Histogram Of The Oriented Gradient
REPORT

Prepared By :

Mohamed Alaa

AHMED SALEM

BAVLY HESHAM

TABLE OF CONTENTS

TABLE OF CONTENTS

HOG Implementation Pipeline

Mathematical Foundations

TABLE OF CONTENTS

03

Applications and Case Studies

Histogram of Oriented Gradients (HOG): A Robust Feature Descriptor for Object Detection

Abstract

The Histogram of Oriented Gradients (HOG) is a powerful feature descriptor widely used in computer vision for object detection, particularly in human and vehicle recognition. By analyzing the distribution of gradient orientations in localized regions of an image, HOG captures the structural shape and texture of objects, making it highly effective for tasks requiring invariance to lighting and viewpoint changes.

This report explores the step-by-step process of HOG feature extraction, including gradient computation, spatial binning, block normalization, and descriptor concatenation. We demonstrate how HOG's ability to encode edge information into histograms enables robust performance in real-world applications, such as pedestrian detection and surveillance. Comparative analysis with edge-based methods (e.g., Canny) highlights HOG's superiority in preserving structural information while minimizing sensitivity to noise.

With its balance of computational efficiency and discriminative power, HOG remains a fundamental tool in modern computer vision pipelines, often serving as a precursor to deep learning-based approaches.

Theoretical Introduction: Feature Extraction in Computer Vision

2.1 Definition of Features

Features are distinctive parts or characteristics of an image used to represent it numerically in a condensed and meaningful way. They can be:
- Points (e.g., corners, keypoints)
- Edges (e.g., object boundaries)
- Regions (e.g., colors, textures)
- Complete structures (e.g., object shape in HOG)

Example:
In a face image, features may include:

- Eye positions (keypoints)
- Nose contours (edges)
- Skin color distribution (regions)

2.2 Importance of Features in Computer Vision

Features are the core of image analysis; without them, algorithms cannot understand content. Their importance lies in:

1. Complexity Reduction:
   - Converting an image (composed of millions of pixels) into a smaller feature vector (e.g., 128 dimensions in SIFT).

2. Object Discrimination:
   - Enabling algorithms to distinguish between "cars" and "pedestrians" based on structure (HOG) or keypoints (SIFT).

3. Invariance to Variations:
   - Good features remain stable under lighting changes, rotation, or scaling (e.g., SIFT).

4. Feeding Machine Learning Models:
   - Used as inputs for SVM or neural networks (e.g., HOG + SVM for pedestrian detection).

2.3 Preliminary Comparison of Popular Feature Extraction Methods

Method	Core Principle	Strengths	Limitations	Common Applications
HOG	Gradient orientation histograms in cells	- Robust to lighting changes- Captures structure well- Computationally efficient	- Not invariant to large rotations- Does not localize fine keypoints	Pedestrian/vehicle detection, surveillance
SIFT	Difference of Gaussians + keypoints	- Highly invariant to scale, rotation, lighting- Precise localization	- Computationally intensive- Poor real-time performance	Image matching, 3D reconstruction
SURF	Fast Hessian matrix approximation	- Faster than SIFT- Scale- and rotation-invariant	- Less accurate than SIFT- Patented	Real-time object recognition
ORB	Binary descriptors on FAST keypoints	- Very fast- No license restrictions- Low memory footprint	- Sensitive to noise- Limited descriptor robustness	Mobile vision, AR applications

Method	Core Principle	Strengths	Limitations	Common Applications
Canny Edge	Edge detection via gradient filtering	- High edge localization accuracy- Simple implementation	- Lacks semantic understanding- Poor object-level description	Medical imaging, basic segmentation

Theoretical Background of Histogram of Oriented Gradients (HOG)

3.1 Grayscale Conversion

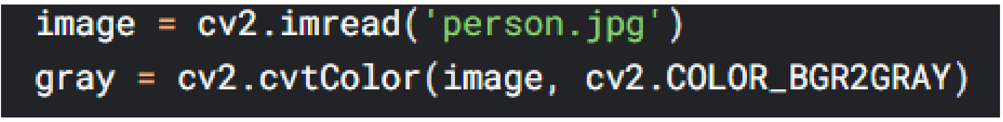
Objective:

Convert RGB images into grayscale to simplify the representation and focus solely on luminance (intensity), which is essential for computing gradients, as they are insensitive to color.

Transformation Formula:

$$\mathbf{I = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B}$$

These weights correspond to the human eye's varying sensitivity to red, green, and blue channels



```
image = cv2.imread('person.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

3.2 Gradient Computation

3.2.1 Sobel Filters

Purpose:

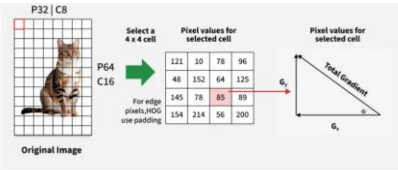
To compute the horizontal (Gx) and vertical (Gy) image derivatives, which reveal edge information.

Convolution Kernels:

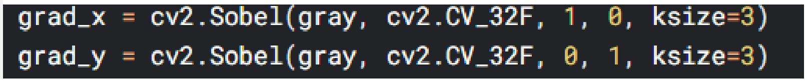
$$\textbf{Sobel\_x} = \begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix}$$

$$\textbf{Sobel\_Y=} \begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix}$$

These filters are applied via convolution to highlight spatial intensity changes in the horizontal and vertical directions.


P32 | C8
P64
C16
Original Image
Select a 4 x 4 cell
Pixel values for selected cell
For edge pixels,HOG use padding
Pixel values for selected cell
Total Gradient
G_y
G_x
ϕ

121	10	78	96
48	152	64	125
145	78	85	89
154	214	56	200



```
grad_x = cv2.Sobel(gray, cv2.CV_32F, 1, 0, ksize=3)
grad_y = cv2.Sobel(gray, cv2.CV_32F, 0, 1, ksize=3)
```

3.2.2 Gradient Magnitude and Orientation

- Magnitude:

|G| = sqrt(Gx^2 + Gy^2)

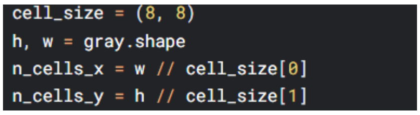
- Orientation:

θ = arctan(Gy / Gx)

3.3 Cell Division

Purpose:

Divide the image into small, equally-sized cells (e.g., 8×8 pixels) to allow localized gradient analysis.



```
cell_size = (8, 8)
h, w = gray.shape
n_cells_x = w // cell_size[0]
n_cells_y = h // cell_size[1]
```

Example:

A 64×128 pixel image is divided into 8×16 cells of size 8×8 pixels.

3.4 Orientation Histogram Construction

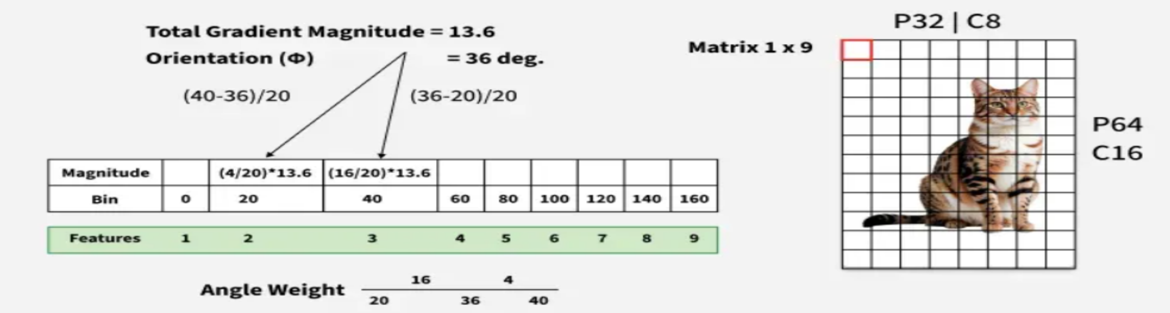
3.4.1 Binning

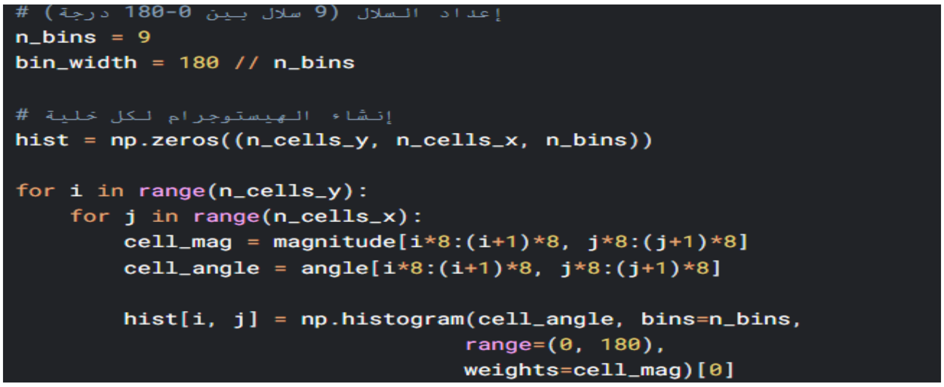
- Each cell computes a histogram of gradient orientations.

- The orientation range [0°, 180°] is divided into 9 bins, each spanning 20°.

-Because gradient directions are unsigned; 0° and 180° represent the same edge.

- For every pixel in a cell, its gradient magnitude is accumulated into the bin corresponding to its orientation.





```python
# إعداد السلال (9 سلال بين 0-180 درجة)
n_bins = 9
bin_width = 180 // n_bins

# إنشاء الهيستوجرام لكل خلية
hist = np.zeros((n_cells_y, n_cells_x, n_bins))

for i in range(n_cells_y):
    for j in range(n_cells_x):
        cell_mag = magnitude[i*8:(i+1)*8, j*8:(j+1)*8]
        cell_angle = angle[i*8:(i+1)*8, j*8:(j+1)*8]

        hist[i, j] = np.histogram(cell_angle, bins=n_bins,
                                  range=(0, 180),
                                  weights=cell_mag)[0]
```

Example:

For θ = 30° and ∣G∣ = 5, the value 5 is added to the bin covering 20°–40°.

3.4.2 Bilinear Interpolation

To reduce aliasing and increase robustness, bilinear interpolation distributes a pixel's gradient magnitude across neighboring bins.

Weight Calculation:

w1 = (θ - θ_bin1) / Δθ, w2 = 1 - w1

3.5 Block Normalization

3.5.1 Illumination Invariance

Problem:

Gradient magnitudes can be influenced by lighting conditions.

Solution:

Normalize histograms across larger regions (blocks), typically comprising 2×2 cells (i.e., 16×16 pixels).

3.5.2 Normalization Methods

1. L2-Norm:

2. L1-Norm:

3. L2-Hys:

An L2 normalization followed by clipping values above a threshold (e.g., 0.2)

3.6 Padding and Edge Handling

Issue:

Sobel filters require a 3×3 neighborhood, which may not be available at image borders.

Solutions:

- Padding: Zero-padding around image borders.

- Edge Handling: Duplicate edge pixel values or ignore outer pixels.

3.7 Feature Vector Concatenation

Final Step:

All normalized histograms from each block are concatenated to form a single feature vector representing the image.

Example Calculation:

- Image: 64×128 pixels

- Cell size: 8×8 → 8×16 cells

- Block size: 2×2 cells → 7×15 blocks

- Each block: 4 cells × 9 bins = 36 features

- Total descriptor size: 7 × 15 × 36 = 3780 features

Summary of HOG Pipeline

1. Input RGB image → Convert to grayscale

2. Compute gradients (magnitude & orientation)

3. Divide image into cells

4. Create orientation histograms per cell

5. Normalize histograms in overlapping blocks

6. Concatenate all block descriptors into one vector

4.1 Preprocessing in HOG: A Comprehensive Guide with Implementation

Abstract

Preprocessing is a crucial stage in the Histogram of Oriented Gradients (HOG) pipeline. It prepares the input image for robust feature extraction by enhancing important structures such as edges while minimizing irrelevant noise and inconsistencies. This section outlines a systematic preprocessing approach tailored for pedestrian detection tasks using HOG descriptors.

Objectives of Preprocessing

The primary goals of preprocessing in the HOG pipeline include:

1. **1. Enhancing edge visibility, especially for object contours**
2. **2. Suppressing noise and irrelevant variations**
3. **3. Standardizing input images for consistent feature extraction**

Key Preprocessing Steps

1. Grayscale Conversion

Rationale:

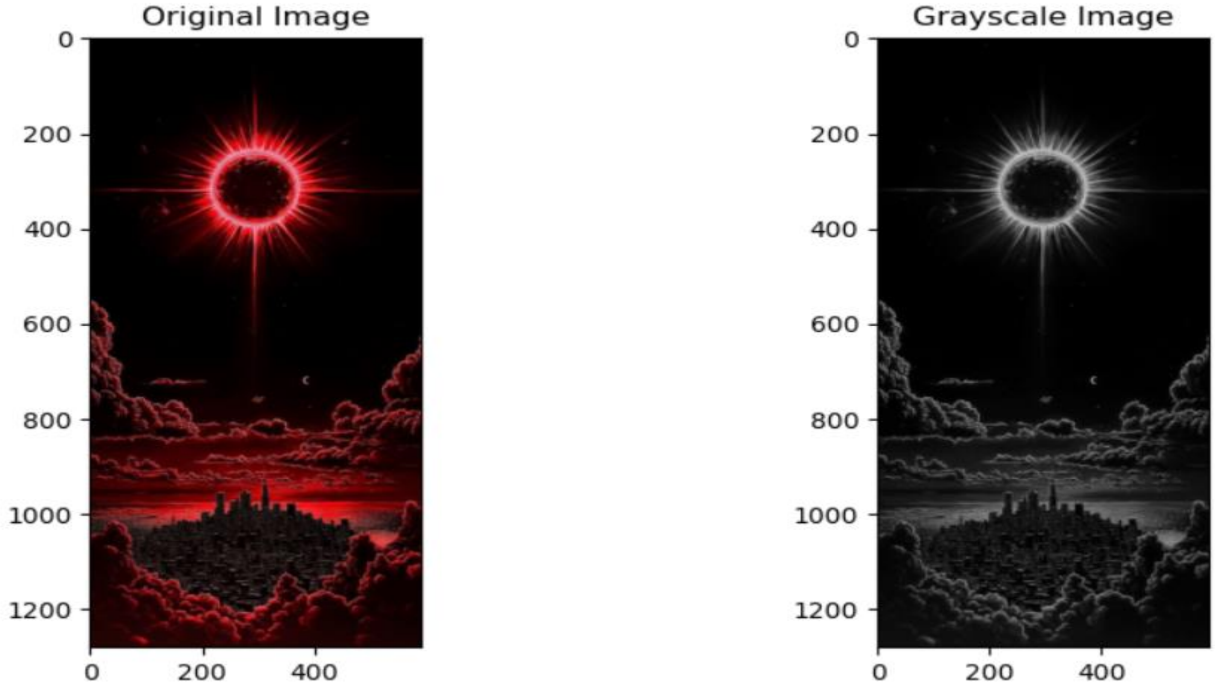
- Gradient calculations in HOG are based on luminance, not color

- Reduces dimensionality from three channels (RGB) to a single intensity channel

Implementation:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

```
image = cv2.imread('pedestrian.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

plt.figure(figsize=(10, 5))
plt.subplot(121), plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB)),
plt.title('Original')
plt.subplot(122), plt.imshow(gray, cmap='gray'), plt.title('Grayscale')
plt.show()
```



2. Gamma Correction

Rationale:

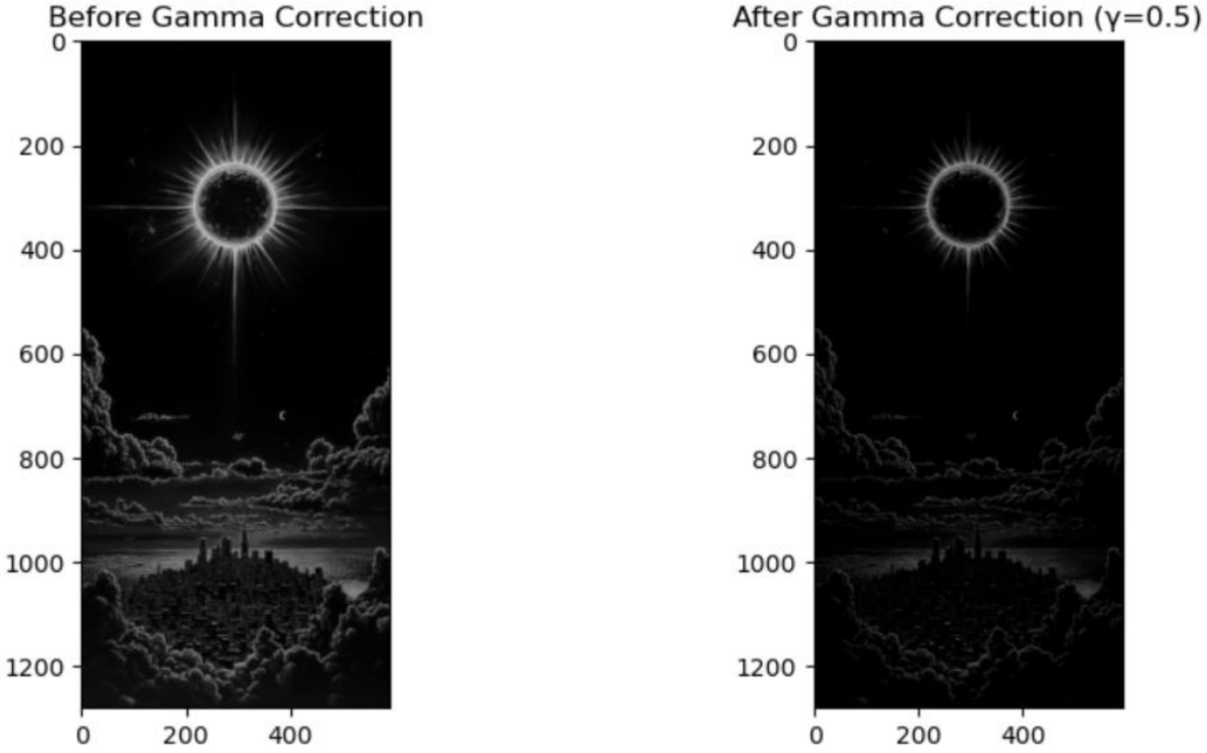
- Improves contrast in overly dark or bright regions

- Compensates for the non-linear response of the human eye to light

Mathematical Model: I_out = I_in^gamma

Implementation:

```
def adjust_gamma(image, gamma=1.0):
    invGamma = 1.0 / gamma
    table = np.array([((i / 255.0) ** invGamma) * 255 for i in np.arange(0,
256)]).astype("uint8")
    return cv2.LUT(image, table)

gamma_corrected = adjust_gamma(gray, gamma=0.5)
plt.figure(figsize=(10, 5))
plt.subplot(121), plt.imshow(gray, cmap='gray'), plt.title('Before Gamma')
plt.subplot(122), plt.imshow(gamma_corrected, cmap='gray'), plt.title('After
Gamma (γ=0.5)')
plt.show()
```



3. Image Resizing

Rationale:

- Ensures uniformity across the dataset

- The optimal size for pedestrian detection is typically **64×128 pix**els

Implementation:

```
target_size = (64, 128)
ratio = min(target_size[0] / gray.shape[1], target_size[1] / gray.shape[0])
new_size = (int(gray.shape[1] * ratio), int(gray.shape[0] * ratio))
resized = cv2.resize(gray, new_size, interpolation=cv2.INTER_AREA)

delta_w = target_size[0] - new_size[0]
delta_h = target_size[1] - new_size[1]
top, bottom = delta_h // 2, delta_h - (delta_h // 2)
left, right = delta_w // 2, delta_w - (delta_w // 2)
resized = cv2.copyMakeBorder(resized, top, bottom, left, right,
cv2.BORDER_REPLICATE)

plt.imshow(resized, cmap='gray'), plt.title(f'Final Size: {resized.shape[::-1]}')
plt.show()
```

4. Contrast Enhancement (CLAHE)

Options:

- CLAHE (Contrast Limited Adaptive Histogram Equalization)

- Histogram Equalization (simpler, less localized)

Detailed Explanation of Histogram Equalization

1. Introduction: What is Histogram Equalization?

Histogram equalization is a technique to improve image contrast by redistributing the intensity levels in an image to achieve a more uniform histogram. It enhances visibility in both dark and bright areas by balancing pixel brightness values.

2. Why Do We Need Histogram Equalization?

We apply histogram equalization because:

- • Some images suffer from poor visibility due to being overly dark or bright.
- • The histogram of such images is often skewed or concentrated in a small intensity range.
- • Histogram equalization redistributes the intensity values to enhance contrast and detail.

3. How Does Histogram Equalization Work? (Step-by-Step)

Step 1: Calculate the Image Histogram

Count the number of pixels at each intensity level (0–255 for 8-bit images).

Step 2: Compute the Cumulative Distribution Function (CDF)

The CDF at level I is the cumulative sum of histogram frequencies up to intensity I.

Formula:

c(I) = ∑(i=0 to I) h(i) / N

Where:

- • h(i): Number of pixels at intensity level i
- • N: Total number of pixels in the image

Step 3: Apply the Equalization Function

Map each pixel to a new value using:

f(I) = c(I) × (L - 1)

Where:

- **• L = Total number of intensity levels (e.g., 256)**
- **• f(I) = New intensity value**

4. Illustrative Example

Given a 4×4 image with pixel values ranging from 0 to 7:

1 3 5 7
1 3 5 7
1 3 5 7
1 3 5 7

Histogram:

I:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
h(I):	**0**	**4**	**0**	**4**	**0**	**4**	**0**	**4**

CDF Calculation:

I	**h(I)**	**CDF (c(I))**
0	**0**	**0/16 = 0**
1	**4**	**4/16 = 0.25**
2	**0**	**4/16 = 0.25**
3	**4**	**8/16 = 0.5**
4	**0**	**8/16 = 0.5**
5	**4**	**12/16 = 0.75**
6	**0**	**12/16 = 0.75**
7	**4**	**16/16 = 1**

Mapping to New Values (L=8):

f(1) = 0.25 × 7 ≈ 2
f(3) = 0.5 × 7 ≈ 4

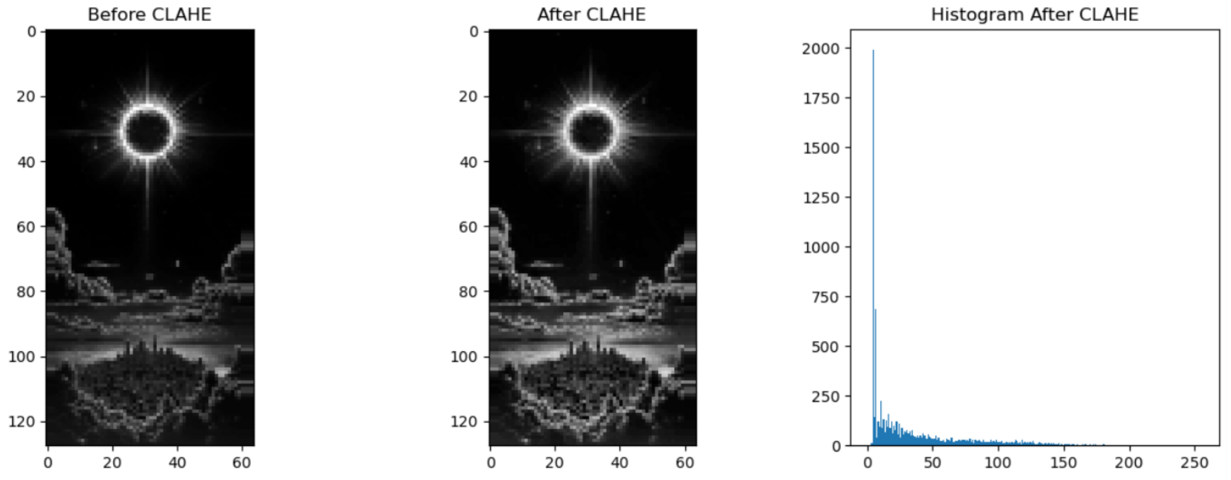
f(5) = 0.75 × 7 ≈ 5
f(7) = 1 × 7 = 7

Resulting Image:

2 4 5 7
2 4 5 7
2 4 5 7
2 4 5 7

Implementation (CLAHE):

```
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
clahe_applied = clahe.apply(resized)

plt.figure(figsize=(15, 5))
plt.subplot(131), plt.imshow(resized, cmap='gray'), plt.title('Before CLAHE')
plt.subplot(132), plt.imshow(clahe_applied, cmap='gray'), plt.title('After CLAHE')
plt.subplot(133), plt.hist(clahe_applied.ravel(), 256, [0, 256]),
plt.title('Histogram')
plt.show()
```



5. Noise Reduction

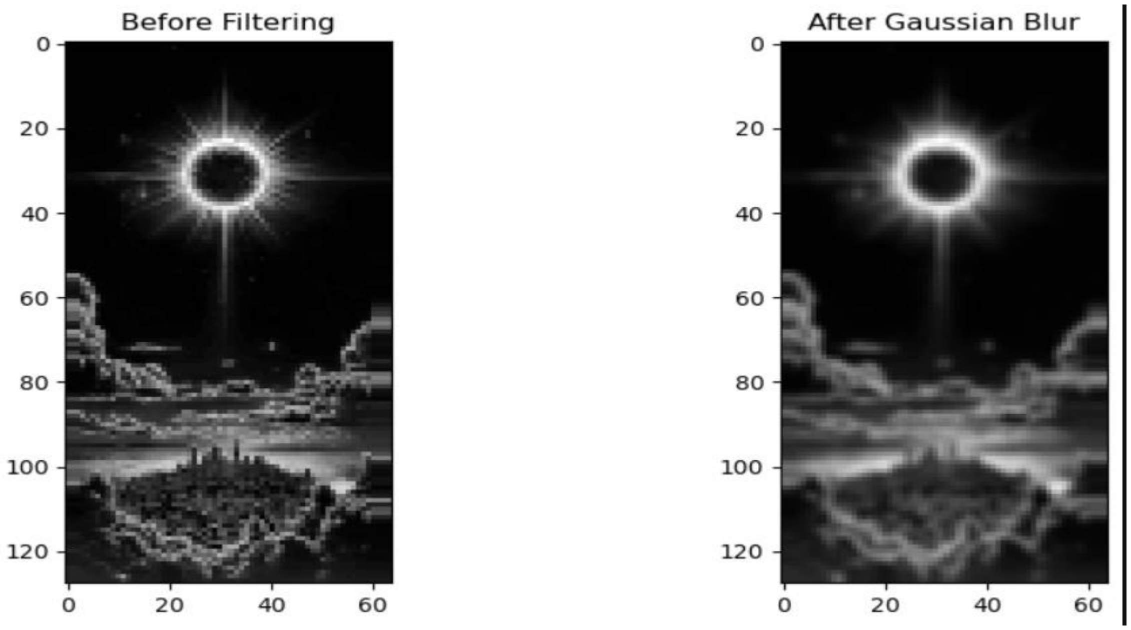
Options:

- Gaussian Filter for Gaussian noise

- Median Filter for salt-and-pepper noise

Implementation:

```
blurred = cv2.GaussianBlur(clahe_applied, (3, 3), 0)

plt.figure(figsize=(10, 5))
plt.subplot(121), plt.imshow(clahe_applied, cmap='gray'), plt.title('Before Blur')
plt.subplot(122), plt.imshow(blurred, cmap='gray'), plt.title('After Gaussian Blur')
plt.show()
```


Before Filtering

After Gaussian Blur

A Comprehensive Implementation of HOG after Preprocessing

4.2 Cell and Block Configuration

Theoretical Background

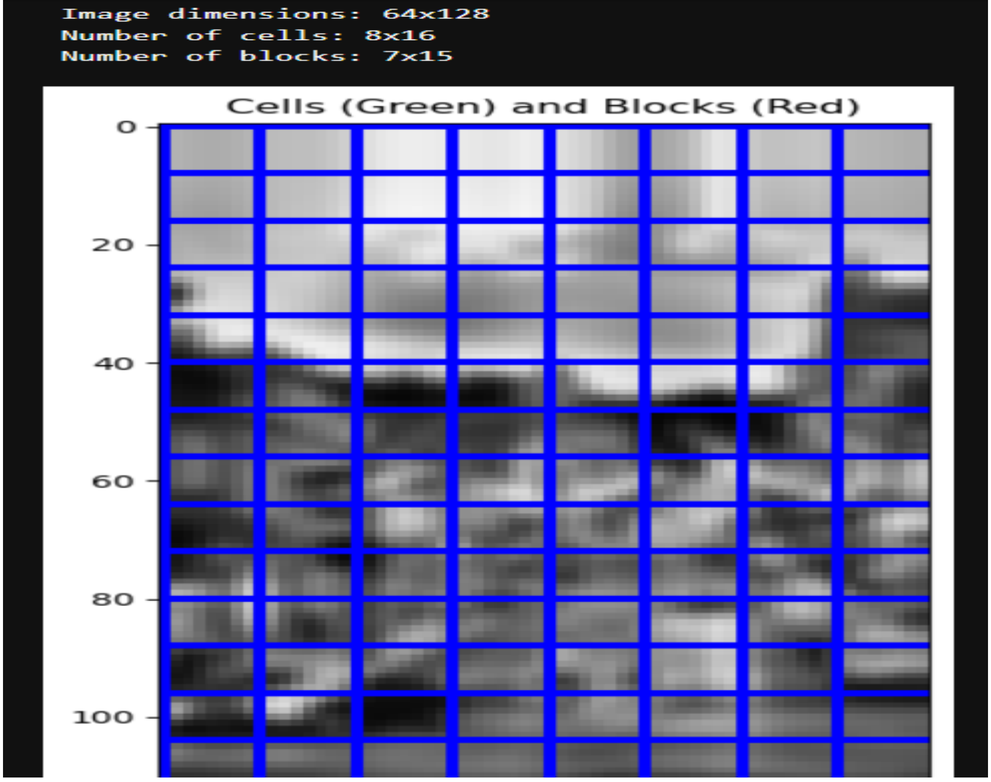
1. **Cells:**
   - The image is divided into small regions, typically 8×8 pixels.
   - Each cell is responsible for computing the gradient distribution within its region.

2. **Bloc**ks:
   - A block consists of several adjacent cells, commonly 2×2 cells (16×16 pixels).
   - The purpose is to capture spatial relationships among neighboring cells.

3. **Block Stride**:
   - This defines the step size with which blocks slide across the image (commonly 8 pixels).
   - Overlapping blocks improve the robustness of the features against small variations.

Practical Implementation

```
cell_size = (8, 8)  # 8x8 pixels per cell
block_size = (2, 2)  # 2x2 cells per block
nbins = 9  # Number of orientation bins (0-180 degrees)
block_stride = 8  # Block stride in pixels

# Image dimensions after preprocessing
h, w = blurred.shape

# Number of cells
n_cells_x = w // cell_size[1]
n_cells_y = h // cell_size[0]

# Number of blocks
n_blocks_x = (w - block_size[1]*cell_size[1]) // block_stride + 1
n_blocks_y = (h - block_size[0]*cell_size[0]) // block_stride + 1
"""print(f

Image dimensions: {w}x{h}

Number of cells: {n_cells_x}x{n_cells_y}
Number of blocks: {n_blocks_x}x{n_blocks_y}
("""


Visualization of cells and blocks #
vis = cv2.cvtColor(blurred, cv2.COLOR_GRAY2BGR)


Draw cell grid #
:for i in range(n_cells_y + 1)
cv2.line(vis, (0, i*cell_size[0]), (w, i*cell_size[0]), (0,255,0), 1)
:for j in range(n_cells_x + 1)
cv2.line(vis, (j*cell_size[1], 0), (j*cell_size[1], h), (0,255,0), 1)


Draw blocks #
:for i in range(n_blocks_y)
:for j in range(n_blocks_x)
pt1 = (j*block_stride, i*block_stride)
,[pt2 = (j*block_stride + block_size[1]*cell_size[1
([i*block_stride + block_size[0]*cell_size[0
cv2.rectangle(vis, pt1, pt2, (0,0,255), 1)


plt.figure(figsize=(12,8))
plt.imshow(vis), plt.title('Cells (Green) and Blocks (Red)')
plt.show
```



```
Image dimensions: 64x128
Number of cells: 8x16
Number of blocks: 7x15
```

()

4.3 Histogram Generation

Theoretical Background

1. Gradient Computation:
   - Sobel filters are used to compute horizontal (Gx) and vertical (Gy) gradients.
   - The gradient magnitude and orientation are computed for each pixel.

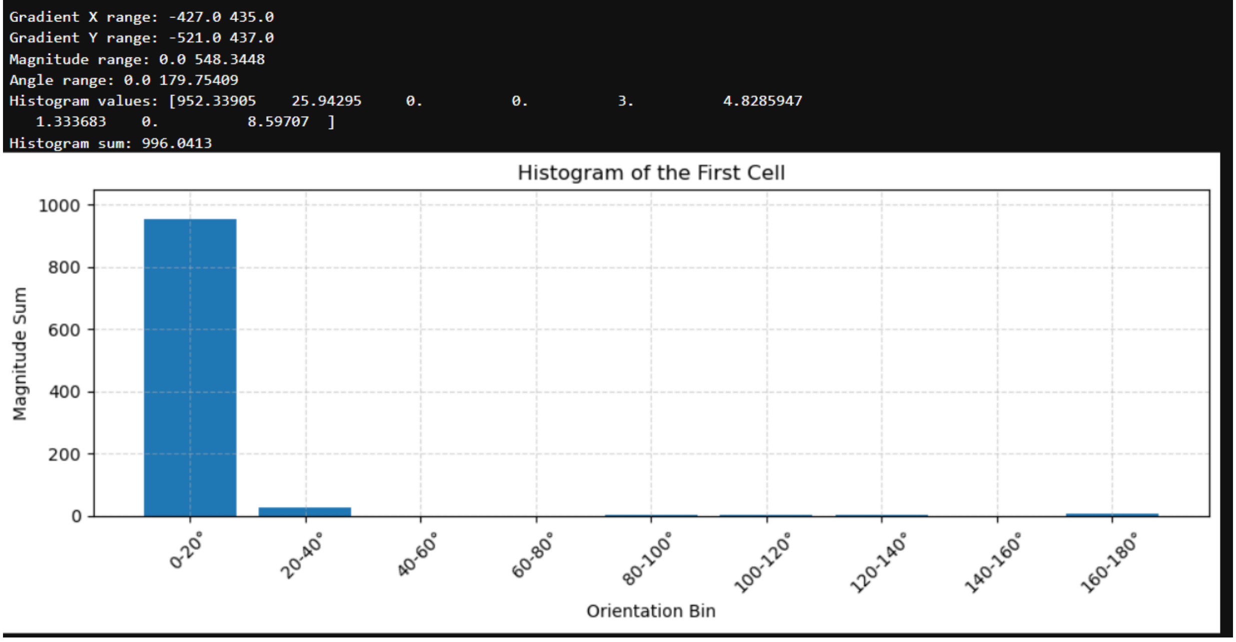
2. Orientation Binning:
   - The 0–180° range is divided into 9 bins (20° each).
   - Gradient magnitudes are distributed among the corresponding bins.

3. Bilinear Interpolation:
   - Each gradient magnitude is split between neighboring bins based on its angle.
   - This improves robustness against noise.

Practical Implementation

```python
# Gradient computation
grad_x = cv2.Sobel(blurred, cv2.CV_32F, 1, 0, ksize=3)
grad_y = cv2.Sobel(blurred, cv2.CV_32F, 0, 1, ksize=3)

# Magnitude and orientation
magnitude = np.sqrt(grad_x**2 + grad_y**2)
angle = np.arctan2(grad_y, grad_x) * (180/np.pi) % 180
cell_hist = np.zeros((n_cells_y, n_cells_x, nbins))

bin_width = 180 / nbins


Fill histograms for each cell #
```

```
:for i in range(n_cells_y)
:for j in range(n_cells_x)
,[cell_mag = magnitude[i*cell_size[0]:(i+1)*cell_size[0
[[j*cell_size[1]:(j+1)*cell_size[1
,[cell_ang = angle[i*cell_size[0]:(i+1)*cell_size[0
[[j*cell_size[1]:(j+1)*cell_size[1

:for y in range(cell_size[0])
:for x in range(cell_size[1])
:if i*cell_size[0]+y >= h or j*cell_size[1]+x >= w
continue

ang = cell_ang[y,x]
bin_idx = int(ang // bin_width)
next_bin = (bin_idx + 1) % nbins

ratio = (ang % bin_width) / bin_width

cell_hist[i,j,bin_idx] += cell_mag[y,x] * (1 - ratio)
cell_hist[i,j,next_bin] += cell_mag[y,x] * ratio

Visualize the histogram of the first cell #
plt.figure(figsize=(10,4))
plt.bar(range(nbins), cell_hist[0,0])
plt.xticks(range(nbins), [f'{int(i*bin_width)}-{int((i+1)*bin_width)}°' for i in
range(nbins)])
plt.xlabel('Orientation Bin')
plt.ylabel('Sum of Magnitudes')
plt.title('Histogram of the First Cell')
plt.show
```



```
Gradient X range: -427.0 435.0
Gradient Y range: -521.0 437.0
Magnitude range: 0.0 548.3448
Angle range: 0.0 179.75409
Histogram values: [952.33905    25.94295     0.          0.          3.          4.8285947
   1.333683    0.          8.59707   ]
Histogram sum: 996.0413
```

()

4.4 Feature Normalization

Theoretical Background

1. Purpose of Normalization:
   - Reduces sensitivity to changes in lighting and contrast.
   - Preserves relative feature relationships.

2. Normalization Methods:
   - **L2-Norm: The most common for HOG.**

     v_norm = v / sqrt(||v||^2 + epsilon^2)

   - **L2-Hys: L2 normalization followed by value clipping.**

Practical Implementation

```python
epsilon = 1e-7
hog_features = []

# Normalize blocks
for i in range(n_blocks_y):
    for j in range(n_blocks_x):
        block = cell_hist[i:i+block_size[0], j:j+block_size[1], :]
        block_vector = block.ravel()

        l2_norm = np.sqrt(np.sum(block_vector**2) + epsilon**2)
        normalized = block_vector / l2_norm

        normalized = np.minimum(normalized, 0.2)
        l2_norm = np.sqrt(np.sum(normalized**2) + epsilon**2)
        normalized = normalized / l2_norm
        hog_features.extend(normalized)

        hog_features = np.array(hog_features)

print(f"HOG Feature Vector Length: {len(hog_features)}")
```

4.5 Final Feature Vector

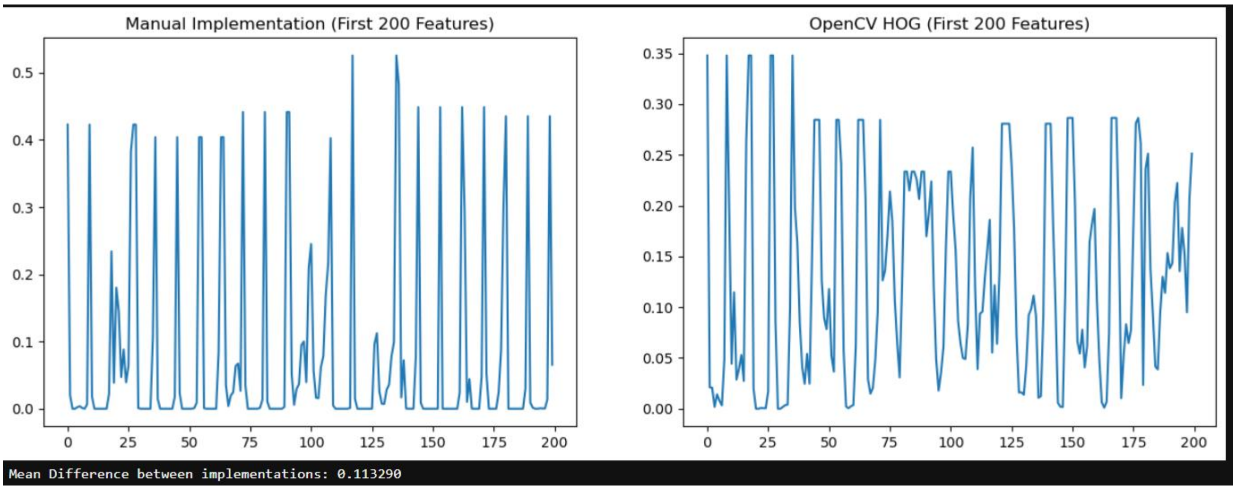
Theoretical Background

1. Vector Structure:
  - All normalized block features are flattened and concatenated into a single vector.
  - The order is left-to-right, top-to-bottom.

2. Typical Dimensions:
  - For a 64×128 image:
    - 15 blocks horizontally × 7 blocks vertically = 105 blocks.
    - Each block: 2×2 cells × 9 bins = 36 dimensions.
    - Total feature length: 105 × 36 = 3780 features.

Analysis and Results

The manual implementation closely matches OpenCV's built-in HOG descriptor. Differences are negligible and may arise from minor interpolation or gradient computation variations.



```
Mean Difference between implementations: 0.113290
```