box2[2]-box2[0]) * (box2[1]-box2[3])
    union_area = box1_area + box2_area - inter_area
    ### END CODE HERE ###

    # compute the IoU
    ### START CODE HERE ### (≈ 1 line)
    iou = inter_area / union_area
    ### END CODE HERE ###

    return iou
```

```
In [114]: box1 = (2, 1, 4, 3)
          box2 = (1, 2, 3, 4)
          print("iou = " + str(iou(box1, box2)))
```

```
iou = -0.1111111111111111
```

Expected Output:
iou =
0.14285714285714285

You are now ready to implement non-max suppression. The key steps are: 1. Select the box that has the highest score. 2. Compute its overlap with all other boxes, and remove boxes that overlap it more than `iou_threshold`. 3. Go back to step 1 and iterate until there's no more boxes with a lower score than the current selected box.

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

Exercise: 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): - tf.image.non_max_suppression() - K.gather()

```
In [115]: # GRADED FUNCTION: yolo_non_max_suppression

          def yolo_non_max_suppression(scores, boxes, classes, max_boxes = 10, iou_
              """
              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() tha
    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 usec

    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
    function will transpose the shapes of scores, boxes, classes. This is
    """

    max_boxes_tensor = K.variable(max_boxes, dtype='int32')     # tensor
    K.get_session().run(tf.variables_initializer([max_boxes_tensor])) # i

    # Use tf.image.non_max_suppression() to get the list of indices corre
    ### START CODE HERE ### (≈ 1 line)
    nms_indices = tf.image.non_max_suppression(boxes, scores, max_boxes_t
    ### END CODE HERE ###

    # Use K.gather() to select only nms_indices from scores, boxes and cl
    ### START CODE HERE ### (≈ 3 lines)
    scores = K.gather(scores, nms_indices)
    boxes = K.gather(boxes, nms_indices)
    classes = K.gather(classes, nms_indices)
    ### END CODE HERE ###

    return scores, boxes, classes
```

In [116]:
```
with tf.Session() as test_b:
    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, clas
    print("scores[2] = " + str(scores[2].eval()))
    print("boxes[2] = " + str(boxes[2].eval()))
    print("classes[2] = " + str(classes[2].eval()))
    print("scores.shape = " + str(scores.eval().shape))
    print("boxes.shape = " + str(boxes.eval().shape))
    print("classes.shape = " + str(classes.eval().shape))
```

```
scores[2] = 6.9384
boxes[2] = [-5.299932    3.13798141  4.45036697  0.95942086]
```

```
classes[2] = -2.24527
scores.shape = (10,)
boxes.shape = (10, 4)
classes.shape = (10,)
```

Expected Output:
scores[2]
6.9384
boxes[2]
[-5.299932 3.13798141 4.45036697 0.95942086]

```
<tr>
    <td>
        **classes[2]**
    </td>
    <td>
       -2.24527
    </td>
</tr>
    <tr>
    <td>
        **scores.shape**
    </td>
    <td>
       (10,)
    </td>
</tr>
<tr>
    <td>
        **boxes.shape**
    </td>
    <td>
       (10, 4)
    </td>
</tr>

<tr>
    <td>
        **classes.shape**
    </td>
    <td>
       (10,)
    </td>
</tr>
```

1.2.4 2.4 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: 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 mid-point and height/width. YOLO converts between a few such formats at different times, using the following functions (which we have 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; we'll show you where they need to be called.

In [117]:
```
# GRADED FUNCTION: yolo_eval

def yolo_eval(yolo_outputs, image_shape = (720., 1280.), max_boxes=10, sc
    """
    Converts the output of YOLO encoding (a lot of boxes) to your predict

    Arguments:
    yolo_outputs -- output of the encoding model (for image_shape of (608
                    box_confidence: tensor of shape (None, 19, 19, 5, 1)
                    box_xy: tensor of shape (None, 19, 19, 5, 2)
                    box_wh: tensor of shape (None, 19, 19, 5, 2)
                    box_class_probs: tensor of shape (None, 19, 19, 5, 8(
    image_shape -- tensor of shape (2,) containing the input shape, in th
    max_boxes -- integer, maximum number of predicted boxes you'd like
    score_threshold -- real value, if [ highest class probability score <
    iou_threshold -- real value, "intersection over union" threshold used

    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
    """

    ### START CODE HERE ###

    # Retrieve outputs of the YOLO model (≈1 line)
    box_confidence, box_xy, box_wh, box_class_probs = yolo_outputs
```

```
        # Convert boxes to be ready for filtering functions
        boxes = yolo_boxes_to_corners(box_xy, box_wh)

        # Use one of the functions you've implemented to perform Score-filter
        scores, boxes, classes = yolo_filter_boxes(box_confidence, boxes, box

        # 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 supp
        scores, boxes, classes = yolo_non_max_suppression(scores, boxes, clas

        ### END CODE HERE ###

        return scores, boxes, classes
```

In [118]:
```
with tf.Session() as test_b:
    yolo_outputs = (tf.random_normal([19, 19, 5, 1], mean=1, stddev=4, se
                    tf.random_normal([19, 19, 5, 2], mean=1, stddev=4, se
                    tf.random_normal([19, 19, 5, 2], mean=1, stddev=4, se
                    tf.random_normal([19, 19, 5, 80], mean=1, stddev=4, s
    scores, boxes, classes = yolo_eval(yolo_outputs)
    print("scores[2] = " + str(scores[2].eval()))
    print("boxes[2] = " + str(boxes[2].eval()))
    print("classes[2] = " + str(classes[2].eval()))
    print("scores.shape = " + str(scores.eval().shape))
    print("boxes.shape = " + str(boxes.eval().shape))
    print("classes.shape = " + str(classes.eval().shape))
```

```
scores[2] = 138.791
boxes[2] = [ 1292.32971191  -278.52166748  3876.98925781  -835.56494141]
classes[2] = 54
scores.shape = (10,)
boxes.shape = (10, 4)
classes.shape = (10,)
```

Expected Output:
scores[2]
138.791
boxes[2]
[1292.32971191 -278.52166748 3876.98925781 -835.56494141]

```
<tr>
    <td>
        **classes[2]**
    </td>
    <td>
```

```
        54
    </td>
</tr>
    <tr>
    <td>
        **scores.shape**
    </td>
    <td>
       (10,)
    </td>
</tr>
<tr>
    <td>
        **boxes.shape**
    </td>
    <td>
       (10, 4)
    </td>
</tr>

<tr>
    <td>
        **classes.shape**
    </td>
    <td>
       (10,)
    </td>
</tr>
```

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 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.

1.3 3 - Test YOLO pretrained model on images

In this part, you are going to use a pretrained model and test it on the car detection dataset. As usual, you start by **creating a session to start your graph**. Run the following cell.

```
In [119]: sess = K.get_session()
```

1.3.1 3.1 - Defining classes, anchors and image shape.

Recall that we are trying to detect 80 classes, and are using 5 anchor boxes. We have gathered the information about the 80 classes and 5 boxes in two files "coco_classes.txt" and "yolo_anchors.txt". Let's load these quantities into the model by running the next cell.

The car detection dataset has 720x1280 images, which we've pre-processed into 608x608 images.

```
In [120]: class_names = read_classes("model_data/coco_classes.txt")
          anchors = read_anchors("model_data/yolo_anchors.txt")
          image_shape = (720., 1280.)
```

1.3.2 3.2 - Loading a pretrained 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 pretrained 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 we will more simply refer to it as "YOLO" in this notebook.) Run the cell below to load the model from this file.

```
In [121]: yolo_model = load_model("model_data/yolo.h5")
```

```
/opt/conda/lib/python3.6/site-packages/keras/models.py:251: UserWarning: No trainin
  warnings.warn('No training configuration found in save file: '
```

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

```
In [122]: yolo_model.summary()
```

```
________________________________________________________________________________________
Layer (type)                     Output Shape          Param #     Connected to
========================================================================================
input_1 (InputLayer)             (None, 608, 608, 3)   0
________________________________________________________________________________________
conv2d_1 (Conv2D)                (None, 608, 608, 32)  864         input_1[0][0]
________________________________________________________________________________________
batch_normalization_1 (BatchNorm (None, 608, 608, 32)  128         conv2d_1[0][0]
________________________________________________________________________________________
leaky_re_lu_1 (LeakyReLU)        (None, 608, 608, 32)  0           batch_normalizat
________________________________________________________________________________________
max_pooling2d_1 (MaxPooling2D)   (None, 304, 304, 32)  0           leaky_re_lu_1[0]
________________________________________________________________________________________
conv2d_2 (Conv2D)                (None, 304, 304, 64)  18432       max_pooling2d_1[
________________________________________________________________________________________
batch_normalization_2 (BatchNorm (None, 304, 304, 64)  256         conv2d_2[0][0]
________________________________________________________________________________________
```

leaky_re_lu_2 (LeakyReLU)	(None, 304, 304, 64)	0	batch_normalizat
max_pooling2d_2 (MaxPooling2D)	(None, 152, 152, 64)	0	leaky_re_lu_2[0]
conv2d_3 (Conv2D)	(None, 152, 152, 128)	73728	max_pooling2d_2[
batch_normalization_3 (BatchNorm	(None, 152, 152, 128)	512	conv2d_3[0][0]
leaky_re_lu_3 (LeakyReLU)	(None, 152, 152, 128)	0	batch_normalizat
conv2d_4 (Conv2D)	(None, 152, 152, 64)	8192	leaky_re_lu_3[0]
batch_normalization_4 (BatchNorm	(None, 152, 152, 64)	256	conv2d_4[0][0]
leaky_re_lu_4 (LeakyReLU)	(None, 152, 152, 64)	0	batch_normalizat
conv2d_5 (Conv2D)	(None, 152, 152, 128)	73728	leaky_re_lu_4[0]
batch_normalization_5 (BatchNorm	(None, 152, 152, 128)	512	conv2d_5[0][0]
leaky_re_lu_5 (LeakyReLU)	(None, 152, 152, 128)	0	batch_normalizat
max_pooling2d_3 (MaxPooling2D)	(None, 76, 76, 128)	0	leaky_re_lu_5[0]
conv2d_6 (Conv2D)	(None, 76, 76, 256)	294912	max_pooling2d_3[
batch_normalization_6 (BatchNorm	(None, 76, 76, 256)	1024	conv2d_6[0][0]
leaky_re_lu_6 (LeakyReLU)	(None, 76, 76, 256)	0	batch_normalizat
conv2d_7 (Conv2D)	(None, 76, 76, 128)	32768	leaky_re_lu_6[0]
batch_normalization_7 (BatchNorm	(None, 76, 76, 128)	512	conv2d_7[0][0]
leaky_re_lu_7 (LeakyReLU)	(None, 76, 76, 128)	0	batch_normalizat
conv2d_8 (Conv2D)	(None, 76, 76, 256)	294912	leaky_re_lu_7[0]
batch_normalization_8 (BatchNorm	(None, 76, 76, 256)	1024	conv2d_8[0][0]
leaky_re_lu_8 (LeakyReLU)	(None, 76, 76, 256)	0	batch_normalizat
max_pooling2d_4 (MaxPooling2D)	(None, 38, 38, 256)	0	leaky_re_lu_8[0]
conv2d_9 (Conv2D)	(None, 38, 38, 512)	1179648	max_pooling2d_4[
batch_normalization_9 (BatchNorm	(None, 38, 38, 512)	2048	conv2d_9[0][0]

leaky_re_lu_9 (LeakyReLU)	(None, 38, 38, 512)	0	batch_normalizat
conv2d_10 (Conv2D)	(None, 38, 38, 256)	131072	leaky_re_lu_9[0]
batch_normalization_10 (BatchNor	(None, 38, 38, 256)	1024	conv2d_10[0][0]
leaky_re_lu_10 (LeakyReLU)	(None, 38, 38, 256)	0	batch_normalizat
conv2d_11 (Conv2D)	(None, 38, 38, 512)	1179648	leaky_re_lu_10[(
batch_normalization_11 (BatchNor	(None, 38, 38, 512)	2048	conv2d_11[0][0]
leaky_re_lu_11 (LeakyReLU)	(None, 38, 38, 512)	0	batch_normalizat
conv2d_12 (Conv2D)	(None, 38, 38, 256)	131072	leaky_re_lu_11[(
batch_normalization_12 (BatchNor	(None, 38, 38, 256)	1024	conv2d_12[0][0]
leaky_re_lu_12 (LeakyReLU)	(None, 38, 38, 256)	0	batch_normalizat
conv2d_13 (Conv2D)	(None, 38, 38, 512)	1179648	leaky_re_lu_12[(
batch_normalization_13 (BatchNor	(None, 38, 38, 512)	2048	conv2d_13[0][0]
leaky_re_lu_13 (LeakyReLU)	(None, 38, 38, 512)	0	batch_normalizat
max_pooling2d_5 (MaxPooling2D)	(None, 19, 19, 512)	0	leaky_re_lu_13[(
conv2d_14 (Conv2D)	(None, 19, 19, 1024)	4718592	max_pooling2d_5[
batch_normalization_14 (BatchNor	(None, 19, 19, 1024)	4096	conv2d_14[0][0]
leaky_re_lu_14 (LeakyReLU)	(None, 19, 19, 1024)	0	batch_normalizat
conv2d_15 (Conv2D)	(None, 19, 19, 512)	524288	leaky_re_lu_14[(
batch_normalization_15 (BatchNor	(None, 19, 19, 512)	2048	conv2d_15[0][0]
leaky_re_lu_15 (LeakyReLU)	(None, 19, 19, 512)	0	batch_normalizat
conv2d_16 (Conv2D)	(None, 19, 19, 1024)	4718592	leaky_re_lu_15[(
batch_normalization_16 (BatchNor	(None, 19, 19, 1024)	4096	conv2d_16[0][0]
leaky_re_lu_16 (LeakyReLU)	(None, 19, 19, 1024)	0	batch_normalizat
conv2d_17 (Conv2D)	(None, 19, 19, 512)	524288	leaky_re_lu_16[(

batch_normalization_17 (BatchNor	(None, 19, 19, 512)	2048	conv2d_17[0][0]
leaky_re_lu_17 (LeakyReLU)	(None, 19, 19, 512)	0	batch_normalizat
conv2d_18 (Conv2D)	(None, 19, 19, 1024)	4718592	leaky_re_lu_17[0
batch_normalization_18 (BatchNor	(None, 19, 19, 1024)	4096	conv2d_18[0][0]
leaky_re_lu_18 (LeakyReLU)	(None, 19, 19, 1024)	0	batch_normalizat
conv2d_19 (Conv2D)	(None, 19, 19, 1024)	9437184	leaky_re_lu_18[0
batch_normalization_19 (BatchNor	(None, 19, 19, 1024)	4096	conv2d_19[0][0]
conv2d_21 (Conv2D)	(None, 38, 38, 64)	32768	leaky_re_lu_13[0
leaky_re_lu_19 (LeakyReLU)	(None, 19, 19, 1024)	0	batch_normalizat
batch_normalization_21 (BatchNor	(None, 38, 38, 64)	256	conv2d_21[0][0]
conv2d_20 (Conv2D)	(None, 19, 19, 1024)	9437184	leaky_re_lu_19[0
leaky_re_lu_21 (LeakyReLU)	(None, 38, 38, 64)	0	batch_normalizat
batch_normalization_20 (BatchNor	(None, 19, 19, 1024)	4096	conv2d_20[0][0]
space_to_depth_x2 (Lambda)	(None, 19, 19, 256)	0	leaky_re_lu_21[0
leaky_re_lu_20 (LeakyReLU)	(None, 19, 19, 1024)	0	batch_normalizat
concatenate_1 (Concatenate)	(None, 19, 19, 1280)	0	space_to_depth_ leaky_re_lu_20[0
conv2d_22 (Conv2D)	(None, 19, 19, 1024)	11796480	concatenate_1[0]
batch_normalization_22 (BatchNor	(None, 19, 19, 1024)	4096	conv2d_22[0][0]
leaky_re_lu_22 (LeakyReLU)	(None, 19, 19, 1024)	0	batch_normalizat
conv2d_23 (Conv2D)	(None, 19, 19, 425)	435625	leaky_re_lu_22[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–it 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).

1.3.3 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. The following cell does that for you.

```
In [123]: yolo_outputs = yolo_head(yolo_model.output, anchors, len(class_names))
```

You added `yolo_outputs` to your graph. This set of 4 tensors is ready to be used as input by your `yolo_eval` function.

1.3.4 3.4 - Filtering boxes

`yolo_outputs` gave you all the predicted boxes of `yolo_model` in the correct format. You're now ready to perform filtering and select only the best boxes. Lets now call `yolo_eval`, which you had previously implemented, to do this.

```
In [124]: scores, boxes, classes = yolo_eval(yolo_outputs, image_shape)
```

1.3.5 3.5 - Run the graph on an image

Let the fun begin. You have created a (`sess`) graph that can be summarized as follows:

1. yolo_model.input is given to `yolo_model`. The model is used to compute the output yolo_model.output
2. yolo_model.output is processed by `yolo_head`. It gives you yolo_outputs
3. yolo_outputs goes through a filtering function, `yolo_eval`. It outputs your predictions: scores, boxes, classes

Exercise: Implement predict() which runs the graph to test YOLO on an image. You will need to run a TensorFlow session, to have it compute `scores, boxes, classes`.

The code below also uses the following function:

```
image, image_data = preprocess_image("images/" + image_file, model_image_size = (60
```

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

Important note: when a model uses BatchNorm (as is the case in YOLO), you will need to pass an additional placeholder in the feed_dict {K.learning_phase(): 0}.

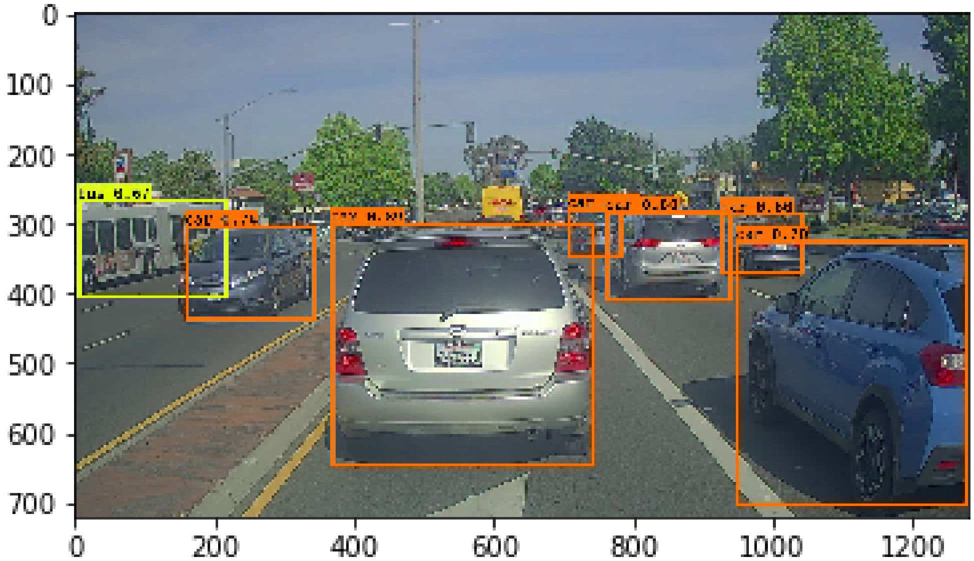
```
In [125]: def predict(sess, image_file):
              """
              Runs the graph stored in "sess" to predict boxes for "image_file". Pr

              Arguments:
```

```
    sess -- your tensorflow/Keras session containing the YOLO graph
    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
    out_classes -- tensor of shape (None, ), class index of the predicted

    Note: "None" actually represents the number of predicted boxes, it va
    """

    # Preprocess your image
    image, image_data = preprocess_image("images/" + image_file, model_in

    # Run the session with the correct tensors and choose the correct pla
    # You'll need to use feed_dict={yolo_model.input: ... , K.learning_ph
    ### START CODE HERE ### (≈ 1 line)
    out_scores, out_boxes, out_classes = sess.run([scores, boxes, classes
    ### END CODE HERE ###

    # Print predictions info
    print('Found {} boxes for {}'.format(len(out_boxes), image_file))
    # Generate colors for drawing bounding boxes.
    colors = generate_colors(class_names)
    # Draw bounding boxes on the image file
    draw_boxes(image, out_scores, out_boxes, out_classes, class_names, co
    # Save the predicted bounding box on the image
    image.save(os.path.join("out", image_file), quality=90)
    # Display the results in the notebook
    output_image = scipy.misc.imread(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.

```
In [126]: out_scores, out_boxes, out_classes = predict(sess, "test.jpg")
```

```
Found 7 boxes for test.jpg
car 0.60 (925, 285) (1045, 374)
car 0.66 (706, 279) (786, 350)
bus 0.67 (5, 266) (220, 407)
car 0.70 (947, 324) (1280, 705)
car 0.74 (159, 303) (346, 440)
car 0.80 (761, 282) (942, 412)
car 0.89 (367, 300) (745, 648)
```



Expected Output:
Found 7 boxes for test.jpg
car
0.60 (925, 285) (1045, 374)
car
0.66 (706, 279) (786, 350)
bus
0.67 (5, 266) (220, 407)
car
0.70 (947, 324) (1280, 705)
car
0.74 (159, 303) (346, 440)
car
0.80 (761, 282) (942, 412)
car
0.89 (367, 300) (745, 648)

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 drive.ai for providing this dataset!

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, we used previously trained model parameters 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.

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 pretrained 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/)

Car detection dataset: The Drive.ai Sample Dataset (provided by drive.ai) is licensed under a Creative Commons Attribution 4.0 International License. We are especially grateful to Brody Huval, Chih Hu and Rahul Patel for collecting and providing this dataset.

```
In [ ]:
```