

Electric Experiment Report

조현종 201410935 담당교수 김원준

2019 09 23

1 Abstract

gradient 필터를 컨볼루션하여 magnitude와 phase 값을 구할수 있으며, 이를 통한 HOG 알고리즘을 간단하게 구현하여 인물을 구분할수 있을지 확인해본다.

2 Experiment

2.1 Source code

```
1 #include <opencv2/imgproc.hpp>
2 #include <opencv2/highgui.hpp>
3 #include <math.h>
4 #include <iostream>
5 #include <stdlib.h>
6 #pragma warning(disable : 4996)
7
8 //define constant
9 #define PI 3.1416
10 #define CHANNEL 3
11 #define FILTER_H 3
12 #define FILTER_W 3
13
14 #define DEBUG_
15
16 //struct definition
17 struct pixel {
18     float H;
19     float W;
20     float phase;
21     float magnitude;
22 }typedef pixel;
23
24 using namespace cv;
25
26 //gradient filter definition
27 int filterX[9] = {
28     -1, -1, -1,
29     0, 0, 0,
30     1, 1, 1
31 };
32 int filterY[9] = {
33     -1, 0, 1,
34     -1, 0, 1,
35     -1, 0, 1
36 };
37
38 int main() {
39     //open img
```

```

40 Mat origImg = imread("ref6.jpg", cv::IMREAD_COLOR);
41 int height = origImg.rows;
42 int width = origImg.cols;
43
44 //define iteration variable
45 int i, j,h,w;
46
47 Mat origImgGray = Mat::zeros(height, width, CV_8UC1);
48
49 //conv. color to gray
50 for (i = 0; i < height; i++) {
51     //X
52     for (j = 0; j < width; j++) {
53         origImgGray.at<uchar>(i, j) = (origImg.at<Vec3b>(i, j)[0] + origImg.at<Vec3b>(i, j)[1] +
54             → origImg.at<Vec3b>(i, j)[2]) / 3;
55     }
56 }
57
58 Mat outputImg(height, width, CV_8UC1);
59
60 //allocation output pixel type
61 pixel* output = (pixel*)calloc(height * width, sizeof(pixel));
62
63 int max = INT_MIN, min = INT_MAX ;
64
65 float comp;
66 for (i = 0; i < height; i++) {
67     //X
68     for (j = 0; j < width; j++) {
69         //convolution calculation
70         for (h = 0; h < FILTER_H; h++) {
71             for (w = 0; w < FILTER_W; w++) {
72                 //countinue when outbound
73                 if ((i - 1 + h < 0) || (i - 1 + h >= height)) continue;
74                 if ((j - 1 + w < 0) || (j - 1 + w >= width)) continue;
75                 //calc MAC gradient filter
76                 output[i * width + j].W += (origImgGray.at<uchar>(i - 1 + h, j - 1 + w)) * (filterX[h *
77                     → FILTER_W + w]);
78                 output[i * width + j].H += (origImgGray.at<uchar>(i - 1 + h, j - 1 + w)) * (filterY[h *
79                     → FILTER_W + w]);
80             }
81         }
82
83         //calc phase
84         output[i * width + j].phase = 180 * (atan2(output[i * width + j].H, output[i * width + j].W) /
85             → PI);
86         if (output[i * width + j].phase < 0) {
87             output[i * width + j].phase = output[i * width + j].phase + 180;
88         }
89         if (output[i * width + j].phase >= 180) {
90             output[i * width + j].phase = 0;
91         }
92
93         //Phase quantization by 20 degree(unsigned)
94         output[i * width + j].phase = (int)(output[i * width + j].phase / 20);
95
96         //calc magnitude
97         output[i * width + j].magnitude = sqrt(pow(output[i * width + j].W,2) + pow(output[i * width +
98             → j].H,2));
99
100        //set max, min for normalize
101        max = __max(output[i * width + j].magnitude, max);
102        min = __min(output[i * width + j].magnitude, min);
103    }
104 }
105
106 //magnitude normalize
107 for (i = 0; i < height; i++) {

```

```

103     for (j = 0; j < width; j++) {
104         output[i * width + j].magnitude = 255 * (output[i * width + j].magnitude - min) / (max - min);
105         outputImg.at<uchar>(i, j) = output[i * width + j].magnitude;
106     }
107 }
108
109 int tileW = 16, tileH = 16, blkI, blkJ, sumBlk = 0;
110 float block[15][7][9] = { 0, };
111
112 //hog weight calc.
113 for (i = 0; i <= height - tileH; i += 8) {
114     for (j = 0; j <= width - tileW; j += 8) {
115         blkI = i / 8;
116         blkJ = j / 8;
117 #ifdef DEBUG
118         std::cout << i << "," << j << std::endl;
119 #endif // DEBUG
120         //calculation sum of demention's magnitude
121         for (h = 0; h < tileH; h++) {
122             for (w = 0; w < tileW; w++) {
123                 block[blkI][blkJ][((int)output[(h + i) * width + (w + j)].phase)] +=
124                     output[(h+i)* width + (w+j)].magnitude;
125             }
126         }
127         //sum of all block pixel's magnitude
128         for (int k = 0; k<9; k++) {
129             sumBlk += pow(block[blkI][blkJ][k],2) + 0.000001;
130         }
131
132         //calc. L-2 norm.
133         for (int k = 0; k < 9; k++) {
134             block[blkI][blkJ][k] = block[blkI][blkJ][k] / sqrt(sumBlk);
135         }
136         sumBlk = 0;
137     }
138 }
139 //save hog weight
140 FILE * file = fopen("output.csv", "w");
141 fprintf(file, ",\n");
142 for (i = 0; i <= height - tileH; i += 8) {
143     for (j = 0; j <= width - tileW; j += 8) {
144         blkI = i / 8;
145         blkJ = j / 8;
146         for (int k = 0; k < 9; k++) {
147             fprintf(file, "%f,\n", block[blkI][blkJ][k]);
148         }
149     }
150 }
151
152 #ifdef DEBUG
153     for (i = 0; i < height; i++) {
154         for (j = 0; j < width; j++) {
155             std::cout << output[i * width + j].phase;
156         }
157         std::cout << std::endl;
158     }
159     //show result for debug
160     imwrite("output.bmp", outputImg);
161     imshow("orig", origImgGray);
162     imshow("result", outputImg);
163     waitKey(5000);
164 #endif // DEBUG
165 }

```

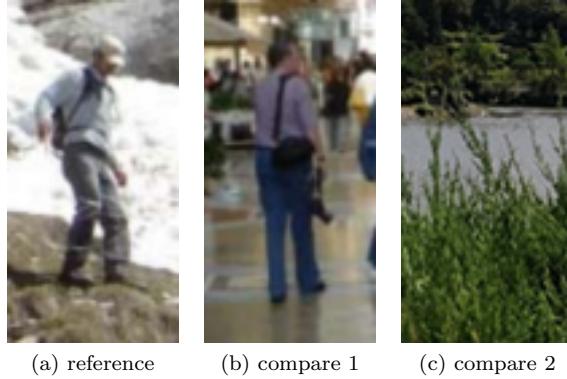


Figure 1: input으로 주어진 이미지

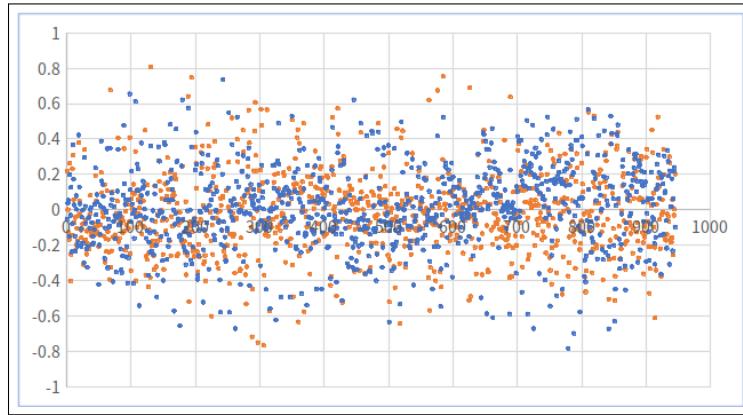


Figure 2: compare 1, 2와 Ref.와의 HOG weight 차 비교 그래프

2.2 Result

인풋 이미지 Figure 1에서, 각 이미지를 HOG weight로 변환 한뒤 ref.이미지와 각 compare 1, compare 2 이미지를 뺀 값을 그래프 Figure 2로, 뺀값의 절대값을 구한 값을 그래프 Figure 3 나타내었다. 파란색은 *reference-compare1* 사람간을 비교한 값이며, 주황색은 *reference-compare2* 사람과 풀을 비교한값이다.

3 discussion

본 코드는 gradient filter를 기반으로한 edgedetecting을 기본으로 하여, 각 픽셀의 magnitude와 radian을 기반으로 유사 hog 알고리즘 구현을 목표하였다. gradient filter란,

$$x \text{ 매트릭스인 } \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \text{와, } y \text{ 매트릭스인 } \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

이며, 이를 컨볼루션 하는 것으로 주변 상하, 좌우 픽셀과의 차를 구하고 두 좌표의 원점으로부터 길이를 구해 magnitude, $\tan^{-1}()$ 하는것을 통해 radian을 구할수 있다.

HOG는 이미지를 일정 크기의 block으로 나눈뒤, radian을 기준으로 양자화해 블록별로 합산하는것으로 얻어진다. 위 코드에서는 128*64의 인풋 이미지를 입력받으며, 16*16의 블럭 사이즈, stride를 8, 양자화를 9

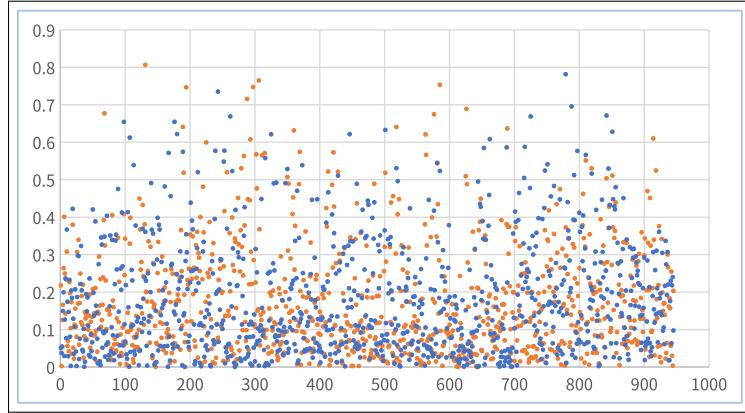


Figure 3: compare 1, 2와 Ref.와의 HOG weight 차의 절대값 비교 그래프



Figure 4: 가설 검증을 위한 새 input 이미지

단계로 설정하였다. 결과적으로 $(\frac{64}{8} - 1) \times (\frac{128}{8} - 1) \times 9 = 945$ 개의 결과가 나오게 되며, 이 값을 비교하면 이미지의 유사도를 판단할수 있다.

그래프 Figure2를 보면 한눈에 두 이미지 compare 1, 2간의 차이가 보이지는 않는 편이다. 하지만, 각 값의 합이 0에 얼마나 가까운지를 보는것으로 각 값이 얼마나 유사한지를 확인할수 있다. 위 코드를 이용해 얻은 reference 이미지와 compare1 이미지 HOG weight의 총합은 7.7이며, compare 2 이미지와의 총합은 -20.5의 결과가 나와, 사람간의 비교가 0에 가까웠지만, 값들의 절대값들의 총합은 170과 166으로 오히려 사람과 사람을 비교했을때가 값이 높았다.

실제 이론과 결과를 비교하여 보았을때, 생각보다 명확한 차이가 나타난것 같지는 않았다. 몇가지 이유가 있을것으로 예상하였는데, 첫째로 기준이 되는 ref. 이미지에서 사람이 아닌 배경에도 옛지가 명확한 부분이 있어 기준 데이터로 삼기 어려웠을것이고, 두번째로 HOG 구현방식이 논문보다 축약된 방식이였으며, 세번째로 사람 여부의 분류를 weight의 학습을 통한 분류가 아닌 사람이 분류하기 때문이다. 실제로, HOG weight을 사용하여 사람을 구분하는경우 SVM과 같은 머신러닝을 통해 분류한다.

가설을 검증해 보기 위해 약간의 실험을 해보았다. 위의 이유가 맞다면, reference 이미지로 다른이미지를 넣는다면 더 정확한 결과가 나올 가능성성이 있다. 그리하여, Figure 4에 보이는 대로, 기존 reference 이미지와 다르게 배경이 단색에 가까운 사람의 전신사진을 대신 input으로 집어넣고 결과를 분석하였다.

그래프 Figure 5를 보면, 눈으로 보기에도 파란색 점인 $|reference - compare1|$ 이 0에 가까워 보이며 실제로도 $|reference - compare1| < 0.1$ 을 만족하는 값은 409개인 반면 $|reference - compare2| < 0.1$ 을

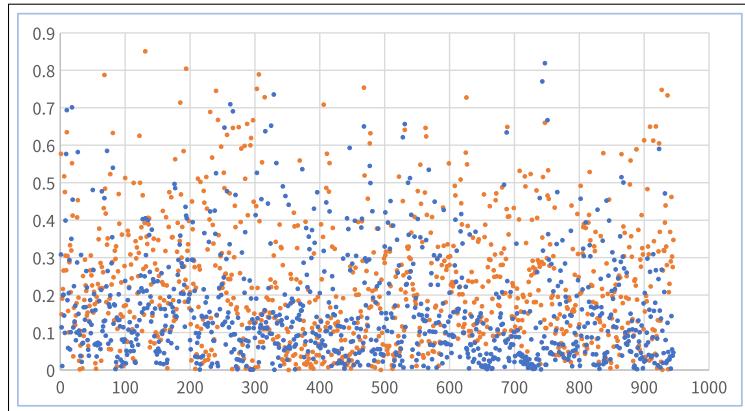


Figure 5: compare 1, 2와 new Ref.와의 HOG weight 차의 절대값 비교 그래프

만족하는 값은 224개에 불구하다. 전체값의 평균도 사람과 비교하였을때가 149, 풀과 비교하였을때가 219로 사람과 비교하였을 때의 값이 낮은것으로 나타났다.

위와같은 결과를 보아, 이번 코드는 HOG 알고리즘을 잘 구현하였으며, 기준이 되는 ref 이미지를 어떻게 잡는지도 중요하다는 것을 확인 하였다.