2.7: The initial training phase

Our **previous lesson (https://gurus.pyimagesearch.com/lessons/preparing-your-training-data/)** discussed how to define an appropriate sliding window size and set our **HOG (https://gurus.pyimagesearch.com/lessons/histogram-of-oriented-gradients/)** parameters. Given our sliding window and HOG parameters, we then extracted HOG feature vectors from *both* our positive and negative training examples.

We are now going to take our dataset of extracted feature vectors (and associated labels) and utilize them to take a first pass at constructing an object detector (i.e, **Step 3** of our object detection framework).

This detector won't be perfect — we still have a ways to go, including *non-maxima suppression*, *hard-negative mining*, and *detector re-training*, but by the end of this, you ***will*** be able to see our classifier detecting cars in images.

Objectives:

In this lesson, we will:

- Utilize extracted HOG feature vectors and associated class labels to train a Linear SVM for object detection.

Initial training phase

At this point, we are ready to take a first pass at training our Linear SVM to detect objects in images. Let's start by opening up our `cars.json` configuration file and adding in a section called *"Linear SVM"*:

cars.json Shell

```
{
    /****
    * DATASET PATHS
    ****/
    "image_dataset": "datasets/caltech101/101_ObjectCategories/car_side",
    "image_annotations": "datasets/caltech101/Annotations/car_side",
    "image_distractions": "datasets/sceneclass13",

    /****
    * FEATURE EXTRACTION
    ****/
    "features_path": "output/cars/car_features.hdf5",
    "percent_gt_images": 0.5,
    "offset": 5,
    "use_flip": true,
    "num_distraction_images": 500,
    "num_distractions_per_image": 10,

    /****
    * HISTOGRAM OF ORIENTED GRADIENTS DESCRIPTOR
    ****/
    "orientations": 9,
    "pixels_per_cell": [4, 4],
    "cells_per_block": [2, 2],
    "normalize": true,

    /****
    * OBJECT DETECTOR
    ****/
    "window_step": 4,
    "overlap_thresh": 0.3,
    "pyramid_scale": 1.5,
    "window_dim": [96, 32],
    "min_probability": 0.7,

    /****
    * LINEAR SVM
    ****/
    "classifier_path": "output/cars/model.cpickle",
    "C": 0.01
}
```

The *"Linear SVM"* section can be found at the very bottom of the file. We have added two parameters for our classifier. The first is the output `classifier_path` — this is the path to where our classifier will be stored after training.

Also, take a look at the `C` parameter of the SVM. As you know from our **Support Vector Machine lesson (https://gurus.pyimagesearch.com/topic/support-vector-machines/)**, the coefficient `C` controls how "strict" the SVM is. Larger values of `C` indicate that the SVM is not allowed to make many

mistakes. While this can lead to higher accuracy on the training data, it causes *over-fitting* where the model does not generalize well to data points it has not seen before. Conversely, smaller values of `C` allow the SVM to make more mistakes — this is called a "soft-classifier". When training our own custom object detectors, we tend to use very small values of `C` , allowing our SVM to mis-classify regions, knowing that the downstream steps of non-maxima suppression and hard-negative mining can help rectify these mistakes.

Now that our configuration file has been updated, we are now ready to train our object detector:

train_model.py Python

```
1  # import the necessary packages
2  from __future__ import print_function
3  from pyimagesearch.utils import dataset
4  from pyimagesearch.utils import Conf
5  from sklearn.svm import SVC
6  import numpy as np
7  import argparse
8  import pickle
9
10 # construct the argument parser and parse the command line arguments
11 ap = argparse.ArgumentParser()
12 ap.add_argument("-c", "--conf", required=True,
13     help="path to the configuration file")
14 ap.add_argument("-n", "--hard-negatives", type=int, default=-1,
15     help="flag indicating whether or not hard negatives should be used")
16 args = vars(ap.parse_args())
```

On **Lines 2-8**, we import our necessary packages. We'll be using the `dataset` functions from the `utils` sub-module to aid us in loading our extracted feature vectors from disk.

Lines 11-16 parse our command line arguments. The only required argument is `--conf` , the path to our JSON configuration file. A second (optional) argument can be supplied, `--hard-negatives` , which indicates whether or not our extracted hard-negative feature vectors should be included in the training process. Since we do not have any hard-negatives, the value of this flag is set to `-1` ; however, once we review our lesson on hard-negative mining later in this module, all we need to do is specify a positive integer value and the hard-negatives will be used.

train_model.py Python

Feedback

```python
# load the configuration file and the initial dataset
print("[INFO] loading dataset...")
conf = Conf(args["conf"])
(data, labels) = dataset.load_dataset(conf["features_path"], "features")

# check to see if the hard negatives flag was supplied
if args["hard_negatives"] > 0:
    print("[INFO] loading hard negatives...")
    (hardData, hardLabels) = dataset.load_dataset(conf["features_path"], "hard_negatives")
    data = np.vstack([data, hardData])
    labels = np.hstack([labels, hardLabels])
```

On **Line 20**, we load our JSON configuration. **Line 21** utilizes the `load_dataset` function to grab the feature vectors and labels that we extracted in the **previous lesson (https://gurus.pyimagesearch.com/lessons/preparing-your-training-data/)**.

If a positive integer value for `--hard-negatives` has been supplied, we also load the hard-negative feature vectors and labels, appending them to their respective original lists (**Lines 24-28**). Again, we'll review hard-negative mining in detail later in this module. For the time being, simply ignore this section of code.

Now that our data is loaded, we can train our Linear SVM:

train_model.py — Python

```python
# train the classifier
print("[INFO] training classifier...")
model = SVC(kernel="linear", C=conf["C"], probability=True, random_state=42)
model.fit(data, labels)

# dump the classifier to file
print("[INFO] dumping classifier...")
f = open(conf["classifier_path"], "wb")
f.write(pickle.dumps(model))
f.close()
```

Line 32 handles the training of our Support Vector Machine. We indicate that we want to use a linear kernel, and that probabilities should be calculated when predicting new class labels. We'll also supply the coefficient `C` as specified in our configuration file.

Finally, **Lines 37-39** dump the trained classifier to file.

To train our object detector, just execute the following command:

train_model.py — Shell

```
$ python train_model.py --conf conf/cars.json
```

You'll now have a `model.cpickle` in your `output/cars` directory, as specified by our `cars.json` file. This `model.cpickle` file is our trained object detector.

But before we can apply our classifier to detect objects in images, we first need to define an `ObjectDetector` class to encapsulate:

1. Looping over all layers of the image pyramid.
2. Applying our sliding window at each layer of the pyramid.
3. Extracting HOG features from each window.
4. Passing the extracted HOG feature vectors to our model for classification.
5. Maintaining a list of bounding boxes that are reported to contain an object of interest with sufficient probability.

Surprisingly, we can accomplish all five of these goals in very few lines of code thanks to our pre-defined methods in the `helpers` sub-module.

Let's go ahead and implement our `ObjectDetector` class in the `object_detection` sub-module of our project:

objectdetector.py — Python

```
1 # import the necessary packages
2 from . import helpers
3
4 class ObjectDetector:
5     def __init__(self, model, desc):
6         # store the classifier and HOG descriptor
7         self.model = model
8         self.desc = desc
```

Line 2 imports the only package we'll need — our `helpers` sub-module.

Line 5 defines the constructor to our `ObjectDetector` class. This constructor requires two arguments. The first argument is the `model` , or our trained Linear SVM. The second argument is the instantiated `HOG` descriptor.

We are now ready to define `detect` , the heart of the `ObjectDetector` class:

objectdetector.py — Python

```
```

Feedback

```
    def detect(self, image, winDim, winStep=4, pyramidScale=1.5, minProb=0.7):
        # initialize the list of bounding boxes and associated probabilities
        boxes = []
        probs = []

        # loop over the image pyramid
        for layer in helpers.pyramid(image, scale=pyramidScale, minSize=winDim):
            # determine the current scale of the pyramid
            scale = image.shape[0] / float(layer.shape[0])

            # loop over the sliding windows for the current pyramid layer
            for (x, y, window) in helpers.sliding_window(layer, winStep, winDim):
                # grab the dimensions of the window
                (winH, winW) = window.shape[:2]

                # ensure the window dimensions match the supplied sliding window dimensions
                if winH == winDim[1] and winW == winDim[0]:
                    # extract HOG features from the current window and classify whether or
                    # not this window contains an object we are interested in
                    features = self.desc.describe(window).reshape(1, -1)
                    prob = self.model.predict_proba(features)[0][1]

                    # check to see if the classifier has found an object with sufficient
                    # probability
                    if prob > minProb:
                        # compute the (x, y)-coordinates of the bounding box using the current
                        # scale of the image pyramid
                        (startX, startY) = (int(scale * x), int(scale * y))
                        endX = int(startX + (scale * winW))
                        endY = int(startY + (scale * winH))

                        # update the list of bounding boxes and probabilities
                        boxes.append((startX, startY, endX, endY))
                        probs.append(prob)

        # return a tuple of the bounding boxes and probabilities
        return (boxes, probs)
```


The `detect` method requires two parameters and three optional ones. The first parameter, `image`, is the image we are trying to detect objects in. Then we have `winDim`, which are the dimensions of our **sliding window (https://gurus.pyimagesearch.com/topic/sliding-windows/)**.

We can also supply values of `winStep`, which is the *step size* of the sliding window. The `pyramidScale` controls the scale of our **image pyramid (https://gurus.pyimagesearch.com/topic/image-pyramids/)**. Larger values will result in *less* layers of the pyramid being evaluated, while smaller values will *increase* the number of layers in the pyramid.

Finally, `minProb` is the minimum probability returned from our SVM for a region to be considered "containing an object of interest".

Now that the `detect` function signature is defined, we initialize two lists on **Lines 12 and 13**: `boxes` and `probs`. The `boxes` list will contain the bounding boxes of objects in our image, while the `probs` list will store the `boxes`' associated probabilities.

Line 16 starts looping over the layers in our image pyramid. Then, **Line 21** applies a sliding window to each layer in the pyramid.

Line 26 makes a check to ensure the dimensions of the current window match our desired window dimensions supplied in the `detect` function signature. If the dimensions do not match up (e.g., when we are examining the boundaries of the image), we simply ignore the window.

Provided that the window dimensions do match up, **Lines 29 and 30** extract HOG features from the window and pass the feature vector on to our classifier, where we predict the probability of the class label.

In this case, the `predict_proba` method will return a 2-tuple containing the probability of: (1) the window *not containing* an object of interest and (2) the window *actually containing* an object of interest. Since we are only interested in case #2, we grab the second value from the 2-tuple.

Given the probability, **Line 34** checks to see if the probability returned by the classifier is greater than the supplied `minProb`. If the probability is sufficiently larger, then we can use the bounding box coordinates of the sliding window along with the current scale of the image pyramid to obtain the bounding box *with respect to the original image size*. The `boxes` and `probs` lists are then updated on **Lines 42 and 43**.

We then return a tuple of the bounding boxes and associated probabilities to the calling function.

Now it's finally time to put all the pieces together and use our `ObjectDetector` class to scan and detect the presence of objects in images:

test_model_no_nms.py — Python

```python
# import the necessary packages
from pyimagesearch.object_detection import ObjectDetector
from pyimagesearch.descriptors import HOG
from pyimagesearch.utils import Conf
import imutils
import argparse
import pickle
import cv2

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-c", "--conf", required=True, help="path to the configuration file")
ap.add_argument("-i", "--image", required=True, help="path to the image to be classified")
args = vars(ap.parse_args())

# load the configuration file
conf = Conf(args["conf"])
```

Lines 2-8 simply import our required packages. **Lines 11-14** parse our command line arguments. We only need two arguments here, the path to our configuration file (`--conf`) and the path to the `--image` we want to detect objects in.

test_model_no_nms.py — Python

```python
# load the classifier, then initialize the Histogram of Oriented Gradients descriptor
# and the object detector
model = pickle.loads(open(conf["classifier_path"], "rb").read())
hog = HOG(orientations=conf["orientations"], pixelsPerCell=tuple(conf["pixels_per_cell"]),
    cellsPerBlock=tuple(conf["cells_per_block"]), normalize=conf["normalize"])
od = ObjectDetector(model, hog)
```

Here, we start by loading our classifier from disk on **Line 21**. The HOG descriptor is then instantiated using the parameters supplied from our configuration file (**Lines 22 and 23**). Now that both our `model` and `hog` descriptor have been loaded and configured, we can construct our `ObjectDetector` by passing in the classifier and descriptor.

The last code block handles perform the actual object detection and draws the results on our image:

test_model_no_nms.py — Python

```
26 # load the image and convert it to grayscale
27 image = cv2.imread(args["image"])
28 image = imutils.resize(image, width=min(260, image.shape[1]))
29 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
30
31 # detect objects in the image
32 (boxes, probs) = od.detect(gray, conf["window_dim"], winStep=conf["window_step"],
33     pyramidScale=conf["pyramid_scale"], minProb=conf["min_probability"])
34
35 # loop over the bounding boxes and draw them
36 for (startX, startY, endX, endY) in boxes:
37     cv2.rectangle(image, (startX, startY), (endX, endY), (0, 0, 255), 2)
38
39 # show the output images
40 cv2.imshow("Image", image)
41 cv2.waitKey(0)
```

Lines 27-29 simply load our image, resize it to have a maximum width of 260px (to speed up detection time), and convert our image to grayscale.

Now that our image has been pre-processed, we can detect objects in the image by making a call to the `detect` method of the `ObjectDetector` class. Notice how all arguments from the `detect` method are defined and specified inside our `cars.json` file. According to our JSON configuration file, we'll be using a `window_step` of four pixels, a `pyramid_scale` of 1.5, a `window_dim` of *96 x 32* pixels, and a `min_probability` of 0.7, implying that our classifier must be 70% certain that a given bounding box contains an object of interest.

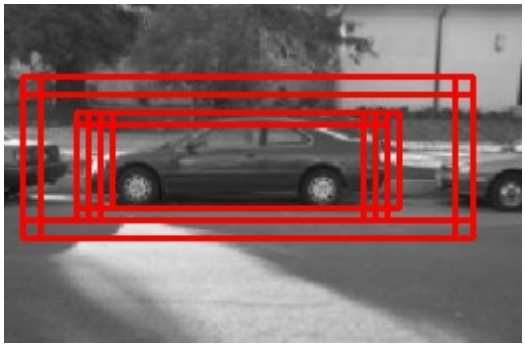
The `detect` method returns a 2-tuple, consisting of the bounding box locations (and associated probabilities) of objects in our images.

Given these boxes, we loop over them, draw them on our `image`, and display the output to our screen (**Lines 32-41**).

To test our object detector, use the following command:

test_model_no_nms.py — Python

```
1 $ python test_model_no_nms.py --conf conf/cars.json \
2     --image datasets/caltech101/101_ObjectCategories/car_side/image_0004.jpg
```

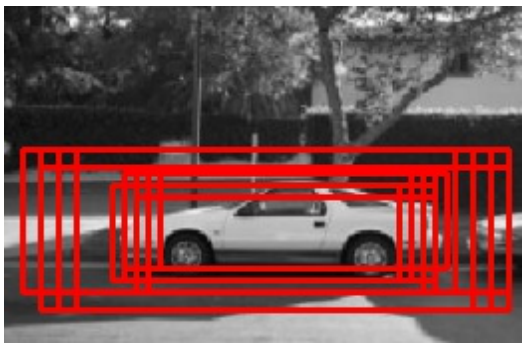


(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/image_0004.jpg)

FIGURE 1: APPLYING OUR CUSTOM OBJECT DETECTOR — NOTICE HOW WE HAVE DETECTED MULTIPLE, OVERLAPPING BOUNDING BOXES SURROUNDING THE CAR.

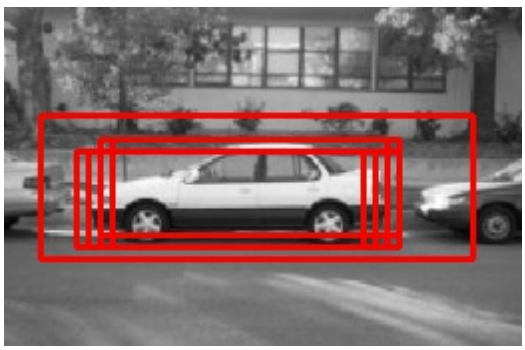
Notice how ***multiple*** bounding boxes have been detected, but there is only *one* car in the image.

The same is true for the following three images:



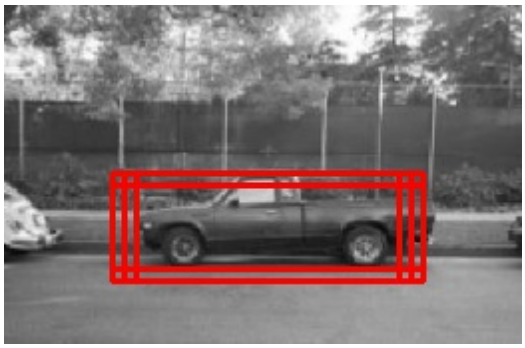
(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/image_0012.jpg)

FIGURE 2: WE HAVE ALSO DETECTED MULTIPLE BOUNDING BOXES SURROUNDING THIS CAR AS WELL.



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/image_0009.jpg)

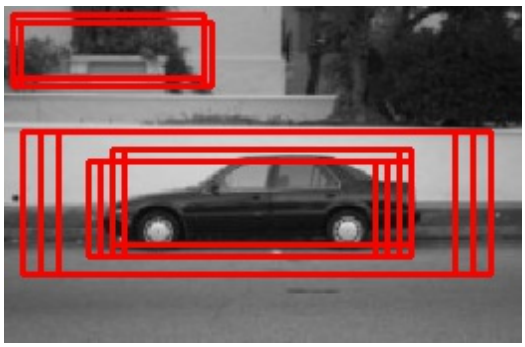
FIGURE 3: AGAIN, NOTICE THAT EVEN THOUGH THERE IS ONLY ONE CAR IN THE IMAGE, MULTIPLE BOUNDING BOXES ARE RETURNED.



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/image_0020.jpg)

FIGURE 4: A FINAL EXAMPLE OF THE MULTIPLE, OVERLAPPING BOUNDING BOX PROBLEM.

There are also cases where our detector misfires and classifies an image region as a car, when in reality, it is not:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/initial_training_phase_image_0060.jpg)

FIGURE 5: AN EXAMPLE OF A FALSE-POSITIVE DETECTION. OUR OBJECT DETECTOR HAS REPORTED THE REGIONS OUTLINED IN RED AT THE TOP-RIGHT CORNER AS CONTAINING A CAR, BUT IN REALITY, IT DOES NOT.

Does this mean our object detector isn't working properly?

Of course not! In the case of multiple, overlapping bounding boxes, this is actually a *good thing*. This behavior demonstrates that our classifier is capable of detecting car-like regions in images. Intuitively, this makes sense, since each step along a sliding window path only differs by a few pixels in size. Simply put, each *(x, y)*-coordinate surrounding the boundary of a car can be considered a *positive classification* of a car region. Thus, it makes sense that our detector is reporting *multiple bounding boxes* surrounding the car.

To compress these multiple, overlapping bounding boxes into a single bounding box, all we need to do is apply non-maxima suppression, which we'll cover in the next lesson.

As for the false-positive detections, we can remedy this problem by using hard-negative mining.

Overall, our classifier performed quite well; we just need to tidy up the results a bit.

Summary

In this lesson, we learned how to train a Linear SVM on top of our HOG features to create our object detector. We also defined an `ObjectDetector` class to encapsulate the process of constructing an image pyramid, applying sliding windows, extracting features, and obtaining bounding boxes for candidate objects in an image.

Given our trained Linear SVM, we then applied it to a few test images in our car dataset. In many cases, our object detector performed quite well. But as we saw, most images contain *multiple*, *overlapping* bounding boxes. While this implies that our object detector is working well, we still need a method to compress these multiple bounding boxes into a *single* bounding box representing the object in an image. To accomplish this, we'll apply non-maxima suppression, which is the exact topic of our next lesson.

Finally, you may have noticed there were times that our object detector misfired and falsely reported a location as containing a car where it clearly did not. To remedy this, we'll need to apply a technique called *hard-negative mining*, which we'll cover later in this module.

Downloads:

Download the Code (https://gurus.pyimagesearch.com/protected/code/object_detector/initial_trainin

Quizzes		Status
1	The Initial Training Phase Quiz (https://gurus.pyimagesearch.com/quizzes/the-initial-training-phase-quiz/)	

← Previous Lesson (https://gurus.pyimagesearch.com/lessons/constructing-your-hog-descriptor/) Next Lesson → (https://gurus.pyimagesearch.com/lessons/non-maxima-suppression/)

Course Progress

Ready to continue the course?

Click the button below to **continue your journey to computer vision guru**.

I'm ready, let's go! (/pyimagesearch-gurus-course/)

Resources & Links

- PyImageSearch Gurus Community (https://community.pyimagesearch.com/)
- PyImageSearch Virtual Machine (https://gurus.pyimagesearch.com/pyimagesearch-virtual-machine/)
- Setting up your own Python + OpenCV environment (https://gurus.pyimagesearch.com/setting-up-your-python-opencv-development-environment/)
- Course Syllabus & Content Release Schedule (https://gurus.pyimagesearch.com/course-syllabus-content-release-schedule/)
- Member Perks & Discounts (https://gurus.pyimagesearch.com/pyimagesearch-gurus-discounts-perks/)
- Your Achievements (https://gurus.pyimagesearch.com/achievements/)
- Official OpenCV documentation (http://docs.opencv.org/index.html)

Your Account

- Account Info (https://gurus.pyimagesearch.com/account/)
- Support (https://gurus.pyimagesearch.com/contact/)
- Logout (https://gurus.pyimagesearch.com/wp-login.php?action=logout&redirect_to=https%3A%2F%2Fgurus.pyimagesearch.com%2F&_wpnonce=5736b21cae)

Search

Feedback