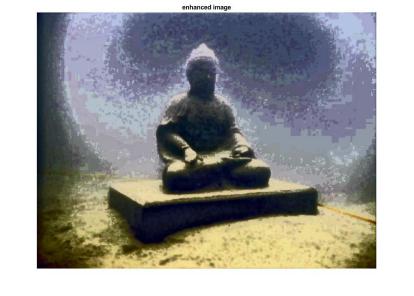
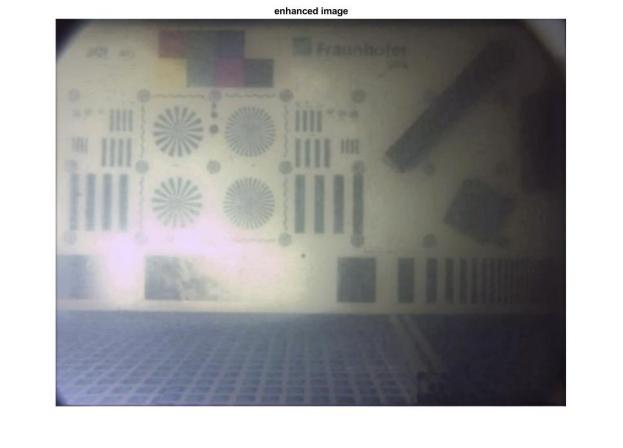
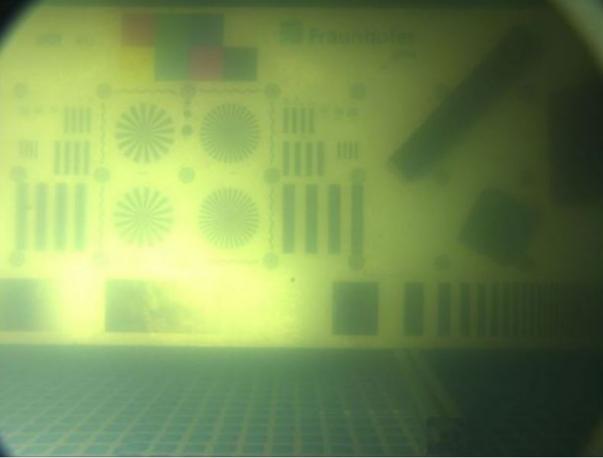
**Image Deblurring**

****

****

****

****

Bilateral Filter

function output = bilateralFilter( data, edge, edgeMin, edgeMax,...

sigmaSpatial, sigmaRange, samplingSpatial, samplingRange )

if( ndims( data ) > 2 ),

error( 'data must be a greyscale image with size [ height, width ]' );

end

if( ~isa( data, 'double' ) ),

error( 'data must be of class "double"' );

end

if ~exist( 'edge', 'var' ),

edge = data;

elseif isempty( edge ),

edge = data;

end

if( ndims( edge ) > 2 ),

error( 'edge must be a greyscale image with size [ height, width ]' );

end

if( ~isa( edge, 'double' ) ),

error( 'edge must be of class "double"' );

end

inputHeight = size( data, 1 );

inputWidth = size( data, 2 );

if ~exist( 'edgeMin', 'var' ),

edgeMin = min( edge( : ) );

%warning( 'edgeMin not set! Defaulting to: %f\n', edgeMin );

end

if ~exist( 'edgeMax', 'var' ),

edgeMax = max( edge( : ) );

%warning( 'edgeMax not set! Defaulting to: %f\n', edgeMax );

end

edgeDelta = edgeMax - edgeMin;

if ~exist( 'sigmaSpatial', 'var' ),

sigmaSpatial = min( inputWidth, inputHeight ) / 16;

%fprintf( 'Using default sigmaSpatial of: %f\n', sigmaSpatial );

end

if ~exist( 'sigmaRange', 'var' ),

sigmaRange = 0.1 \* edgeDelta;

%fprintf( 'Using default sigmaRange of: %f\n', sigmaRange );

end

if ~exist( 'samplingSpatial', 'var' ),

samplingSpatial = sigmaSpatial;

end

if ~exist( 'samplingRange', 'var' ),

samplingRange = sigmaRange;

end

if size( data ) ~= size( edge ),

error( 'data and edge must be of the same size' );

end

% parameters

derivedSigmaSpatial = sigmaSpatial / samplingSpatial;

derivedSigmaRange = sigmaRange / samplingRange;

paddingXY = floor( 2 \* derivedSigmaSpatial ) + 1;

paddingZ = floor( 2 \* derivedSigmaRange ) + 1;

% allocate 3D grid

downsampledWidth = floor( ( inputWidth - 1 ) / samplingSpatial )...

+ 1 + 2 \* paddingXY;

downsampledHeight = floor( ( inputHeight - 1 ) / samplingSpatial )...

+ 1 + 2 \* paddingXY;

downsampledDepth = floor( edgeDelta / samplingRange ) + 1 + 2 \* paddingZ;

gridData = zeros( downsampledHeight, downsampledWidth, downsampledDepth );

gridWeights = zeros( downsampledHeight, downsampledWidth, downsampledDepth );

% compute downsampled indices

[ jj, ii ] = meshgrid( 0 : inputWidth - 1, 0 : inputHeight - 1 );

di = round( ii / samplingSpatial ) + paddingXY + 1;

dj = round( jj / samplingSpatial ) + paddingXY + 1;

dz = round( ( edge - edgeMin ) / samplingRange ) + paddingZ + 1;

for k = 1 : numel( dz ),

dataZ = data( k ); % traverses the image column wise, same as di( k )

if ~isnan( dataZ ),

dik = di( k );

djk = dj( k );

dzk = dz( k );

gridData( dik, djk, dzk ) = gridData( dik, djk, dzk ) + dataZ;

gridWeights( dik, djk, dzk ) = gridWeights( dik, djk, dzk ) + 1;

end

end

% make gaussian kernel

kernelWidth = 2 \* derivedSigmaSpatial + 1;

kernelHeight = kernelWidth;

kernelDepth = 2 \* derivedSigmaRange + 1;

halfKernelWidth = floor( kernelWidth / 2 );

halfKernelHeight = floor( kernelHeight / 2 );

halfKernelDepth = floor( kernelDepth / 2 );

[gridX, gridY, gridZ] = meshgrid( 0 : kernelWidth - 1,...

0 : kernelHeight - 1, 0 : kernelDepth - 1 );

gridX = gridX - halfKernelWidth;

gridY = gridY - halfKernelHeight;

gridZ = gridZ - halfKernelDepth;

gridRSquared = ( gridX .\* gridX + gridY .\* gridY ) /...

( derivedSigmaSpatial \* derivedSigmaSpatial ) +...

( gridZ .\* gridZ ) / ( derivedSigmaRange \* derivedSigmaRange );

kernel = exp( -0.5 \* gridRSquared );

% convolve

blurredGridData = convn( gridData, kernel, 'same' );

blurredGridWeights = convn( gridWeights, kernel, 'same' );

blurredGridWeights( blurredGridWeights == 0 ) = -2;

normalizedBlurredGrid = blurredGridData ./ blurredGridWeights;

normalizedBlurredGrid( blurredGridWeights < -1 ) = 0;

[ jj, ii ] = meshgrid( 0 : inputWidth - 1, 0 : inputHeight - 1 );

di = ( ii / samplingSpatial ) + paddingXY + 1;

dj = ( jj / samplingSpatial ) + paddingXY + 1;

dz = ( edge - edgeMin ) / samplingRange + paddingZ + 1;

output = interpn( normalizedBlurredGrid, di, dj, dz );

Convolution Function

function [ output\_data ] = convolution(input\_data, weights\_conv, biases\_conv)

weights\_conv=double(weights\_conv);

biases\_conv=double(biases\_conv);

hei = size(input\_data,1);

wid = size(input\_data,2);

[conv\_channels,conv\_patchsize2,conv\_filters] = size(weights\_conv);

conv\_patchsize = sqrt(conv\_patchsize2);

output\_data = zeros(hei, wid, conv\_filters);

for i = 1 : conv\_filters

for j = 1 : conv\_channels

conv\_subfilter = reshape(weights\_conv(j,:,i), conv\_patchsize, conv\_patchsize);

output\_data(:,:,i) = output\_data(:,:,i) + imfilter(input\_data(:,:,j), conv\_subfilter, 'same', 'symmetric');

end

output\_data(:,:,i) = output\_data(:,:,i) + biases\_conv(i);

end

End

Dehaze Function 1

function [ radiance ] = dehaze( image, omega, win\_size, lambda )

if ~exist('omega', 'var')

omega = 0.95;

end

if ~exist('win\_size', 'var')

win\_size = 15;

end

if ~exist('lambda', 'var')

lambda = 0.0001;

end

[m, n, ~] = size(image);

dark\_channel = get\_dark\_channel(image, win\_size);

atmosphere = get\_atmosphere(image, dark\_channel);

trans\_est = get\_transmission\_estimate(image, atmosphere, omega, win\_size);

L = get\_laplacian(image);

A = L + lambda \* speye(size(L));

b = lambda \* trans\_est(:);

x = A \ b;

transmission = reshape(x, m, n);

radiance = get\_radiance(image, transmission, atmosphere);

end

Dehaze Function 2

function [ radiance1,radiance2 ] = dehaze\_fast( image, omega, win\_size )

if ~exist('omega', 'var')

omega = 0.95;

end

if ~exist('win\_size', 'var')

win\_size = 30;

end

r = 15;

res = 0.001;

[m, n, ~] = size(image);

dark\_channel = get\_dark\_channel(image, win\_size);

atmosphere = get\_atmosphere(image, dark\_channel);

trans\_est = get\_transmission\_estimate(image, atmosphere, omega, win\_size);

% ????

x1 = guided\_filter(rgb2gray(image), trans\_est, r, res);

r = 0;

N=[1,-2,1];

M=[1;-2;1];

dim=size(trans\_est);

for i=1:(dim(1)/2)

for j=1:(dim(2)/2)

p=abs(conv(M,conv(N,trans\_est(i,j))));

r=p+r;

end

end

q=1.66\*p/(dim(1)\*dim(2))

w = 3;

sigma = [3 q];

x2=bilateral\_filter(abs(trans\_est),w,sigma);%????????

transmission1 = reshape(x1, m, n);

transmission2 = reshape(x2, m, n);

radiance1 = get\_radiance(image, transmission1, atmosphere);

radiance2 = get\_radiance(image, transmission2, atmosphere);

End

Window Function

function sum\_img = window\_sum\_filter(image, r)

[h, w] = size(image);

sum\_img = zeros(size(image));

% Y axis

im\_cum = cumsum(image, 1);

sum\_img(1:r+1, :) = im\_cum(1+r:2\*r+1, :);

sum\_img(r+2:h-r, :) = im\_cum(2\*r+2:h, :) - im\_cum(1:h-2\*r-1, :);

sum\_img(h-r+1:h, :) = repmat(im\_cum(h, :), [r, 1]) - im\_cum(h-2\*r:h-r-1, :);

% X axis

im\_cum = cumsum(sum\_img, 2);

sum\_img(:, 1:r+1) = im\_cum(:, 1+r:2\*r+1);

sum\_img(:, r+2:w-r) = im\_cum(:, 2\*r+2:w) - im\_cum(:, 1:w-2\*r-1);

sum\_img(:, w-r+1:w) = repmat(im\_cum(:, w), [1, r]) - im\_cum(:, w-2\*r:w-r-1);

end

**YOLOv5 Training**

"""Yolov5-Manual\_Training.ipynb

Automatically generated by Colaboratory.

"""

!git clone https://github.com/ultralytics/yolov5

!cd yolov5/

!ls

cd yolov5

!ls

!pip install -r requirements.txt

!ls

# Commented out IPython magic to ensure Python compatibility.

# %cd ..

from google.colab import drive

drive.mount('/content/gdrive')

!ln -s /content/gdrive/My\ Drive/ /mydrive

!ls /mydrive/yolov3

!cp /mydrive/yolov3/UNDERWATER\_DATA.zip ../

!ls

!unzip UNDERWATER\_DATA.zip

!python train.py --img 768 --batch 8 --epochs 50 --data underwater.yaml --cfg models/yolov5s.yaml --weights yolov5s.pt --device 0

!wandb login --relogin # use 'wandb disabled' or 'wandb enabled' to disable or enable

!wandb enabled

!python detect.py --source Sample/ --weights yolov5s.pt --conf 0.5

