2.6: Constructing your HOG descriptor

In our **previous lesson (https://gurus.pyimagesearch.com/lessons/preparing-your-training-data/)**, we explored the components of our object detection framework. We also learned how to determine the average width, average height, and aspect ratio of a collection of objects based on their supplied annotations.

But the question still remains: *"How do we use these dimensions and aspect ratio to construct our sliding window size?"*

The first part of this lesson will address the window dimension problem. The second half will then detail how to implement **Step 2** of our object detection framework: **HOG feature extraction from positive and negative samples.**

Objectives:

In this lesson, we will:

- Take our average object dimensions and use them to choose appropriate HOG descriptor dimensions along with sliding window size.
- Extract HOG features from our positive and negative image example training images.

Choosing sliding window dimensions and appropriate HOG parameters

In our previous lesson on **preparing our experiment and training data (https://gurus.pyimagesearch.com/lessons/preparing-your-training-data/)**, we discovered that our car side views *annotations dataset* (i.e., car bounding boxes) have an average width of 184px, an average height of 62px, and an aspect ratio of 2.97, implying that the width of the side view of a car is approximately 3x larger than the height.

We now need to take these dimensions and use them to define an appropriate sliding window size, along with the associated HOG descriptor parameters.

Defining sliding window dimensions and HOG parameters is a bit of a "black art" that takes much practice, persistence, and noting what does (and more importantly, what does not) work. That said, the first thing I always do when constructing an object detector is take the average width and height and ***divide them by two***, respectively.

This leaves us with a width of 92px and a height of 31px.

So why did I divide by two?

Primarily for two reasons:

- **Reason #1:** HOG feature vectors can grow to be *extremely large* as window size increases, so it's important to strike a balance between window size, feature vector length, and discriminability of the feature vector.
- **Reason #2:** To aid in multi-scale detection. If my window size is too large, I can easily miss out on smaller objects. Conversely, if my window size is too small, I waste billions of CPU cycles on ROIs far too small to contain an object of interest — this also implies that an object can only be found at the smaller scales of an image pyramid (which, again, leads to wasted CPU cycles). By dividing by two, I strike a nice balance between a reasonable window size and feature vector length, while (1) ensuring I am not carelessly evaluating a multitude of ROIs and (2) relying on my image pyramid to catch objects that are either smaller or larger than my window dimensions.

Now that I have an adjusted width and height of 92px and 31px respectively, I can move on to defining my HOG parameters.

The two most important parameters to consider in our HOG descriptor are `pixels_per_cell` and `cells_per_block` — *our sliding window* ***dimensions*** *must be divisible by these values*. If the sliding window dimensions *are not* divisible by `pixels_per_cell` and `cells_per_block`, then the HOG descriptor will

not “fit” into the window size.

Again, tuning HOG parameters is a bit of a black art that you’ll acquire with practice, but it’s very common to have your `pixels_per_cell` be a multiple of four. Your `cells_per_block` is also *almost always* in the set *{1, 2, 3}*. You may find that multiples of another integer work better (such as with our **car logo recognition lesson (https://gurus.pyimagesearch.com/lessons/histogram-of-oriented-gradients/)** where we used `pixels_per_cell=(10, 10)`) — and that’s perfectly okay. Just keep in mind the relation of your window size to `pixels_per_cell` , and you’ll be okay.

For our car detection example, I decided to start with a `pixels_per_cell=(4, 4)` and `cells_per_block=(2, 2)` — these tend to be good, common first parameter choices when building an object detector.

This now implies that *both* my sliding window width and sliding window height must be divisible by four. To accomplish this, I’ll round 92px and 31px to the nearest multiple of four, which gives me 96px and 32px, respectively. Examining the aspect ratio, I see that *96 / 32 = 3*, which is conveniently close to the aspect ratio of the average dimensions of our car training data (*2.97*).

Let’s update our `cars.json` configuration file to include our newly derived sliding window dimensions and HOG parameters:

cars.json JavaScript

```
{
    /****
    * 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
}
```

Feedback

As you can see, our `cars.json` file has greatly expanded. Ignore most of it for now and concentrate on **Line 33** where we have defined our sliding window size to be *(96, 32)*.

From there, examine **Lines 22-25**, the *"Histogram of Oriented Gradients"* section. We'll be using `pixels_per_cell=(4, 4)`, `cells_per_block=(2, 2)`, 9 orientations per histogram, and square-root normalization applied to the image prior to description.

Now that our sliding window dimensions and HOG parameters have been set, let's move on to the actual process of feature extraction.

HOG feature extraction

The first thing we should do before getting too far into feature extraction is define a few helper methods.

The first two, `dump_dataset` and `load_dataset`, will live in the `dataset.py` of the `utils` sub-package:

dataset.py Python

```python
# import the necessary packages
import numpy as np
import h5py

def dump_dataset(data, labels, path, datasetName, writeMethod="w"):
    # open the database, create the dataset, write the data and labels to dataset,
    # and then close the database
    db = h5py.File(path, writeMethod)
    dataset = db.create_dataset(datasetName, (len(data), len(data[0]) + 1), dtype="float")
    dataset[0:len(data)] = np.c_[labels, data]
    db.close()

def load_dataset(path, datasetName):
    # open the database, grab the labels and data, then close the dataset
    db = h5py.File(path, "r")
    (labels, data) = (db[datasetName][:, 0], db[datasetName][:, 1:])
    db.close()

    # return a tuple of the data and labels
    return (data, labels)
```

As the function names suggest, these methods allow us to quickly and easily load feature vectors and labels from a dataset residing on disk. We'll be using HDF5 and the `h5py` library to facilitate efficient data storage, so if you haven't read the Content-Based Image Retrieval lesson on **feature extraction and indexing (https://gurus.pyimagesearch.com/lessons/extracting-keypoints-and-local-invariant-descriptors/)**, you'll want to stop and go do that now.

Line 5 defines our `dump_dataset` function. This method requires four parameters, followed by an optional fifth argument:

- `data` : The list of feature vectors that will be written to the HDF5 dataset.
- `labels` : The list of labels associated with each feature vector. The label values will be contained in *{-1, 1}*, where *-1* indicates that the feature vector *is not representative* of the object we want to detect, and a value of *1* indicates the feature vector *is representative* of the object we want to detect.
- `path` : This is the path to where our HDF5 dataset will be stored on disk.
- `datasetName` : The name of the dataset within the HDF5 file.
- `writeMethod` : This parameter is entirely optional — it is simply the write mode of the file. We specify a value of `w` here by default, indicating that the database should be opened for writing. Later in this module, we'll supply a value of `a` , allowing us to append hard-negative features to the dataset.

Line 13 then defines our `load_dataset` method. This function simply loads the feature vectors and labels associated with `datasetName` .

Let's define another convenience function, `crop_ct101_bb` in `helpers.py` of the `object_detection` sub-module:

helpers.py Python

```python
# import the necessary packages
import imutils
import cv2

def crop_ct101_bb(image, bb, padding=10, dstSize=(32, 32)):
    # unpack the bounding box, extract the ROI from the image, while taking into account
    # the supplied offset
    (y, h, x, w) = bb
    (x, y) = (max(x - padding, 0), max(y - padding, 0))
    roi = image[y:h + padding, x:w + padding]

    # resize the ROI to the desired destination size
    roi = cv2.resize(roi, dstSize, interpolation=cv2.INTER_AREA)

    # return the ROI
    return roi
```

As mentioned in our previous lesson, our object detection framework will work out of the box on any of the 101 categories from the CALTECH-101 dataset. The `crop_ct101_bb` method allows us to easily take an input image (along with its associated bounding box) and crop the ROI to a fixed width and height. We can also specify optional padding surrounding the ROI.

While optional, adding padding prior to cropping allows us to add "context" to the ROI. As the work of Dalal and Triggs (https://gurus.pyimagesearch.com/wp-content/uploads/2015/05/dalal_2005.pdf) suggests, adding a bit of padding can actually *increase* the accuracy of our HOG detector.

Now we are ready to put all the pieces together in `extract_features.py` :

extract_features.py Python

Feedback

```python
# import the necessary packages
from __future__ import print_function
from sklearn.feature_extraction.image import extract_patches_2d
from pyimagesearch.object_detection import helpers
from pyimagesearch.descriptors import HOG
from pyimagesearch.utils import dataset
from pyimagesearch.utils import Conf
from imutils import paths
from scipy import io
import numpy as np
import progressbar
import argparse
import random
import cv2

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

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

Lines 1-14 handle importing our necessary packages. We then parse our command line arguments on **Lines 17-19**. We'll need only a single switch here, `--conf`, which is the path to our JSON configuration file. Finally, **Line 22** loads our configuration file from disk.

extract_features.py Python

```python
# initialize the HOG descriptor along with the list of data and labels
hog = HOG(orientations=conf["orientations"], pixelsPerCell=tuple(conf["pixels_per_cell"]),
    cellsPerBlock=tuple(conf["cells_per_block"]), normalize=conf["normalize"])
data = []
labels = []

# grab the set of ground-truth images and select a percentage of them for training
trnPaths = list(paths.list_images(conf["image_dataset"]))
trnPaths = random.sample(trnPaths, int(len(trnPaths) * conf["percent_gt_images"]))
print("[INFO] describing training ROIs...")
```

Here, we start off by initializing our HOG descriptor on **Lines 25 and 26**. Our HOG descriptor is simply a wrapper around the scikit-learn implementation, only in *class* rather than *function* form. We also initialize our list of feature vectors and associated training labels on **Lines 27 and 28**.

Lines 31 and 32 randomly sample car side views for training data. Based on the contents of our `cars.json` file, we'll sample 50% of the data for training and leave the rest for testing.

Now we are ready to extract features from our positive training examples:

extract_features.py Python

Feedback

```
35 # set up the progress bar
36 widgets = ["Extracting: ", progressbar.Percentage(), " ", progressbar.Bar(), " ", progressbar.ET
37 pbar = progressbar.ProgressBar(maxval=len(trnPaths), widgets=widgets).start()
38
39 # loop over the training paths
40 for (i, trnPath) in enumerate(trnPaths):
41     # load the image, convert it to grayscale, and extract the image ID from the path
42     image = cv2.imread(trnPath)
43     image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
44     imageID = trnPath[trnPath.rfind("_") + 1:].replace(".jpg", "")
45
46     # load the annotation file associated with the image and extract the bounding box
47     p = "{}/annotation_{}.mat".format(conf["image_annotations"], imageID)
48     bb = io.loadmat(p)["box_coord"][0]
49     roi = helpers.crop_ct101_bb(image, bb, padding=conf["offset"], dstSize=tuple(conf["window_di
50
51     # define the list of ROIs that will be described, based on whether or not the
52     # horizontal flip of the image should be used
53     rois = (roi, cv2.flip(roi, 1)) if conf["use_flip"] else (roi,)
54
55     # loop over the ROIs
56     for roi in rois:
57         # extract features from the ROI and update the list of features and labels
58         features = hog.describe(roi)
59         data.append(features)
60         labels.append(1)
61
62     # update the progress bar
63     pbar.update(i)
```

We start looping over each of our training images on **Line 40**. **Lines 41-44** then load the image from disk, convert it to grayscale, and extract the filename from the image path.

However, we also need to load the bounding box annotations for the current image, which we do in **Lines 47 and 48**. A call to `crop_ct101_bb` conveniently crops the bounding box ROI from our image and resizes the ROI to a fixed, canonical size (in this case, *96 x 32* pixels, as specified by our `cars.json` file).

Line 53 determines if we should use the *horizontal flip* of the ROI as additional training data. For many objects, using the horizontal flip of an ROI is very beneficial, since it can be used as additional training data. As we can see, a horizontal mirror of a car is still a car:

(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/constructing_hog_descriptor_horizontal_flip.png)

FIGURE 1: A HORIZONTAL MIRROR OF A CAR STILL LOOKS LIKE A CAR, AND THUS WE CAN USE THE HORIZONTAL FLIP AS ADDITIONAL TRAINING DATA.

However, a vertical flip of a car is not a car (unless we wanted to detect cars that were upside-down):

(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/constructing_hog_descriptor_vertical_flip.png)

FIGURE 2: HOWEVER, A VERTICAL FLIP OF OUR IMAGE DOES NOT RESEMBLE THE SEMANTIC CONTEXT IN WHICH WE ACCEPT TO SEE CARS.

Lines 56-60 then loop over the `rois`, extract HOG features from them, and update the `data` and `labels` lists, respectively.

The last step in HOG feature extraction is to describe our *negative image samples*:

`extract_features.py` Python

```
65 # grab the distraction image paths and reset the progress bar
66 pbar.finish()
67 dstPaths = list(paths.list_images(conf["image_distractions"]))
68 pbar = progressbar.ProgressBar(maxval=conf["num_distraction_images"], widgets=widgets).start()
69 print("[INFO] describing distraction ROIs...")
70
71 # loop over the desired number of distraction images
72 for i in np.arange(0, conf["num_distraction_images"]):
73     # randomly select a distraction image, load it, convert it to grayscale, and
74     # then extract random patches from the image
75     image = cv2.imread(random.choice(dstPaths))
76     image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
77     patches = extract_patches_2d(image, tuple(conf["window_dim"]),
78         max_patches=conf["num_distractions_per_image"])
79
80     # loop over the patches
81     for patch in patches:
82         # extract features from the patch, then update the data and label list
83         features = hog.describe(patch)
84         data.append(features)
85         labels.append(-1)
86
87     # update the progress bar
88     pbar.update(i)
```

We start off on **Line 72** by looping over the desired number of distraction images (in this case, *500* images, as specified in our `cars.json` file). We then randomly choose a distraction image, load it from disk, and use the `extract_patches_2d` function to randomly sample patches from the image. We sample *10* random ROIs from the image, resizing them to a fixed size of *96 x 32* pixels, the same as our training data.

Note: The `extract_patches_2d` function is a convenient function for random ROI sampling implemented in the scikit-learn library. **Click here to read more about it. (http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.image.extract_patches_2d.html)**

Once we have our randomly sampled patches, we loop over them (**Line 81**), extract HOG features (**Line 83**), and update our `data` and `labels` lists (**Lines 84 and 85**). Notice how we supply a value of *1* for *positive image examples* and a value of *-1* for *negative image examples*. It's important that we have different label values for these two classes, as we'll later need to train a Linear SVM to discriminate between positive and negative image patches.

All that's left to do now is dump our dataset to disk:

extract_features.py — Python

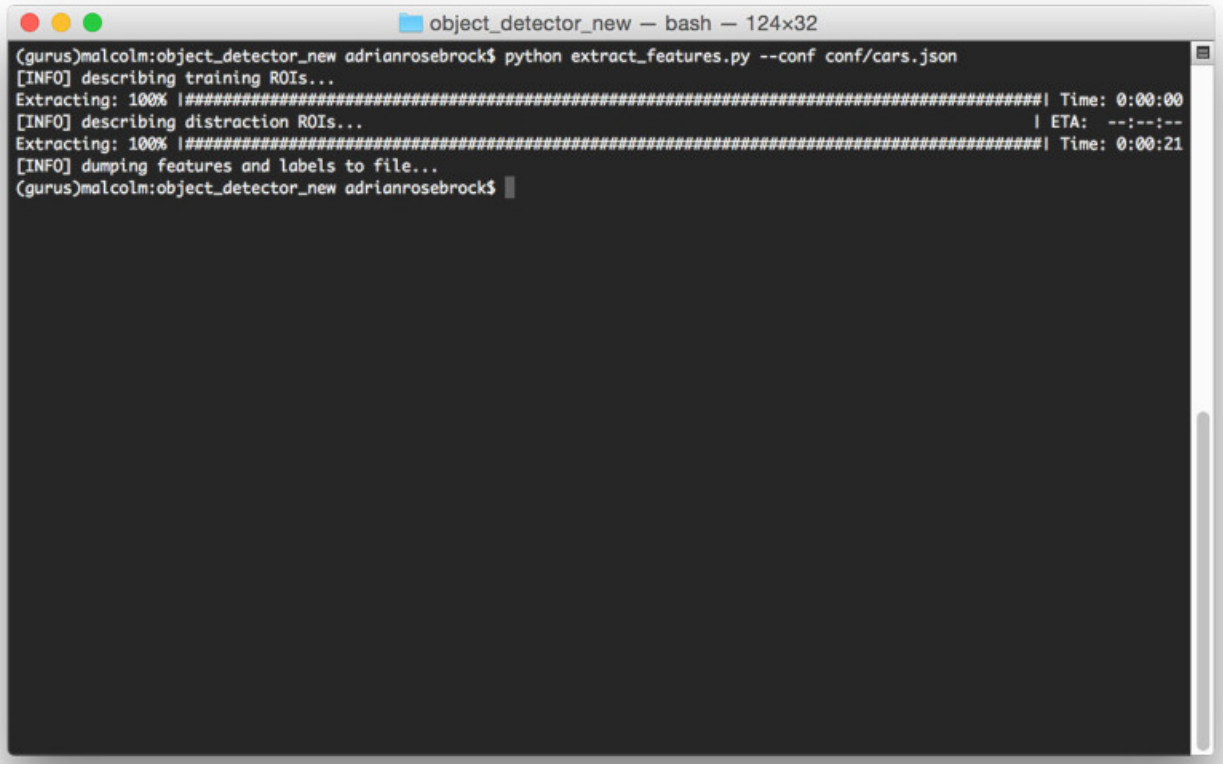
```
90 # dump the dataset to file
91 pbar.finish()
92 print("[INFO] dumping features and labels to file...")
93 dataset.dump_dataset(data, labels, conf["features_path"], "features")
```

Here, we dump our `data` and `labels` to an HDF5 dataset named *"features"* stored in `output/cars/car_features.hdf5`.

To execute our `extract_features.py` script, just issue the following command:

extract_features.py — Shell

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



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

FIGURE 3: AFTER EXECUTING OUR COMMAND, WE NOW HAVE A DATABASE OF HOG FEATURE VECTORS EXTRACTED FROM OUR DATASET OF POSITIVE AND NEGATIVE IMAGES.

Inside the `output/cars directory`, we'll see our `car_features.hdf5` database:

Extracting HOG features — Shell

```
$ ls output/cars/
car_features.hdf5
```

Let's fire up a Python shell and explore the features:

Exploring our HOG features — Shell

```
$ cd output/cars
$ python
>>> import h5py
>>> db = h5py.File("car_features.hdf5")
>>> list(db.keys())
['features']
>>> db["features"].shape
(5122, 5797)
>>>
```

Here, we can see that 5,122 HOG feature vectors have been extracted from our positive and negative training set.

The *first* entry in each row is the class label, while the *remaining entries* constitute the HOG feature vector:

Exploring our HOG features — Shell

```
>>> row = db["features"][0]
>>> (label, features) = (row[0], row[1:])
>>> label
1.0
>>> features.shape
(5796,)
>>>
```

In our next lesson, we'll move on to **Step 3** of our object detection framework — making a first pass at training our object detector.

Feedback

Summary

In this lesson, we used our average object dimensions to define Histogram of Oriented Gradients parameters, along with an optimal sliding window size. Given these parameters, we updated our `cars.json` configuration file to reflect these definitions. Hopefully by now, you can see the benefit of using a configuration file — imagine having to specify all these parameters as command line arguments!

From there, we defined `extract_features.py`, a Python script to extract HOG features from *both* our positive and negative training examples. We used 50% of our car side view images for training, followed by extracting HOG features from *500 x 10 = 5,000* negative training ROIs. These feature vectors and associated labels were then dumped to disk inside an HDF5 dataset.

The next step in our object detection framework is to take our extracted feature vectors and train a Linear SVM on top of them. This will serve as our "first pass" for our custom object detector. There will still be lots of room for improvement, but after our next lesson, you'll be able to see a fully functioning object detector used to detect the presence of cars in images.

Downloads:

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

Download the Scene Class 13 Dataset (https://gurus.pyimagesearch.com/protected/code/object_detector/sceneclass13

	Quizzes	Status
1	Constructing your HOG Descriptor Quiz (https://gurus.pyimagesearch.com/quizzes/constructing-your-hog-descriptor-quiz/)	

← Previous Lesson (https://gurus.pyimagesearch.com/lessons/preparing-your-training-data/) Next Lesson → (https://gurus.pyimagesearch.com/lessons/the-initial-training-phase/)

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

/