DLS_C4_week3_Assignment 1 _SI

Autonomous driving application Car detection

jupyter Autonomous_driving_application_Car_detection 最后检查: 8 分钟前 (自动保存)

File Edit View Insert Cell Kernel Widgets Help 可信的 Python 3

Autonomous Driving - Car Detection

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

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

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

Important Note on Submission to the AutoGrader

Before submitting your assignment to the AutoGrader, please make sure you are not doing the following:

1. You have not added any *extra* `print` statement(s) in the assignment.
2. You have not added any *extra* code cell(s) in the assignment.
3. You have not changed any of the function parameters.
4. You are not using any global variables inside your graded exercises. Unless specifically instructed to do so, please refrain from it and use the local variables instead.
5. You are not changing the assignment code where it is not required, like creating *extra* variables.

If you do any of the following, you will get something like, `Grader not found` (or similarly unexpected) error upon submitting your assignment. Before asking for help/debugging the errors in your assignment, check for these first. If this is the case, and you don't remember the changes you have made, you can get a fresh copy of the assignment by following these instructions.

Table of Contents

Packages

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

In [1]:

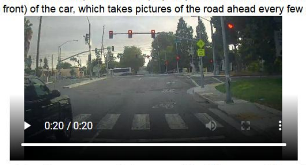
```
import argparse
import os
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow
import scipy.io
import scipy.misc
import numpy as np
import pandas as pd
import PIL
from PIL import ImageFont, ImageDraw, Image
import tensorflow as tf
from tensorflow.python.framework.ops import EagerTensor

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

%matplotlib inline
```

1 - Problem Statement

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



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

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

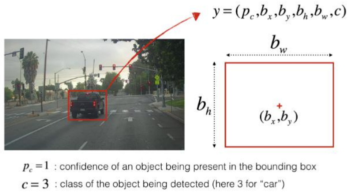


Figure 1: Definition of a box

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

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

2 - YOLO

"You Only Look Once" (YOLO) is a popular algorithm because it achieves high accuracy while also being able to run in real time. This algorithm "only looks once" at the image in the sense that it requires only one forward propagation pass through the network to make predictions. After non-max suppression, it then outputs recognized objects together with the bounding boxes.

2.1 - Model Details

Inputs and outputs

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

Anchor Boxes

- Anchor boxes are chosen by exploring the training data to choose reasonable height/width ratios that represent the different classes. For this assignment, 5 anchor boxes were chosen for you (to cover the 80 classes), and stored in the file './model_data/yolo_anchors.txt'
- The dimension of the encoding tensor of the second to last dimension based on the anchor boxes is $(m, n_H, n_W, anchors, classes)$.
- 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.

Encoding

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

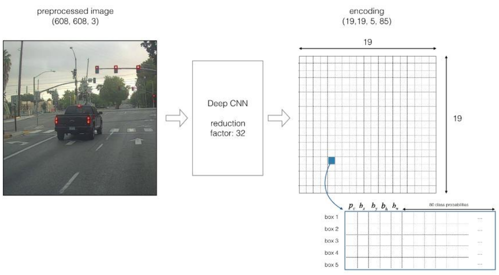


Figure 2 : Encoding architecture for YOLO

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

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

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

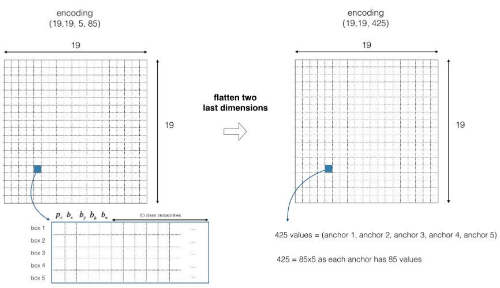


Figure 3 : Flattening the last two last dimensions

Class score

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

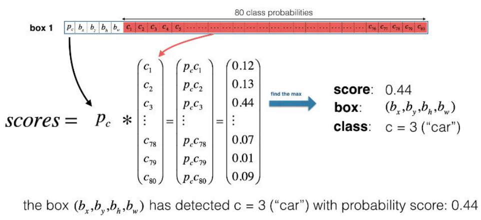


Figure 4: Find the class detected by each box

Example of figure 4

- In figure 4, let's say for box 1 (cell 1), the probability that an object exists is $p_1 = 0.60$. So there's a 60% chance that an object exists in box 1 (cell 1).
- The probability that the object is the class "category 3 (a car)" is $c_3 = 0.73$.
- The score for box 1 and for category "3" is $score_{1,3} = 0.60 \times 0.73 = 0.44$.
- Let's say you calculate the score for all 80 classes in box 1, and find that the score for the car class (class 3) is the maximum. So you'll assign the score 0.44 and class "3" to this box "1".

Visualizing classes

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

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

Doing this results in this picture:

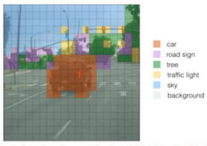


Figure 5: Each one of the 19x19 grid cells is colored according to which class has the largest predicted probability in that cell.

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

Visualizing bounding boxes

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

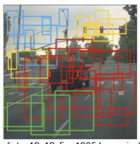


Figure 6: Each cell gives you 5 boxes. In total, the model predicts: 19x19x5 = 1805 boxes just by looking once at the image (one forward pass through the network)! Different colors denote different classes.

Non-Max suppression

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

To do so, you'll use **non-max suppression**. Specifically, you'll carry out these steps:

- Get rid of boxes with a low score. Meaning, the box is not very confident about detecting a class, either due to the low probability of any object, or low probability of this particular class.
- Select only one box when several boxes overlap with each other and detect the same object.

2.2 - Filtering with a Threshold on Class Scores

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

The model gives you a total of 19x19x5x85 numbers, with each box described by 85 numbers. It's convenient to rearrange the (19,19,5,85) (or (19,19,425)) dimensional tensor into the following variables:

- `box_confidence` : tensor of shape $(19, 19, 5, 1)$ containing p_c (confidence probability that there's some object) for each of the 5 boxes predicted in each of the 19x19 cells.
- `boxes` : tensor of shape $(19, 19, 5, 4)$ containing the midpoint and dimensions (b_x, b_y, b_h, b_w) for each of the 5 boxes in each cell.
- `box_class_probs` : tensor of shape $(19, 19, 5, 80)$ containing the "class probabilities" $(c_1, c_2, \dots c_{80})$ for each of the 80 classes for each of the 5 boxes per cell.

Exercise 1 - yolo_filter_boxes

Implement `yolo_filter_boxes()`.

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

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

   This is an example of **broadcasting** (multiplying vectors of different sizes).
2. For each box, find:
   - the index of the class with the maximum box score
   - the corresponding box score

   Useful References
   - tf.math.argmax
   - tf.math.reduce_max

   Helpful Hints
   - For the `axis` parameter of `argmax` and `reduce_max`, if you want to select the **last** axis, one way to do so is to set `axis=-1`. This is similar to Python array indexing, where you can select the last position of an array using `arrayname[-1]`.
   - Applying `reduce_max` normally collapses the axis for which the maximum is applied. `keepdims=False` is the default option, and allows that dimension to be removed. You don't need to keep the last dimension after applying the maximum here.
3. Create a mask by using a threshold. As a reminder: `([0.9, 0.3, 0.4, 0.5, 0.1] < 0.4)` returns: `[False, True, False, False, True]`. The mask should be `True` for the boxes you want to keep.
4. Use TensorFlow to apply the mask to `box_class_scores`, `boxes` and `box_classes` to filter out the boxes you don't want. You should be left with just the subset of boxes you want to keep.

   One more useful reference:
   - tf.boolean mask

   And one more helpful hint: :)
   - For the `tf.boolean_mask`, you can keep the default `axis=None`.

In [4]:
```
# UNQ_C1 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: yolo_filter_boxes

def yolo_filter_boxes(boxes, box_confidence, box_class_probs, threshold = .6):
```

File Edit View Insert Cell Kernel Widgets Help 可信的 Python 3

运行 Markdown

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).

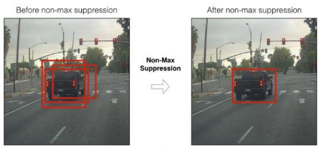


Figure 7 : In this example, the model has predicted 3 cars, but it's actually 3 predictions of the same car. Running non-max suppression (NMS) will select only the most accurate (highest probability) of the 3 boxes.

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

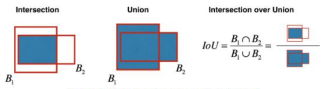


Figure 8 : Definition of "Intersection over Union".