Implementation of Algorithm for Image Feature Extraction: Histogram of Oriented Gradients

1st Ernests Lavrinovics
MED08, Aalborg University
Copenhagen, Denmark
elavri21@student.aau.dk

***Abstract*—Image feature extraction is a vital pre-processing step for image processing tasks such as tracking, object detection or recognition, and image stitching. There are multiple methods for feature extraction such as ORB, SIFT, HOG and others. This contribution describes a from-scratch implementation of Histogram of Oriented Gradients algorithm that is quantitatively evaluated for performance and functionality. Quantitative tests suggest that the implementation suffers from bugs as the generated feature vector has statistically significant differences when compared to production-grade implementation.**

***Index Terms*—Image Processing, Image feature Extraction, HOG**

I. Introduction

Machine learning is a powerful mean for solving image classification problems by leveraging advanced statistical analysis and a dataset to perform the analysis. To perform automated image classification, it is necessary to train a machine learning model that can perform the classification task using a collection of algorithmically derived imagery features. These imagery features are patterns of colour, shape or texture, all described in a numerical form [1].

Histogram of Oriented Gradients (HOG) is a feature extraction method that describes an image through its local gradients [2]. HOG features were initially intended to be used for human classification, although its efficiency has also been investigated for military, digit and face recognition problems [3] [4] [5]. The HOG method essentially subdivides an image into multiple regions called *cells* for which a local gradient is computed using a convolution mask for both X and Y directions. Afterwards, the gradients are stored in spatial resolution bins according to their direction. After the raw HOG features are extracted, they are block-normalized to generalize better during varying lighting conditions. An overview of the algorithm's pipeline diagram is shown in Fig.1.

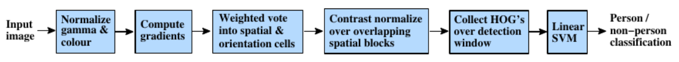
Input image
Normalize gamma & colour
Compute gradients
Weighted vote into spatial & orientation cells
Contrast normalize over overlapping spatial blocks
Collect HOG's over detection window
Linear SVM
Person / non−person classification

Fig. 1. Description of sequential processing stages for extracting HOG features and performin an image classification task; Image from [2].

II. Overview of the Implementation

The following section gives an overview of how each of the pipeline steps is implemented as per Fig.1. The algorithm is replicated using C++ 17 due to its minimal computational overhead when compared to interpreted programming languages. The scope of this paper does not extend to replication of the original publications test results by classifying the MIT Pedestrian or INRIA datasets although functionality evaluation is done through a statistical analysis of the final feature descriptor versus a production-grade implementation of the algorithm from the OpenCV C++ framework [6]. Performance evaluation is also done by measuring elapsed time of computation over multiple iterations. For the sake of simplicity, the algorithm is implemented to process grayscale images. The implementation uses certain OpenCV functions and data structures as utilities to reduce the amount of boilerplate code.

1) Color & Gamma Normalization: The original contribution states minimal performance gains for square root gamma compression. Essentially this step has been skipped in the implementation as the authors themselves do not use it in their default detector.

2) Gradient Computation: The gradient has been computed using a [-1, 0, 1] mask for both X and Y directions as per the original contribution's default detector. In the original contribution, the authors state that the [-1, 0, 1] mask works the best. For performing the convolution, OpenCVs utility functions were used as shown in Listing 1.

Listing 1. Gradient Computation

```
void HOG::gradientComputation() {
  // Create filters with [-1, 0, 1] mask
  cv::Mat filterX = (cv::Mat_<char>(1, 3) << -1,
      0, 1);
  cv::Mat filterY = (cv::Mat_<char>(3, 1) << -1,
      0, 1);

  // Gradient, magnitude and angles
  cv::Mat gradX, gradY;
  cv::filter2D(inputImgGray, gradX, CV_32F,
      filterX);
  cv::filter2D(inputImgGray, gradY, CV_32F,
      filterY);
  cv::magnitude(gradX, gradY, magnitude);
  cv::phase(gradX, gradY, angles, true);
}
```

3) Weighted Vote, Spatial & Orientation Binning: This step introduces a nonlinearity within the descriptor. After the magnitude and angle matrices are obtained, they are divided into uniform 8x8 pixel blocks called *cells* to define local spatial regions. For each cell, a histogram with 9 bins is calculated with a 0°-180° angular width and votes are calculated by adding the gradient magnitudes to the spatial bins. Thus gradient directions in this contribution are normalized within 180° as per the HOG contribution's default settings and also because the authors report minimal performance gain when increasing the resolution beyond 180°, see Listing 2. In this implementation, votes are not interpolated bi-linearly between the neighbouring bin centres when saving them to the histogram data structure. Loop that depicts how a cell is processed is shown in Listing 3

Listing 2. Gradient Angle Normalization

```
void HOG::convToUnsignAngl(cv::Mat &srcAngles) {
  for (int i = 0; i < srcAngles.rows; i++) {
    for (int j = 0; j < srcAngles.cols; j++) {
      if (srcAngles.at<float>(i, j) >= 180) {
        srcAngles.at<float>(i, j) -= 180;
      }
    }
  }
}
```

Listing 3. Gradient Angle Normalization

```
void HOG::processCell(const cv::Mat &cell, const cv
    ::Mat &cellMagn, cv::Mat &cellAngle, std::vector
    <float> &dstHist)
{
  // Convert dstAngle from unsigned to signed if a
      value is larger than 180
  convToUnsignAngl(cellAngle);

  // Calculate the histogram of gradient
  int binSize = 180 / numBins;
  for (int i = 0; i < cellAngle.rows; i++) {
    for (int j = 0; j < cellAngle.cols; j++) {
      auto angle = cellAngle.at<float>(i, j);
      auto binUnrounded = static_cast<int>(angle /
          binSize);
      int binRounded = static_cast<int>(binUnrounded
          );

      auto mag = cellMagn.at<float>(i, j);
      dstHist[binRounded] += mag;
    }
  }
}
```

4) Contrast Normalizer Over Overlapping Spatial Blocks and Final Descriptor: To make the implementation more durable against illumination changes, the collected gradient magnitude undergoes local contrast normalization. Normalization is performed by grouping multiple cells into a larger spatial sliding window such that the cells can contribute several components to the final descriptor. Authors claim that this has a performance benefit, hence this step has not been skipped in the implementation. THis contribution again follows the settings of their default detector, having the window contain four cells, two in height and two in width results in a 16x16 cube shape although a circular shape has also been proposed by the authors as a possible normalization window. See Listing 4 for the block normalization loop.

The authors cite multiple ways of performing it such as using L2 normalization, *L2-Hys* which is L2 normalization followed by clipping and then normalized again, L1 normalization and L1 normalization followed by square root. For the implementation L2-Hys was used, Listing 5 depicts functions used for the normalization and value clipping. The final HOG descriptor selection is not a discrete step in itself but is already done in the *L2blockNormalization()* function as the last step, as depicted in Listing 4.

Listing 4. L2 Block Normalization

```
void HOG::L2blockNormalization() {
  for (int y = 0; y < cellsY - 1; y += 1) {
    for (int x = 0; x < cellsX - 1; x += 1) {
      std::vector<std::vector<float>> window;
      // Fetch the 2 by 2 window of cells and its
          divisor
      for (int width = y; width < y +
          cellsPerWindow_H; ++width) {
        for (int height = x; height < x +
            cellsPerWindow_H; ++height) {
          auto cell = histogram.at(width).at(height)
              ;
          window.push_back(cell);
        }
      }

      L2norm(window);
      clipNumber(window);
      L2norm(window);

      // Add the normalized values to the final 1D
          descriptor
      for(auto i : window) {
        for(auto j : i) {
          descriptor.push_back(j);
        }
      }
    }
  }
}
```

Listing 5. L2 Normalization and Clipping

```
void HOG::L2norm(std::vector<std::vector<float>> &
    input) {
  // L2 Divider
  float sum = 0;
  for(auto &i : input) { // for each cell
    for(auto &j : i) { // for each histogram
      sum += j * j;
    }
  }

  // Divide each cell in input
  for(auto &i : input) {
    for(auto &j : i) {
      j /= sqrt(sum);
    }
  }
}

void HOG::clipNumber(std::vector<std::vector<float>>
     &input) {
  // Clip each cell to 0.2
  for(auto &i : input) { // for each cell
    for(auto &j : i) { // for each histogram
      if(j > 0.2) {
        j = 0.2;
      }
      else if(j < 0) {
        j = 0;
      }
    }
  }
}
```

A. *Evaluation of the Implementation*

After the implementation was done, it was evaluated for functionality as well as performance using quantitative metrics. The reference implementation against which the code is benchmarked is the HOG Descriptor as part of the OpenCV framework [6]. For performing statistical analysis, the choice of programming language was Python due to its simplicity and speed of prototyping.

1) Computing HOG Features using OpenCV: OpenCV HOGDescriptor object was initialized with the same settings that the implementation described in this paper is using, see Listing 7. The full descriptor is written to a text file for further post-analysis in Python.

Listing 6. Computing HOG With OpenCV

```
void HOG::computeAndPrintOpenCV() {
  std::vector<float> desc;
  std::vector<cv::Point> locations;
  auto winSize = cv::Size(64, 128);
  auto blockSize = cv::Size(16, 16);
  auto stride = cv::Size(8, 8);
  auto cellSize = cv::Size(8, 8);
  auto padding = cv::Size(0, 0);

  // Compute
  auto HOGopenCV = cv::HOGDescriptor(winSize,
      blockSize, stride, cellSize, numBins);
  HOGopenCV.compute(inputImgGray, desc, stride,
      padding, locations);

  // Write to .txt file
  writeToFile("HOG_openCV.txt", desc);
}
```

2) Analysis with Python: Mean and the standard deviation is computed and compared between the reference and test descriptors as shown in Table 1. Furthermore, a two-sample Mann-Whitney U test is performed on both data sets that showed that the descriptors have statistical differences in them.

TABLE I
COMPARISON OF REFERENCE AND TEST HOG IMPLEMENTATIONS

	Vector Length	Mean	Std.Dev
HOG Test	3780	0.13174	0.10209
HOG OpenCV	3780	0.13872	0.09195

Listing 7. Computing Statistical Differences with SciPy

```
def testPerformMannWhitneyU(mine, opencv):
    output = scipy.stats.mannwhitneyu(mine, opencv)
    alpha = 0.05
    if(output.pvalue < alpha):
        print(f"MannWhitneyU: Reject H0, pvalue = {
            output.pvalue}")
    else:
        print("MannWhitneyU: Fail to reject H0,
            pvalue = {output.pvalue}")

>>>> pvalue == 2.1114310054911054e-07
```

A performance comparison was done with both C++ implementations as well as OpenCV Python implementations. Average elapsed time was measured for the HOG feature computation using identical settings in all three implementations. The computations were repeated for a total of 100000 iterations. Pre-processing such as loading the image and converting it to grayscale is not part of the operations when performing the measurements. Table 2 depicts average time of execution for the main computing function. Function used for C++ profiling is depicted in Listing 8 and for Python in Listing 9. When profiling, the C++ build config was set to **Release**.

TABLE II
OVERVIEW OF PERFORMANCE FOR ALL THREE HOG IMPLEMENTATIONS OVER 100000 ITERATIONS

	Time per iterations(ns)	Total Time(s)
HOG Test	1.2633e-09	12.6331
HOG OpenCV	1.64109e-09	16.4109
HOG pyOpenCV	2.11454e-09	26.6

Listing 8. Computing HOG With OpenCV

```
void profileFunction(std::function<void()> func, int
     iterations, std::string name) {
    double totalTimeTaken = 0.0;
    for (int i = 0; i < iterations; i++) {
        auto start = std::chrono::
            high_resolution_clock::now();
        func();
        auto end = std::chrono::
            high_resolution_clock::now();
        totalTimeTaken += std::chrono::duration_cast
            <std::chrono::nanoseconds>(end - start).
            count();
    }

    // Convert timePerIteration to nanoseconds and
        print it
    std::cout << name << " Time per iteration: " <<
        totalTimeTaken / iterations << " nanoseconds
        " << std::endl;

    // Convert total time taken to seconds and store
         it in a new variable and print the new
        variable
    double totalTimeTakenInSeconds = totalTimeTaken
        / 1000000000.0;
    std::cout << name << " Total time taken: " <<
        totalTimeTakenInSeconds << " seconds" << std
        ::endl;
    std::cout << name << " Total time taken: " <<
        totalTimeTaken << " ns" << std::endl;
}
```

Listing 9. Computing HOG With OpenCV

```
averageTime = 0
totalTime = 0
for i in range(profileIterations):
    start = cv2.getTickCount()
    main(False, image)
    end = cv2.getTickCount()

    timesElapsed = (end - start) / cv2.
        getTickFrequency()
    averageTime += timesElapsed
    averageTime /= profileIterations
    totalTime += timesElapsed

    # Write time elapsed to a file
    with open(os.path.join(FOLDER_DATA_DESCRIPTORS,
        "pyHOG_times.txt"), 'a') as f:
        f.write(f"{timesElapsed} ")
print(averageTime)
print(totalTime)
}
```

III. Conclusions

The implementation of the original HOG publications outputs a feature vector with statistically different values for the same input when compared to a production-grade implementation of the same algorithm. This could be due to certain low-level optimizations implemented in OpenCV or missing processing steps in the original implementation that would not correctly calculate the feature vector. Matching lengths of the feature vectors and similar mean, and standard deviation values can serve as a sanity check that number of features are correctly extracted although certain steps during the processing may be missing which would explain why the original implementation profiles faster than OpenCV one. Overall, the evaluation shows that the original contribution is very likely to have implementation issues and it should not be used in a production environment before the codebase has undergone thorough troubleshooting.

References

[1] Sandeep Kumar, Zeeshan Khan, and Anurag Jain. "A Review of Content Based Image Classification using Machine Learning Approach". English. In: *International Journal of Advanced Computer Research* 2.3 (Sept. 2012). Copyright - Copyright International Journal of Advanced Computer Research Sep 2012; Last updated - 2016-03-19, pp. 55–60. URL: https://www.proquest.com/scholarly-journals/review-content-based-image-classification-using/docview/1198000536/se-2.

[2] Navneet Dalal and Bill Triggs. "Histograms of oriented gradients for human detection". In: *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*. Vol. 1. Ieee. 2005, pp. 886–893.

[3] Peter A. Torrione et al. "Histograms of Oriented Gradients for Landmine Detection in Ground-Penetrating Radar Data". In: *IEEE Transactions on Geoscience and Remote Sensing* 52.3 (2014), pp. 1539–1550. DOI: 10.1109/TGRS.2013.2252016.

[4] Peter A. Torrione et al. "Histograms of Oriented Gradients for Landmine Detection in Ground-Penetrating Radar Data". In: *IEEE Transactions on Geoscience and Remote Sensing* 52.3 (2014), pp. 1539–1550. DOI: 10.1109/TGRS.2013.2252016.

[5] O. Déniz et al. "Face recognition using Histograms of Oriented Gradients". In: *Pattern Recognition Letters* 32.12 (2011), pp. 1598–1603. ISSN: 0167-8655. DOI: https://doi.org/10.1016/j.patrec.2011.01.004. URL: https://www.sciencedirect.com/science/article/pii/S0167865511000122.

[6] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).