COT 5930 Digital Image Processing Spring 2021

Christian Garbin

Image denoising using deep learning

(C) Oge Marques, PhD - 2020-2021

Goal: Build and evaluate image denoising solutions using deep learning architectures.

Learning objectives:

- Learn how to implement an Image Processing **workflow** in MATLAB
- Learn how to implement and evaluate **contemporary** (deep-learning-based) image denoising techniques in MATLAB
- Get acquainted with representative **datasets** and problems in image denoising

Table of Contents

Part 1: Noise types and image quality metrics

Effects of different noise types

`imnoise()` allows you to add noise to an image. You can choose from different noise types: Gaussian, Poisson, salt-and-pepper.

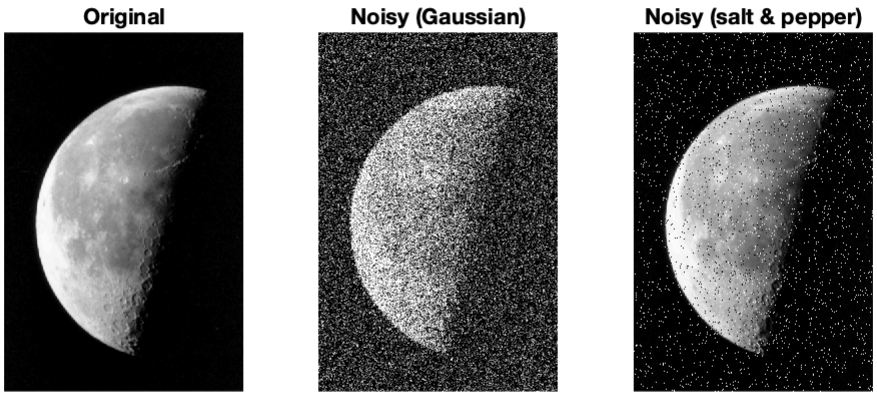
Suggested steps:

1. Load test image and add noise.
2. Try out a few noise types, and play with associated parameters.
3. Visualize the image before and after adding noise.

Example code:

```
I = imread('moon.tif');
Jg = imnoise(I, 'gaussian', 0, 0.2);     % Add gaussian noise with zero mean and varianc
Jsp = imnoise(I,'salt & pepper');        % Add salt-and-pepper noise
```

```
% Display images
figure('Name','Adding noise to an image');
subplot(1,3,1), imshow(I), title('Original')
subplot(1,3,2), imshow(Jg), title ('Noisy (Gaussian)')
subplot(1,3,3), imshow(Jsp), title ('Noisy (salt & pepper)')
```



Assess different image quality metrics

"Image quality can degrade due to distortions during image acquisition and processing. Examples of distortion include noise, blurring, ringing, and compression artifacts.

Efforts have been made to create objective measures of quality. For many applications, a valuable quality metric correlates well with the subjective perception of quality by a human observer. Quality metrics can also track unperceived errors as they propagate through an image processing pipeline, and can be used to compare image processing algorithms."

Check Image Quality Metrics for more details.

Suggested steps:

1. Load test image and add noise.
2. Measure the quality of the resulting (noisy) image (and compare with the original, when appropriate).
3. Visualize the image before and after adding noise.

Example code:

Mean-squared error (`immse`)

```
I = imread('moon.tif');
Jg = imnoise(I, 'gaussian');         % Add gaussian noise
Jsp = imnoise(I,'salt & pepper');    % Add salt-and-pepper noise

err1 = immse(Jg, I);
fprintf('\n The mean-squared error between the noisy image and the original image is %0
```

```
 The mean-squared error between the noisy image and the original image is 421.3349
```

Peak Signal-to-noise ratio (`psnr`)

```
[peaksnr, snr] = psnr(Jg, I);
fprintf('\n The Peak-SNR value is %0.4f', peaksnr);
```

```
 The Peak-SNR value is 21.8845
```

```
fprintf('\n The SNR value is %0.4f \n', snr);
```

```
 The SNR value is 13.2401
```

Structural similarity index (`ssim`)

```
[ssimval,ssimmap] = ssim(Jg,I);
% figure('Name','Local SSIM Map');
figure
imshow(ssimmap,[])
```

```
% title(['Local SSIM Map with Global SSIM Value: ',num2str(ssimval)])  % Commented out
fprintf('\n The Global SSIM Value is %0.6f \n', ssimval);
```

```
The Global SSIM Value is 0.178550
```

Blind/Referenceless Image Spatial Quality Evaluator (`brisque`), no-reference image quality score

```
brisqueI = brisque(I);
fprintf('\nBRISQUE score for original image is %0.4f.\n',brisqueI)
```

```
BRISQUE score for original image is 14.0315.
```

```
brisqueJg = brisque(Jg);
fprintf('BRISQUE score for noisy (Gaussian) image is %0.4f.\n',brisqueJg)
```

```
BRISQUE score for noisy (Gaussian) image is 44.3032.
```

```
brisqueJsp = brisque(Jsp);
fprintf('BRISQUE score for noisy (Salt & Pepper) image is %0.4f.\n',brisqueJsp)
```

```
BRISQUE score for noisy (Salt & Pepper) image is 47.1000.
```

Naturalness Image Quality Evaluator (`niqe`), no-reference image quality score

```
niqeI = niqe(I);
fprintf('\nNIQE score for original image is %0.4f.\n',niqeI)
```

```
NIQE score for original image is 2.8985.
```

```
niqeJg = niqe(Jg);
fprintf('NIQE score for noisy (Gaussian) image is %0.4f.\n',niqeJg)
```

```
NIQE score for noisy (Gaussian) image is 17.0867.
```

```
niqeJsp = niqe(Jsp);
fprintf('NIQE score for noisy (Salt & Pepper) image is %0.4f.\n',niqeJsp)
```

```
NIQE score for noisy (Salt & Pepper) image is 20.7053.
```

Perception based Image Quality Evaluator (`piqe`), no-reference image quality score

```
piqeI = piqe(I);
fprintf('\nPIQE score for original image is %0.4f.\n',piqeI)
```

```
PIQE score for original image is 23.1441.
```

```
piqeJg = piqe(Jg);
fprintf('PIQE score for noisy (Gaussian) image is %0.4f.\n',piqeJg)
```

```
PIQE score for noisy (Gaussian) image is 74.8382.
```

```
piqeJsp = piqe(Jsp);
fprintf('PIQE score for noisy (Salt & Pepper) image is %0.4f.\n',piqeJsp)
```

```
PIQE score for noisy (Salt & Pepper) image is 73.0916.
```

Your turn (step 3 of the guidelines):

(OPTIONAL) Expand "Part 1" of the starter code to include other test images, noise types (and their parameters) and image quality metrics.

Select an image:

```
[file, path] = uigetfile({'*.png;*.jpg;*.jpeg;*.tif;*,bmp', 'Pictures'},...
               'Select an image');

% Set to the default file if none was picked, so the rest of the code works
if file == 0
    file = 'aeroflot_sukhoi_superjet_100.png';
    [path, name, ext] = fileparts(matlab.desktop.editor.getActiveFilename);
    path = append(path, '/');
end
```

Load the original image as a grayscale image.

```
imgOriginal = im2gray(imread([path file]));
imshow(imgOriginal);
```

Apply different types of noises to the image.

```
imgGaussian = imnoise(imgOriginal, 'gaussian');
imgSaltPepper05 = imnoise(imgOriginal, 'salt & pepper', 0.05);
imgSaltPepper10 = imnoise(imgOriginal, 'salt & pepper', 0.1);
imgSpeckle = imnoise(imgOriginal, 'speckle');
```

Show the original and noisy images. To make it easy to compare the noisy image with the original one, the top row shows the original image (repeated) and the bottom row is one of the modified images.

```
montage({imgOriginal, imgOriginal, imgOriginal, imgOriginal, ...
```

```
    imgGaussian, imgSaltPepper05, imgSaltPepper10, imgSpeckle}, ...
    'Size', [2 4], 'BorderSize', [1 10], 'BackgroundColor', 'white');
```

The following code calculates and shows the image quality metrics (see table after the code block).

Note that the first row is the original image. Some of the metrics (e.g. MSE, SNR, and SSIM) may not make much sense in that case. It was done this one to simplify the code.

We can make a few observations, based on the metrics shown in the following table:

1. The original image has a significantly lower NIQE, as expected.
2. Speckle noise with the default variance of 0.05 and a 5% salt and pepper noise produce similar quality metrics, except for SSIM and NIQUE. The larger NIQE for the speckle image can be noticed by the much more perturbed sky behind the airplane's tail (last column above).

```
images = {imgOriginal imgGaussian imgSaltPepper05 imgSaltPepper10 imgSpeckle};
imageTitle = {'Original' 'Gaussian' '5% Salt & Pepper' '10% Salt & Pepper' 'Speckle'};
numImages = numel(images);

qualityMetrics = repmat(struct('Image', "", 'MSE', 0.0, 'PeakSNR', 0.0, ...
   'SNR', 0.0, 'SSIM', 0.0, 'BRISQUE', 0.0, 'NIQE', 0.0, 'PIQE', 0.0), ...
   1, numImages);

for i = 1:numImages
    qualityMetrics(i).Image = imageTitle(i);
    img = images{:, i};

    qualityMetrics(i).MSE = immse(img, imgOriginal);
    [peaksnr, snr] = psnr(img, imgOriginal);
    qualityMetrics(i).PeakSNR = peaksnr;
    qualityMetrics(i).SNR = snr;
    [ssimval, ssimmap] = ssim(img, imgOriginal);
    qualityMetrics(i).SSIM = ssimval;
    qualityMetrics(i).BRISQUE = brisque(img);
    qualityMetrics(i).NIQE = niqe(img);
    qualityMetrics(i).PIQE = piqe(img);
```

```
end

struct2table(qualityMetrics)
```

```
ans = 5×8 table
```

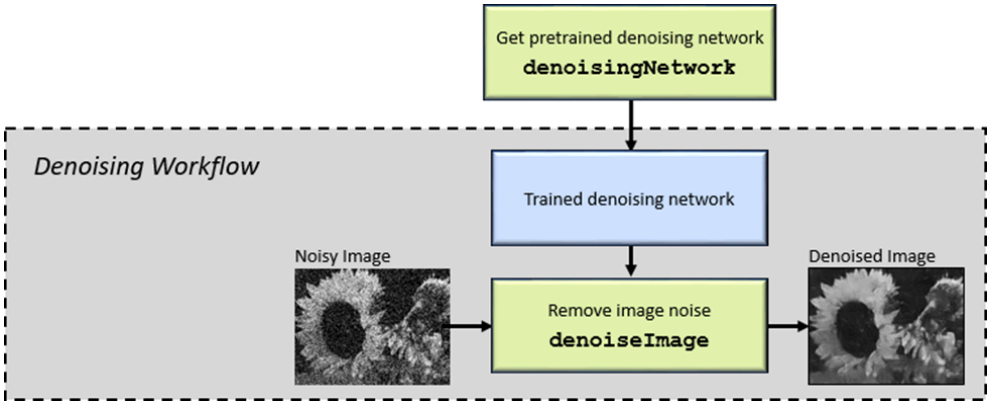
	Image	MSE	PeakSNR	SNR	SSIM	BRISQUE	NIQE	PIQE
1	'Original'	0	Inf	Inf	1.0000	25.1869	3.6370	52.6345
2	'Gaussian'	615.1974	20.2407	16.4625	0.2432	47.5849	15.8531	70.8777
3	'5% Salt & ...	1.0258e+03	18.0204	14.2422	0.3135	47.0069	21.8349	73.9226
4	'10% Salt &...	2.0777e+03	14.9550	11.1769	0.1649	43.4649	23.9142	77.0393
5	'Speckle'	1.1820e+03	17.4045	13.6264	0.2326	45.2730	55.1568	73.6300

Go back to top

Part 2: Denoising using pretrained deep neural network

There are many ways to remove noise from images. The simplest workflow for removing image noise from an individual image using deep learning is to use a pretrained denoising neural network (`DnCNN`), and is illustrated in the figure below.

Please refer to MATLAB documentation for more details.



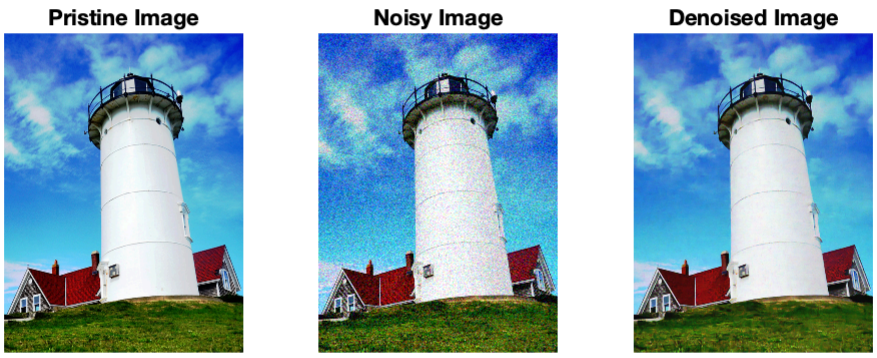
Suggested steps:

1. Load test image, covert it to appropriate class (if needed) and split it into R, G, and B channels.
2. Load the pretrained DnCNN network.
3. Use the DnCNN network to remove noise from each color channel.
4. Recombine the denoised color channels to form the denoised RGB image.
5. Display the original, noisy, and denoised color image.
6. (OPTIONAL) Explore the details of the denoising convolutional neural network (DnCNN).

Example code:

Run this section of the starter code and ensure it works as intended.

```
% Load RGB image
pristineRGB = imread('lighthouse.png');
% Convert to double
pristineRGB = im2double(pristineRGB);
% Add Gaussian noise
noisyRGB = imnoise(pristineRGB,'gaussian',0,0.01);

% Split the noisy RGB image into its individual color channels.
[noisyR,noisyG,noisyB] = imsplit(noisyRGB);

% Load the pretrained DnCNN network
preTrainedDnCNN = denoisingNetwork('dncnn');

% Use the DnCNN network to remove noise from each color channel
denoisedR = denoiseImage(noisyR,preTrainedDnCNN);
denoisedG = denoiseImage(noisyG,preTrainedDnCNN);
denoisedB = denoiseImage(noisyB,preTrainedDnCNN);

% Recombine the denoised color channels to form the denoised RGB image
denoisedRGB = cat(3,denoisedR,denoisedG,denoisedB);

% Display the original, noisy, and denoised color image
figure('Name','Denoising with pretrained DnCNN');
subplot(1,3,1), imshow(pristineRGB), title('Pristine Image')
subplot(1,3,2), imshow(noisyRGB), title('Noisy Image')
subplot(1,3,3), imshow(denoisedRGB), title('Denoised Image')
```



Your turn (step 5 of the guidelines):

Write code to apply the DnCNN-based denoising technique of "Part 2" to at least 2 (two) different test images of your choice, using both Gaussian and salt-and-pepper noise, and compare their results using at least 2 (two) image quality metrics.

Summarize your results in a table (see *Guidelines* for additional information).

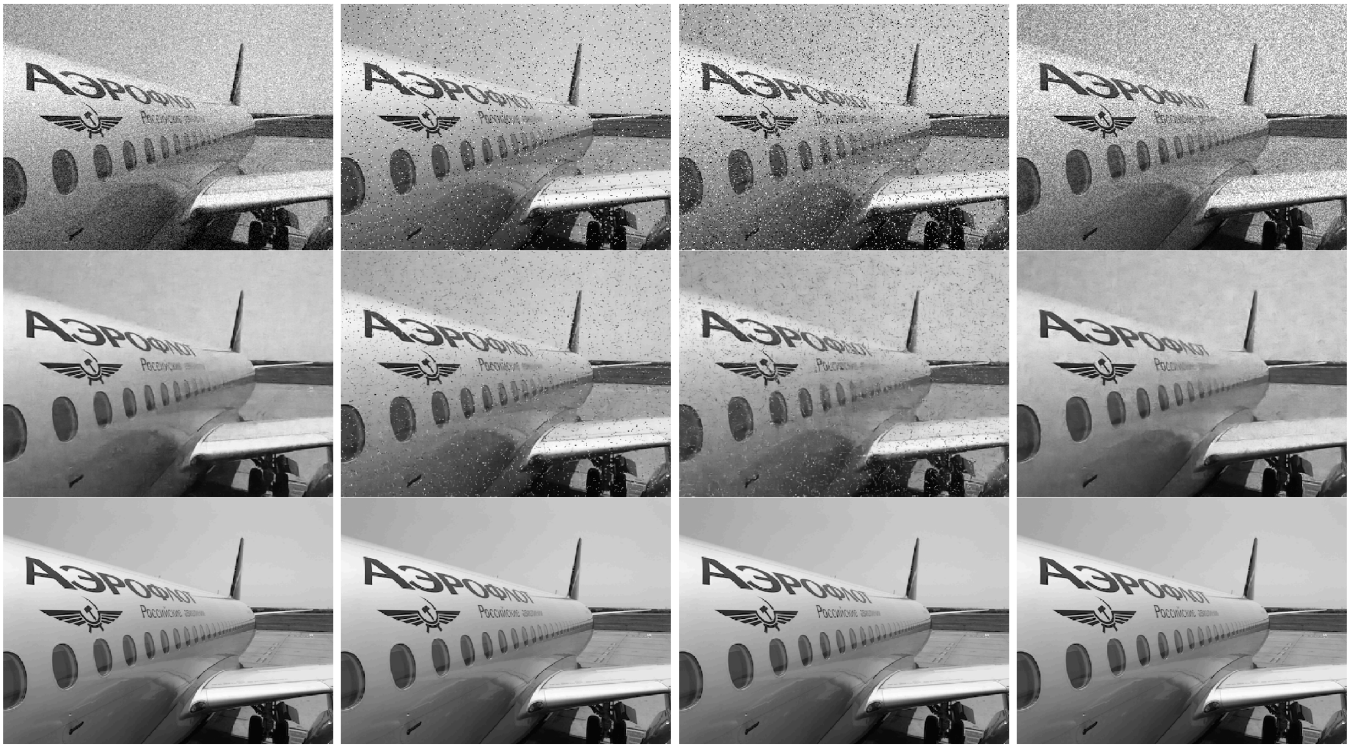
Part 1 - Denoise images

```
imgGaussianDns = denoiseImage(imgGaussian, preTrainedDnCNN);
imgSaltPepper05Dns = denoiseImage(imgSaltPepper05, preTrainedDnCNN);
imgSaltPepper10Dns = denoiseImage(imgSaltPepper10, preTrainedDnCNN);
imgSpeckleDns = denoiseImage(imgSpeckle, preTrainedDnCNN);
```

Part 2 - Show images

Show the noisy image (top row), denoised image (middle row), and the original image (repeated in the bottom row).

```
montage({imgGaussian, imgSaltPepper05, imgSaltPepper10, imgSpeckle, ...
    imgGaussianDns, imgSaltPepper05Dns, imgSaltPepper10Dns, imgSpeckleDns, ...
    imgOriginal, imgOriginal, imgOriginal, imgOriginal}, ...
    'Size', [3 4], 'BorderSize', [1 10], 'BackgroundColor', 'white');
```



Part 3 - Show metrics

The code below calculates and shows the image qulaity metrics. The first row of the resulting table is the original image. The following rows show one of the noisy images, followed by its denoised version (prefixed with "(D)").

We can make a few observations, based on the metrics shown in the following table:

1. From the MSE values: Gaussian and speckle noise recovered significantly better than salt and pepper.
2. This is also visible in the SSIM values, where salt and pepper are in the mid 0.5 range, while Gaussian and speckle are over 0.8.
3. Interestingly, the subjective PIQE index indicates that the denoised Gaussian and speckle images are of better quality than the original image. Something to investigate later, if time permits.

The observation above can be confirmed visually in the image montage. The Gaussian image is the best denoised version ("best" = subjective user perception). All other images have readily noticed artifacts, especially the salt and pepper ones.

It seems that the neural network is better at recovering "natural" noise than structured noise.

```
images = {imgOriginal imgGaussian imgGaussianDns imgSaltPepper05 ...
    imgSaltPepper05Dns imgSaltPepper10 imgSaltPepper10Dns imgSpeckle ...
    imgSpeckleDns};
imageTitle = {'Original' 'Gaussian' '(D) Gaussian' '5% Salt & Pepper' ...
    '(D) 5% Salt & Pepper' '10% Salt & Pepper' '(D) 10% Salt & Pepper' ...
    'Speckle' '(D) Speckle'};
numImages = numel(images);

qualityMetrics = repmat(struct('Image', "", 'MSE', 0.0, 'PeakSNR', 0.0, ...
   'SNR', 0.0, 'SSIM', 0.0, 'BRISQUE', 0.0, 'NIQE', 0.0, 'PIQE', 0.0), ...
   1, numImages);

for i = 1:numImages
    qualityMetrics(i).Image = imageTitle(i);
    img = images{:, i};

    qualityMetrics(i).MSE = immse(img, imgOriginal);
    [peaksnr, snr] = psnr(img, imgOriginal);
    qualityMetrics(i).PeakSNR = peaksnr;
    qualityMetrics(i).SNR = snr;
    [ssimval, ssimmap] = ssim(img, imgOriginal);
    qualityMetrics(i).SSIM = ssimval;
    qualityMetrics(i).BRISQUE = brisque(img);
    qualityMetrics(i).NIQE = niqe(img);
    qualityMetrics(i).PIQE = piqe(img);
end

struct2table(qualityMetrics)
```

ans = 9×8 table

	Image	MSE	PeakSNR	SNR	SSIM	BRISQUE	NIQE	PIQE
1	'Original'	0	Inf	Inf	1.0000	25.1869	3.6370	52.6345
2	'Gaussian'	615.1974	20.2407	16.4625	0.2432	47.5849	15.8531	70.8777

	Image	MSE	PeakSNR	SNR	SSIM	BRISQUE	NIQE	PIQE
3	'(D) Gaussian'	48.3718	31.2849	27.5068	0.8718	25.8007	3.4973	14.9477
4	'5% Salt & ...	1.0258e+03	18.0204	14.2422	0.3135	47.0069	21.8349	73.9226
5	'(D) 5% Sal...	260.3123	23.9759	20.1977	0.5672	49.7801	17.4271	46.7170
6	'10% Salt &...	2.0777e+03	14.9550	11.1769	0.1649	43.4649	23.9142	77.0393
7	'(D) 10% Sa...	267.8598	23.8517	20.0736	0.5720	45.9773	9.1498	38.2220
8	'Speckle'	1.1820e+03	17.4045	13.6264	0.2326	45.2730	55.1568	73.6300
9	'(D) Speckle'	77.4803	29.2389	25.4608	0.8390	43.4562	5.5616	15.1407

Your turn (step 7 of the guidelines):

(OPTIONAL) Add code to inspect/analyze the selected pretrained neural network in more detail.

Uncomment the line below to see an interactive network explorer.

```
%analyzeNetwork(preTrainedDnCNN)
```

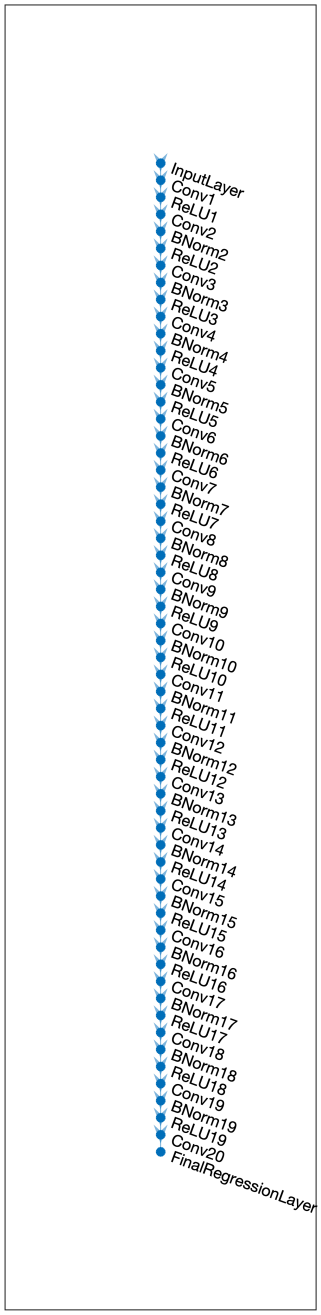
The next pieces of code explore the network programattically. Ths is not the best method for this relatively large network. It is shown here as an example for cases where it may make more sense (smaller networks).

```
layers = dnCNNLayers
```

```
layers =
  1×59 Layer array with layers:

     1   'InputLayer'            Image Input           50×50×1 images
     2   'Conv1'                 Convolution           64 3×3×1 convolutions with stride [1  1] and paddi
     3   'ReLU1'                 ReLU                  ReLU
     4   'Conv2'                 Convolution           64 3×3×64 convolutions with stride [1  1] and padd
     5   'BNorm2'                Batch Normalization   Batch normalization with 64 channels
     6   'ReLU2'                 ReLU                  ReLU
     7   'Conv3'                 Convolution           64 3×3×64 convolutions with stride [1  1] and padd
     8   'BNorm3'                Batch Normalization   Batch normalization with 64 channels
     9   'ReLU3'                 ReLU                  ReLU
    10   'Conv4'                 Convolution           64 3×3×64 convolutions with stride [1  1] and padd
    11   'BNorm4'                Batch Normalization   Batch normalization with 64 channels
    12   'ReLU4'                 ReLU                  ReLU
    13   'Conv5'                 Convolution           64 3×3×64 convolutions with stride [1  1] and padd
    14   'BNorm5'                Batch Normalization   Batch normalization with 64 channels
    15   'ReLU5'                 ReLU                  ReLU
    16   'Conv6'                 Convolution           64 3×3×64 convolutions with stride [1  1] and padd
    17   'BNorm6'                Batch Normalization   Batch normalization with 64 channels
    18   'ReLU6'                 ReLU                  ReLU
    19   'Conv7'                 Convolution           64 3×3×64 convolutions with stride [1  1] and padd
    20   'BNorm7'                Batch Normalization   Batch normalization with 64 channels
    21   'ReLU7'                 ReLU                  ReLU
    22   'Conv8'                 Convolution           64 3×3×64 convolutions with stride [1  1] and padd
    23   'BNorm8'                Batch Normalization   Batch normalization with 64 channels
    24   'ReLU8'                 ReLU                  ReLU
    25   'Conv9'                 Convolution           64 3×3×64 convolutions with stride [1  1] and padd
    26   'BNorm9'                Batch Normalization   Batch normalization with 64 channels
    27   'ReLU9'                 ReLU                  ReLU
    28   'Conv10'                Convolution           64 3×3×64 convolutions with stride [1  1] and padd
    29   'BNorm10'               Batch Normalization   Batch normalization with 64 channels
    30   'ReLU10'                ReLU                  ReLU
```

```
 31   'Conv11'                Convolution           64 3×3×64 convolutions with stride [1  1] and padd
 32   'BNorm11'               Batch Normalization   Batch normalization with 64 channels
 33   'ReLU11'                ReLU                  ReLU
 34   'Conv12'                Convolution           64 3×3×64 convolutions with stride [1  1] and padd
 35   'BNorm12'               Batch Normalization   Batch normalization with 64 channels
 36   'ReLU12'                ReLU                  ReLU
 37   'Conv13'                Convolution           64 3×3×64 convolutions with stride [1  1] and padd
 38   'BNorm13'               Batch Normalization   Batch normalization with 64 channels
 39   'ReLU13'                ReLU                  ReLU
 40   'Conv14'                Convolution           64 3×3×64 convolutions with stride [1  1] and padd
 41   'BNorm14'               Batch Normalization   Batch normalization with 64 channels
 42   'ReLU14'                ReLU                  ReLU
 43   'Conv15'                Convolution           64 3×3×64 convolutions with stride [1  1] and padd
 44   'BNorm15'               Batch Normalization   Batch normalization with 64 channels
 45   'ReLU15'                ReLU                  ReLU
 46   'Conv16'                Convolution           64 3×3×64 convolutions with stride [1  1] and padd
 47   'BNorm16'               Batch Normalization   Batch normalization with 64 channels
 48   'ReLU16'                ReLU                  ReLU
 49   'Conv17'                Convolution           64 3×3×64 convolutions with stride [1  1] and padd
 50   'BNorm17'               Batch Normalization   Batch normalization with 64 channels
 51   'ReLU17'                ReLU                  ReLU
 52   'Conv18'                Convolution           64 3×3×64 convolutions with stride [1  1] and padd
 53   'BNorm18'               Batch Normalization   Batch normalization with 64 channels
 54   'ReLU18'                ReLU                  ReLU
 55   'Conv19'                Convolution           64 3×3×64 convolutions with stride [1  1] and padd
 56   'BNorm19'               Batch Normalization   Batch normalization with 64 channels
 57   'ReLU19'                ReLU                  ReLU
 58   'Conv20'                Convolution           1 3×3×64 convolutions with stride [1  1] and paddi
 59   'FinalRegressionLayer'  Regression Output     mean-squared-error
```

```
lgraph = layerGraph(layers);
figure
set(gcf,'position',[0, 0, 200, 800])
plot(lgraph)
```

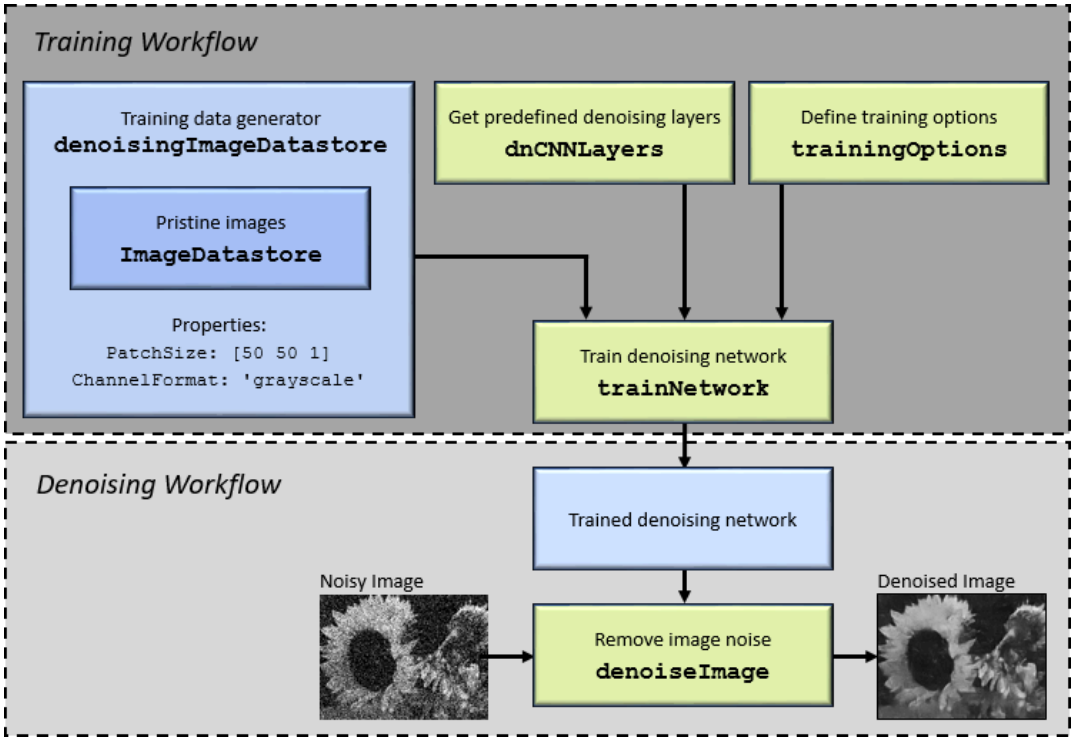


Go back to top

Part 3 (OPTIONAL): Training your own denoising network

In this part, we will move to a more comprehensive workflow, where you will train your denoising network from scratch before using it, as illustrated in the figure below.

Please refer to MATLAB documentation for more details.



Note: this is a very time-consuming operation that can only be successfully completed if you have powerful GPUs.

You can skip it entirely and it will not affect your grade.

If you decide to give it a try, you'll have to uncomment all code blocks within this part.

Suggested steps:

1. Create an `imageDatastore` object that stores pristine images.
2. Create a `denoisingImageDatastore` object.
3. Get the predefined denoising layers using the `dnCNNLayers` function.
4. Define training options.
5. Train the network.
6. Test and compare with the network in Part 2.

Step 1: Create an `imageDatastore` object that stores pristine images. We will use 984 color images of flowers (tulips), of size `224x224` pixels.

```
% imds2 = imageDatastore(fullfile('tulip'),...
```

```
% 'IncludeSubfolders',true,'FileExtensions','.jpg','LabelSource','foldernames');
```

Step 2: Create a `denoisingImageDatastore` object

`denoisingImageDatastore(imds)` creates a denoising image datastore, `dnimds` using images from image datastore `imds`. To generate noisy image patches, the denoising image datastore randomly crops pristine images from `imds` then adds zero-mean Gaussian white noise with a standard deviation of `0.1` to the image patches.

```
% dnimds2 = denoisingImageDatastore(imds2,'PatchSize', 50, 'ChannelFormat', 'grayscale'
```

Step 3: Get the predefined denoising layers using the `dnCNNLayers` function

```
% layers2 = dnCNNLayers;
```

Step 4: Define training options

```
% options = trainingOptions('adam', ...
%     'InitialLearnRate',0.1, ...
%     'LearnRateSchedule',"none", ...
%     'MiniBatchSize',256, ...
%     'MaxEpochs',2, ...
%     'Plots','training-progress', ...
%     'Verbose',false);
```

Step 5: Train the network

```
% dnCNN2 = trainNetwork(dnimds2,layers2,options);
```

Step 6: Test and compare with the network in Part 2.

```
% % Load RGB image
% pristineRGB = imread('lighthouse.png');
% % Convert to double
% pristineRGB = im2double(pristineRGB);
% % Add Gaussian noise
% noisyRGB = imnoise(pristineRGB,'gaussian',0,0.01);
%
% % Split the noisy RGB image into its individual color channels.
% [noisyR,noisyG,noisyB] = imsplit(noisyRGB);
%
% % Use the DnCNN2 network to remove noise from each color channel
% denoisedR2 = denoiseImage(noisyR,dnCNN2);
% denoisedG2 = denoiseImage(noisyG,dnCNN2);
% denoisedB2 = denoiseImage(noisyB,dnCNN2);
%
% % Recombine the denoised color channels to form the denoised RGB image
% denoisedRGB2 = cat(3,denoisedR2,denoisedG2,denoisedB2);
%
```

```
% % Display the original, noisy, and denoised color image
% figure('Name','Denoising with DnCNN trained from scratch');
% subplot(1,3,1), imshow(pristineRGB), title('Pristine Image')
% subplot(1,3,2), imshow(noisyRGB), title('Noisy Image')
% subplot(1,3,3), imshow(denoisedRGB2), title('Denoised Image')
```

Your turn (step 10 of the guidelines):

Write code to apply the DnCNN-based denoising technique of "Part 3" to at least 2 (two) different test images of your choice, using both Gaussian and salt-and-pepper noise, and compare their results using at least 2 (two) image quality metrics.

Summarize your results in a table (see *Guidelines* for additional information).

```
% ENTER YOUR CODE HERE
% ...
% ...
% ...
```

Go back to top

Part 4: Denoising using autoencoders

In this part, we will use a convolutional autoencoder architecture for denoising and apply it to images from a built-in dataset containing handwritten digits (similar to MNIST).

Please refer to MATLAB documentation for more details.

Suggested steps:

1. Load test images and add noise.
2. Prepare the data for network training, validation, and testing.
3. Visualize the images before and after adding noise.
4. Define a convolutional autoencoder network.
5. Train the network.
6. (Visually) evaluate the results.
7. Measure the quality of the resulting (denoised) image (and compare with the original).

Example code:

Based on (and adapted from): https://www.mathworks.com/help/deeplearning/ug/image-to-image-regression-using-deep-learning.html

Notes:

- The datastore contains 10,000 synthetic images of digits from 0 to 9.
- The images are generated by applying random transformations to digit images created with different fonts.
- Each digit image is 28-by-28 pixels.

- The datastore contains an equal number of images per category.

Build `imds`

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet', ...
    'nndemos','nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');

imds.ReadSize = 500;
rng(0)
imds = shuffle(imds);
```

Split `imds` into three image datastores containing pristine images for training, validation, and testing

```
[imdsTrain,imdsVal,imdsTest] = splitEachLabel(imds,0.95,0.025);
```

Create noisy versions of each input image using the helper function `addNoise` (see https://www.mathworks.com/help/deeplearning/ug/image-to-image-regression-using-deep-learning.html)

```
dsTrainNoisy = transform(imdsTrain,@addNoise);
dsValNoisy = transform(imdsVal,@addNoise);
dsTestNoisy = transform(imdsTest,@addNoise);
```

Combine the noisy images and pristine images into a single datastore that feeds data to `trainNetwork`

This combined datastore reads batches of data into a two-column cell array as expected by `trainNetwork`.

```
dsTrain = combine(dsTrainNoisy,imdsTrain);
dsVal = combine(dsValNoisy,imdsVal);
dsTest = combine(dsTestNoisy,imdsTest);
```

Perform additional preprocessing operations that are common to both the input and response datastores

The `commonPreprocessing` helper function (see https://www.mathworks.com/help/deeplearning/ug/image-to-image-regression-using-deep-learning.html) resizes input and response images to 32-by-32 pixels to match the input size of the network, and normalizes the data in each image to the range [0, 1].

```
dsTrain = transform(dsTrain,@commonPreprocessing);
dsVal = transform(dsVal,@commonPreprocessing);
dsTest = transform(dsTest,@commonPreprocessing);
```

Perform image data augmentation

The `augmentImages` helper function (see https://www.mathworks.com/help/deeplearning/ug/image-to-image-regression-using-deep-learning.html) applies randomized 90 degree rotations to the data. Identical rotations are applied to the network input and corresponding expected responses.

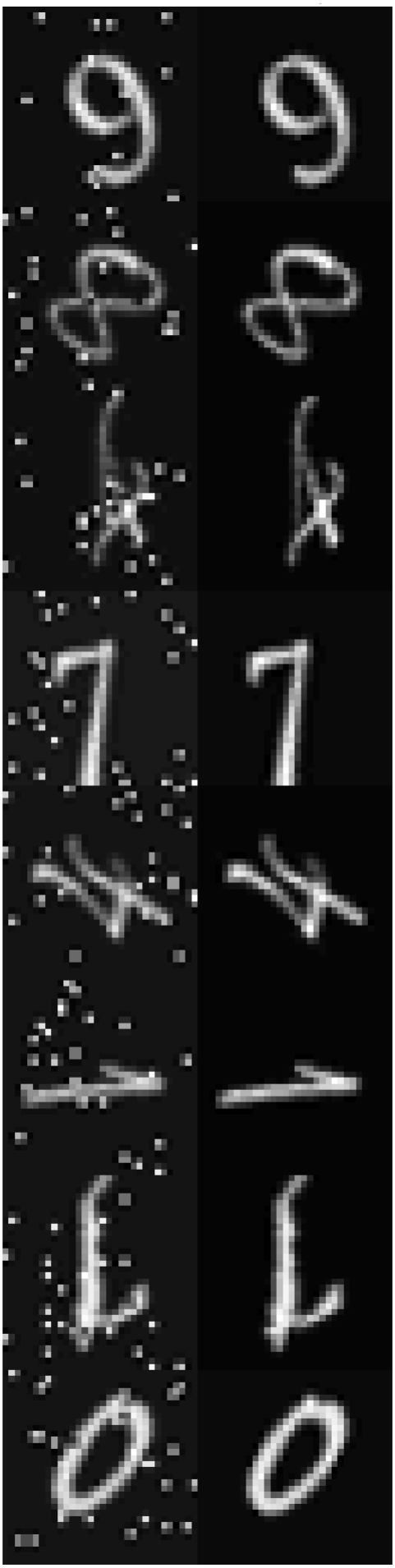
```
dsTrain = transform(dsTrain,@augmentImages);
```

Preview preprocessed data

The code below shows examples of paired noisy and pristine images using the `montage` function.

The training data looks correct. Salt and pepper noise appears in the input images in the left column. Randomized 90 degree rotation is applied to both input and response images in the same way.

```
exampleData = preview(dsTrain);
inputs = exampleData(:,1);
responses = exampleData(:,2);
minibatch = cat(2,inputs,responses);
figure
montage(minibatch','Size',[8 2])
title('Inputs (Left) and Responses (Right)')
```



Define convolutional autoencoder network

See https://www.mathworks.com/help/deeplearning/ug/image-to-image-regression-using-deep-learning.html for details.

```
imageLayer = imageInputLayer([32,32,1]);

encodingLayers = [ ...
    convolution2dLayer(3,16,'Padding','same'), ...
    reluLayer, ...
    maxPooling2dLayer(2,'Padding','same','Stride',2), ...
    convolution2dLayer(3,8,'Padding','same'), ...
    reluLayer, ...
    maxPooling2dLayer(2,'Padding','same','Stride',2), ...
    convolution2dLayer(3,8,'Padding','same'), ...
    reluLayer, ...
    maxPooling2dLayer(2,'Padding','same','Stride',2)];

decodingLayers = [ ...
    createUpsampleTransponseConvLayer(2,8), ...
    reluLayer, ...
    createUpsampleTransponseConvLayer(2,8), ...
    reluLayer, ...
    createUpsampleTransponseConvLayer(2,16), ...
    reluLayer, ...
    convolution2dLayer(3,1,'Padding','same'), ...
    clippedReluLayer(1.0), ...
    regressionLayer];

layers = [imageLayer,encodingLayers,decodingLayers];
```

Define training options

```
% options = trainingOptions('adam', ...
%     'MaxEpochs',30, ...
%     'MiniBatchSize',imds.ReadSize, ...
%     'ValidationData',dsVal, ...
%     'Shuffle','never', ...
%     'Plots','training-progress', ...
%     'Verbose',false);

% Add parallel execution and remove shuffling (or can't use parallel execution)
% options = trainingOptions('adam', ...
%     'MaxEpochs',30, ...
%     'MiniBatchSize',imds.ReadSize, ...
%     'ValidationData',dsVal, ...
%     'Plots','training-progress', ...
%     'ExecutionEnvironment', 'parallel', ...
%     'Verbose',false);

% Changes to the hyperparameters (with parallel processing):
%  - 'LearnRateSchedule','piecewise'
```

```
%
 options = trainingOptions('adam', ...
     'MaxEpochs',50, ...
     'MiniBatchSize',imds.ReadSize, ...
     'ValidationData',dsVal, ...
     'Plots','training-progress', ...
     'ExecutionEnvironment', 'parallel', ...
     'LearnRateSchedule','piecewise', ...
     'InitialLearnRate',0.008, ...
     'LearnRateDropFactor',0.5, ...
     'LearnRateDropPeriod',10, ...
     'Verbose',false);
```

Train the network

```
autoEncoder = trainNetwork(dsTrain,layers,options);
```

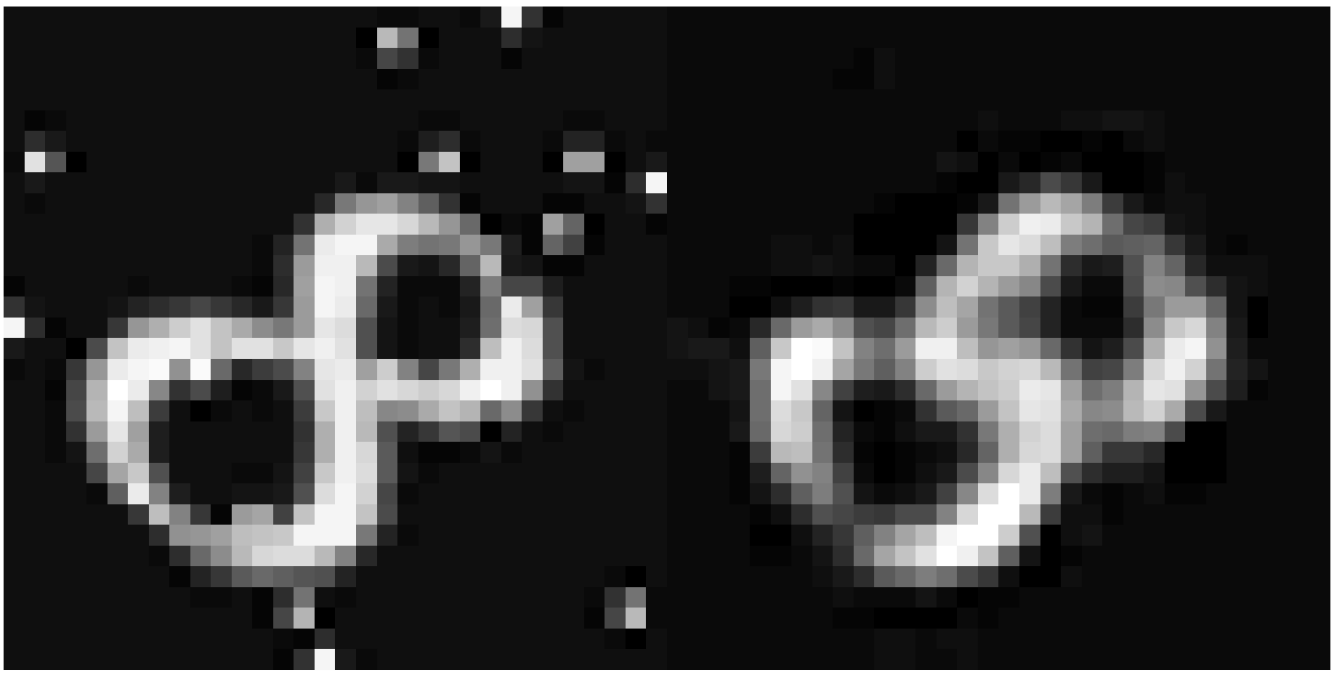
```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 4).
```

Evaluate the performance of the network on the test set

```
ypred = predict(autoEncoder,dsTest);
```

Perform visual inspection of selected results

```
inputImageExamples = preview(dsTest);
figure
montage({inputImageExamples{1},ypred(:,:,:,1)});
```



Calculate PSNR

```
ref = inputImageExamples{1,2};
originalNoisyImage = inputImageExamples{1,1};
psnrNoisy = psnr(originalNoisyImage,ref);
psnrDenoised = psnr(ypred(:,:,:,1),ref);

fprintf('\nThe PSNR of the noisy image is %.4f', psnrNoisy);
```

```
The PSNR of the noisy image is 20.0818
```

```
fprintf('\nThe PSNR of the denoised image is %.4f\n', psnrDenoised);
```

```
The PSNR of the denoised image is 19.2377
```

Calculate PIQE

```
piqeNoisy = piqe(originalNoisyImage);
piqeDenoised = piqe(ypred(:,:,:,1));

fprintf('\nThe PIQE of the noisy image is %.4f', piqeNoisy);
```

```
The PIQE of the noisy image is 83.7633
```

```
fprintf('\nThe PIQE of the denoised image is %.4f\n', piqeDenoised);
```

```
The PIQE of the denoised image is 88.4083
```

Your turn (step 16 of the guidelines):

(OPTIONAL) Write code to compute other figures of merit for selected images.

Part 1 - Select image

Select a reference (original) image and its noisy version, then denoise it.

```
imageNumber = 7;
referenceImage = inputImageExamples{imageNumber, 2};
noisyImage = inputImageExamples{imageNumber, 1};
denoisedImage = ypred(:,:,:, imageNumber);
```

Part 2 - Show metrics

Show quality metrics.

```
fprintf('\nPSNR of the noisy image=%.4f, denoised image=%.4f',...
   psnr(noisyImage, referenceImage), psnr(denoisedImage, referenceImage));
```

```
PSNR of the noisy image=17.8861, denoised image=20.6402
```

```
fprintf('\nPIQE of the noisy image=%.4f, denoised image=%.4f',...
   piqe(noisyImage), piqe(denoisedImage));
```

```
PIQE of the noisy image=84.6347, denoised image=90.1717
```

```
fprintf('\nMSE of noisy image=%.4f, denoised image=%.4f',...
   immse(noisyImage, referenceImage), immse(denoisedImage, referenceImage));
```

```
MSE of noisy image=0.0163, denoised image=0.0086
```

```
fprintf('\nSSIM of noisy image=%.4f, denoised image=%.4f',...
   ssim(noisyImage, referenceImage), ssim(denoisedImage, referenceImage));
```

```
SSIM of noisy image=0.5247, denoised image=0.8142
```

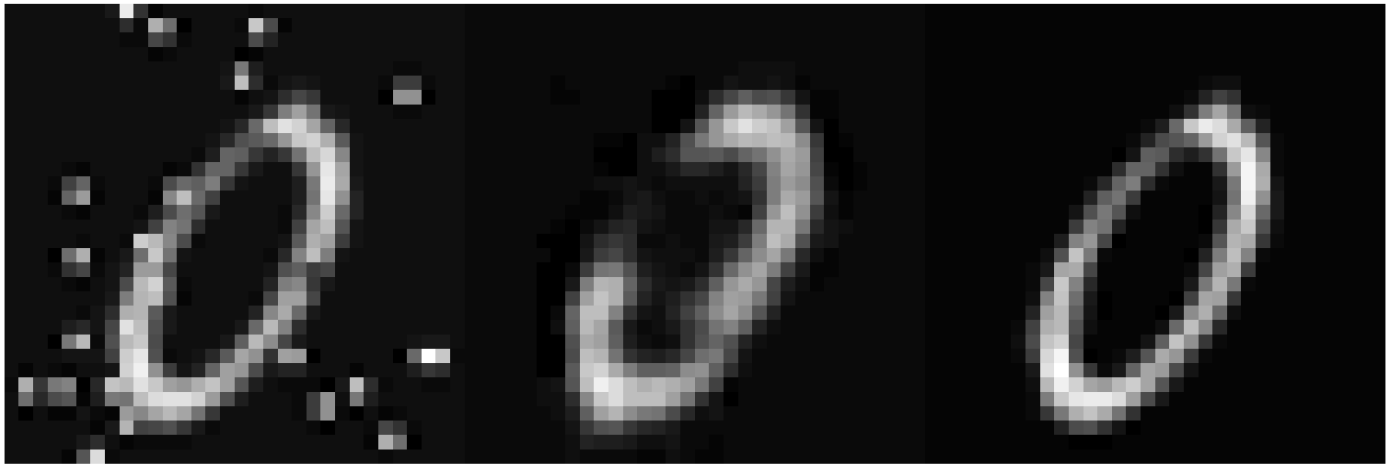
```
fprintf('\nBRISQUE of noisy image=%.4f, denoised image=%.4f',...
   brisque(noisyImage), brisque(denoisedImage));
```

```
BRISQUE of noisy image=43.4582, denoised image=45.1617
```

Part 3 - Show images

Show the images (noisy, denoised, and original image).

```
montage({noisyImage, denoisedImage, referenceImage}, 'Size', [1 3])
```



Go back to top

```
% Helper functions
```

```
function dataOut = addNoise(data)
    dataOut = data;
    for idx = 1:size(data,1)
       dataOut{idx} = imnoise(data{idx},'salt & pepper');
    end
end

function dataOut = commonPreprocessing(data)
    dataOut = cell(size(data));
    for col = 1:size(data,2)
```

```
        for idx = 1:size(data,1)
            temp = single(data{idx,col});
            temp = imresize(temp,[32,32]);
            temp = rescale(temp);
            dataOut{idx,col} = temp;
        end
    end
end

function dataOut = augmentImages(data)
    dataOut = cell(size(data));
    for idx = 1:size(data,1)
        rot90Val = randi(4,1,1)-1;
        dataOut(idx,:) = {rot90(data{idx,1},rot90Val),rot90(data{idx,2},rot90Val)};
    end
end

function out = createUpsampleTransponseConvLayer(factor,numFilters)
    filterSize = 2*factor - mod(factor,2);
    cropping = (factor-mod(factor,2))/2;
    numChannels = 1;

    out = transposedConv2dLayer(filterSize,numFilters, ...
        'NumChannels',numChannels,'Stride',factor,'Cropping',cropping);
end
```