Northeastern University
EECE 5639 Computer Vision HW 2 Submission

Yash Mewada

Feb 7, 2023

1 Solution 1 EST NOISE algorithm and filtering noise with BOX filter

Initially, the images were created with zero noise, this is evident from the fact that all the images were of the same grayscale and size of 256*256. After implementing the $EST-NOISE$ algorithm on the noisy images of zero mean and standard deviation of 2.0 we got an estimated noise value of around that ranged from **1.91 - 2.1** and the singular estimated noise of the whole image turned out to be **1.918832** for noisy images. Although the singular value should reach near 2.0 due to the random number generator in c++ the image even had a 2.0 standard deviation the numbers generated were repetitive. To test the correctness of this algorithm it was tested on zero noise images and it gave an output of **0.0** as singular noise, which makes sense. The algorithm for estimating noise and generating noisy images as well as the implementation of the Box filter is listed below.

```

  void noisy_img_gen() {
   // This algorithm generates a noisy image with zero mean and standard
  deviation of 2.0
   default_random_engine generator;
   normal_distribution<double> distribution(0.0, 2.0);

   Mat base_image(256, 256, CV_8UC1, Scalar(128));
   generator.seed(time(0));
   for (int i = 0; i < 10; i++) {
     Mat noisy_image = base_image.clone();
     for (int r = 0; r < 256; r++) {
       for (int c = 0; c < 256; c++) {
         int noise = (int)distribution(generator);
         noisy_image.at<uchar>(r, c) =
  saturate_cast<uchar>(noisy_image.at<uchar>(r, c) + noise);
       }
     }
     String filename = "noisy_image_" + to_string(i) + ".jpg";
     imwrite(filename, noisy_image);
   }
   }
   double EST_NOISE(String image_name){
   double bgrPixel[256][256] = {0};
   double sigma[256][256] = {0};

   for(uint8_t s = 0;s<10;s++){
       string file_name = (String)image_name + to_string(s) + ".jpg";
       Mat image = imread(file_name, IMREAD_COLOR);
       for (int i = 0; i < image.rows; i++) {
           for (int j = 0; j < image.cols; j++) {
               bgrPixel[i][j] += (sum(image.at<Vec3b>(i, j))/3)[0];
```

```
            }
        }
    }
    for (int i = 0; i < 256; i++) {
        for (int j = 0; j < 256; j++) {
            bgrPixel[i][j] /= 10;
            // cout << bgrPixel[i][j] << ",";
        }
    }
    for(uint8_t s = 0;s<10;s++){
        string file_name = (String)image_name + to_string(s) + ".jpg";
        Mat image = imread(file_name, IMREAD_COLOR);
        for (int i = 0; i < image.rows; i++) {
            for (int j = 0; j < image.cols; j++) {
                double diff = (bgrPixel[i][j] - (sum(image.at<Vec3b>(i,
  j))/3)[0]);
                sigma[i][j] += pow(diff,2);
                // cout << sigma[i][j] << ",";
            }
        }
    }
    for (int i = 0; i < 256; i++) {
        for (int j = 0; j < 256; j++) {
            sigma[i][j] /= 9;
            sigma[i][j] = pow(sigma[i][j],0.5);
        }
    }
    double sig_est = 0.0;
    for (int i = 0; i < 256; i++) {
        for (int j = 0; j < 256; j++) {
            sig_est += sigma[i][j];
            // cout << bgrPixel.at<Vec3f>(i, j) << endl;
        }
    }
    return sig_est/65536;
    }void box_filter(){
    Mat kernel = Mat::ones(1, 3, CV_32F) / 9;
    Mat filteredImage;
    for(uint8_t s = 0;s<10;s++){
        string file_name = "noisy_image_" + to_string(s) + ".jpg";
        Mat image = imread(file_name);
        filter2D(image, filteredImage, -1, kernel, Point(-1, -1), 0,
  BORDER_DEFAULT);
        imwrite("filtered_image.jpg", filteredImage);
        for(uint8_t i = 0;i<10;i++){
            string file_name = "filter_image_" + to_string(i) + ".jpg";
            imwrite(file_name,image);
        }
    }
    }
    int main(){
    noisy_img_gen();
    double est_noisy = EST_NOISE("noisy_image_");
    box_filter();
    double est_filtered = EST_NOISE("filter_image_");
    printf("%f %f",est_noisy,est_filtered);
    }

    noisy image noise 1.918832 and filtered image noise 0.000000
```
Listing 1: Algorithm for estimated noise

```

   void box_filter(){
    Mat kernel = Mat::ones(1, 3, CV_32F) / 9;
    Mat filteredImage;
    for(uint8_t s = 0;s<10;s++){
        string file_name = "noisy_image_" + to_string(s) + ".jpg";
        Mat image = imread(file_name);
        filter2D(image, filteredImage, -1, kernel, Point(-1, -1), 0,
   BORDER_DEFAULT);
        imwrite("filtered_image.jpg", filteredImage);
        for(uint8_t i = 0;i<10;i++){
            string file_name = "filter_image_" + to_string(i) + ".jpg";
            imwrite(file_name,image);
        }
    }
}
```
Listing 2: Algorithm for Box filter

2 Solution 2 Generate a 2D Gaussian Filter mask.

We saw in class that Gaussian and Box filters are separable i.e they can be operated over rows and columns of an image. The equation for gaussian distribution is.

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{1}$$

Separating the above equation for two 1D filters, we can rewrite the equation as.

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}, G(y) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{y^2}{2\sigma^2}} \tag{2}$$

Now the general thumb rule for the size of the filter is between 6σ to 3σ. But for optimality, we consider here that 5σ = 7x7 Box filter should be designed. Below is the algorithm for a 2D gaussian mask.

```
float sigma = 1.4;
    int kernel_size = 7;
    Mat gx = getGaussianKernel(kernel_size, sigma, CV_32F);
    Mat gy = getGaussianKernel(kernel_size, sigma, CV_32F);

    cout << "gx = " << gx << endl;
    cout << "gy = " << gy << endl;
```
Listing 3: CV2 gaussian filter generator.

3 Solution 3 Linear Filtering

3.1 Apply two linear filters to the given image.

With the filter (a) - [6, 8, 10, 16, 22, 28, 34, 40, 32, 24] - [X, X, 10, 16, 22, 28, 34, 40, X, X]
With the filter (b) - [7, 9, 10, 13, 19, 31, 37, 40, 36, 28] - [X, X, 10, 13, 19, 31, 37, 40, X, X]

3.2 Compare and discuss the results

In terms of computational cost, filter a took around **0.043ms** while filter b took **0.07ms**. Hence this tells that filter b required more computational time than filter (a) also filter b had nonuniform weights. Calculating the variance of the filtered images..
with filter (a) the variance was - **164.44**
with filter (b) the variance was - **137.77**.
Variance without the filter is **250**. Having said that the variance before the filter is...

$$E[(I_i - E(T_i)^2)] = \sigma^2 = 250$$

4 Solution 4

The [1x3] matrix with a center value of 2 has a grayscale value of 2. $P(salt) = 0.7$ and $P(pepper) = 0.3$. Now the probability of the output matrix to be as per these probability values will be...

$$P(I = \left[\begin{array}{c} saltonbothsides \end{array}\right]) = P(O = -100) = 0.49$$
$$P(I = \left[\begin{array}{c} pepperonbothsides \end{array}\right]) = P(O = 100) = 0.09$$

Neglecting the cases where the probability distributions are independent of one another.

5 Solution 5

$$Input - image = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 1 & 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 2 & 1 & 0 & 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 & 0 & 1 & 2 & 3 \\ 5 & 4 & 3 & 2 & 1 & 0 & 1 & 2 \\ 6 & 5 & 4 & 3 & 2 & 1 & 0 & 1 \\ 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \end{bmatrix}$$

As we know that the median filter takes the median value of the mask by sorting the corresponding values of the pixel. Hence the output is as below, It also evident from the output image that the sudden change in the input image is changed to systematic change, **causing the image to smooth**

$$Output - image = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 1 & 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 1 & 1 & 1 & 2 & 3 & 4 & 5 \\ 3 & 2 & 1 & 1 & 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 & 1 & 1 & 2 & 3 \\ 5 & 4 & 3 & 2 & 1 & 1 & 1 & 2 \\ 6 & 5 & 4 & 3 & 2 & 1 & 1 & 1 \\ 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \end{bmatrix}$$

6 Solution 6 Compare the results from the median filter and the averaging filter

As per the equation, the input image becomes (keeping the border pixels as it is) ...

$$Input - image = \begin{bmatrix} 4 & 4 & 4 & 4 & 8 & 8 & 8 & 8 \end{bmatrix}$$
$$Median - filtered - image = \begin{bmatrix} 4 & 4 & 4 & 4 & 8 & 8 & 8 & 8 \end{bmatrix}$$
$$averaging - filter - image = \begin{bmatrix} 4 & 4 & 4 & 5 & 7 & 8 & 8 & 8 \end{bmatrix}$$
$$averaging - filter - image_with_border = \begin{bmatrix} 3 & 4 & 4 & 5 & 7 & 8 & 8 & 6 \end{bmatrix}$$

As the image is already smoothed and in terms of median filter, i.e it only has two frames of color, the median filter fails to smooth the image further, but when we apply the averaging filter we can see that it further smoothens the image. Hence the difference between the filters.

7 Solution 7 Apply the soble and Prewitt masks

Below is the algorithm with the output of images with Prewitt masks and Soble masks.

```
Image = 0 1 2 3 4 5 6 7
        1 0 1 2 3 4 5 6
        2 1 0 1 2 3 4 5
        3 2 1 0 1 2 3 4
        4 3 2 1 0 1 2 3
        5 4 3 2 1 0 1 2
        6 5 4 3 2 1 0 1
        7 6 5 4 3 2 1 0
Prewitt_magnitude = X X X X X X X X
                    X 0 5.65685 8.48528 8.48528 8.48528 8.48528 X
                    X 5.65685 0 5.65685 8.48528 8.48528 8.48528 X
                    X 8.48528 5.65685 0 5.65685 8.48528 8.48528 X
                    X 8.48528 8.48528 5.65685 0 5.65685 8.48528 X
                    X 8.48528 8.48528 8.48528 5.65685 0 5.65685 X
                    X 8.48528 8.48528 8.48528 8.48528 5.65685 0 X
                    X X X X X X X X
Prewitt_oreintation =
X X X X X X X X
X 0 0.785398 0.785398 0.785398 0.785398 0.785398 X
X -2.35619 0 0.785398 0.785398 0.785398 0.785398 X
X -2.35619 -2.35619 0 0.785398 0.785398 0.785398 X
X -2.35619 -2.35619 -2.35619 0 0.785398 0.785398 X
X -2.35619 -2.35619 -2.35619 -2.35619 0 0.785398 X
X -2.35619 -2.35619 -2.35619 -2.35619 -2.35619 0 X
X X X X X X X X
```

Listing 4: prewitt output

```

  void prewitt_mask(){
   Mat prewitt_x = (Mat_<float>(3,3) << -1,0,1,-1,0,1,-1,0,1);
   Mat prewitt_y = (Mat_<float>(3,3) << 1,1,1,0,0,0,-1,-1,-1);
   Mat gradient_x;
   Mat gradient_y;
   Mat img = Mat::zeros(8,8,CV_32F);
   for (uint8_t i = 0; i < 9; i++) {
       for (uint8_t r = 0; r < 9; r++) {
           img.at<float>(i,r) = abs(i - r);
           // cout << img.at<float>(i,r) << endl;
     }
   }
   filter2D(img,gradient_x,CV_32F,prewitt_x);
   filter2D(img,gradient_y,CV_32F,prewitt_y);
   Mat magnitude = Mat::zeros(img.rows, img.cols, CV_32F);
   Mat orientation = Mat::zeros(img.rows, img.cols, CV_32F);
   for (int i = 0; i < img.rows; i++) {
     for (int j = 0; j < img.cols; j++) {
       float gx = gradient_x.at<float>(i, j);
       float gy = gradient_y.at<float>(i, j);
       magnitude.at<float>(i, j) = sqrt(gx * gx + gy * gy);
       orientation.at<float>(i, j) = atan2(gy, gx);
     }
   }
   for (int i = 0; i < img.rows; i++) {
       for (int j = 0; j < img.cols; j++) {
           cout << orientation.at<float>(i, j) << " ";
       }
       cout << endl;
 }

}
```

Listing 5: Algorithm for gradient calculations with Prewitt masks

```

    void soble_masks(){
     Mat soble_x = (Mat_<float>(3,3) << -1,0,1,-2,0,2,-1,0,1);
     Mat soble_y = (Mat_<float>(3,3) << -1,-2,-1,0,0,0,1,2,1);
     Mat gradient_x;
     Mat gradient_y;
     Mat img = Mat::zeros(8,8,CV_32F);
     for (uint8_t i = 0; i < 9; i++) {
         for (uint8_t r = 0; r < 9; r++) {
             img.at<float>(i,r) = abs(i - r);
             // cout << img.at<float>(i,r) << endl;
       }
     }
     filter2D(img,gradient_x,CV_32F,soble_x);
     filter2D(img,gradient_y,CV_32F,soble_y);
     Mat magnitude = Mat::zeros(img.rows, img.cols, CV_32F);
     Mat orientation = Mat::zeros(img.rows, img.cols, CV_32F);
     for (int i = 0; i < img.rows; i++) {
       for (int j = 0; j < img.cols; j++) {
         float gx = gradient_x.at<float>(i, j);
         float gy = gradient_y.at<float>(i, j);
         magnitude.at<float>(i, j) = sqrt(gx * gx + gy * gy);
         orientation.at<float>(i, j) = atan2(gy, gx);
       }
     }
     for (int i = 0; i < img.rows; i++) {
         for (int j = 0; j < img.cols; j++) {
           cout << img.at<float>(i, j) << " ";
         }
         cout << endl;
  }

}
```

As from the below output is is evident that the Sobel mask will make edges more visible as it will give more weight to the change unlike Prewitt which just gives uniform weight to the change.

```
Magnitude =
 X X X X X X X X
X 0 8.48528 11.3137 11.3137 11.3137 11.3137 X
X 8.48528 0 8.48528 11.3137 11.3137 11.3137 X
X 11.3137 8.48528 0 8.48528 11.3137 11.3137 X
X 11.3137 11.3137 8.48528 0 8.48528 11.3137 X
X 11.3137 11.3137 11.3137 8.48528 0 8.48528 X
X 11.3137 11.3137 11.3137 11.3137 8.48528 0 X
X X X X X X X X


Orientation =
 X X X X X X X X
X 0 -0.785398 -0.785398 -0.785398 -0.785398 -0.785398 X
X 2.35619 0 -0.785398 -0.785398 -0.785398 -0.785398 X
X 2.35619 2.35619 0 -0.785398 -0.785398 -0.785398 X
X 2.35619 2.35619 2.35619 0 -0.785398 -0.785398 X
X 2.35619 2.35619 2.35619 2.35619 0 -0.785398 X
X 2.35619 2.35619 2.35619 2.35619 2.35619 0 X
 X X X X X X X X
```
Listing 6: output of Sobel masks

8 Solution 8 Corner detection with checkrboard

```
void corner_detecy(){

    Mat Zeroes = Mat::zeros(10,10,CV_32F);
    Mat Onees = 40*Mat::ones(10,10,CV_32F);
    Mat img;
    vconcat(Zeroes, Onees, img);
    Mat img3;
    vconcat(Onees,Zeroes, img3);
    Mat img2;
    hconcat(img, img3, img2);
    Mat Ix = Mat::zeros(20,20,CV_32F);
    Mat Iy = Mat::zeros(20,20,CV_32F);
    for (int i = 0; i < img2.rows; i++) {
          for (int j = 0; j < img2.cols; j++) {
            cout << img2.at<float>(i, j) << " ";
          }
          cout << endl;
    }
    Mat prewitt_x = (Mat_<float>(3,3) << -1,0,1,-1,0,1,-1,0,1);
    Mat prewitt_y = (Mat_<float>(3,3) << 1,1,1,0,0,0,-1,-1,-1);
    Mat gradient_x;
    Mat gradient_y;
    filter2D(img2,gradient_x,CV_32F,prewitt_x);
    filter2D(img2,gradient_y,CV_32F,prewitt_y);
    Mat C = Mat::zeros(img2.rows, img2.cols, CV_32F);
    for (int i = 0; i < img2.rows; i++) {
      for (int j = 0; j < img2.cols; j++) {
        float gx = gradient_x.at<float>(i, j);
        float gy = gradient_y.at<float>(i, j);
        C.at<float>(i, j) = gx * gx + gy * gy;
        // cout << C.at<Vec2f>(i, j) << " ";
      }
    }
    Mat eigenvalues = Mat::zeros(20,20,CV_32F);
    eigen(C,eigenvalues);
    cout << eigenvalues << endl;
}
```

Listing 7: Algorithm for Corner detection

output for the eigenvalues.

```
[89659.234;
 0.00028402629;
 2.181193e-10;
 8.4831979e-13;
 2.6463508e-18;
 4.6475055e-19;
 2.4648248e-19;
 4.0297945e-20;
 3.6269624e-21;
 -7.5241356e-22;
 -2.2927854e-19;
 -3.3085412e-19;
 -6.2439268e-19;
 -1.1899289e-18;
 -3.647931e-18;
 -3.1345627e-12;
 −4.4699196e−11;
 −2.0790447e−10;
 −0.0046110945;
 −83259.234]
```

Listing 8: Eigenvalues

Since we consider a neighborhood of 7×7, the border of width 3 can also be disregarded, and thus we only calculated those valid pixels of the 14×14 matrix in the center. For each pixel, its C matrix can be given by

$$C = \begin{bmatrix} \sum E_x^2 & \sum E_x E_y \\ \sum E_x E_y & \sum E_y^2 \end{bmatrix}$$

For each C matrix, we can calculate its eigenvalues λ_1, λ_2, and select $min\{\lambda_1, \lambda_2\}$.