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

Updates

If you were working on the notebook before this update...

- The current notebook is version "3a".
- You can find your original work saved in the notebook with the previous version name ("v3")
- To view the file directory, go to the menu "File->Open", and this will open a new tab that shows the file directory.

List of updates

- Clarified "YOLO" instructions preceding the code.
- Added details about anchor boxes.
- Added explanation of how score is calculated.
- `yolo_filter_boxes`: added additional hints. Clarify syntax for argmax and max.
- `iou`: clarify instructions for finding the intersection.
- `iou`: give variable names for all 8 box vertices, for clarity. Adds `width` and `height` variables for clarity.
- `iou`: add test cases to check handling of non-intersecting boxes, intersection at vertices, or intersection at edges.
- `yolo_non_max_suppression`: clarify syntax for tf.image.non_max_suppression and keras.gather.
- "convert output of the model to usable bounding box tensors": Provides a link to the definition of `yolo_head`.
- `predict`: hint on calling sess.run.
- Spelling, grammar, wording and formatting updates to improve clarity.

Import libraries

Run the following cell to load the packages and dependencies that you will find useful as you build the object detector!

In [ ]:
```
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, preprocess_im
from yad2k.models.keras_yolo import yolo_head, yolo_boxes_to_corners, preprocess_

%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(...)`.

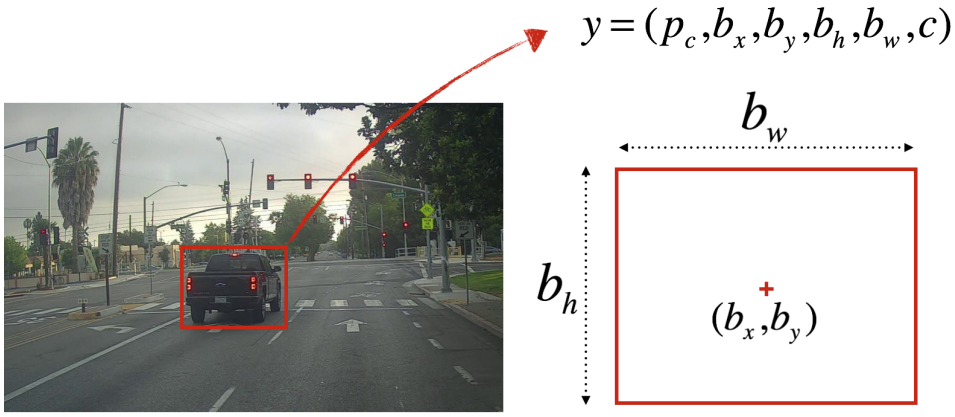
1 - Problem Statement

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.

0:00 / 0:20

Pictures taken from a car-mounted camera while driving around Silicon Valley.
We thank drive.ai for providing this dataset.

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.



$p_c = 1$: confidence of an object being present in the bounding box

$c = 3$: class of the object being detected (here 3 for "car")

Figure 1 : **Definition of a box**

If you have 80 classes that 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 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 "You Only Look Once" (YOLO) performs object detection, and 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.

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.

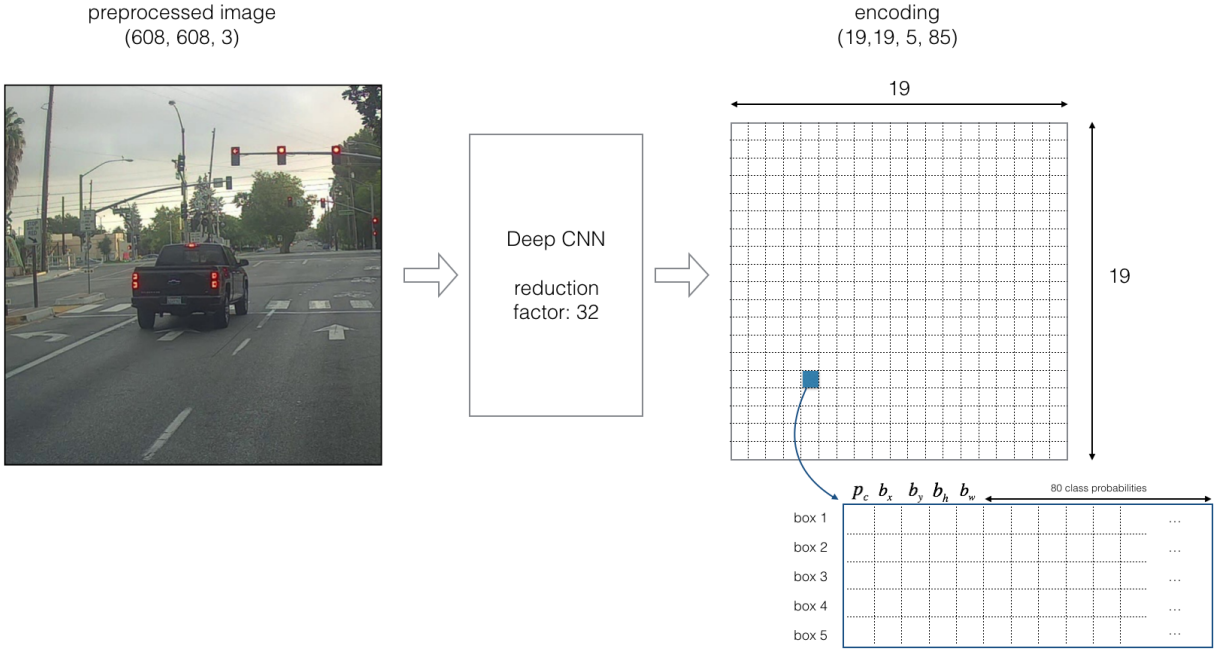


Figure 2 : **Encoding architecture for YOLO**

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

Since 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).

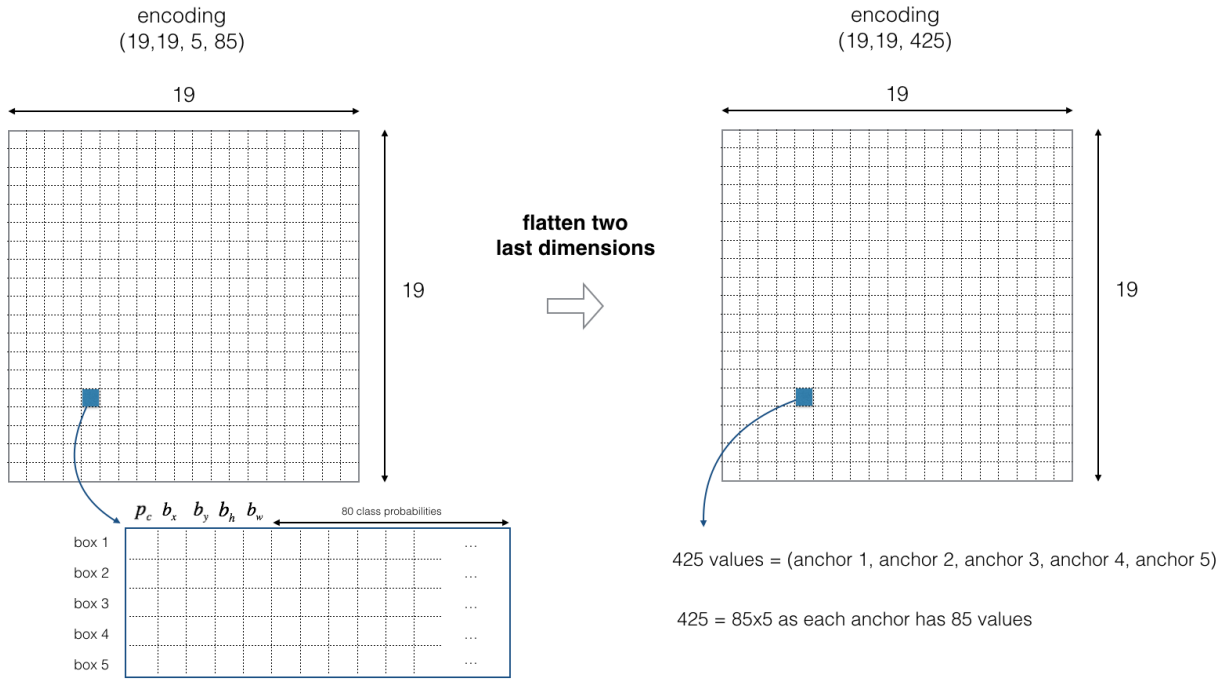


Figure 3 : **Flattening the last two last dimensions**

Class score

Now, for each box (of each cell) we will 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.

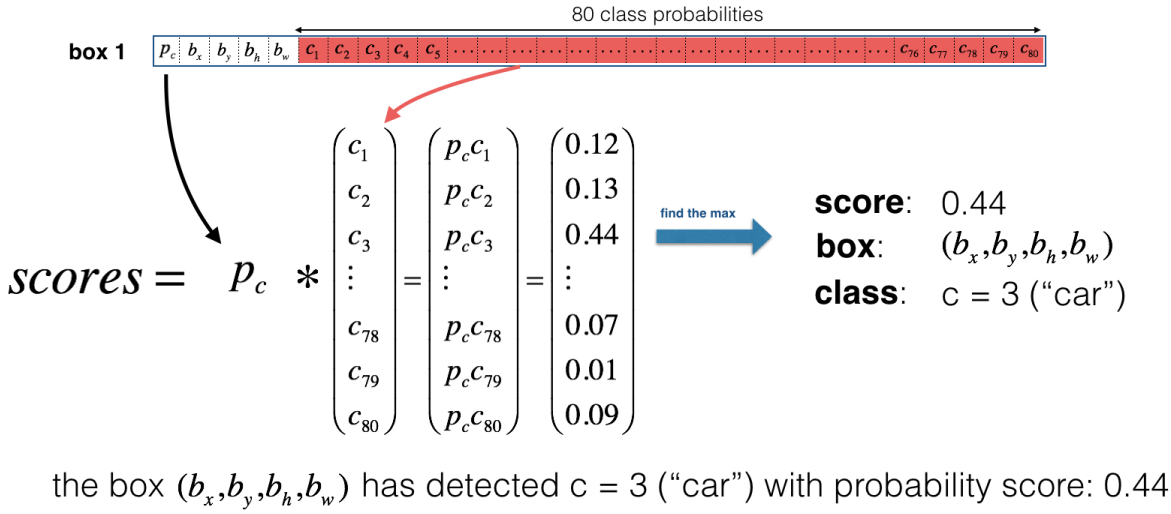


Figure 4 : **Find the class detected by each box**

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 we 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 we'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:

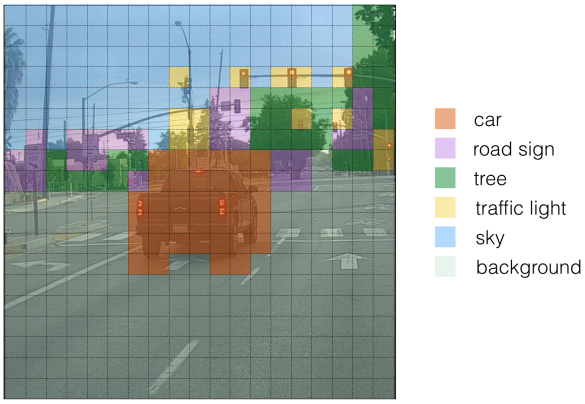


Figure 5 : Each one of the 19x19 grid cells is colored according to which class has the largest predicted probability in that cell.

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, we plotted only boxes 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 are going to first apply a 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 is 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 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 $(19 \times 19, 5, 80)$ containing the "class probabilities" $(c_1, c_2, \ldots c_{80})$ 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 ($p \times c$). 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

   - Keras argmax (https://keras.io/backend/#argmax)
   - Keras max (https://keras.io/backend/#max)

   Additional Hints

   - For the `axis` parameter of `argmax` and `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 `max` normally collapses the axis for which the maximum is applied. `keepdims=False` is the default option, and allows that dimension to be removed. We don't need to keep the last dimension after applying the maximum here.
- Even though the documentation shows `keras.backend.argmax`, use `keras.argmax`. Similarly, use `keras.max`.

1. 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.
2. 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.

   Useful reference:

   - boolean mask (https://www.tensorflow.org/api_docs/python/tf/boolean_mask)

   Additional Hints:

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

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

In [ ]:
```
# GRADED FUNCTION: yolo_filter_boxes

def yolo_filter_boxes(box_confidence, boxes, box_class_probs, threshold = .6):
    """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 < threshold], t

    Returns:
    scores -- tensor of shape (None,), containing the class probability score for
    boxes -- tensor of shape (None, 4), containing (b_x, b_y, b_h, b_w) coordinat
    classes -- tensor of shape (None,), containing the index of the class detecte

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

    # Step 1: Compute box scores
    ### START CODE HERE ### (≈ 1 line)
    box_scores = None
    ### END CODE HERE ###

    # Step 2: Find the box_classes using the max box_scores, keep track of the co
    ### START CODE HERE ### (≈ 2 lines)
    box_classes = None
    box_class_scores = None
    ### END CODE HERE ###

    # Step 3: Create a filtering mask based on "box_class_scores" by using "thres
    # same dimension as box_class_scores, and be True for the boxes you want to k
    ### START CODE HERE ### (≈ 1 line)
    filtering_mask = None
    ### END CODE HERE ###

    # Step 4: Apply the mask to box_class_scores, boxes and box_classes
    ### START CODE HERE ### (≈ 3 lines)
    scores = None
    boxes = None
    classes = None
    ### END CODE HERE ###

    return scores, boxes, classes
```

In [ ]:
```
with tf.Session() as test_a:
    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 =
    scores, boxes, classes = yolo_filter_boxes(box_confidence, boxes, box_class_p
    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))
```

Expected Output:

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

Note In the test for `yolo_filter_boxes`, we'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.

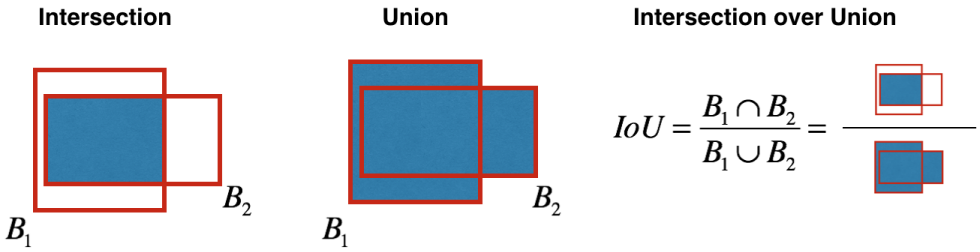


Figure 8 : Definition of "Intersection over Union".

Exercise: Implement iou(). Some hints:

- 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) is the lower-right corner. In other words, the (0,0) origin starts at the top left corner of the image. As x increases, we move to the right. As y increases, we move down.
- For this exercise, we define a box 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)`

In [ ]:
```
# 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 (box1_x1, box1_y1, box1_x2, b
    box2 -- second box, list object with coordinates (box2_x1, box2_y1, box2_x2,
    """

    # Assign variable names to coordinates for clarity
    (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 
    ### START CODE HERE ### (≈ 7 lines)
    xi1 = None
    yi1 = None
    xi2 = None
    yi2 = None
    inter_width = None
    inter_height = None
    inter_area = None
    ### END CODE HERE ###

    # Calculate the Union area by using Formula: Union(A,B) = A + B - Inter(A,B)
    ### START CODE HERE ### (≈ 3 lines)
    box1_area = None
    box2_area = None
    union_area = None
    ### END CODE HERE ###

    # compute the IoU
    ### START CODE HERE ### (≈ 1 line)
    iou = None
    ### END CODE HERE ###

    return iou
```

In [ ]:
```
## Test case 1: boxes intersect
box1 = (2, 1, 4, 3)
box2 = (1, 2, 3, 4)
print("iou for intersecting boxes = " + str(iou(box1, box2)))

## 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)))

## 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)))

## 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)))
```

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
```

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: 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() (https://www.tensorflow.org/api_docs/python/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.*

- K.gather() (https://www.tensorflow.org/api_docs/python/tf/keras/backend/gather)
  Even though the documentation shows `tf.keras.backend.gather()`, you can use `keras.gather()`.

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

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

def yolo_non_max_suppression(scores, boxes, classes, max_boxes = 10, iou_threshol
    """
    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 b
    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

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

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

    # Use tf.image.non_max_suppression() to get the list of indices corresponding
    ### START CODE HERE ### (≈ 1 line)
    nms_indices = None
    ### END CODE HERE ###

    # Use K.gather() to select only nms_indices from scores, boxes and classes
    ### START CODE HERE ### (≈ 3 lines)
    scores = None
    boxes = None
    classes = None
    ### END CODE HERE ###

    return scores, boxes, classes
```

In [ ]:
```
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, classes)
    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))
```

Expected Output:

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,)

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 midpoint 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 [ ]:
```
# GRADED FUNCTION: yolo_eval

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

    Arguments:
    yolo_outputs -- output of the encoding model (for image_shape of (608, 608, 3
                    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, 80)
    image_shape -- tensor of shape (2,) containing the input shape, in this noteb
    max_boxes -- integer, maximum number of predicted boxes you'd like
    score_threshold -- real value, if [ highest class probability score < thresho
    iou_threshold -- real value, "intersection over union" threshold used for NMS

    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 = None

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

    # Use one of the functions you've implemented to perform Score-filtering with
    scores, boxes, classes = None

    # 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 u
    # maximum number of boxes set to max_boxes and a threshold of iou_threshold (=
    scores, boxes, classes = None

    ### END CODE HERE ###

    return scores, boxes, classes
```

In [ ]:
```
with tf.Session() as test_b:
    yolo_outputs = (tf.random_normal([19, 19, 5, 1], 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, 2], 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].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))
```

Expected Output:

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,)

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.

3 - Test YOLO pre-trained model on images

In this part, you are going to use a pre-trained model and test it on the car detection dataset. We'll need a session to execute the computation graph and evaluate the tensors.

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

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 on the 80 classes and 5 boxes in two files "coco_classes.txt" and "yolo_anchors.txt".
- We'll read class names and anchors from text files.
- The car detection dataset has 720x1280 images, which we've pre-processed into 608x608 images.

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

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 we will simply refer to it as "YOLO" in this notebook.

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

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

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

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

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).

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.

If you are curious about how `yolo_head` is implemented, you can find the function definition in the file 'keras_yolo.py' (https://github.com/allanzelener/YAD2K/blob/master/yad2k/models/keras_yolo.py). The file is located in your workspace in this path 'yad2k/models/keras_yolo.py'.

In [ ]:
```
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.

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. Let's now call `yolo_eval`, which you had previously implemented, to do this.

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

3.5 - Run the graph on an image

Let the fun begin. You have created a 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 = (608, 608))
```

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}.

Hint: Using the TensorFlow Session object

- Recall that above, we called `K.get_Session()` and saved the Session object in `sess`.

- To evaluate a list of tensors, we call `sess.run()` like this:

```
sess.run(fetches=[tensor1,tensor2,tensor3],
         feed_dict={yolo_model.input: the_input_variable,
                    K.learning_phase():0
         }
```

- Notice that the variables `scores, boxes, classes` are not passed into the `predict` function, but these are global variables that you will use within the `predict` function.

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

    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 boxes
    out_classes -- tensor of shape (None, ), class index of the predicted boxes

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

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

    # Run the session with the correct tensors and choose the correct placeholders
    # You'll need to use feed_dict={yolo_model.input: ... , K.learning_phase(): 0}
    ### START CODE HERE ### (≈ 1 line)
    out_scores, out_boxes, out_classes = None
    ### 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, colors)
    # 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 [ ]:
```
out_scores, out_boxes, out_classes = predict(sess, "test.jpg")
```

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" t
o go on your Coursera Hub.
2. Add your image to this Jupyter Notebook's directory, in the "images" f
older
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 (https://www.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 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 (https://arxiv.org/abs/1506.02640) (2015)
- Joseph Redmon, Ali Farhadi - YOLO9000: Better, Faster, Stronger (https://arxiv.org/abs/1612.08242) (2016)
- Allan Zelener - YAD2K: Yet Another Darknet 2 Keras (https://github.com/allanzelener/YAD2K)
- The official YOLO website (https://pjreddie.com/darknet/yolo/ (https://pjreddie.com/darknet/yolo/))

Car detection dataset:

In [ ]: