[**l'Log**](https://lilianweng.github.io/)

* [Posts](https://lilianweng.github.io/)
* [Archive](https://lilianweng.github.io/archives)
* [Search](https://lilianweng.github.io/search/)
* [Tags](https://lilianweng.github.io/tags/)
* [FAQ](https://lilianweng.github.io/faq)
* [emojisearch.app](https://www.emojisearch.app/)

**Object Detection for Dummies Part 1: Gradient Vector, HOG, and SS**

October 29, 2017 · 15 min · Lilian Weng

Table of Contents

I’ve never worked in the field of computer vision and has no idea how the magic could work when an autonomous car is configured to tell apart a stop sign from a pedestrian in a red hat. To motivate myself to look into the maths behind object recognition and detection algorithms, I’m writing a few posts on this topic “Object Detection for Dummies”. This post, part 1, starts with super rudimentary concepts in image processing and a few methods for image segmentation. Nothing related to deep neural networks yet. Deep learning models for object detection and recognition will be discussed in [Part 2](https://lilianweng.github.io/posts/2017-12-15-object-recognition-part-2/) and [Part 3](https://lilianweng.github.io/posts/2017-12-31-object-recognition-part-3/).

Disclaimer: When I started, I was using “object recognition” and “object detection” interchangeably. I don’t think they are the same: the former is more about telling whether an object exists in an image while the latter needs to spot where the object is. However, they are highly related and many object recognition algorithms lay the foundation for detection.

Links to all the posts in the series: [[Part 1](https://lilianweng.github.io/posts/2017-10-29-object-recognition-part-1/)] [[Part 2](https://lilianweng.github.io/posts/2017-12-15-object-recognition-part-2/)] [[Part 3](https://lilianweng.github.io/posts/2017-12-31-object-recognition-part-3/)] [[Part 4](https://lilianweng.github.io/posts/2018-12-27-object-recognition-part-4/)].

**Image Gradient Vector**

First of all, I would like to make sure we can distinguish the following terms. They are very similar, closely related, but not exactly the same.

|  | **Derivative** | **Directional Derivative** | **Gradient** |
| --- | --- | --- | --- |
| Value type | Scalar | Scalar | Vector |
| Definition | The rate of change of a function �(�,�,�,…) at a point (�0,�0,�0,…), which is the slope of the tangent line at the point. | The instantaneous rate of change of �(�,�,�,…) in the direction of an unit vector �→. | It points in the direction of the greatest rate of increase of the function, containing all the partial derivative information of a multivariable function. |

In the image processing, we want to know the direction of colors changing from one extreme to the other (i.e. black to white on a grayscale image). Therefore, we want to measure “gradient” on pixels of colors. The gradient on an image is discrete because each pixel is independent and cannot be further split.

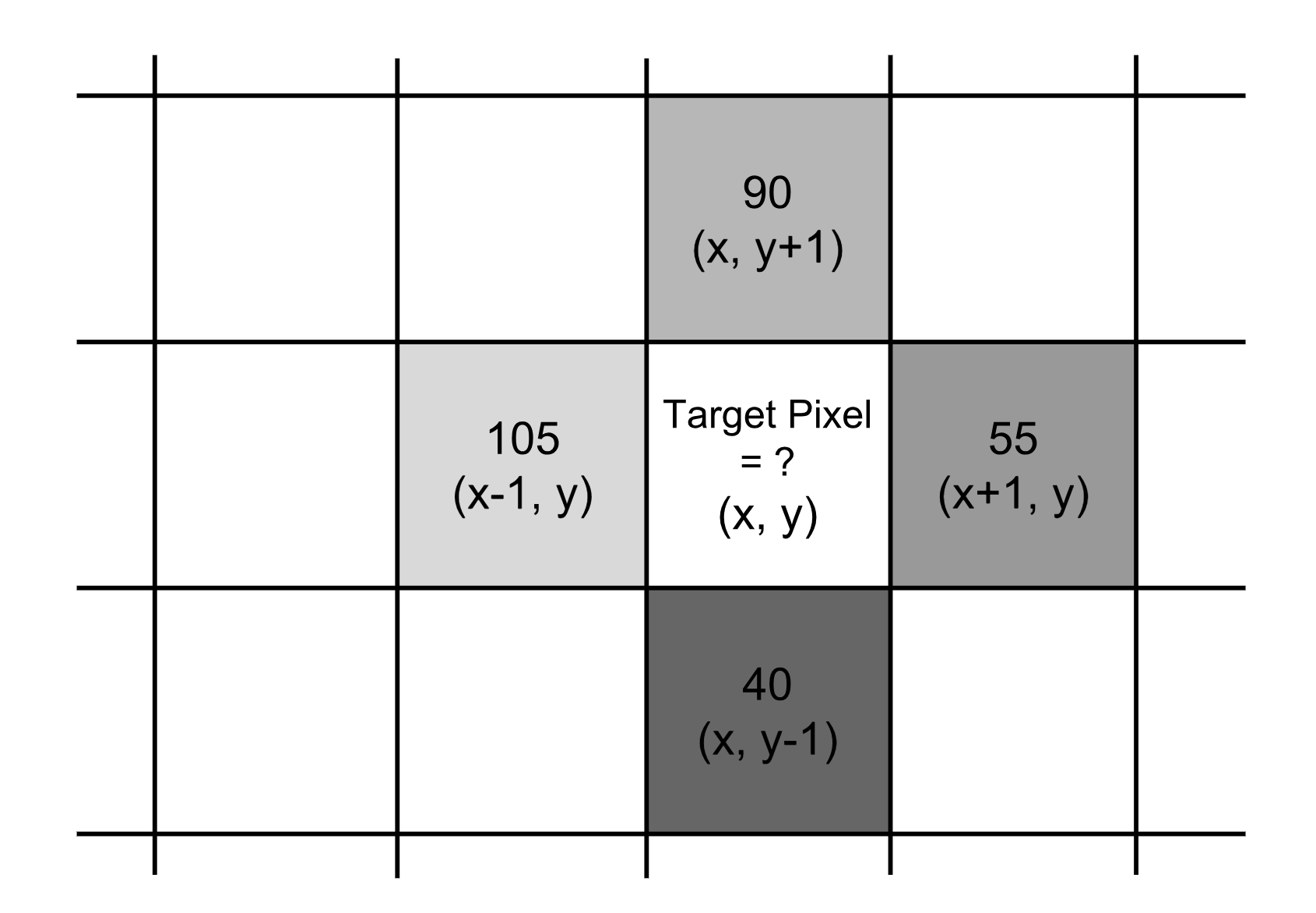
The [image gradient vector](https://en.wikipedia.org/wiki/Image_gradient) is defined as a metric for every individual pixel, containing the pixel color changes in both x-axis and y-axis. The definition is aligned with the gradient of a continuous multi-variable function, which is a vector of partial derivatives of all the variables. Suppose f(x, y) records the color of the pixel at location (x, y), the gradient vector of the pixel (x, y) is defined as follows:

∇�(�,�)=[����]=[��������]=[�(�+1,�)−�(�−1,�)�(�,�+1)−�(�,�−1)]

The ���� term is the partial derivative on the x-direction, which is computed as the color difference between the adjacent pixels on the left and right of the target, f(x+1, y) - f(x-1, y). Similarly, the ���� term is the partial derivative on the y-direction, measured as f(x, y+1) - f(x, y-1), the color difference between the adjacent pixels above and below the target.

There are two important attributes of an image gradient:

* **Magnitude** is the L2-norm of the vector, �=��2+��2.
* **Direction** is the arctangent of the ratio between the partial derivatives on two directions, �=arctan⁡(��/��).

Fig. 1. To compute the gradient vector of a target pixel at location (x, y), we need to know the colors of its four neighbors (or eight surrounding pixels depending on the kernel).

The gradient vector of the example in Fig. 1. is:

∇�=[�(�+1,�)−�(�−1,�)�(�,�+1)−�(�,�−1)]=[55−10590−40]=[−5050]

Thus,

* the magnitude is 502+(−50)2=70.7107, and
* the direction is arctan⁡(−50/50)=−45∘.

Repeating the gradient computation process for every pixel iteratively is too slow. Instead, it can be well translated into applying a convolution operator on the entire image matrix, labeled as � using one of the specially designed convolutional kernels.

Let’s start with the x-direction of the example in Fig 1. using the kernel [−1,0,1] sliding over the x-axis; ∗ is the convolution operator:

��=[−1,0,1]∗[105,255,55]=−105+0+55=−50

Similarly, on the y-direction, we adopt the kernel [+1,0,−1]⊤:

��=[+1,0,−1]⊤∗[9025540]=90+0−40=50

Try this in python:

import numpy as np

import scipy.signal as sig

data = np.array([[0, 105, 0], [40, 255, 90], [0, 55, 0]])

G\_x = sig.convolve2d(data, np.array([[-1, 0, 1]]), mode='valid')

G\_y = sig.convolve2d(data, np.array([[-1], [0], [1]]), mode='valid')

These two functions return array([[0], [-50], [0]]) and array([[0, 50, 0]]) respectively. (Note that in the numpy array representation, 40 is shown in front of 90, so -1 is listed before 1 in the kernel correspondingly.)

**Common Image Processing Kernels**

[Prewitt operator](https://en.wikipedia.org/wiki/Prewitt_operator): Rather than only relying on four directly adjacent neighbors, the Prewitt operator utilizes eight surrounding pixels for smoother results.

��=[−10+1−10+1−10+1]∗� and ��=[+1+1+1000−1−1−1]∗�

[Sobel operator](https://en.wikipedia.org/wiki/Sobel_operator): To emphasize the impact of directly adjacent pixels more, they get assigned with higher weights.

��=[−10+1−20+2−10+1]∗� and ��=[+1+2+1000−1−2−1]∗�

Different kernels are created for different goals, such as edge detection, blurring, sharpening and many more. Check [this wiki page](https://en.wikipedia.org/wiki/Kernel_(image_processing)) for more examples and references.

**Example: Manu in 2004**

Let’s run a simple experiment on the photo of Manu Ginobili in 2004 [[Download Image]({{ ‘/assets/data/manu-2004.jpg’ | relative\_url }}){:target="\_blank"}] when he still had a lot of hair. For simplicity, the photo is converted to grayscale first. For colored images, we just need to repeat the same process in each color channel respectively.

Fig. 2. Manu Ginobili in 2004 with hair. (Image source: [Manu Ginobili's bald spot through the years](http://ftw.usatoday.com/2013/05/manu-ginobilis-bald-spot-through-the-years))

import numpy as np

import scipy

import scipy.signal as sig

# With mode="L", we force the image to be parsed in the grayscale, so it is

# actually unnecessary to convert the photo color beforehand.

img = scipy.misc.imread("manu-2004.jpg", mode="L")

# Define the Sobel operator kernels.

kernel\_x = np.array([[-1, 0, 1],[-2, 0, 2],[-1, 0, 1]])

kernel\_y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])

G\_x = sig.convolve2d(img, kernel\_x, mode='same')

G\_y = sig.convolve2d(img, kernel\_y, mode='same')

# Plot them!

fig = plt.figure()

ax1 = fig.add\_subplot(121)

ax2 = fig.add\_subplot(122)

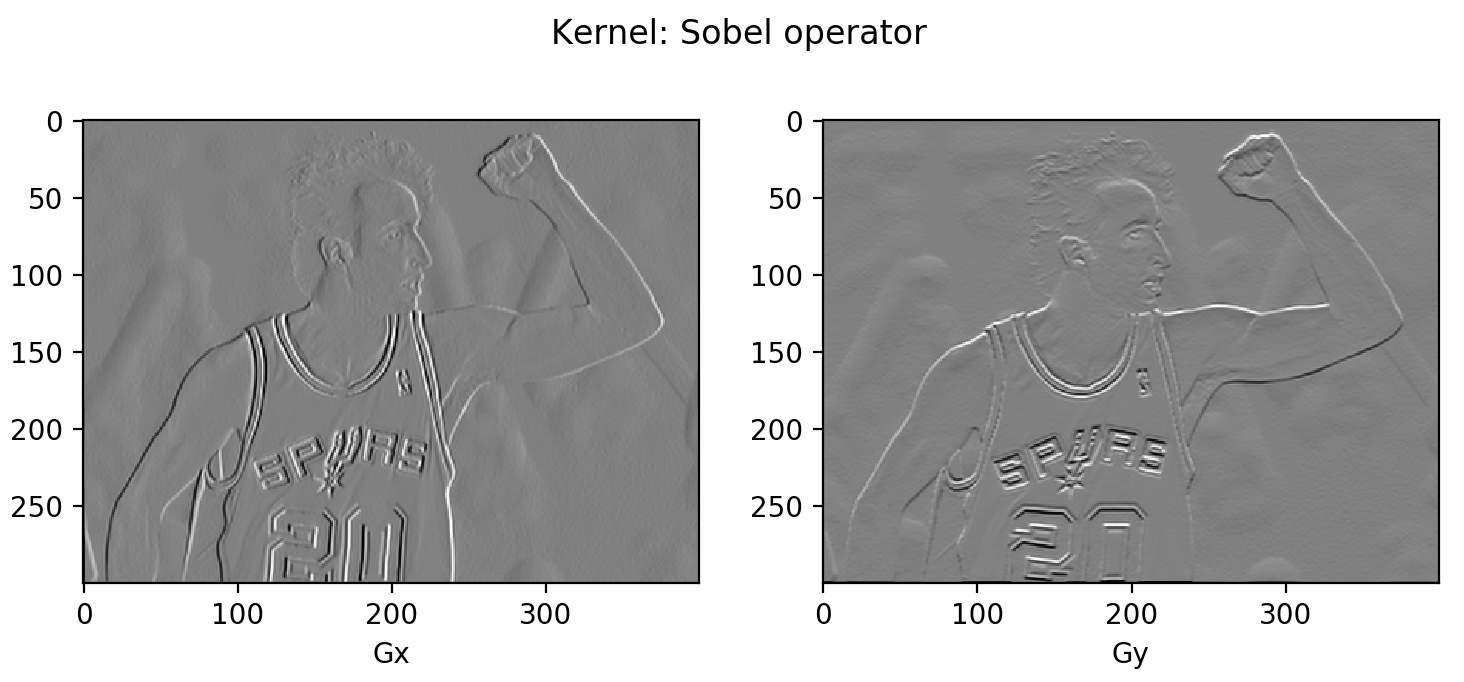
# Actually plt.imshow() can handle the value scale well even if I don't do

# the transformation (G\_x + 255) / 2.

ax1.imshow((G\_x + 255) / 2, cmap='gray'); ax1.set\_xlabel("Gx")

ax2.imshow((G\_y + 255) / 2, cmap='gray'); ax2.set\_xlabel("Gy")

plt.show()

Fig. 3. Apply Sobel operator kernel on the example image.

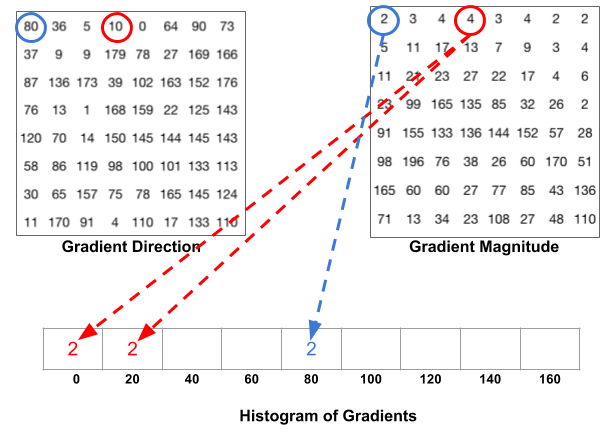
You might notice that most area is in gray. Because the difference between two pixel is between -255 and 255 and we need to convert them back to [0, 255] for the display purpose. A simple linear transformation (� + 255)/2 would interpret all the zeros (i.e., constant colored background shows no change in gradient) as 125 (shown as gray).

**Histogram of Oriented Gradients (HOG)**

The Histogram of Oriented Gradients (HOG) is an efficient way to extract features out of the pixel colors for building an object recognition classifier. With the knowledge of image gradient vectors, it is not hard to understand how HOG works. Let’s start!

**How HOG works**

1. Preprocess the image, including resizing and color normalization.
2. Compute the gradient vector of every pixel, as well as its magnitude and direction.
3. Divide the image into many 8x8 pixel cells. In each cell, the magnitude values of these 64 cells are binned and cumulatively added into 9 buckets of unsigned direction (no sign, so 0-180 degree rather than 0-360 degree; this is a practical choice based on empirical experiments).  
     
   For better robustness, if the direction of the gradient vector of a pixel lays between two buckets, its magnitude does not all go into the closer one but proportionally split between two. For example, if a pixel’s gradient vector has magnitude 8 and degree 15, it is between two buckets for degree 0 and 20 and we would assign 2 to bucket 0 and 6 to bucket 20.  
     
   This interesting configuration makes the histogram much more stable when small distortion is applied to the image.

Fig. 4. How to split one gradient vector's magnitude if its degress is between two degree bins. (Image source: https://www.learnopencv.com/histogram-of-oriented-gradients/)

1. Then we slide a 2x2 cells (thus 16x16 pixels) block across the image. In each block region, 4 histograms of 4 cells are concatenated into one-dimensional vector of 36 values and then normalized to have an unit weight. The final HOG feature vector is the concatenation of all the block vectors. It can be fed into a classifier like SVM for learning object recognition tasks.

**Example: Manu in 2004**

Let’s reuse the same example image in the previous section. Remember that we have computed �� and �� for the whole image.

N\_BUCKETS = 9

CELL\_SIZE = 8 # Each cell is 8x8 pixels

BLOCK\_SIZE = 2 # Each block is 2x2 cells

def assign\_bucket\_vals(m, d, bucket\_vals):

left\_bin = int(d / 20.)

# Handle the case when the direction is between [160, 180)

right\_bin = (int(d / 20.) + 1) % N\_BUCKETS

assert 0 <= left\_bin < right\_bin < N\_BUCKETS

left\_val= m \* (right\_bin \* 20 - d) / 20

right\_val = m \* (d - left\_bin \* 20) / 20

bucket\_vals[left\_bin] += left\_val

bucket\_vals[right\_bin] += right\_val

def get\_magnitude\_hist\_cell(loc\_x, loc\_y):

# (loc\_x, loc\_y) defines the top left corner of the target cell.

cell\_x = G\_x[loc\_x:loc\_x + CELL\_SIZE, loc\_y:loc\_y + CELL\_SIZE]

cell\_y = G\_y[loc\_x:loc\_x + CELL\_SIZE, loc\_y:loc\_y + CELL\_SIZE]

magnitudes = np.sqrt(cell\_x \* cell\_x + cell\_y \* cell\_y)

directions = np.abs(np.arctan(cell\_y / cell\_x) \* 180 / np.pi)

buckets = np.linspace(0, 180, N\_BUCKETS + 1)

bucket\_vals = np.zeros(N\_BUCKETS)

map(

lambda (m, d): assign\_bucket\_vals(m, d, bucket\_vals),

zip(magnitudes.flatten(), directions.flatten())

)

return bucket\_vals

def get\_magnitude\_hist\_block(loc\_x, loc\_y):

# (loc\_x, loc\_y) defines the top left corner of the target block.

return reduce(

lambda arr1, arr2: np.concatenate((arr1, arr2)),

[get\_magnitude\_hist\_cell(x, y) for x, y in zip(

[loc\_x, loc\_x + CELL\_SIZE, loc\_x, loc\_x + CELL\_SIZE],

[loc\_y, loc\_y, loc\_y + CELL\_SIZE, loc\_y + CELL\_SIZE],

)]

)

The following code simply calls the functions to construct a histogram and plot it.

# Random location [200, 200] as an example.

loc\_x = loc\_y = 200

ydata = get\_magnitude\_hist\_block(loc\_x, loc\_y)

ydata = ydata / np.linalg.norm(ydata)

xdata = range(len(ydata))

bucket\_names = np.tile(np.arange(N\_BUCKETS), BLOCK\_SIZE \* BLOCK\_SIZE)

assert len(ydata) == N\_BUCKETS \* (BLOCK\_SIZE \* BLOCK\_SIZE)

assert len(bucket\_names) == len(ydata)

plt.figure(figsize=(10, 3))

plt.bar(xdata, ydata, align='center', alpha=0.8, width=0.9)

plt.xticks(xdata, bucket\_names \* 20, rotation=90)

plt.xlabel('Direction buckets')

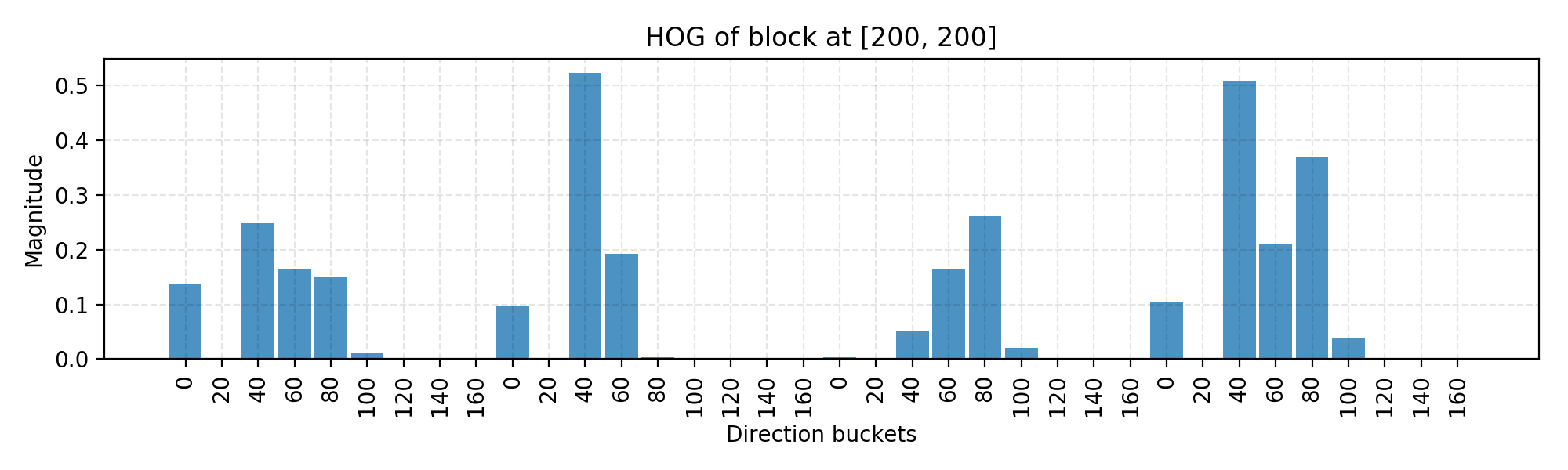
plt.ylabel('Magnitude')

plt.grid(ls='--', color='k', alpha=0.1)

plt.title("HOG of block at [%d, %d]" % (loc\_x, loc\_y))

plt.tight\_layout()

In the code above, I use the block with top left corner located at [200, 200] as an example and here is the final normalized histogram of this block. You can play with the code to change the block location to be identified by a sliding window.

Fig. 5. Demonstration of a HOG histogram for one block.

The code is mostly for demonstrating the computation process. There are many off-the-shelf libraries with HOG algorithm implemented, such as [OpenCV](https://github.com/opencv/opencv), [SimpleCV](http://simplecv.org/) and [scikit-image](http://scikit-image.org/).

**Image Segmentation (Felzenszwalb’s Algorithm)**

When there exist multiple objects in one image (true for almost every real-world photos), we need to identify a region that potentially contains a target object so that the classification can be executed more efficiently.

Felzenszwalb and Huttenlocher ([2004](http://cvcl.mit.edu/SUNSeminar/Felzenszwalb_IJCV04.pdf)) proposed an algorithm for segmenting an image into similar regions using a graph-based approach. It is also the initialization method for Selective Search (a popular region proposal algorithm) that we are gonna discuss later.

Say, we use a undirected graph �=(�,�) to represent an input image. One vertex ��∈� represents one pixel. One edge �=(��,��)∈� connects two vertices �� and ��. Its associated weight �(��,��) measures the dissimilarity between �� and ��. The dissimilarity can be quantified in dimensions like color, location, intensity, etc. The higher the weight, the less similar two pixels are. A segmentation solution � is a partition of � into multiple connected components, {�}. Intuitively similar pixels should belong to the same components while dissimilar ones are assigned to different components.

**Graph Construction**

There are two approaches to constructing a graph out of an image.

* **Grid Graph**: Each pixel is only connected with surrounding neighbours (8 other cells in total). The edge weight is the absolute difference between the intensity values of the pixels.
* **Nearest Neighbor Graph**: Each pixel is a point in the feature space (x, y, r, g, b), in which (x, y) is the pixel location and (r, g, b) is the color values in RGB. The weight is the Euclidean distance between two pixels' feature vectors.

**Key Concepts**

Before we lay down the criteria for a good graph partition (aka image segmentation), let us define a couple of key concepts:

* **Internal difference**: ���(�)=max�∈���(�,�)�(�), where ��� is the minimum spanning tree of the components. A component � can still remain connected even when we have removed all the edges with weights < ���(�).
* **Difference between two components**: ���(�1,�2)=min��∈�1,��∈�2,(��,��)∈��(��,��). ���(�1,�2)=∞ if there is no edge in-between.
* **Minimum internal difference**: ����(�1,�2)=���(���(�1)+�(�1),���(�2)+�(�2)), where �(�)=�/|�| helps make sure we have a meaningful threshold for the difference between components. With a higher �, it is more likely to result in larger components.

The quality of a segmentation is assessed by a pairwise region comparison predicate defined for given two regions �1 and �2:

�(�1,�2)={True if ���(�1,�2)>����(�1,�2)False otherwise

Only when the predicate holds True, we consider them as two independent components; otherwise the segmentation is too fine and they probably should be merged.

**How Image Segmentation Works**

The algorithm follows a bottom-up procedure. Given �=(�,�) and |�|=�,|�|=�:

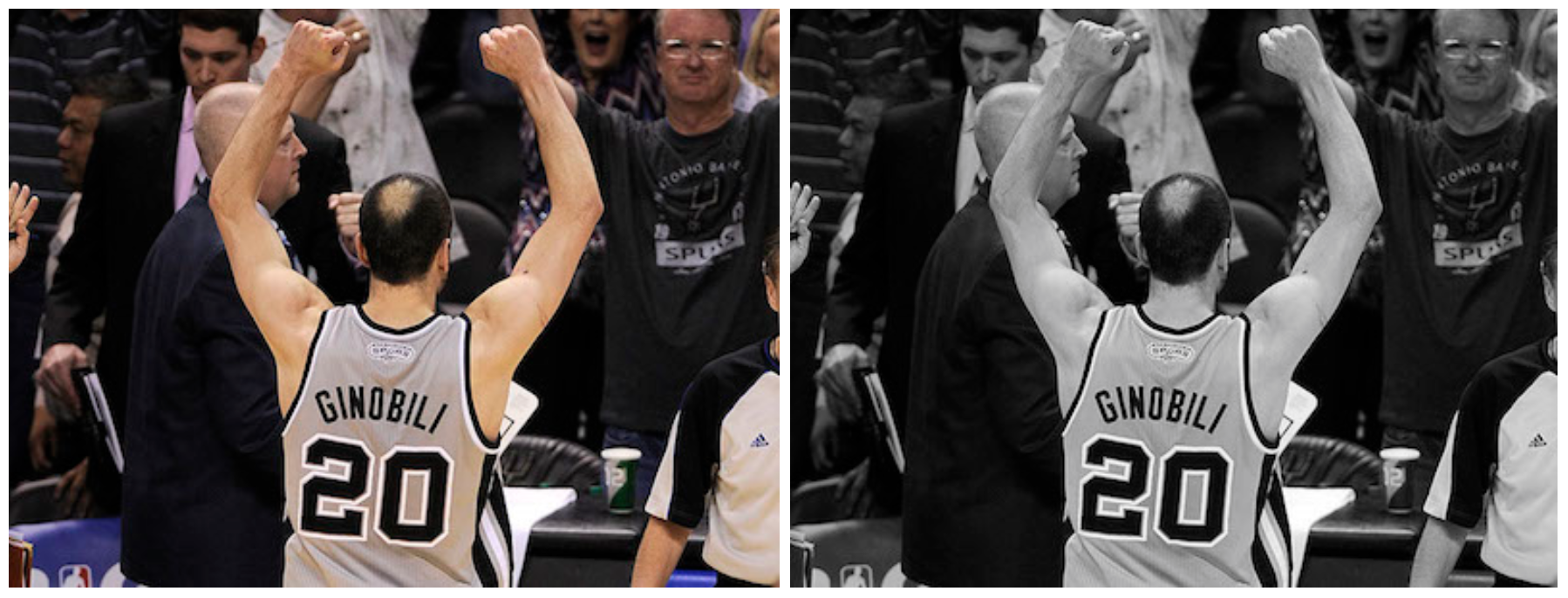
1. Edges are sorted by weight in ascending order, labeled as �1,�2,…,��.
2. Initially, each pixel stays in its own component, so we start with � components.
3. Repeat for �=1,…,�:
   * The segmentation snapshot at the step � is denoted as ��.
   * We take the k-th edge in the order, ��=(��,��).
   * If �� and �� belong to the same component, do nothing and thus ��=��−1.
   * If �� and �� belong to two different components ���−1 and ���−1 as in the segmentation ��−1, we want to merge them into one if �(��,��)≤����(���−1,���−1); otherwise do nothing.

If you are interested in the proof of the segmentation properties and why it always exists, please refer to the [paper](http://fcv2011.ulsan.ac.kr/files/announcement/413/IJCV(2004)%20Efficient%20Graph-Based%20Image%20Segmentation.pdf).

Fig. 6. An indoor scene with segmentation detected by the grid graph construction in Felzenszwalb's graph-based segmentation algorithm (k=300).

**Example: Manu in 2013**

This time I would use the photo of old Manu Ginobili in 2013 [[Image]({{ ‘/assets/data/manu-2013.jpg’ | relative\_url }})] as the example image when his bald spot has grown up strong. Still for simplicity, we use the picture in grayscale.

Fig. 7. Manu Ginobili in 2013 with bald spot. (Image source: [Manu Ginobili's bald spot through the years](http://ftw.usatoday.com/2013/05/manu-ginobilis-bald-spot-through-the-years))

Rather than coding from scratch, let us apply [skimage.segmentation.felzenszwalb](http://scikit-image.org/docs/dev/api/skimage.segmentation.html#skimage.segmentation.felzenszwalb) to the image.

import skimage.segmentation

from matplotlib import pyplot as plt

img2 = scipy.misc.imread("manu-2013.jpg", mode="L")

segment\_mask1 = skimage.segmentation.felzenszwalb(img2, scale=100)

segment\_mask2 = skimage.segmentation.felzenszwalb(img2, scale=1000)

fig = plt.figure(figsize=(12, 5))

ax1 = fig.add\_subplot(121)

ax2 = fig.add\_subplot(122)

ax1.imshow(segment\_mask1); ax1.set\_xlabel("k=100")

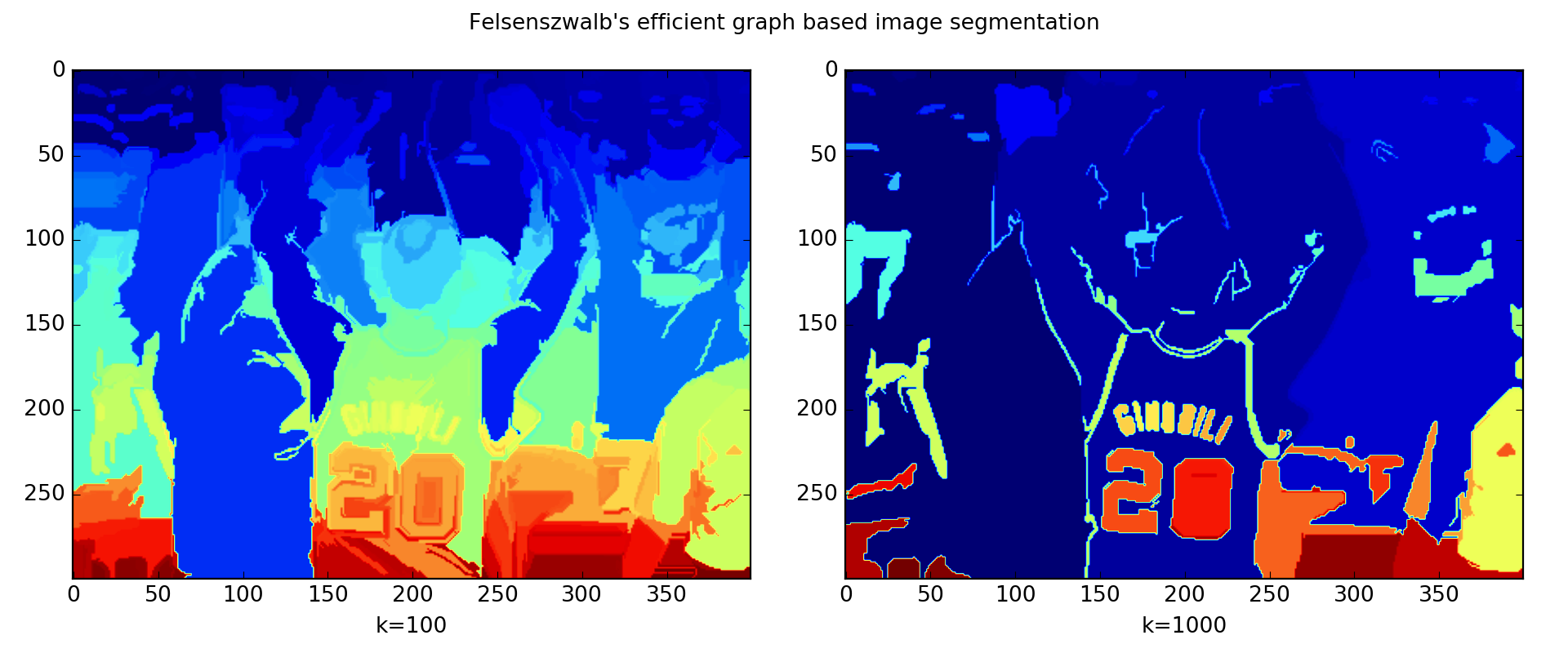
ax2.imshow(segment\_mask2); ax2.set\_xlabel("k=1000")

fig.suptitle("Felsenszwalb's efficient graph based image segmentation")

plt.tight\_layout()

plt.show()

The code ran two versions of Felzenszwalb’s algorithms as shown in Fig. 8. The left k=100 generates a finer-grained segmentation with small regions where Manu’s bald spot is identified. The right one k=1000 outputs a coarser-grained segmentation where regions tend to be larger.

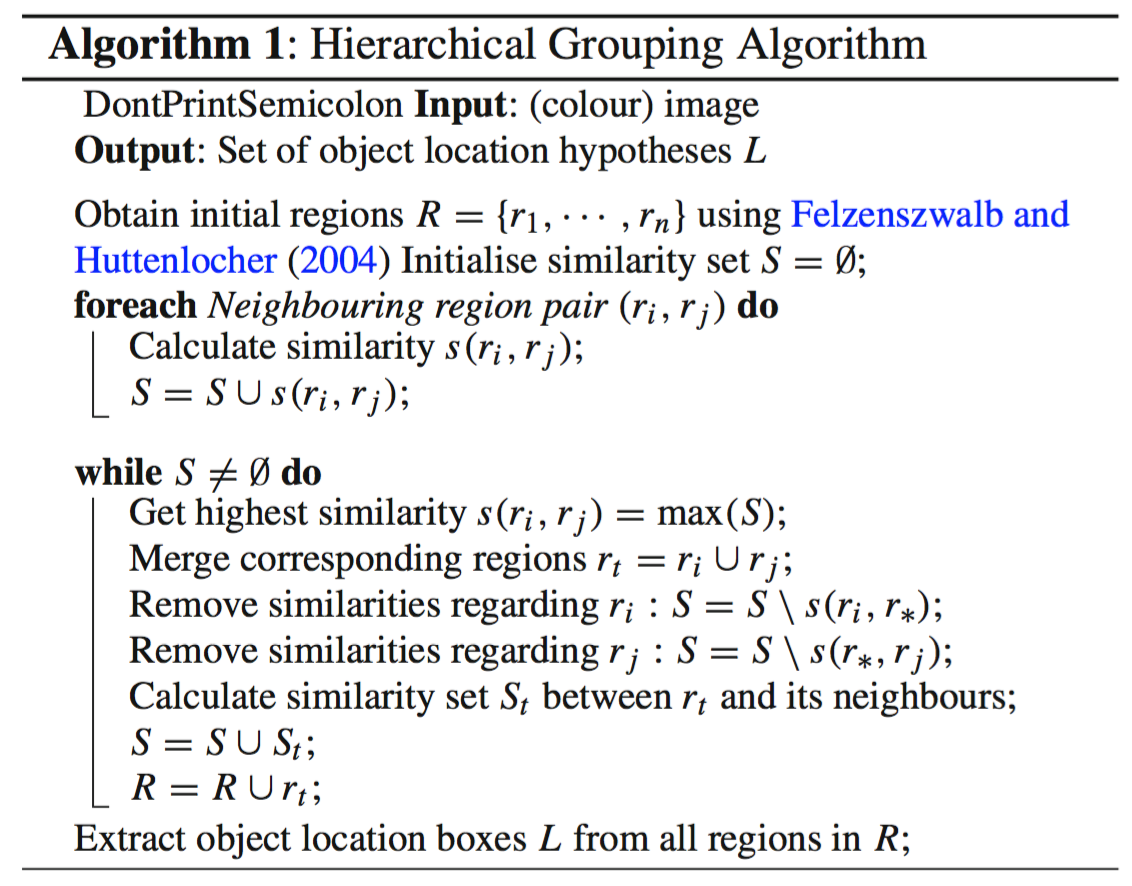
Fig. 8. Felsenszwalb's efficient graph-based image segmentation is applied on the photo of Manu in 2013.

**Selective Search**

Selective search is a common algorithm to provide region proposals that potentially contain objects. It is built on top of the image segmentation output and use region-based characteristics (NOTE: not just attributes of a single pixel) to do a bottom-up hierarchical grouping.

**How Selective Search Works**

1. At the initialization stage, apply Felzenszwalb and Huttenlocher’s graph-based image segmentation algorithm to create regions to start with.
2. Use a greedy algorithm to iteratively group regions together:
   * First the similarities between all neighbouring regions are calculated.
   * The two most similar regions are grouped together, and new similarities are calculated between the resulting region and its neighbours.
3. The process of grouping the most similar regions (Step 2) is repeated until the whole image becomes a single region.

Fig. 9. The detailed algorithm of Selective Search.

**Configuration Variations**

Given two regions (��,��), selective search proposed four complementary similarity measures:

* **Color** similarity
* **Texture**: Use algorithm that works well for material recognition such as [SIFT](http://www.cs.ubc.ca/~lowe/papers/iccv99.pdf).
* **Size**: Small regions are encouraged to merge early.
* **Shape**: Ideally one region can fill the gap of the other.

By (i) tuning the threshold � in Felzenszwalb and Huttenlocher’s algorithm, (ii) changing the color space and (iii) picking different combinations of similarity metrics, we can produce a diverse set of Selective Search strategies. The version that produces the region proposals with best quality is configured with (i) a mixture of various initial segmentation proposals, (ii) a blend of multiple color spaces and (iii) a combination of all similarity measures. Unsurprisingly we need to balance between the quality (the model complexity) and the speed.

Cited as:

@article{weng2017detection1,

title = "Object Detection for Dummies Part 1: Gradient Vector, HOG, and SS",

author = "Weng, Lilian",

journal = "lilianweng.github.io",

year = "2017",

url = "https://lilianweng.github.io/posts/2017-10-29-object-recognition-part-1/"

}

**References**

[1] Dalal, Navneet, and Bill Triggs. [“Histograms of oriented gradients for human detection."](https://hal.inria.fr/file/index/docid/548512/filename/hog_cvpr2005.pdf) Computer Vision and Pattern Recognition (CVPR), 2005.

[2] Pedro F. Felzenszwalb, and Daniel P. Huttenlocher. [“Efficient graph-based image segmentation."](http://cvcl.mit.edu/SUNSeminar/Felzenszwalb_IJCV04.pdf) Intl. journal of computer vision 59.2 (2004): 167-181.

[3] [Histogram of Oriented Gradients by Satya Mallick](https://www.learnopencv.com/histogram-of-oriented-gradients/)

[4] [Gradient Vectors by Chris McCormick](http://mccormickml.com/2013/05/07/gradient-vectors/)

[5] [HOG Person Detector Tutorial by Chris McCormick](http://mccormickml.com/2013/05/09/hog-person-detector-tutorial/)

* [object-detection](https://lilianweng.github.io/tags/object-detection/)

* [object-recognition](https://lilianweng.github.io/tags/object-recognition/)

* [vision-model](https://lilianweng.github.io/tags/vision-model/)

[«  
Object Detection for Dummies Part 2: CNN, DPM and Overfeat](https://lilianweng.github.io/posts/2017-12-15-object-recognition-part-2/)[»  
Learning Word Embedding](https://lilianweng.github.io/posts/2017-10-15-word-embedding/)

© 2023 [Lil'Log](https://lilianweng.github.io/) Powered by [Hugo](https://gohugo.io/) & [PaperMod](https://git.io/hugopapermod)