Vehicle Detection

1. Overview

The project is to detect vehicle on the road using **YOLO** model. Many ideas of this project 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), and my project of *Coursera/Deep Learning/Convolutional Neural Networks/Week 3/Car Detection for Autonomous Driving* with the certificate number `WY7JJSS7Q9B9`

2. Project Introduction

Comparing with the class text, the **YOLO** model is relatively easier but needs higher calculate ability. This model doesn't need a lot of image preprocess, and can directly feed the model with *RGB* images, and the output is classified object and the boxes in the image.

3. Project Pipeline

The pipeline is below:

1. read the test images

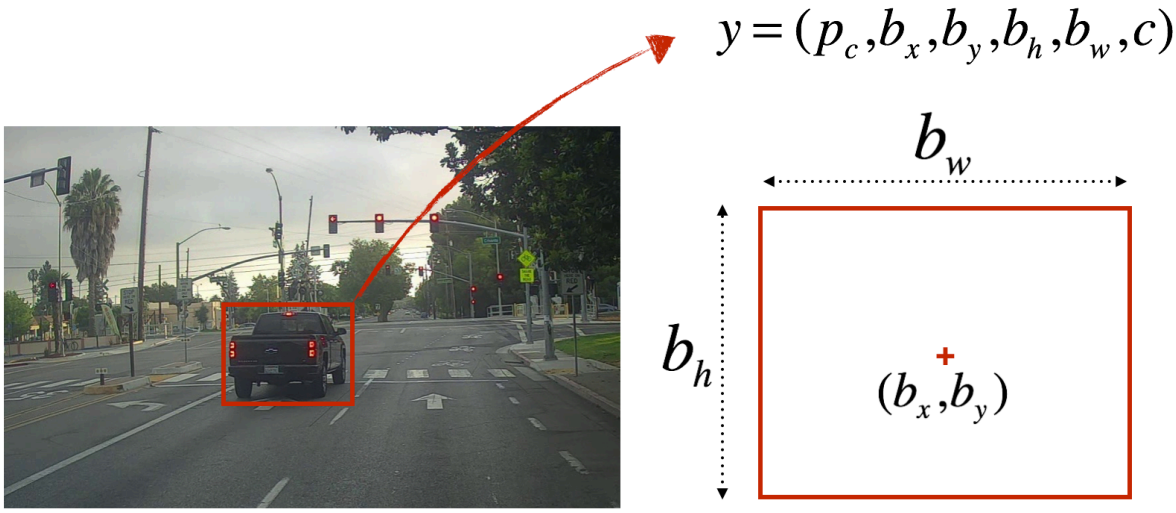
2. load the anchor boxes and the classification
3. define *YOLO* model and load pre-trained *h5* file
4. predict the image
5. define the pipeline and generate the video

3.1 Read images

The read image function shows the original test images.

3.2 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.



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

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

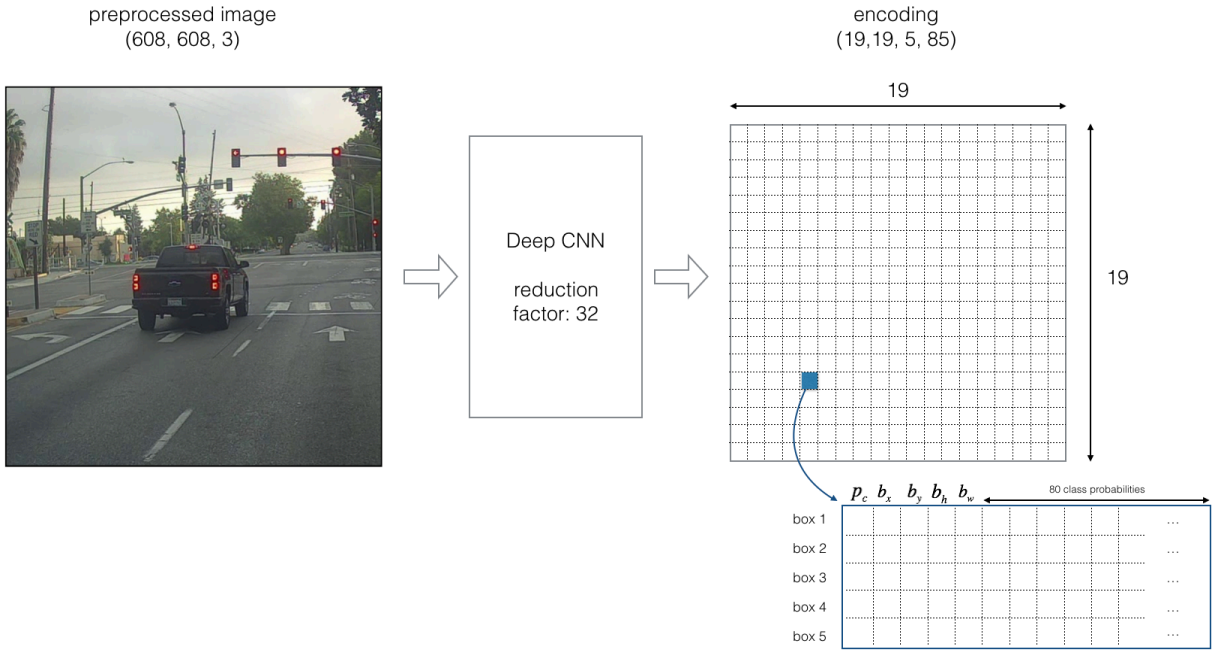
3.2.1 Model details

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.

I am using 5 anchor boxes. So the YOLO architecture is as the following: IMAGE (m, 608, 608, 3) -> DEEP CNN -> ENCODING (m, 19, 19, 5, 85).

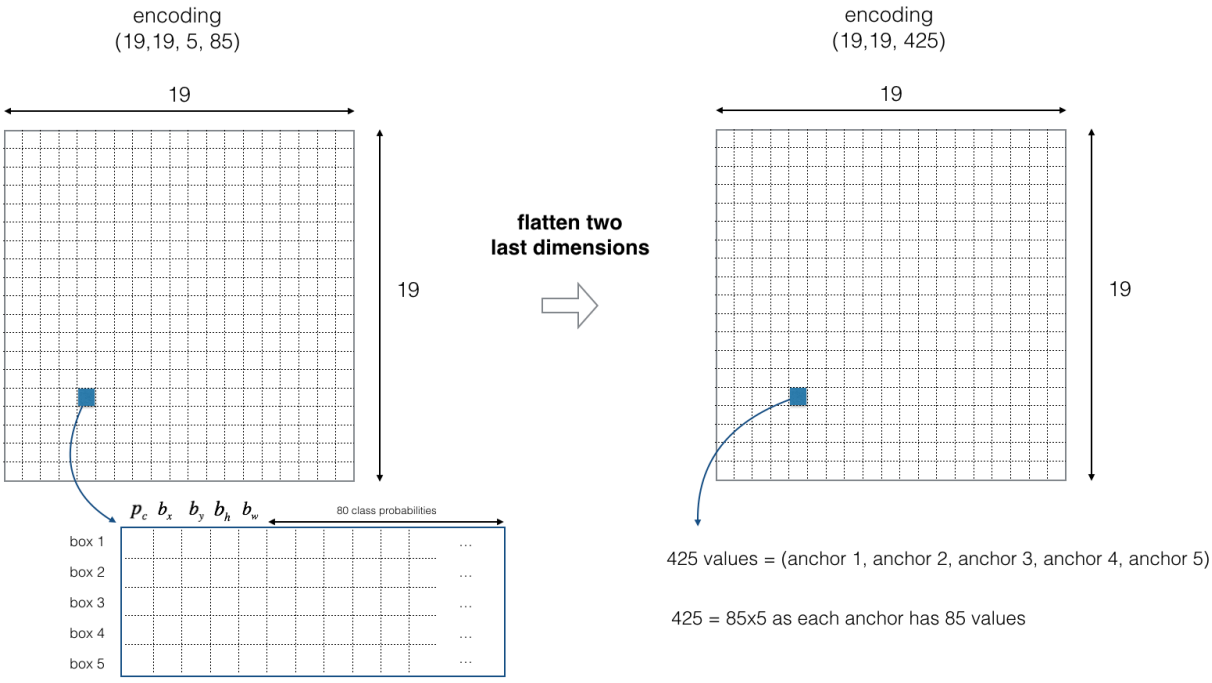
Lets look in greater detail at what this encoding represents.



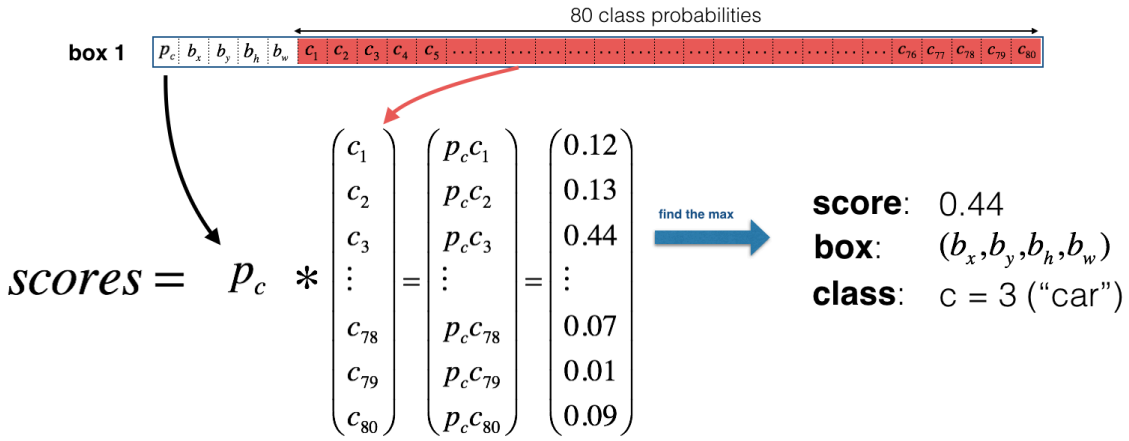
If the center/midpoint of an object falls into a grid cell, that grid cell is responsible for detecting that object.

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).



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.

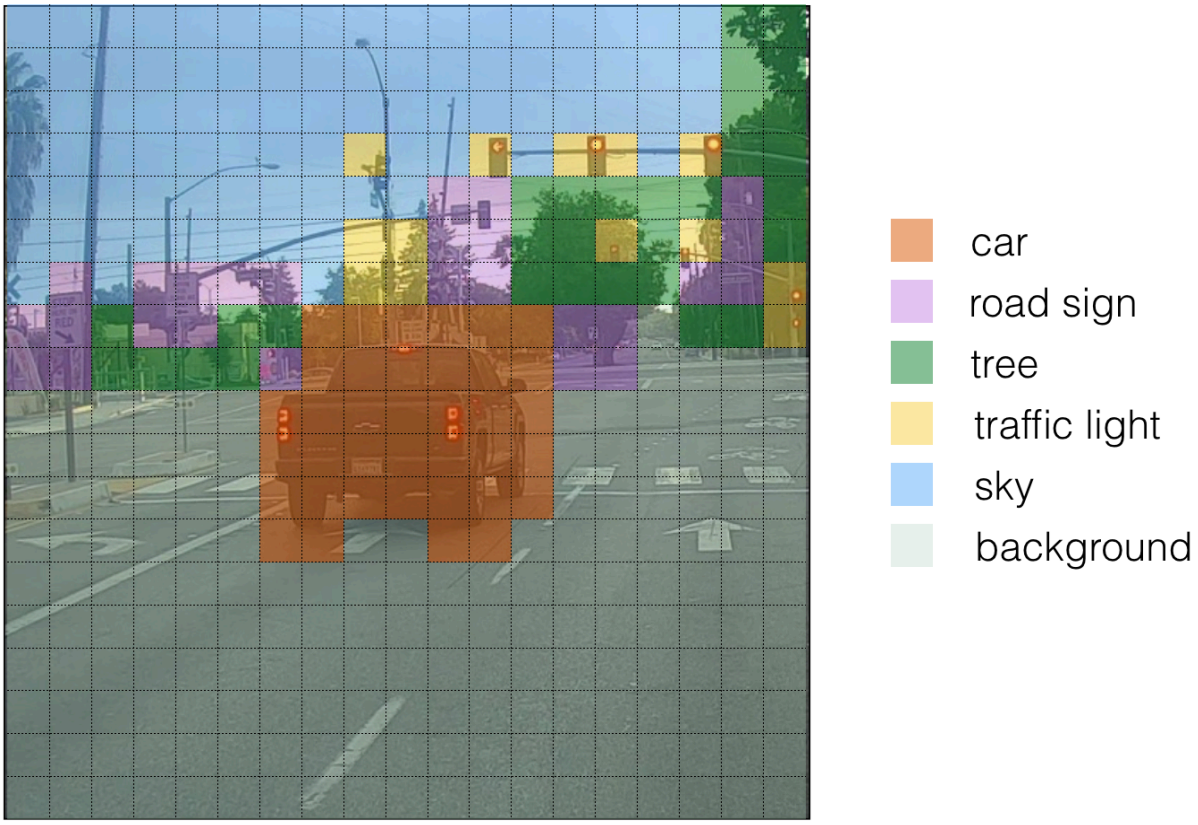


the box (b_x, b_y, b_h, b_w) has detected c = 3 ("car") with probability score: 0.44

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

- For each of the 19×19 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:



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:

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.

3.2.2 Filtering with threshold on class scores

The next step is to get rid of any box for which the class "score" is less than the 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

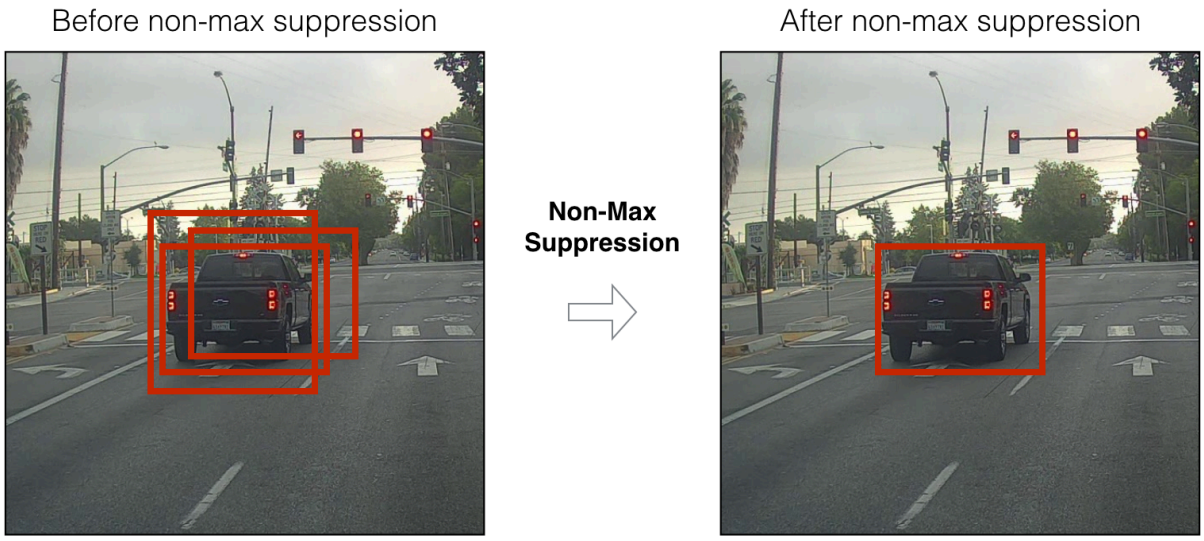
that there's some objects) for each of the 5 boxes predicted in each of the 19×19 cells.

- `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 $(c_1, c_2, \ldots, c_{80})$ for each of 80 classes for each of the 5 boxes per cell.

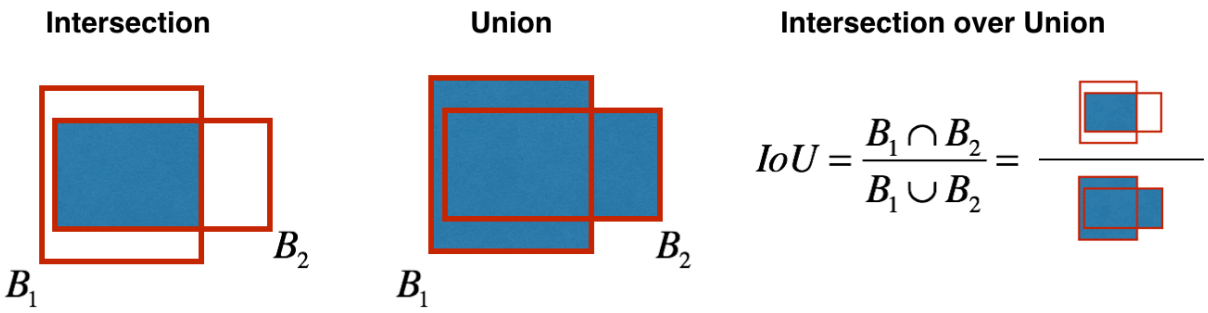
```
def yolo_filter_boxes(box_confidence, boxes, box_class_probs, threshold=.6):
    """Filters YOLO boxes by thresholding on object and class confidence.

    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], 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.
    """

    ### Code Implementation ###

    return scores, boxes, classes
```

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).



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

    ### Code Implementation ###

    return iou
```

Now to implement non-max suppression function, the 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.

```
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.
    """

    ### Code Implementation ###

    return scores, boxes, classes
```

3.2.4 Wrapping up the filtering

This function is to take the output of the $(19 \times 19 \times 5 \times 85)$ dimensional deep CNN and filter through all the boxes using the function just implemented.

The function `yolo_eval()` takes the output of the **YOLO** encoding and filters the boxes using score threshold and NMS.

There are 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 (defined in the `yolo_utils.py`):

```
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.

```
def yolo_eval(yolo_outputs, image_shape = (720., 1280.), max_boxes=10, score_threshold=.5, 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_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 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
    """

    ### Code Implementation ###

    return scores, boxes, classes
```

Conclusion:

- 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
- 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

3.3 Test on images

```
sess = K.get_session()
```

3.3.1 Defining classes, anchors and image shape

```
# TXT includes 80 classes information
class_names = read_classes("model_data/coco_classes.txt")
# TXT includes 5 boxes
anchors = read_anchors("model_data/yolo_anchors.txt")
# Define image shape
image_shape = (720., 1280.)
```

The `coco_classes.txt` includes 80 classes that we want YOLO to recognize, the class label 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.

3.3.2 Loading pertained model

```
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.

```
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 (BatchNor (None, 608, 608, 32) 128
conv2d_1[0][0]
_________________________________________________________________
_________________
leaky_re_lu_1 (LeakyReLU)       (None, 608, 608, 32) 0
batch_normalization_1[0][0]
_________________________________________________________________
_________________
max_pooling2d_1 (MaxPooling2D)  (None, 304, 304, 32) 0
leaky_re_lu_1[0][0]
_________________________________________________________________
_________________
conv2d_2 (Conv2D)               (None, 304, 304, 64) 18432
max_pooling2d_1[0][0]
_________________________________________________________________
_________________
batch_normalization_2 (BatchNor (None, 304, 304, 64) 256
conv2d_2[0][0]
_________________________________________________________________
_________________
leaky_re_lu_2 (LeakyReLU)       (None, 304, 304, 64) 0
batch_normalization_2[0][0]
_________________________________________________________________
_________________
max_pooling2d_2 (MaxPooling2D)  (None, 152, 152, 64) 0
leaky_re_lu_2[0][0]
_________________________________________________________________
_________________
conv2d_3 (Conv2D)               (None, 152, 152, 128 73728
max_pooling2d_2[0][0]
_________________________________________________________________
_________________
batch_normalization_3 (BatchNor (None, 152, 152, 128 512
conv2d_3[0][0]
_________________________________________________________________
_________________
leaky_re_lu_3 (LeakyReLU)       (None, 152, 152, 128 0
batch_normalization_3[0][0]
_________________________________________________________________
_________________
```

```
conv2d_4 (Conv2D)               (None, 152, 152, 64) 8192
leaky_re_lu_3[0][0]
__________________________________________________________________________________
__________________
batch_normalization_4 (BatchNor (None, 152, 152, 64) 256
conv2d_4[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_4 (LeakyReLU)       (None, 152, 152, 64) 0
batch_normalization_4[0][0]
__________________________________________________________________________________
__________________
conv2d_5 (Conv2D)               (None, 152, 152, 128 73728
leaky_re_lu_4[0][0]
__________________________________________________________________________________
__________________
batch_normalization_5 (BatchNor (None, 152, 152, 128 512
conv2d_5[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_5 (LeakyReLU)       (None, 152, 152, 128 0
batch_normalization_5[0][0]
__________________________________________________________________________________
__________________
max_pooling2d_3 (MaxPooling2D)  (None, 76, 76, 128)  0
leaky_re_lu_5[0][0]
__________________________________________________________________________________
__________________
conv2d_6 (Conv2D)               (None, 76, 76, 256)  294912
max_pooling2d_3[0][0]
__________________________________________________________________________________
__________________
batch_normalization_6 (BatchNor (None, 76, 76, 256)  1024
conv2d_6[0][0]
__________________________________________________________________________________
__________________
leaky_re_lu_6 (LeakyReLU)       (None, 76, 76, 256)  0
batch_normalization_6[0][0]
__________________________________________________________________________________
__________________
conv2d_7 (Conv2D)               (None, 76, 76, 128)  32768
leaky_re_lu_6[0][0]
__________________________________________________________________________________
__________________
batch_normalization_7 (BatchNor (None, 76, 76, 128)  512
conv2d_7[0][0]
__________________________________________________________________________________
__________________
```

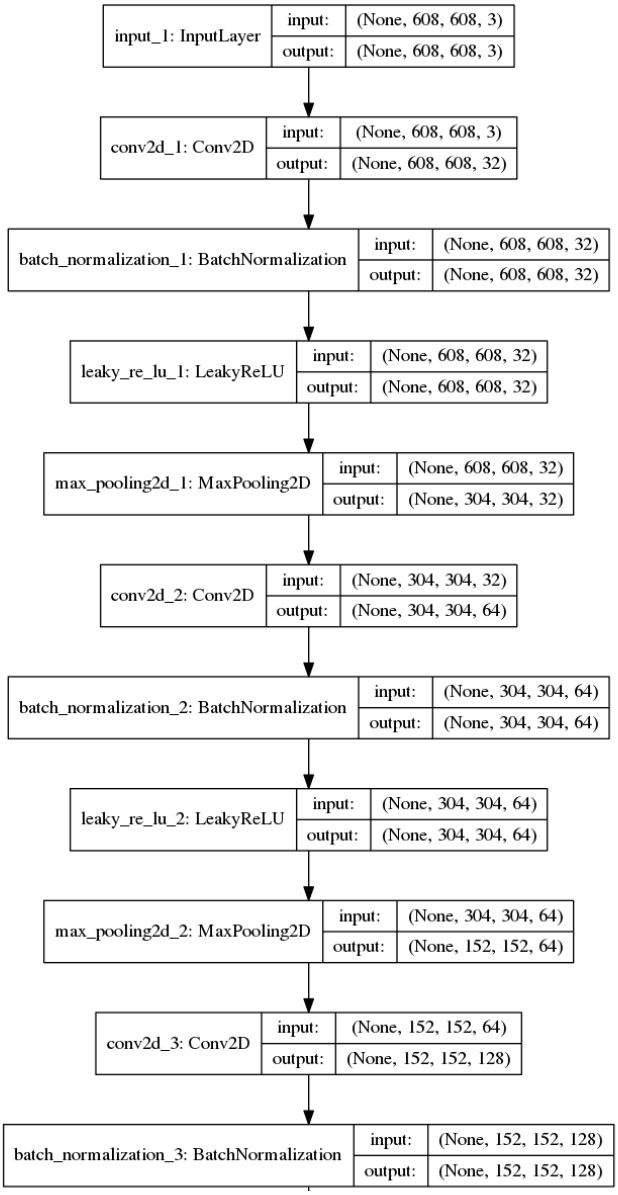
```
leaky_re_lu_7 (LeakyReLU)       (None, 76, 76, 128)  0
batch_normalization_7[0][0]
__________________________________________________________________________________________________
conv2d_8 (Conv2D)               (None, 76, 76, 256)  294912
leaky_re_lu_7[0][0]
__________________________________________________________________________________________________
batch_normalization_8 (BatchNor (None, 76, 76, 256)  1024
conv2d_8[0][0]
__________________________________________________________________________________________________
leaky_re_lu_8 (LeakyReLU)       (None, 76, 76, 256)  0
batch_normalization_8[0][0]
__________________________________________________________________________________________________
max_pooling2d_4 (MaxPooling2D)  (None, 38, 38, 256)  0
leaky_re_lu_8[0][0]
__________________________________________________________________________________________________
conv2d_9 (Conv2D)               (None, 38, 38, 512)  1179648
max_pooling2d_4[0][0]
__________________________________________________________________________________________________
batch_normalization_9 (BatchNor (None, 38, 38, 512)  2048
conv2d_9[0][0]
__________________________________________________________________________________________________
leaky_re_lu_9 (LeakyReLU)       (None, 38, 38, 512)  0
batch_normalization_9[0][0]
__________________________________________________________________________________________________
conv2d_10 (Conv2D)              (None, 38, 38, 256)  131072
leaky_re_lu_9[0][0]
__________________________________________________________________________________________________
batch_normalization_10 (BatchNo (None, 38, 38, 256)  1024
conv2d_10[0][0]
__________________________________________________________________________________________________
leaky_re_lu_10 (LeakyReLU)      (None, 38, 38, 256)  0
batch_normalization_10[0][0]
__________________________________________________________________________________________________
conv2d_11 (Conv2D)              (None, 38, 38, 512)  1179648
leaky_re_lu_10[0][0]
__________________________________________________________________________________________________
```

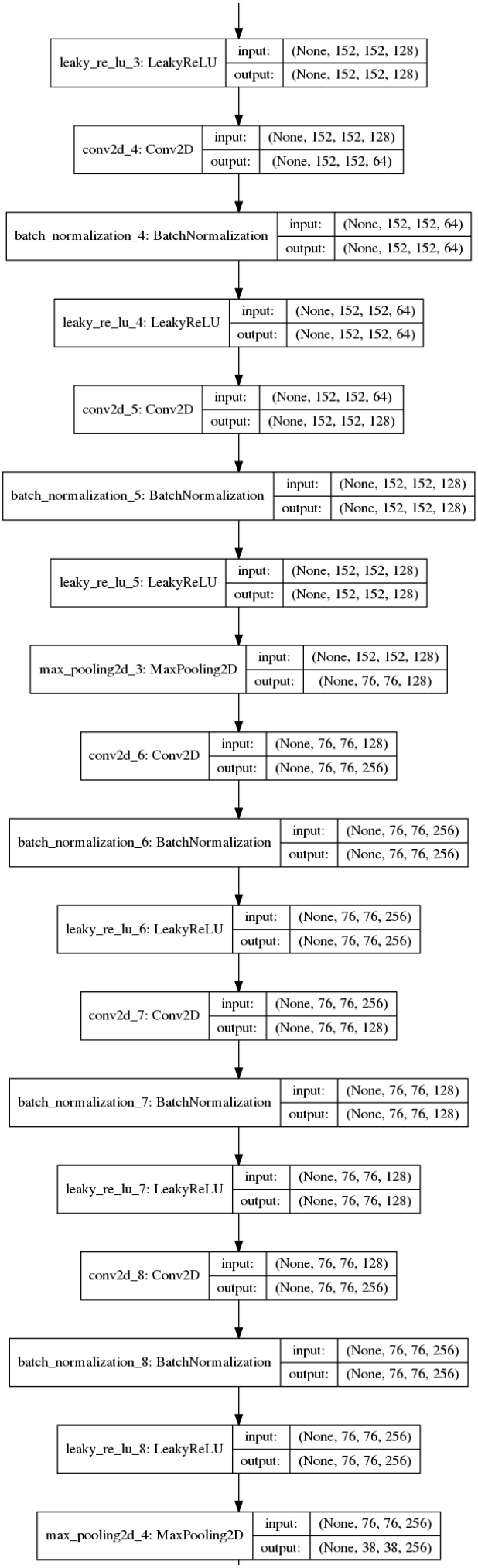
```
batch_normalization_11 (BatchNo (None, 38, 38, 512)  2048
conv2d_11[0][0]
__________________________________________________________________________________________________
leaky_re_lu_11 (LeakyReLU)      (None, 38, 38, 512)  0
batch_normalization_11[0][0]
__________________________________________________________________________________________________
conv2d_12 (Conv2D)              (None, 38, 38, 256)  131072
leaky_re_lu_11[0][0]
__________________________________________________________________________________________________
batch_normalization_12 (BatchNo (None, 38, 38, 256)  1024
conv2d_12[0][0]
__________________________________________________________________________________________________
leaky_re_lu_12 (LeakyReLU)      (None, 38, 38, 256)  0
batch_normalization_12[0][0]
__________________________________________________________________________________________________
conv2d_13 (Conv2D)              (None, 38, 38, 512)  1179648
leaky_re_lu_12[0][0]
__________________________________________________________________________________________________
batch_normalization_13 (BatchNo (None, 38, 38, 512)  2048
conv2d_13[0][0]
__________________________________________________________________________________________________
leaky_re_lu_13 (LeakyReLU)      (None, 38, 38, 512)  0
batch_normalization_13[0][0]
__________________________________________________________________________________________________
max_pooling2d_5 (MaxPooling2D)  (None, 19, 19, 512)  0
leaky_re_lu_13[0][0]
__________________________________________________________________________________________________
conv2d_14 (Conv2D)              (None, 19, 19, 1024) 4718592
max_pooling2d_5[0][0]
__________________________________________________________________________________________________
batch_normalization_14 (BatchNo (None, 19, 19, 1024) 4096
conv2d_14[0][0]
__________________________________________________________________________________________________
leaky_re_lu_14 (LeakyReLU)      (None, 19, 19, 1024) 0
batch_normalization_14[0][0]
__________________________________________________________________________________________________
```

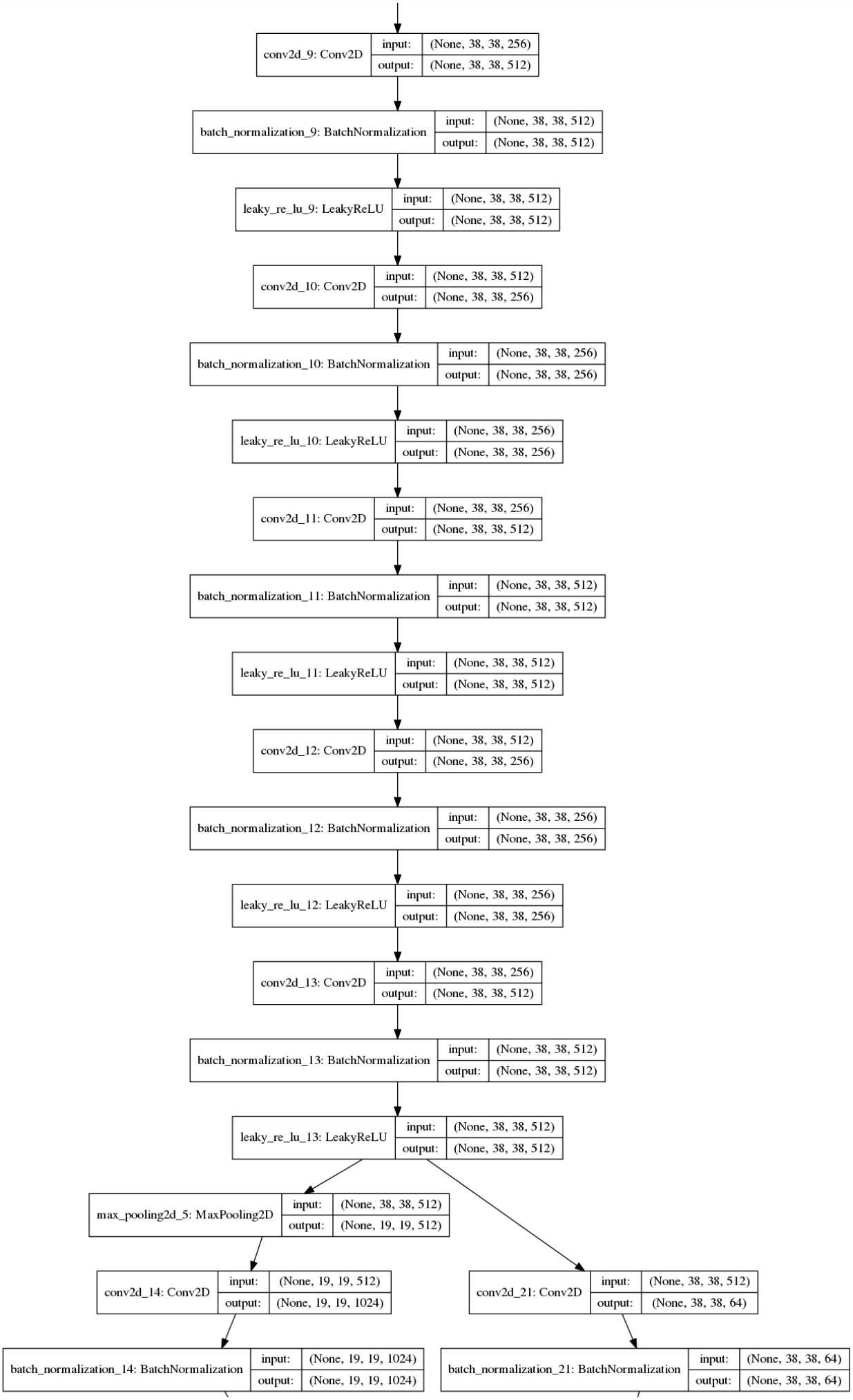
```
conv2d_15 (Conv2D)              (None, 19, 19, 512)  524288
leaky_re_lu_14[0][0]
__________________________________________________________________________________________________
batch_normalization_15 (BatchNo (None, 19, 19, 512)  2048
conv2d_15[0][0]
__________________________________________________________________________________________________
leaky_re_lu_15 (LeakyReLU)      (None, 19, 19, 512)  0
batch_normalization_15[0][0]
__________________________________________________________________________________________________
conv2d_16 (Conv2D)              (None, 19, 19, 1024) 4718592
leaky_re_lu_15[0][0]
__________________________________________________________________________________________________
batch_normalization_16 (BatchNo (None, 19, 19, 1024) 4096
conv2d_16[0][0]
__________________________________________________________________________________________________
leaky_re_lu_16 (LeakyReLU)      (None, 19, 19, 1024) 0
batch_normalization_16[0][0]
__________________________________________________________________________________________________
conv2d_17 (Conv2D)              (None, 19, 19, 512)  524288
leaky_re_lu_16[0][0]
__________________________________________________________________________________________________
batch_normalization_17 (BatchNo (None, 19, 19, 512)  2048
conv2d_17[0][0]
__________________________________________________________________________________________________
leaky_re_lu_17 (LeakyReLU)      (None, 19, 19, 512)  0
batch_normalization_17[0][0]
__________________________________________________________________________________________________
conv2d_18 (Conv2D)              (None, 19, 19, 1024) 4718592
leaky_re_lu_17[0][0]
__________________________________________________________________________________________________
batch_normalization_18 (BatchNo (None, 19, 19, 1024) 4096
conv2d_18[0][0]
__________________________________________________________________________________________________
leaky_re_lu_18 (LeakyReLU)      (None, 19, 19, 1024) 0
batch_normalization_18[0][0]
__________________________________________________________________________________________________
```

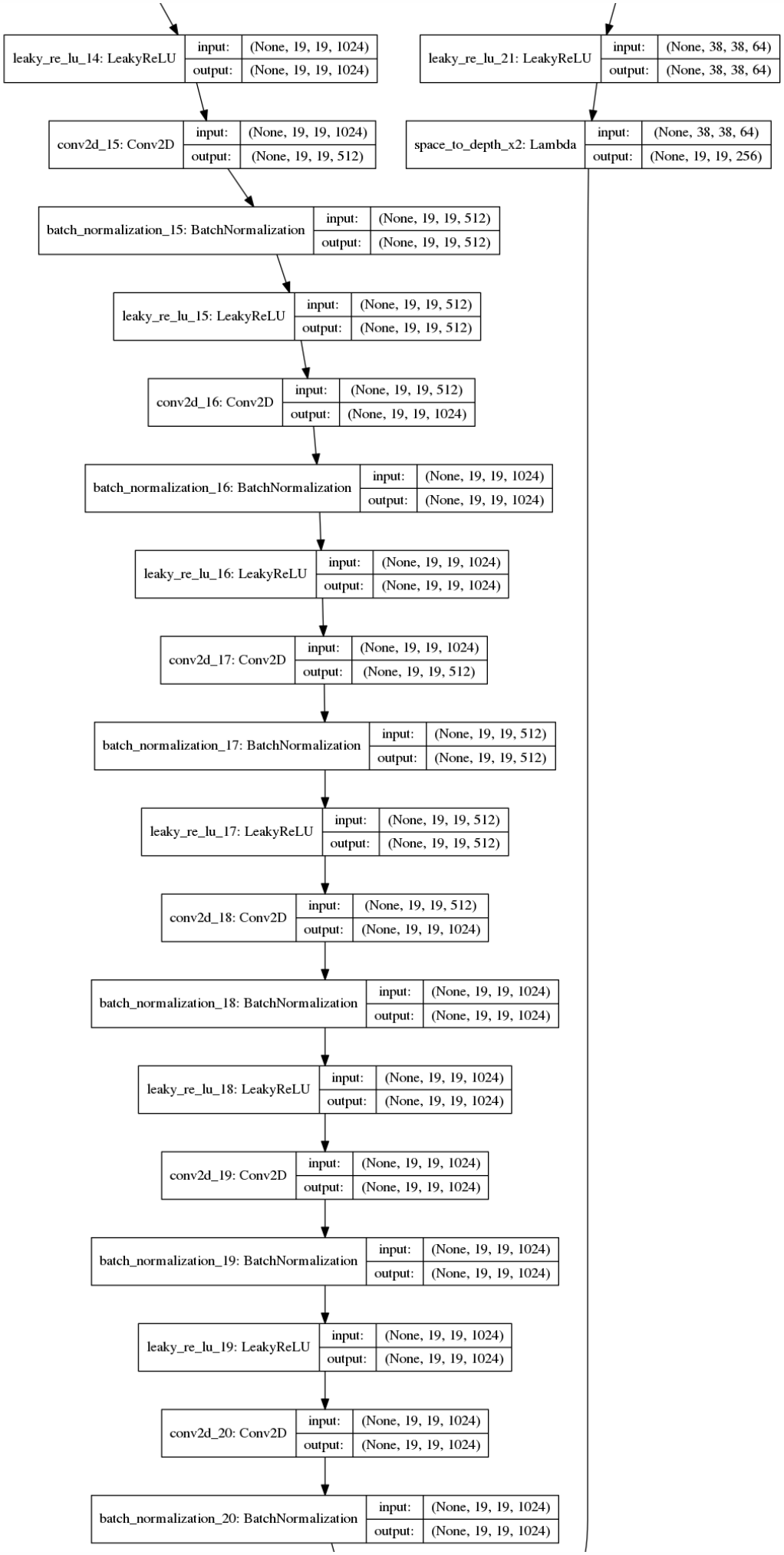
```
conv2d_19 (Conv2D)              (None, 19, 19, 1024) 9437184
leaky_re_lu_18[0][0]
__________________________________________________________________________________________________
batch_normalization_19 (BatchNo (None, 19, 19, 1024) 4096
conv2d_19[0][0]
__________________________________________________________________________________________________
conv2d_21 (Conv2D)              (None, 38, 38, 64)   32768
leaky_re_lu_13[0][0]
__________________________________________________________________________________________________
leaky_re_lu_19 (LeakyReLU)      (None, 19, 19, 1024) 0
batch_normalization_19[0][0]
__________________________________________________________________________________________________
batch_normalization_21 (BatchNo (None, 38, 38, 64)   256
conv2d_21[0][0]
__________________________________________________________________________________________________
conv2d_20 (Conv2D)              (None, 19, 19, 1024) 9437184
leaky_re_lu_19[0][0]
__________________________________________________________________________________________________
leaky_re_lu_21 (LeakyReLU)      (None, 38, 38, 64)   0
batch_normalization_21[0][0]
__________________________________________________________________________________________________
batch_normalization_20 (BatchNo (None, 19, 19, 1024) 4096
conv2d_20[0][0]
__________________________________________________________________________________________________
space_to_depth_x2 (Lambda)      (None, 19, 19, 256)  0
leaky_re_lu_21[0][0]
__________________________________________________________________________________________________
leaky_re_lu_20 (LeakyReLU)      (None, 19, 19, 1024) 0
batch_normalization_20[0][0]
__________________________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 19, 19, 1280) 0
space_to_depth_x2[0][0]

leaky_re_lu_20[0][0]
__________________________________________________________________________________________________
conv2d_22 (Conv2D)              (None, 19, 19, 1024) 11796480
concatenate_1[0][0]
```

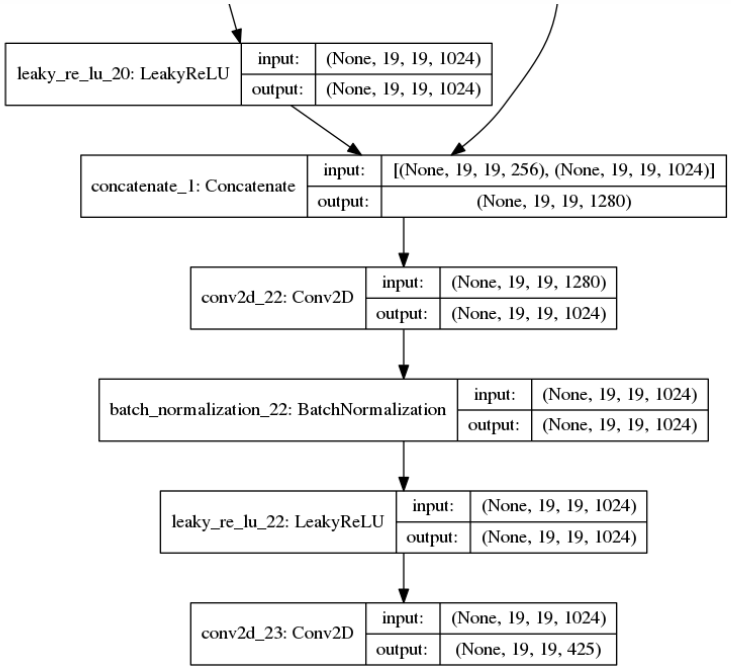
```
147 ____________________________________________________________________________
    __________________
148 batch_normalization_22 (BatchNo (None, 19, 19, 1024) 4096
    conv2d_22[0][0]
149 ____________________________________________________________________________
    __________________
150 leaky_re_lu_22 (LeakyReLU)      (None, 19, 19, 1024) 0
    batch_normalization_22[0][0]
151 ____________________________________________________________________________
    __________________
152 conv2d_23 (Conv2D)              (None, 19, 19, 425)  435625
    leaky_re_lu_22[0][0]
153 ============================================================================
    ================
154 Total params: 50,983,561
155 Trainable params: 50,962,889
156 Non-trainable params: 20,672
157 ____________________________________________________________________________
    __________________
```




leaky_re_lu_3: LeakyReLU
input: (None, 152, 152, 128)
output: (None, 152, 152, 128)
conv2d_4: Conv2D
input: (None, 152, 152, 128)
output: (None, 152, 152, 64)
batch_normalization_4: BatchNormalization
input: (None, 152, 152, 64)
output: (None, 152, 152, 64)
leaky_re_lu_4: LeakyReLU
input: (None, 152, 152, 64)
output: (None, 152, 152, 64)
conv2d_5: Conv2D
input: (None, 152, 152, 64)
output: (None, 152, 152, 128)
batch_normalization_5: BatchNormalization
input: (None, 152, 152, 128)
output: (None, 152, 152, 128)
leaky_re_lu_5: LeakyReLU
input: (None, 152, 152, 128)
output: (None, 152, 152, 128)
max_pooling2d_3: MaxPooling2D
input: (None, 152, 152, 128)
output: (None, 76, 76, 128)
conv2d_6: Conv2D
input: (None, 76, 76, 128)
output: (None, 76, 76, 256)
batch_normalization_6: BatchNormalization
input: (None, 76, 76, 256)
output: (None, 76, 76, 256)
leaky_re_lu_6: LeakyReLU
input: (None, 76, 76, 256)
output: (None, 76, 76, 256)
conv2d_7: Conv2D
input: (None, 76, 76, 256)
output: (None, 76, 76, 128)
batch_normalization_7: BatchNormalization
input: (None, 76, 76, 128)
output: (None, 76, 76, 128)
leaky_re_lu_7: LeakyReLU
input: (None, 76, 76, 128)
output: (None, 76, 76, 128)
conv2d_8: Conv2D
input: (None, 76, 76, 128)
output: (None, 76, 76, 256)
batch_normalization_8: BatchNormalization
input: (None, 76, 76, 256)
output: (None, 76, 76, 256)
leaky_re_lu_8: LeakyReLU
input: (None, 76, 76, 256)
output: (None, 76, 76, 256)
max_pooling2d_4: MaxPooling2D
input: (None, 76, 76, 256)
output: (None, 38, 38, 256)


conv2d_9: Conv2D
input: (None, 38, 38, 256)
output: (None, 38, 38, 512)
batch_normalization_9: BatchNormalization
input: (None, 38, 38, 512)
output: (None, 38, 38, 512)
leaky_re_lu_9: LeakyReLU
input: (None, 38, 38, 512)
output: (None, 38, 38, 512)
conv2d_10: Conv2D
input: (None, 38, 38, 512)
output: (None, 38, 38, 256)
batch_normalization_10: BatchNormalization
input: (None, 38, 38, 256)
output: (None, 38, 38, 256)
leaky_re_lu_10: LeakyReLU
input: (None, 38, 38, 256)
output: (None, 38, 38, 256)
conv2d_11: Conv2D
input: (None, 38, 38, 256)
output: (None, 38, 38, 512)
batch_normalization_11: BatchNormalization
input: (None, 38, 38, 512)
output: (None, 38, 38, 512)
leaky_re_lu_11: LeakyReLU
input: (None, 38, 38, 512)
output: (None, 38, 38, 512)
conv2d_12: Conv2D
input: (None, 38, 38, 512)
output: (None, 38, 38, 256)
batch_normalization_12: BatchNormalization
input: (None, 38, 38, 256)
output: (None, 38, 38, 256)
leaky_re_lu_12: LeakyReLU
input: (None, 38, 38, 256)
output: (None, 38, 38, 256)
conv2d_13: Conv2D
input: (None, 38, 38, 256)
output: (None, 38, 38, 512)
batch_normalization_13: BatchNormalization
input: (None, 38, 38, 512)
output: (None, 38, 38, 512)
leaky_re_lu_13: LeakyReLU
input: (None, 38, 38, 512)
output: (None, 38, 38, 512)
max_pooling2d_5: MaxPooling2D
input: (None, 38, 38, 512)
output: (None, 19, 19, 512)
conv2d_14: Conv2D
input: (None, 19, 19, 512)
output: (None, 19, 19, 1024)
conv2d_21: Conv2D
input: (None, 38, 38, 512)
output: (None, 38, 38, 64)
batch_normalization_14: BatchNormalization
input: (None, 19, 19, 1024)
output: (None, 19, 19, 1024)
batch_normalization_21: BatchNormalization
input: (None, 38, 38, 64)
output: (None, 38, 38, 64)


leaky_re_lu_14: LeakyReLU
input: (None, 19, 19, 1024)
output: (None, 19, 19, 1024)
leaky_re_lu_21: LeakyReLU
input: (None, 38, 38, 64)
output: (None, 38, 38, 64)
conv2d_15: Conv2D
input: (None, 19, 19, 1024)
output: (None, 19, 19, 512)
space_to_depth_x2: Lambda
input: (None, 38, 38, 64)
output: (None, 19, 19, 256)
batch_normalization_15: BatchNormalization
input: (None, 19, 19, 512)
output: (None, 19, 19, 512)
leaky_re_lu_15: LeakyReLU
input: (None, 19, 19, 512)
output: (None, 19, 19, 512)
conv2d_16: Conv2D
input: (None, 19, 19, 512)
output: (None, 19, 19, 1024)
batch_normalization_16: BatchNormalization
input: (None, 19, 19, 1024)
output: (None, 19, 19, 1024)
leaky_re_lu_16: LeakyReLU
input: (None, 19, 19, 1024)
output: (None, 19, 19, 1024)
conv2d_17: Conv2D
input: (None, 19, 19, 1024)
output: (None, 19, 19, 512)
batch_normalization_17: BatchNormalization
input: (None, 19, 19, 512)
output: (None, 19, 19, 512)
leaky_re_lu_17: LeakyReLU
input: (None, 19, 19, 512)
output: (None, 19, 19, 512)
conv2d_18: Conv2D
input: (None, 19, 19, 512)
output: (None, 19, 19, 1024)
batch_normalization_18: BatchNormalization
input: (None, 19, 19, 1024)
output: (None, 19, 19, 1024)
leaky_re_lu_18: LeakyReLU
input: (None, 19, 19, 1024)
output: (None, 19, 19, 1024)
conv2d_19: Conv2D
input: (None, 19, 19, 1024)
output: (None, 19, 19, 1024)
batch_normalization_19: BatchNormalization
input: (None, 19, 19, 1024)
output: (None, 19, 19, 1024)
leaky_re_lu_19: LeakyReLU
input: (None, 19, 19, 1024)
output: (None, 19, 19, 1024)
conv2d_20: Conv2D
input: (None, 19, 19, 1024)
output: (None, 19, 19, 1024)
batch_normalization_20: BatchNormalization
input: (None, 19, 19, 1024)
output: (None, 19, 19, 1024)



3.3.3 Convert output of the model to usable bounding box tensors

```
yolo_outputs = yolo_head(yolo_model.output, anchors, len(class_names))
```

3.3.4 Filtering boxes

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

```
scores, boxes, classes = yolo_eval(yolo_outputs, image_shape)
```

3.3.5 Run on an image

Let the fun begin. A `sess` graph is created that can be summarized as follows:

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

The code below also uses the following function:

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

which outputs:

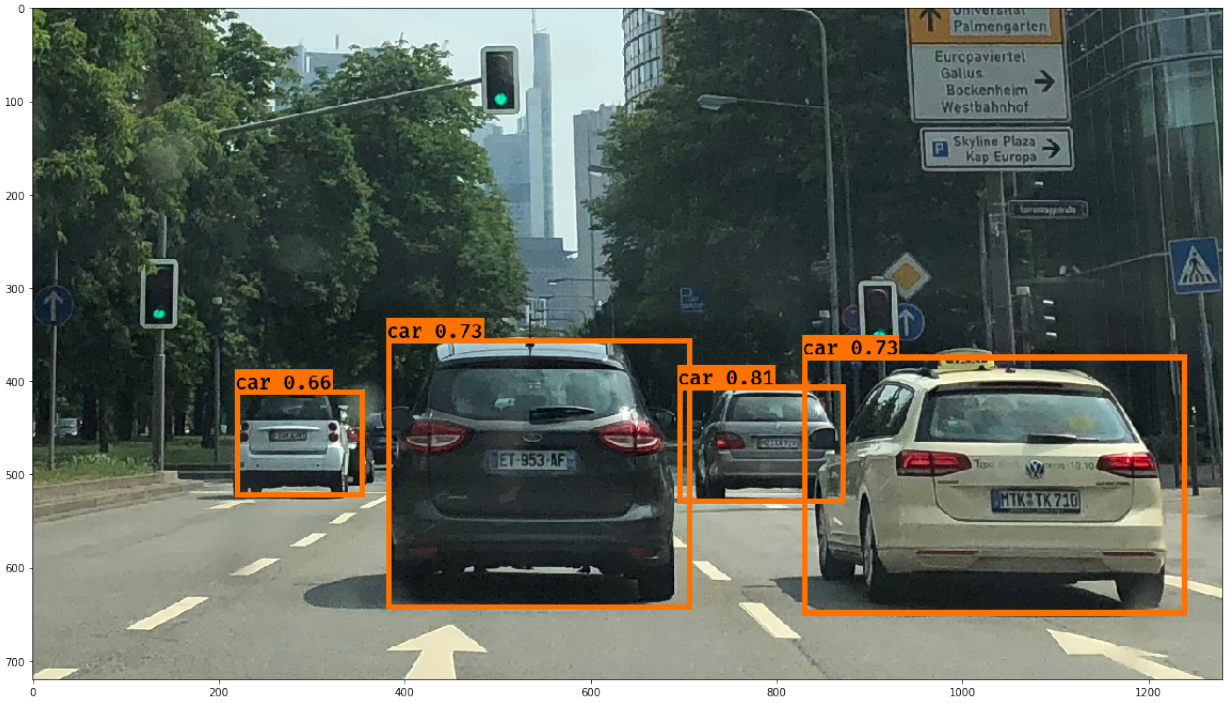
- image: a python (PIL) representation of your image used for drawing boxes. You won't need to use it.
- image_data: a numpy-array representing the image. This will be the input to the CNN.

```
def predict(sess, image, printlabel, scores, boxes, classes, class_names):
    """
    Runs the graph stored in "sess" to predict boxes for "image_file". Prints and plots
    the preditions.

    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 between 0
    and max_boxes.
    """

    ### Code Implementation ###

    return out_scores, out_boxes, out_classes, image
```

The output image shows below:

```
    out_scores, out_boxes, out_classes, output_image = predict(sess, image,True,scores, boxes, all_classes,class_names=classes)
    print('Found {} boxes for {}'.format(len(out_boxes), f))
    plt.imshow(output_image)
```

```
car 0.66 (218, 409) (357, 524)
car 0.73 (381, 354) (709, 645)
car 0.73 (828, 372) (1241, 651)
car 0.81 (694, 404) (874, 531)
Found 4 boxes for test_images/1.jpg
```




```
car 0.81 (1014, 408) (1209, 496)
car 0.81 (818, 419) (944, 496)
Found 2 boxes for test_images/test6.jpg
```

Conclusion:

- 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.
- I 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, I used previously trained model parameters in this exercise.

4. Discussion

There is one image that the vehicle detection is failed.

```
Found 0 boxes for test_images/test3.jpg
```

The car is very clear for human eyes, but the **YOLO** model in this case fails to detect it.

The reason may come from the loaded pertained model. The YOLO model is very computationally expensive to train, that's why I loaded from The official YOLO website.

The way to generate `.h5` file is explained in this github. What I did is as below:

1. Download `.cfg` and `.weights` of *YOLOv2* 608×608 from The official YOLO website
2. Open **Terminal** or **CMD** to run the following command (I defined the `.cfg` as `yolo.cfg`, same as `weights`):

    ```
    python3 yad2k.py yolo.cfg yolo.weights model_data/yolo.h5
    ```

3. Load `.h5` into the project

There are several models in the official website, some other models with input size adaption may give a better results.