Getting Started

In [2]:
```
#%matplotlib notebook
import os
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as td
import torchvision as tv
from PIL import Image
import matplotlib.pyplot as plt
import nntools as nt


device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
```

```
cuda
```

2 Creating noisy images of BSDS dataset with DataSet

In [3]:
```
#Part 1
dataset_root_dir = '/datasets/ee285f-public/bsds/'
```

In [4]:
```
# Part 2

class NoisyBSDSDataset(td.Dataset):
    def __init__(self, root_dir, mode='train', image_size=(180, 180), sigma=30
):
        super(NoisyBSDSDataset, self).__init__()
        self.mode = mode
        self.image_size = image_size
        self.sigma = sigma
        self.images_dir = os.path.join(root_dir, mode)
        self.files = os.listdir(self.images_dir)
    def __len__(self):
        return len(self.files)
    def __repr__(self):
        return "NoisyBSDSDataset(mode={}, image_size={}, sigma={})". \
            format(self.mode, self.image_size, self.sigma)
    def __getitem__(self, idx):
        img_path = os.path.join(self.images_dir, self.files[idx])
        clean= Image.open(img_path).convert('RGB')
        i = np.random.randint(clean.size[0] - self.image_size[0])
        j = np.random.randint(clean.size[1] - self.image_size[1])

        # COMPLETE
        clean = clean.crop((i,j,self.image_size[0]+i,self.image_size[1]+j))

        transform = tv.transforms.Compose([
            # COMPLETE
            tv.transforms.ToTensor(), #Normalizes the image
            tv.transforms.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5]),
            ])

        clean = transform(clean)
        noisy = clean + 2 / 255 * self.sigma * torch.randn(clean.shape)
        return noisy, clean
```

In [5]:
```
# Part 3
def myimshow(image, ax=plt):
    image = image.to('cpu').numpy()
    image = np.moveaxis(image, [0, 1, 2], [2, 0, 1])
    image = (image + 1) / 2
    image[image < 0] = 0
    image[image > 1] = 1
    h = ax.imshow(image)
    ax.axis('off')
    return h


train_set = NoisyBSDSDataset(root_dir = dataset_root_dir, mode = 'train', imag
e_size=(180, 180), sigma=30)
test_set = NoisyBSDSDataset(root_dir = dataset_root_dir, mode = 'test', image_
size=(320, 320), sigma=30)
x, y = train_set[12]

plt.figure()
myimshow(x)
plt.figure()
myimshow(y)
```

Out[5]: <matplotlib.image.AxesImage at 0x7fe113a9d9b0>

3 DnCNN

In [6]:
```
# Part 4
class NNRegressor(nt.NeuralNetwork):
    def __init__(self):
        super(NNRegressor, self).__init__()
        self.MSELoss = nn.MSELoss()
    def criterion(self, y, d):
        return self.MSELoss(y, d)
```

In [10]:
```
#Part 5 + 7

class DnCNN(NNRegressor):
    def __init__(self, D, C=64):
        super(DnCNN, self).__init__()
        self.D = D
        self.conv = nn.ModuleList()
        self.conv.append(nn.Conv2d(3, C, 3, padding=1))
        # COMPLETE
        nn.init.kaiming_normal_(self.conv[0].weight.data)#Part 7


        self.bn = nn.ModuleList()
        for k in range(D):
            self.bn.append(nn.BatchNorm2d(C, C))
            self.conv.append(nn.Conv2d(64, C, 3, padding=1))

            nn.init.constant_(self.bn[k].weight.data, 1.25 * np.sqrt(C))#Part
 7
            nn.init.kaiming_normal_(self.conv[k+1].weight.data)#Part 7


        self.conv.append(nn.Conv2d(C, 3, 3, padding=1))

    def forward(self, x):
        D = self.D
        h = F.relu(self.conv[0](x))
        for k in range(D):
            t = self.bn[k](self.conv[k+1](h))
            h = F.relu(t)
        # COMPLETE
        y = self.conv[D+1](h) + x
        return y
```

In [8]:
```
# Part 6

x, y = train_set[-1]
print(x.shape)
x = x.unsqueeze(0)
print(x.shape)

D_list = [0,1,2,4,8]
for d in D_list:
    net = DnCNN(d)
    y = net.forward(x)

    print('D= ',d)
    plt.figure()
    plt.subplot(1,3,1)
    myimshow(x[0])
    plt.subplot(1,3,2)
    myimshow(y[0].detach())
    plt.subplot(1,3,3)
    myimshow(x[0] - y[0].detach())
```

```
torch.Size([3, 180, 180])
torch.Size([1, 3, 180, 180])
D=  0
D=  1
D=  2
D=  4
D=  8
```

Answer part 6 theorotical questions

We observe that the first image x[0] is the noisy image
The second image is the model's output after forward propogation and adding the original image to it : y[0]
The third layer is the output of the last layer in the model x[0] - y[0]
The third image represents the output of the model or the tranformation done to the image after passing it through the model.
It is obvious that, when not initialized, the output is very grayed and the more added layers the more gray and corrupted the output image is
Backpropogation is to be used for learning the noise in the image.

In [11]:
```
#Part 7 + Part 8
D_list = [0,1,2,4,8]
for d in D_list:
    net = DnCNN(d)
    y = net.forward(x)

    plt.figure()
    plt.subplot(1,3,1)
    myimshow(x[0])
    plt.subplot(1,3,2)
    myimshow(y[0].detach())
    plt.subplot(1,3,3)
    myimshow(x[0] - y[0].detach())
```

Part 8

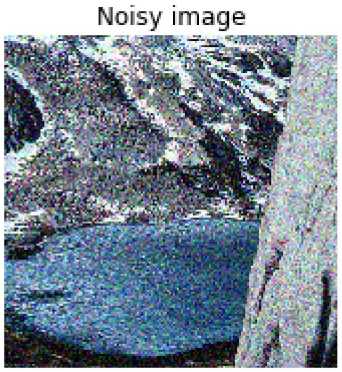
We observe that the first image x[0] is the noisy image
The second image is the model's output after forward propogation and adding the original image to it : y[0]
The third layer is the output of the last layer in the model x[0] - y[0]
The third image represents the output of the model or the tranformation done to the image after passing it through the model.
It is obvious that due to forward pass, when initialized, we keep more image features. The more added layers, the more corrupted (tranformed) the output image gets.
Backpropogation is to be used for learning the noise in the image. When convolution and batch normalization layers are initialized, we will be able to learn more noise since the image wouldn't get as grayed when passing the layers.

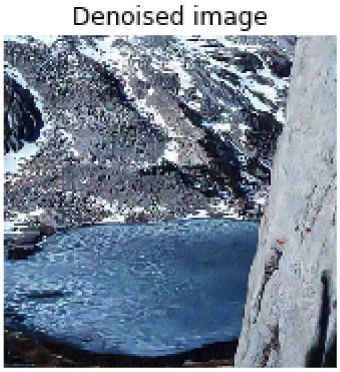
In [12]:
```
#Part 9

class DenoisingStatsManager(nt.StatsManager):
    def __init__(self):
        super(DenoisingStatsManager, self).__init__()
    def init(self):
        super(DenoisingStatsManager, self).init()
        self.running_psnr = 0

    def accumulate(self, loss, x, y, d):
        super(DenoisingStatsManager, self).accumulate(loss, x, y, d)
        n = y.numel()
        norm2 = torch.pow(torch.norm(y-d),2) # not sure if x or d TODO
        self.running_psnr += 10*torch.log10(4*n/norm2)

    def summarize(self):
        loss = super(DenoisingStatsManager, self).summarize()
        avg_psnr = self.running_psnr/self.number_update
        return {'loss': loss, 'avg_psnr': avg_psnr}
```

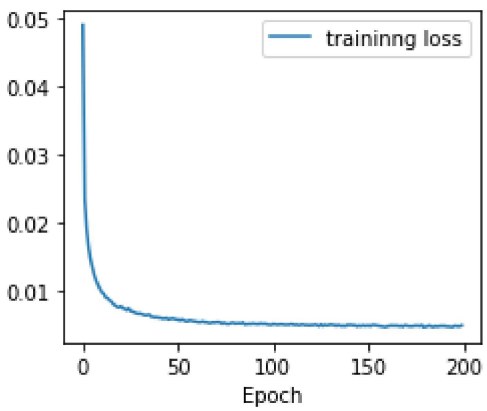
In [13]:
```
#Part 10
D = 6
lr = 1e-3
net = DnCNN(D)
net = net.to(device)
adam = torch.optim.Adam(net.parameters(), lr=lr)
stats_manager = DenoisingStatsManager()
val_set = test_set
exp1 = nt.Experiment(net, train_set, val_set, adam, stats_manager,
                     output_dir="denoising1", batch_size=4, perform_validation_
during_training=False)
```

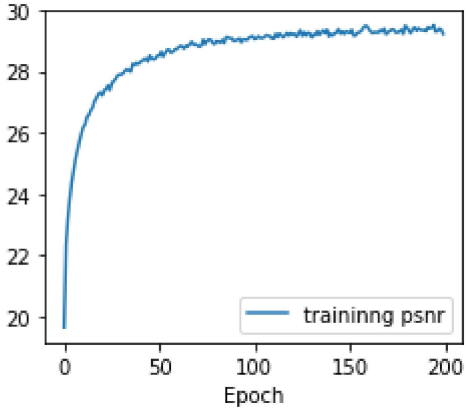
In [14]:
```
#Part 11
def plot(exp, fig, axes, noisy, visu_rate=2):
    if exp.epoch % visu_rate != 0:
        return
    with torch.no_grad():
        denoised = exp.net(noisy[np.newaxis].to(exp.net.device))[0]
    axes[0][0].clear()
    axes[0][1].clear()
    axes[1][0].clear()
    axes[1][1].clear()
    myimshow(noisy, ax=axes[0][0])
    axes[0][0].set_title('Noisy image')
    myimshow(denoised, ax=axes[0][1])
    axes[0][1].set_title('Denoised image')
    axes[1][0].plot([exp.history[k]['loss'] for k in range(exp.epoch)],
    label="traininng loss")
    axes[1][1].plot([exp.history[k]['avg_psnr'] for k in range(exp.epoch)],
    label="traininng psnr")
    axes[1][0].legend()
    axes[1][1].legend()
    axes[1][0].set_xlabel("Epoch")
    axes[1][1].set_xlabel("Epoch")

    plt.tight_layout()
    fig.canvas.draw()

fig, axes = plt.subplots(ncols=2, nrows=2, figsize=(7,6))
exp1.run(num_epochs=200, plot=lambda exp: plot(exp, fig=fig, axes=axes,noisy=t
est_set[73][0]))

plot(exp1, fig=fig, axes=axes,noisy=test_set[73][0]) # Added
```

```
Start/Continue training from epoch 200
Finish training for 200 epochs
```

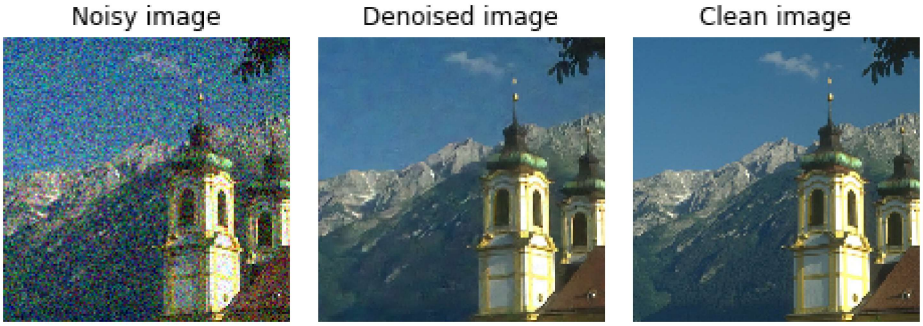








In [15]:
```
#Part 12
#%matplotlib notebook

def plot_comparision(exp, fig, axes, test_image, visu_rate=2):
    noisy = test_image[0]
    clean = test_image[1]
    if exp.epoch % visu_rate != 0:
        return
    with torch.no_grad():
        denoised = exp.net(noisy[np.newaxis].to(exp.net.device))[0]
    axes[0].clear()
    axes[1].clear()
    axes[2].clear()

    myimshow(noisy, ax=axes[0])
    axes[0].set_title('Noisy image')
    myimshow(denoised, ax=axes[1])
    axes[1].set_title('Denoised image')
    myimshow(clean, ax=axes[2])
    axes[2].set_title('Clean image')

    plt.tight_layout()
    fig.canvas.draw()

fig, axes = plt.subplots(ncols=3, nrows=1, sharex='all', sharey='all',figsize=
(7,6))

plot_comparision(exp1, fig=fig, axes=axes,test_image=test_set[34])
plot_comparision(exp1, fig=fig, axes=axes,test_image=test_set[42])
plot_comparision(exp1, fig=fig, axes=axes,test_image=test_set[62])
```



It is obvious from the images and their zoom-ins, that the denoised image looks similar to the clean original image.
The denoised image looks as if the clean image passed through a low pass filter.
The clean image has more high frequency details which are missing in the denoised one
We have lost some of the high frequency details while "cleaning" the image from noise which was high frequency.

Part 13
Number of parameter at each convulational layer = $C_{in}C_{out}n^2 + bias = C_{in}C_{out}n^2 + C_{out}$
Number of parameter at each batch normalization layer = $weights + bias = C_{out} + C_{out}$
Number of parameters in DnCNN(D) = 3x64x9 + 64 + 64x3x9+3 + Dx(64x64x9 + 64) +Dx(64+64)
D=6 =>
3x64x9 + 64 + 64x3x9+3 + 6x(64x64x9 + 64) +6x(64+64)= 225859 Parameters in DnCNN(D=6)
Receptive Field
Due to convulotion process, when we have 1 layer, the reciptive field is equal to the filter's size (3x3)
The number of input of pixels that influence an output pixels is ((D+2)*2 +1)x((D+2)*2 +1)
Since D= 6, the receptive field is 17x17

Part 14
In order to have a receptive field of 33x33, we should have D= 14.
In this case we will have 522307 learnable parameters. It is obvious the number of parameters is a little bit more than double the previous when D=6
Since the number of learnable parameters has doubled, the computation time will also increase. For example, in case time complexity is O(n^k), it will increase by 2^k

4 U-net like CNNs

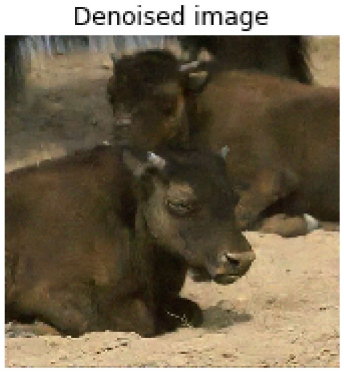
In [16]:
```
#Part 15
import math
class UDnCNN(NNRegressor):
    def __init__(self, D, C=64):
        super(UDnCNN, self).__init__()
        self.D = D
        self.conv = nn.ModuleList()
        self.conv.append(nn.Conv2d(3, C, 3, padding=1))
        nn.init.kaiming_normal_(self.conv[0].weight.data)#Part 7
        self.bn = nn.ModuleList()

        for k in range(D):
            self.bn.append(nn.BatchNorm2d(C, C))
            self.conv.append(nn.Conv2d(64, C, 3, padding=1))

            nn.init.constant_(self.bn[k].weight.data, 1.25 * np.sqrt(C))#Part
 7
            nn.init.kaiming_normal_(self.conv[k+1].weight.data)#Part 7


        self.conv.append(nn.Conv2d(C, 3, 3, padding=1))

    def forward(self, x):
        D = self.D

        h = F.relu(self.conv[0](x))
        saved_h = []
        saved_h.append(h)
        for k in range(D):
            if k >= (D/2)+1:
                h_temp, ind_temp = saved_h.pop()
                h = h + h_temp/1.4142135624

                h = F.max_unpool2d(h, indices=ind_temp, kernel_size=2)


            t = self.bn[k](self.conv[k+1](h))
            h = F.relu(t)

            if k < D/2 -1:
                h, ind = F.max_pool2d(h, kernel_size=2, return_indices=True)
                saved_h.append((h,ind))

        y = self.conv[D+1](h) + x

        return y
```

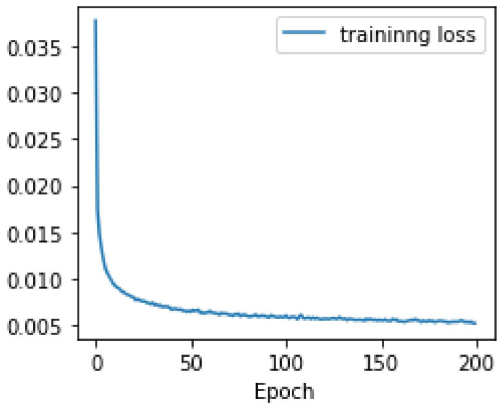
In [ ]:
```

```

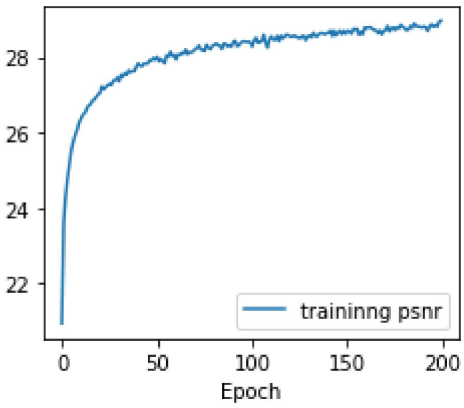
In [17]:
```
#Part 16

D = 6
lr = 1e-3
net = UDnCNN(D)
net = net.to(device)
adam = torch.optim.Adam(net.parameters(), lr=lr)
stats_manager = DenoisingStatsManager()
val_set = test_set
exp2 = nt.Experiment(net, train_set, val_set, adam, stats_manager,
                     output_dir="denoising2", batch_size=4, perform_validation_
during_training=False)
```

In [18]:
```
fig, axes = plt.subplots(ncols=2, nrows=2, figsize=(7,6))
exp2.run(num_epochs=200, plot=lambda exp: plot(exp, fig=fig, axes=axes,noisy=t
est_set[39][0]))
```

```
Start/Continue training from epoch 200
Finish training for 200 epochs
```







Part 17

Number of parameter at each convulational layer = $C_{in}C_{out}n^2 + bias = C_{in}C_{out}n^2 + C_{out}$

Number of parameter at each batch normalization layer = $weights + bias = C_{out} + C_{out}$

Number of parameters in UDnCNN(D) = 3x64x9 + 64 + 64x3x9+3 + Dx(64x64x9 + 64) +Dx(64+64)

D=6 =>

3x64x9 + 64 + 64x3x9+3 + 6x(64x64x9 + 64) +6x(64+64)= 225859 Parameters in UDnCNN(D=6)

Receptive Field

Due to convulotion process, when we have 1 layer, the reciptive field is equal to the filter's size (3x3)

When pooling is implemented, the receptive field is multiplied by 2 for this layer.

Therefore, we should find the total number of pooling layers and multiply each layer of these by 2.

The number of input of pixels that influence an output pixels is

=(D*2*+*22*+(D/2 - 1)*2 +1)x(D2+22+(D/2 - 1)*2 +1) =

=(3D+3)x(3D+3)

Since D= 6, the receptive field is 21x21 in our case

In [19]:
```
#Part 18
print(exp1.evaluate())
print(exp2.evaluate())
```

```
{'loss': 0.0050514051038771865, 'avg_psnr': tensor(29.0821, device='cuda:0')}
{'loss': 0.005568497329950333, 'avg_psnr': tensor(28.6460, device='cuda:0')}
```

Part 18

According to the results above, we see that the loss for DnCNN is less than the loss for the U-net like CNN method.

We can also see that the avg psnr is also higher for DnCNN

This implies that the DnCNN method gave better results which corresponds to the visualized images. We can see that the DnCNN images were denoised slightly better than U-net like CNN counterpart.

Even though the receptive field is larger, the pooling led to the loss of features' exact locations and may introduce artifacts in the results and lower the psnr.

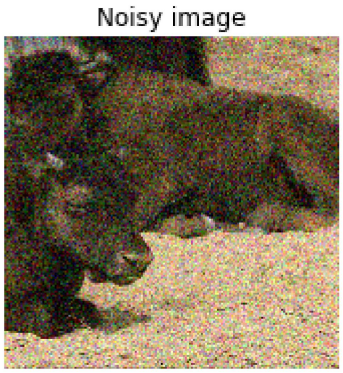
In [20]:
```
# Part 19 + 20

class DUDnCNN(NNRegressor):
    def __init__(self, D, C=64):
        super(DUDnCNN, self).__init__()
        self.D = D
        self.conv = nn.ModuleList()
        self.conv.append(nn.Conv2d(3, C, 3, dilation=1, padding=1))
        nn.init.kaiming_normal_(self.conv[0].weight.data)
        self.bn = nn.ModuleList()
        dilation_list = []
        for k in range(D):
            if k < D/2:
                dilation_list.append(math.pow(2,k))
            else:
                dilation_list.append(math.pow(2,D-k-1))

        dilation = 0
        for k in range(D):

            dilation = int(dilation_list[k])

            self.bn.append(nn.BatchNorm2d(C, C))
            pad = int(((dilation -1)*2 +3)/2)

            self.conv.append(nn.Conv2d(64, C, 3,dilation = dilation,  padding=
pad))


            nn.init.constant_(self.bn[k].weight.data, 1.25 * np.sqrt(C))
            nn.init.kaiming_normal_(self.conv[k+1].weight.data)


        self.conv.append(nn.Conv2d(C, 3, 3, padding=1))

    def forward(self, x):
        D = self.D

        torch.backends.cudnn.benchmark=True
        h = F.relu(self.conv[0](x))
        torch.backends.cudnn.benchmark=False
        saved_h = []
        saved_h.append(h)
        for k in range(D):
            if k >= (D/2)+1:
                h_temp = saved_h.pop()
                h = h + h_temp/1.4142135624

            torch.backends.cudnn.benchmark=True
            t = self.bn[k](self.conv[k+1](h))
            h = F.relu(t)
            torch.backends.cudnn.benchmark=False

            if k < D/2 -1:
                saved_h.append(h)

        torch.backends.cudnn.benchmark=True
```

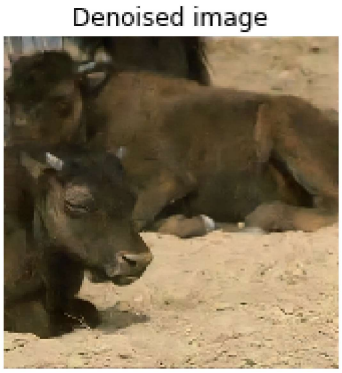
```
        y = self.conv[D+1](h) + x
        torch.backends.cudnn.benchmark=False

        return y
```

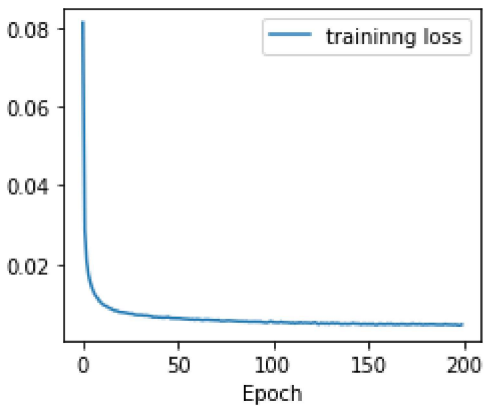
In [21]:
```
#Part 21
D = 6
lr = 1e-3
net = DUDnCNN(D)
net = net.to(device)
adam = torch.optim.Adam(net.parameters(), lr=lr)
stats_manager = DenoisingStatsManager()
val_set = test_set
exp3 = nt.Experiment(net, train_set, val_set, adam, stats_manager,
                     output_dir="denoising3", batch_size=4, perform_validation_
during_training=False)
```

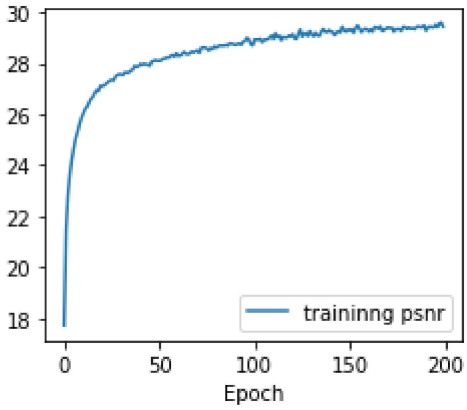
In [22]:
```
fig, axes = plt.subplots(ncols=2, nrows=2, figsize=(7,6))
exp3.run(num_epochs=200, plot=lambda exp: plot(exp, fig=fig, axes=axes,noisy=t
est_set[39][0]))
```

```
Start/Continue training from epoch 200
Finish training for 200 epochs
```



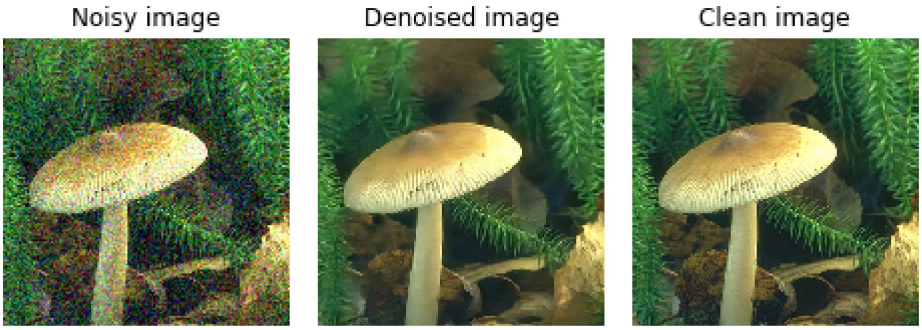






In [23]:
```
#Part 22
print(exp1.evaluate())
print(exp2.evaluate())
print(exp3.evaluate())

fig, axes = plt.subplots(ncols=3, nrows=1, sharex='all', sharey='all',figsize=
(7,6))
plot_comparision(exp3, fig=fig, axes=axes,test_image=test_set[17])
```

```
{'loss': 0.0050079727824777365, 'avg_psnr': tensor(29.1163, device='cuda:0')}
{'loss': 0.005578818619251251, 'avg_psnr': tensor(28.6314, device='cuda:0')}
{'loss': 0.004930539429187775, 'avg_psnr': tensor(29.1969, device='cuda:0')}
```



Part 22
As we obsereve above, The DUDnCNN has the best performance when it comes to loss and average psnr
It has the lowest loss and the best average psnr. It also generates the best denoised images.
This is due to the increased reciptive field while keeping the total number of parameters constant.

Part 23
Number of parameter at each convulational layer = $C_{in}C_{out}n^2 + bias = C_{in}C_{out}n^2 + C_{out}$
Number of parameter at each batch normalization layer = $weights + bias = C_{out} + C_{out}$
Number of parameters in DUDnCNN(D) = 3x64x9 + 64 + 64x3x9+3 + Dx(64x64x9 + 64) +Dx(64+64)
D=6 =>
3x64x9 + 64 + 64x3x9+3 + 6x(64x64x9 + 64) +6x(64+64)= 225859 Parameters in DUDnCNN(D=6)

We can conclude that the number of parameters doesn't change since it is dependent on $C_{in}C_{out}$ and n (filter) in each layer and the number of layers in the models is the same
Receptive Field

Receptive field:
dialation = d = 2^{i-1} for the inner D/2 layers
dialation = d = 2^{D-i} for the outer D/2 layers
d = 1 for the outer layers
filter size at each layer = k = (2*dilation + 1)
Receptive field = 1 + $\Sigma(k-1)$
Since D= 6, the receptive field is 33x33

In [ ]: