In [1]:

```
%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

import math
```

In [2]:

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

```
cuda
```

### Question 1

Create a variable dataset root dir and make it point to the BSDS dataset directory.

In [3]:

```
dataset_root_dir = "/datasets/ee285f-public/bsds/"
```

In [4]:

```
print(dataset_root_dir)
```

```
/datasets/ee285f-public/bsds/
```

### Question 2

In [5]:

```
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])

        clean = clean.crop([i , j , self.image_size[0]+i , self.image_size[1]+j])
#left,top,right,bottom

#         print(clean)

        transform = tv.transforms.Compose([
        tv.transforms.ToTensor(),                #convert numpy array to tensor
        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 [6]:

```
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
```

**Question 3**

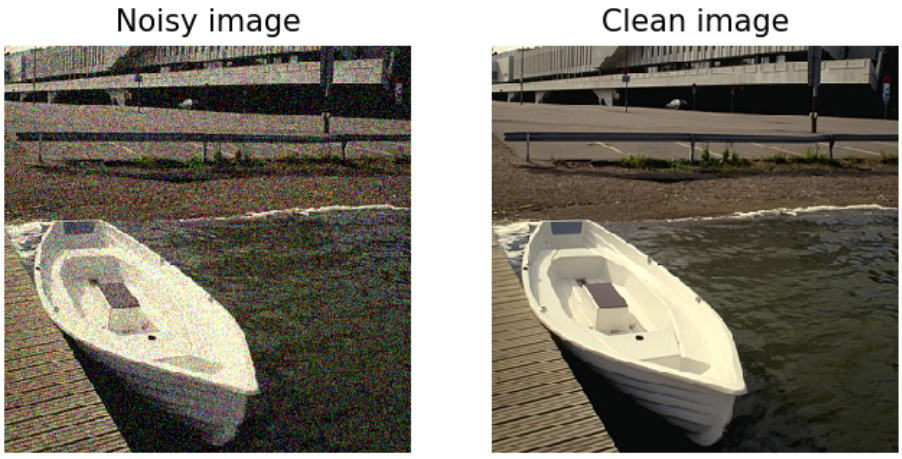
Creation of train_set and test_set

In [9]:

```
train_set = NoisyBSDSDataset(dataset_root_dir , mode = "train" , image_size =(180,180))
```

In [10]:

```
test_set = NoisyBSDSDataset(dataset_root_dir , mode = "test" , image_size =(320,320))

x = test_set.__getitem__(12)

figure = plt.figure()
ax1 = figure.add_subplot(121)
ax1.set_title('Noisy image')
myimshow(x[0])

ax2 = figure.add_subplot(122)
ax2.set_title('Clean image')
myimshow(x[1])

plt.show()
```




In [12]:

```
import nntools as nt
```

### Question 4

In [7]:

```
class NNRegressor(nt.NeuralNetwork):
    def __init__(self):
        super(NNRegressor, self).__init__()
        self.MSE = nn.MSELoss()
    def criterion(self, y, d):
        return self.MSE(y, d)
```

### Question 5

In [16]:

```
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))

        self.bn = nn.ModuleList()

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

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


        for k in range(D):

            self.bn.append(nn.BatchNorm2d(C))

    def forward(self, x):

        D = self.D
        h = F.relu(self.conv[0](x))

        for j in range(D):

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

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

        return y
```

### Question 6

In [17]:

```
class  DenoisingStatsManager(nt.StatsManager):

    def __init__(self):
        super(DenoisingStatsManager, self).__init__()

    def init(self):
        super(DenoisingStatsManager, self).init()
        self.PSNR = 0

    def accumulate(self, loss, x, y, d):
        super(DenoisingStatsManager, self).accumulate(loss, x, y, d)

        n = x.numel()

        self.PSNR += 10*torch.log10(4*n/(torch.norm(y - d)**2))


    def summarize(self):

        loss = super(DenoisingStatsManager, self).summarize()
        PSNR = (self.PSNR / self.number_update)

        return {'loss': loss, 'PSNR': PSNR}
```

### Question 7

In [18]:

```
lr = 1e-3

net = DnCNN(6)

net = net.to(device)

adam = torch.optim.Adam(net.parameters(), lr=lr)

stats_manager = DenoisingStatsManager()

exp1 = nt.Experiment(net, train_set, test_set, adam, stats_manager,
output_dir="denoising1", batch_size=4 , perform_validation_during_training=True)
```

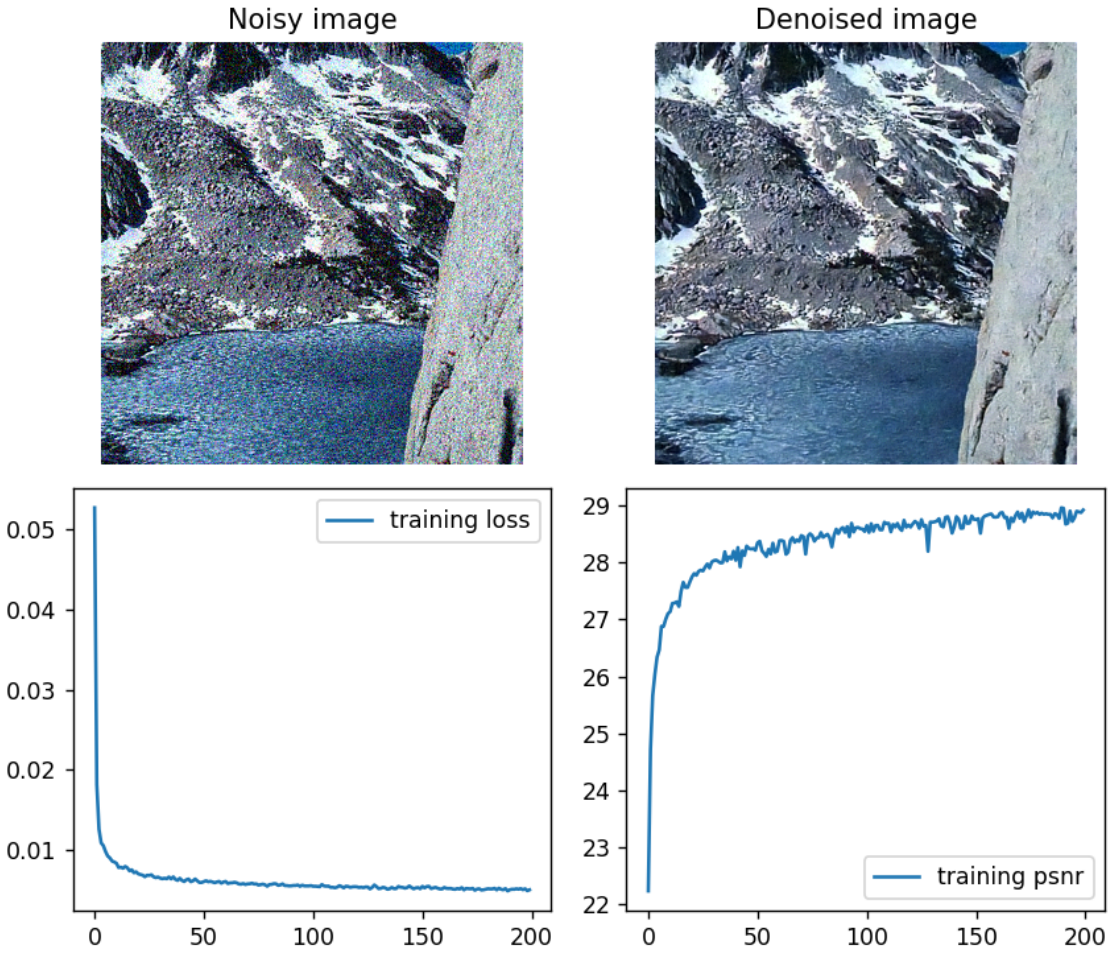
### Question 8

In [19]:

```
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][0]['loss'] for k in range(exp.epoch)],
    label="training loss")

    axes[1][1].plot([exp.history[k][1]['PSNR'] for k in range(exp.epoch)],
    label="training psnr")

    axes[1][0].legend()
    axes[1][1].legend()


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

In [22]:

```
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=test_set[
73][0]))
```

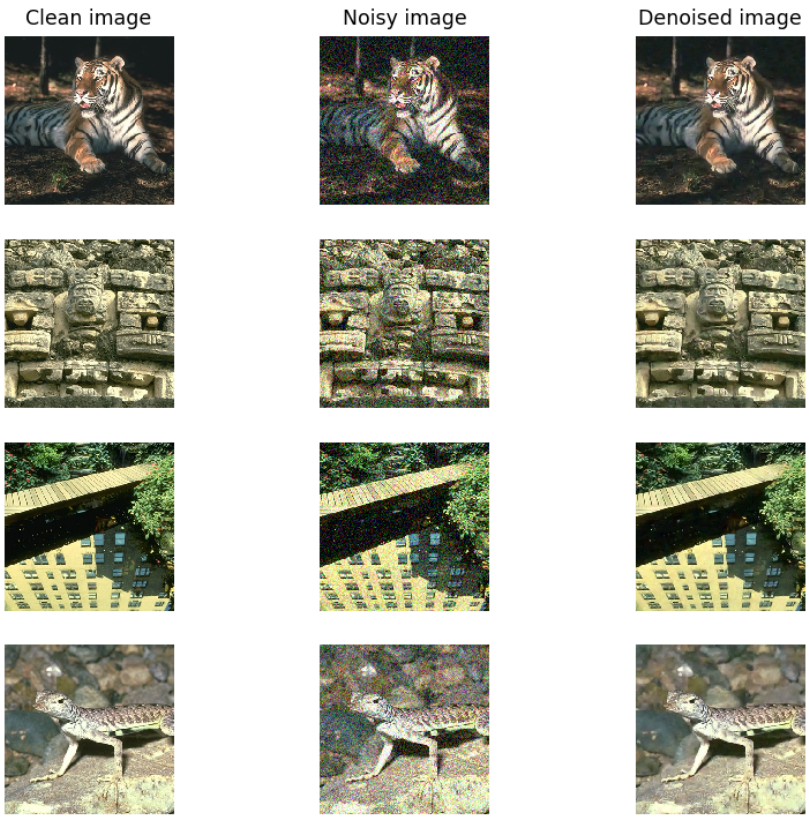



```
Start/Continue training from epoch 200
Finish training for 200 epochs
```

### Question 9

In [24]:

```
fig, axes = plt.subplots(nrows=4, ncols=3, figsize =(8,7) , sharex = 'all' , sharey =
'all')

i = 0

for i in range(len(test_set)):

    if i == 4:
        break

    noisy, clean = test_set[i]
    myimshow(clean, ax =axes[i][0])
    axes[0][0].set_title('Clean image', fontsize = 10)

    myimshow(noisy, ax = axes[i][1])
    axes[0][1].set_title('Noisy image', fontsize = 10)

    with torch.no_grad():
        denoised = exp1.net(noisy[np.newaxis].to(exp1.net.device))[0]
    myimshow(denoised, ax = axes[i][2])
    axes[0][2].set_title('Denoised image' , fontsize = 10)

    #check
```



Evaluate the visual quality of your result . Do you see any artifacts or loss of information ?

We observe that noise has been added to the original image and we obtained the noisy image.

Ultimately , we have used DnCNN to remove the noise from the noisy image.

We observe few artifacts in the denoised image and there is loss of information if we compare it with the original image.

This is because the psnr value is low and there is loss after denoising .

**DnCNN**

In [28]:

```
print(net)        #printing the DnCNN network
```

```
DnCNN(
  (MSE): MSELoss()
  (conv): ModuleList(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): Conv2d(64, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (bn): ModuleList(
    (0): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (5): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  )
)
```

In [29]:

```
for name, param in net.named_parameters():
    print(name, param.size(), param.dtype, param.requires_grad)
```

```
conv.0.weight torch.Size([64, 3, 3, 3]) torch.float32 True
conv.0.bias torch.Size([64]) torch.float32 True
conv.1.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.1.bias torch.Size([64]) torch.float32 True
conv.2.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.2.bias torch.Size([64]) torch.float32 True
conv.3.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.3.bias torch.Size([64]) torch.float32 True
conv.4.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.4.bias torch.Size([64]) torch.float32 True
conv.5.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.5.bias torch.Size([64]) torch.float32 True
conv.6.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.6.bias torch.Size([64]) torch.float32 True
conv.7.weight torch.Size([3, 64, 3, 3]) torch.float32 True
conv.7.bias torch.Size([3]) torch.float32 True
bn.0.weight torch.Size([64]) torch.float32 True
bn.0.bias torch.Size([64]) torch.float32 True
bn.1.weight torch.Size([64]) torch.float32 True
bn.1.bias torch.Size([64]) torch.float32 True
bn.2.weight torch.Size([64]) torch.float32 True
bn.2.bias torch.Size([64]) torch.float32 True
bn.3.weight torch.Size([64]) torch.float32 True
bn.3.bias torch.Size([64]) torch.float32 True
bn.4.weight torch.Size([64]) torch.float32 True
bn.4.bias torch.Size([64]) torch.float32 True
bn.5.weight torch.Size([64]) torch.float32 True
bn.5.bias torch.Size([64]) torch.float32 True
```

**Question 10**

**What is the number of parameters of DnCNN(D)?**

Ans) The number of parameters depend on -

1) The number of input channels (Input feature maps)

2) The number of output channels (Output feature maps)

3) The size of the kernel (Filter size)

4) Bias for each feature map

Let number of input channels be n

Let number of output channels be m

Let kernel size be (k * k )

Number of bias terms = Number of output channels = m

**FORMULA to calculate number of parameters**

$$= n * m * k * k + m$$

In case of the network given here,

1) For input convolutional layer

n = 3 , m = 64 , k = 3

No. of parameters = n *m* k * k + m

= 3 *64* 3 * 3 + 64

= 1728 + 64

= 1792

2) For convolutions with depth D

n = 64 , m = 64 , k = 3

No. of parameters for single convolutional layer = n *m* k * k + m

= 64 *64* 3 * 3 + 64

= 36864 + 64

= 36928

3) We consider batch normalization for each layer

Now, batch normalization has 2 parameters - alpha and beta

No. of output channels = 64

Total number of batch normalization parameters for a single layer

= 2 * 64

= 128

4) To calculate parameters for 'D' layers

Total no. of parameters for a single layer = no. of parameters + Batch normalization parameters

= 36928 + 128

= 37056

Total number of parameters for 'D' Layers

= D * no. of parameters for single layer

= D * 37056

= 37056D

5) For output convolutional layer

n = 64 , m = 3 , k = 3

no. of parameters = n *m* k * k + m

= 64 *3* 3 * 3 + 3

= 1728 + 3

= 1731

6) Total number of parameters for entire network ( DnCNN )

= 1792 + 37056D + 1731

= 37056D + 3523

$$Answer = 37056D + 3523$$

For D = 6 ,

$$= 37056 * 6 + 3523$$
$$= 225859$$

In [37]:

```
param_num = sum([p.numel() for p in exp1.net.parameters() if p.requires_grad])

print(param_num)
```

```
225859
```

**What is the receptive field of DnCNN(D), i.e, how many input pixels do influence an output pixel?**

The receptive field at the input is 1 .

The receptive field increases after each convolution by $2^{k-l+1}$ times the number needed to pad to preerve the spatial resolution .

k and l are the number of max pooling and un pooling placed before the convolution .

The receptive field for DnCNN is

= receptive field at input + receptive field at first convolution + receptive field due to D convolutions + receptive field at the last convolution

= 1 + $2^{0-0+1}$ + ( D ) $2^{0-0+1}$ + $2^{0-0+1}$

= 1 + 2 + 2D + 2

$$Answer = 5 + 2D$$

If D = 6 ,

$$5 + 2(6) = 17$$

$$Receptive field = 17x17$$

**Question 11**

Denoising literature claims that for reducing Gaussian noise of standard deviation σ = 30 efficiently, a pixel should be influenced by at least 33×33 pixels. How large D (how deep) should DnCNN be to satisfy this constraint? What would be the implication on the number of parameters and the computation time?

To find D :

We have obtained equation for receptive field ( i.e. the number of input pixels that influence the output pixels ) as

$$=> 5 + 2D$$

Here , it is being influenced by 33 x 33 pixels. Therefore, receptive field is 33

Hence, the equation becomes

$$5 + 2D = 33$$
$$2D = 28$$
$$D = 14$$

The DnCNN should have a D = 14 ( 14 layers deep ) in order to stisfy the above constraint.

The number of parameters increase as the value of D increases.

They become -

$$37056D + 3523$$

For D = 14 ,

$$= 37056 * 14 + 3523$$
$$= 522307$$

The computation time increases because since there are more layers the number of convolutions have also increased.

**Question 12**

In [39]:

```
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))

        for i in range(D):

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

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


        self.bn = nn.ModuleList()

        for i in range(D):
            self.bn.append(nn.BatchNorm2d(C,C))


    def forward(self, x):
        D = self.D
        h = F.relu(self.conv[0](x))

        contract_feature = []
        index = []
        feature_dim = []

        for k in range(D//2-1):

            feature_dim.append(h.shape)


            h, indices = F.max_pool2d(F.relu(self.bn[k](self.conv[k+1](h))),
                                      kernel_size=(2, 2), return_indices=True)

            contract_feature.append(h)

            index.append(indices)


        for k in range(D//2-1, D//2+1):

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



        for i in range(D//2+1, D):

            j = i - (D // 2 + 1) + 1

            h = F.max_unpool2d(F.relu(self.bn[i](self.conv[i+1]((h+contract_feature[-j
])/np.sqrt(2)))),
                               index[-j], kernel_size=(2, 2), output_size=feature_dim[-
j])
```

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

### Question 13

In [40]:

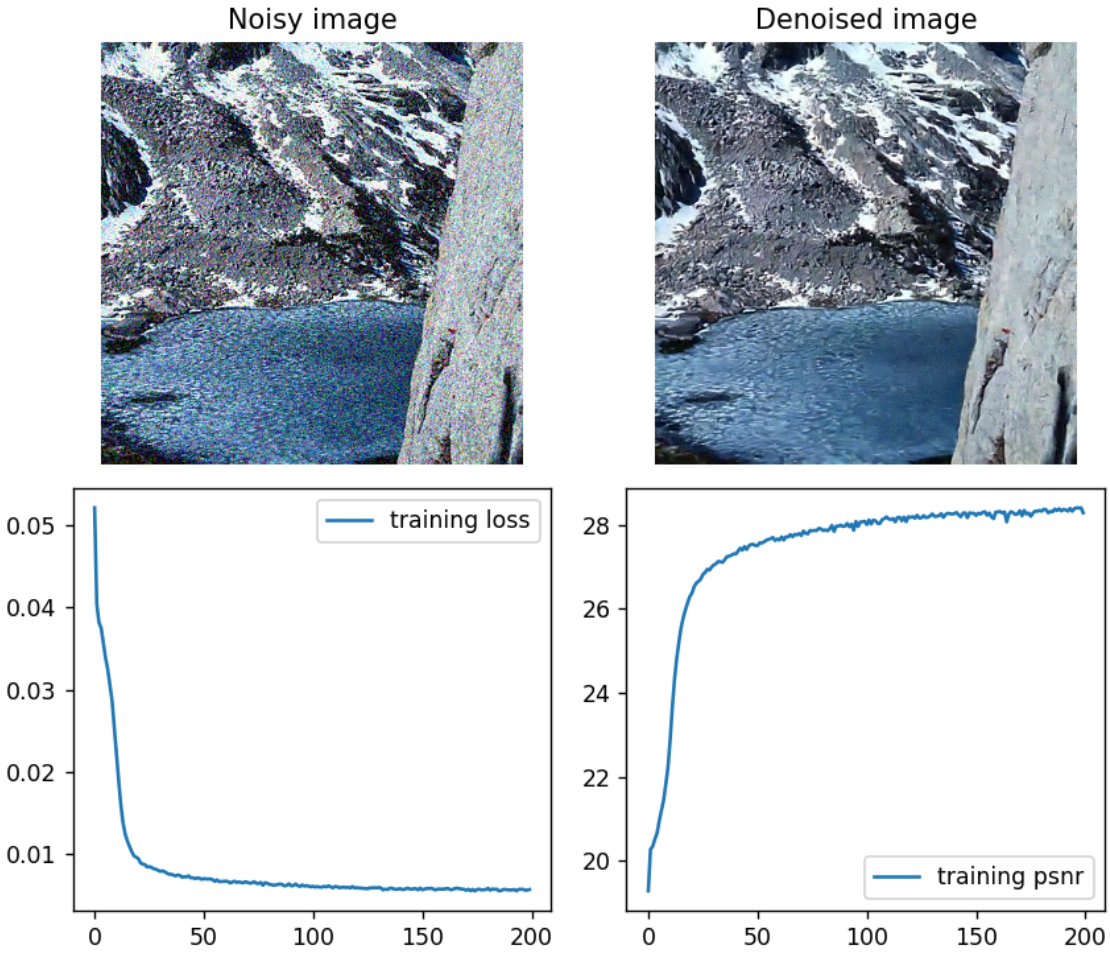
```
lr = 1e-3

net = UDnCNN(6)

net = net.to(device)

adam = torch.optim.Adam(net.parameters(), lr=lr)

stats_manager = DenoisingStatsManager()

exp2 = nt.Experiment(net, train_set, test_set, adam, stats_manager,
output_dir="denoising2", batch_size=4 , perform_validation_during_training=True)
```

In [41]:

```
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][0]['loss'] for k in range(exp.epoch)],
    label="training loss")

    axes[1][1].plot([exp.history[k][1]['PSNR'] for k in range(exp.epoch)],
    label="training psnr")

    axes[1][0].legend()
    axes[1][1].legend()


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

In [42]:

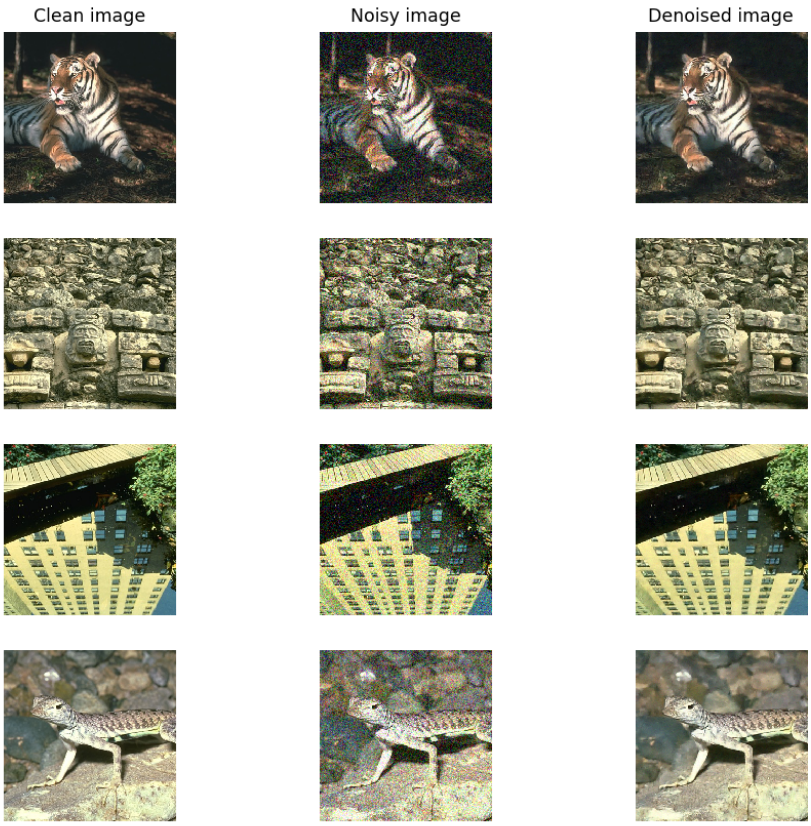
```
fig, axes = plt.subplots(ncols=2, nrows=2, figsize=(7,6))
exp2.run(num_epochs=200, plot=lambda exp: plot(exp, fig=fig, axes=axes, noisy=test_set[
73][0]))
```




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

In [44]:

```
fig, axes = plt.subplots(nrows=4, ncols=3, figsize =(9,8))

i = 0

for i in range(len(test_set)):

    if i == 4:
        break

    noisy, clean = test_set[i]
    myimshow(clean, ax =axes[i][0])
    axes[0][0].set_title('Clean image', fontsize = 10)

    myimshow(noisy, ax = axes[i][1])
    axes[0][1].set_title('Noisy image', fontsize = 10)

    with torch.no_grad():
        denoised = exp2.net(noisy[np.newaxis].to(exp2.net.device))[0]
    myimshow(denoised, ax = axes[i][2])
    axes[0][2].set_title('Denoised image' , fontsize = 10)
```




### Question 14

### UDnCNN

**What is the number of parameters of UDnCNN(D)?**

Ans) The number of parameters depend on -

1) The number of input channels (Input feature maps)

2) The number of output channels (Output feature maps)

3) The size of the kernel (Filter size)

4) Bias for each feature map

Let number of input channels be n

Let number of output channels be m

Let kernel size be (k * k )

Number of bias terms = Number of output channels = m

**FORMULA to calculate number of parameters**

$$= n * m * k * k + m$$

In case of the network given here,

1) For input convolutional layer

n = 3 , m = 64 , k = 3

No. of parameters = n *m* k * k + m

= 3 *64* 3 * 3 + 64

= 1728 + 64

= 1792

2) For convolutions with depth D

n = 64 , m = 64 , k = 3

No. of parameters for single convolutional layer = n *m* k * k + m

= 64 *64* 3 * 3 + 64

= 36864 + 64

= 36928

3) We consider batch normalization for each layer

Now, batch normalization has 2 parameters - alpha and beta

No. of output channels = 64

Total number of batch normalization parameters for a single layer

= 2 * 64

= 128

4) To calculate parameters for 'D' layers

Total no. of parameters for a single layer = no. of parameters + Batch normalization parameters

= 36928 + 128

= 37056

Total number of parameters for 'D' Layers

= D * no. of parameters for single layer

= D * 37056

= 37056D

5) For output convolutional layer

n = 64 , m = 3 , k = 3

no. of parameters = n *m* k * k + m

= 64 *3* 3 * 3 + 3

= 1728 + 3

= 1731

6) Total number of parameters for entire network ( UDnCNN )

= 1792 + 37056D + 1731

= 37056D + 3523

$$Answer = 37056D + 3523$$

For D = 6 ,

$$= 37056 * 6 + 3523$$
$$= 225859$$

In [43]:

```
param_num = sum([p.numel() for p in exp1.net.parameters() if p.requires_grad])

print(param_num)
```

```
225859
```

**What is the receptive field of UDnCNN(D)?**

The receptive field at the input is 1 .

The receptive field increases after each convolution by $2^{k-l+1}$ times the number needed to pad to preerve the spatial resolution .

k and l are the number of max pooling and un pooling placed before the convolution .

The receptive field for DnCNN is

= receptive field at input + receptive field at first convolution + receptive field at second convolution + receptive field at

convolution after 1st max pool + receptive field at convolution after 2nd maxpool + ..... convolution after D/2 max pools +

receptive field at convolution after 1st unpool + receptive field at convolution after 2nd unpool + ..... convolution after D/2 unpools

+receptive field at last convolution

= 1 + $2^{0-0+1}$ + $2^{0-0+1}$ + $2^{1-0+1}$ + $2^{2-0+1}$ + ..... $2^{D/2-1-0+1}$ + $2^{D/2-1-0+1}$ +

$2^{D/2-1-1+1}$ + $2^{D/2-1-2+1}$ + ..... + $2^{(D/2-1)-(D/2-1)}$ + 1 + 2

= 1 + 2 + 2 + $2^2$ + .....$2^{D/2}$ + ..... + $2^{(D/2)}$ + $2^{(D/2-1)}$ + ..... 2 + 2

= 2 + $2^2$ + ..... $2^{D/2}$ ( Geometric series )

= 2 ( \frac{1 - $2^{D/2}$}{1 - 2} )

= 2 ( $2^{D/2}$ - 1 )

Therefore , we get

= 3 + 2 ( $2^{D/2}$ - 1 ) + 2 ( $2^{D/2}$ - 1 ) + 2

= 3 + 4 ( $2^{D/2}$ - 1 ) + 2

if D = 6

= 3 + 4 ( $2^{6/2}$ - 1 ) + 2

= 3 + 4 ( $2^3$ - 1 ) + 2

= 3 + 2 + 4( 8 - 1 )

= 5 + 4 * 7

= 5 + 28

= 33

$$Receptive field = 33 * 33$$

1089 input pixels influence the output pixels

**Do you expect UDnCNN(D) to beat DnCNN(D)?**

Yes, I expect -the UDnCNN network to beat the DnCNN network . The max pooling operations reduces the spatial dimensions of the image . As a result the number of operations that are done in convolution decreases and the time to train the network also reduces.

Also, the UDnCNN has lasrger receptive field than DnCNN . As a result, it considers global features . The local specular noise is less considered as compared to the global features. The skip connections help to preserve the low level features.

Hence, the PSNR is expected to increase and loss is expected to reduce.

In the above two networks, the PSNR and loss for DnCNN and UDnCNN are comparable and have similar values.

In [46]:

```
print(net)
```

```
UDnCNN(
  (MSE): MSELoss()
  (conv): ModuleList(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): Conv2d(64, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (bn): ModuleList(
    (0): BatchNorm2d(64, eps=64, momentum=0.1, affine=True, track_running_
stats=True)
    (1): BatchNorm2d(64, eps=64, momentum=0.1, affine=True, track_running_
stats=True)
    (2): BatchNorm2d(64, eps=64, momentum=0.1, affine=True, track_running_
stats=True)
    (3): BatchNorm2d(64, eps=64, momentum=0.1, affine=True, track_running_
stats=True)
    (4): BatchNorm2d(64, eps=64, momentum=0.1, affine=True, track_running_
stats=True)
    (5): BatchNorm2d(64, eps=64, momentum=0.1, affine=True, track_running_
stats=True)
  )
)
```

In [47]:

```
for name, param in net.named_parameters():
    print(name, param.size(), param.dtype, param.requires_grad)
```

```
conv.0.weight torch.Size([64, 3, 3, 3]) torch.float32 True
conv.0.bias torch.Size([64]) torch.float32 True
conv.1.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.1.bias torch.Size([64]) torch.float32 True
conv.2.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.2.bias torch.Size([64]) torch.float32 True
conv.3.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.3.bias torch.Size([64]) torch.float32 True
conv.4.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.4.bias torch.Size([64]) torch.float32 True
conv.5.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.5.bias torch.Size([64]) torch.float32 True
conv.6.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.6.bias torch.Size([64]) torch.float32 True
conv.7.weight torch.Size([3, 64, 3, 3]) torch.float32 True
conv.7.bias torch.Size([3]) torch.float32 True
bn.0.weight torch.Size([64]) torch.float32 True
bn.0.bias torch.Size([64]) torch.float32 True
bn.1.weight torch.Size([64]) torch.float32 True
bn.1.bias torch.Size([64]) torch.float32 True
bn.2.weight torch.Size([64]) torch.float32 True
bn.2.bias torch.Size([64]) torch.float32 True
bn.3.weight torch.Size([64]) torch.float32 True
bn.3.bias torch.Size([64]) torch.float32 True
bn.4.weight torch.Size([64]) torch.float32 True
bn.4.bias torch.Size([64]) torch.float32 True
bn.5.weight torch.Size([64]) torch.float32 True
bn.5.bias torch.Size([64]) torch.float32 True
```

**Question 15**

Using the method evaluate of Experiment, compare the validation performance of DnCNN and UDnCNN. Interpret the results

In [48]:

```
exp1.evaluate()
```

Out[48]:

```
{'loss': 0.005297933360561728, 'PSNR': tensor(28.8673, device='cuda:0')}
```

In [49]:

```
exp2.evaluate()
```

Out[49]:

```
{'loss': 0.006024486999958754, 'PSNR': tensor(28.2924, device='cuda:0')}
```

It can be observed that the loss for DnCNN network is = 0.00529 and for UDnCNN loss is = 0.00602

Hence, the loss slighly increases for UDnCNN network .

PSNR for DnCNN network is = 28.867 and for UDnCNN PSNR is = 28.292

Hence, the PSNR slighly reduces for UDnCNN .

**Question 16 and 17**

**DUDnCNN**

In [50]:

```
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, padding= 1))

        self.bn = nn.ModuleList()

        for i in range(D):
            self.bn.append(nn.BatchNorm2d(C))

        for k in range(int(D/2)-1):
            d = (2**(k))
            self.conv.append(nn.Conv2d(C, C, 3, padding= d , dilation = d))

        for m in range(2):
            d = (2**(int(D/2)-1))
            self.conv.append(nn.Conv2d(C, C, 3, padding= d , dilation = d))

        for l in range(int(D/2)-1):
            d = (2**(int(D/2)-1 - l))
            self.conv.append(nn.Conv2d(C, C, 3, padding= d , dilation = d))

        self.conv.append(nn.Conv2d(C, 3, 3, padding= 1 , dilation = 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
        features1 = []
        features1.append(h)

        z = 0
        indexs = []
        for i in range(int(D/2) - 1):
            torch.backends.cudnn.benchmark = True
            h = (F.relu(self.bn[i](self.conv[i+1](h))))
            torch.backends.cudnn.benchmark = False

            features1.append(h)

            z = z+1

        torch.backends.cudnn.benchmark = True
        p = (F.relu(self.bn[z](self.conv[z+1](h))))
        torch.backends.cudnn.benchmark = False
        torch.backends.cudnn.benchmark = True
        q = (F.relu(self.bn[z+1](self.conv[z+2](p))))
        torch.backends.cudnn.benchmark = False
        q = q + features1[-1] / 2**.5

        for k in range(int(D/2) - 1):
             torch.backends.cudnn.benchmark = True
             q =  (F.relu(self.bn[D // 2 + 1 + k](self.conv[D // 2 + k + 2](q)))
             torch.backends.cudnn.benchmark = False
             q = q + features1[-(k+2)] / 2**.5
```

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

### Question 18

In [51]:

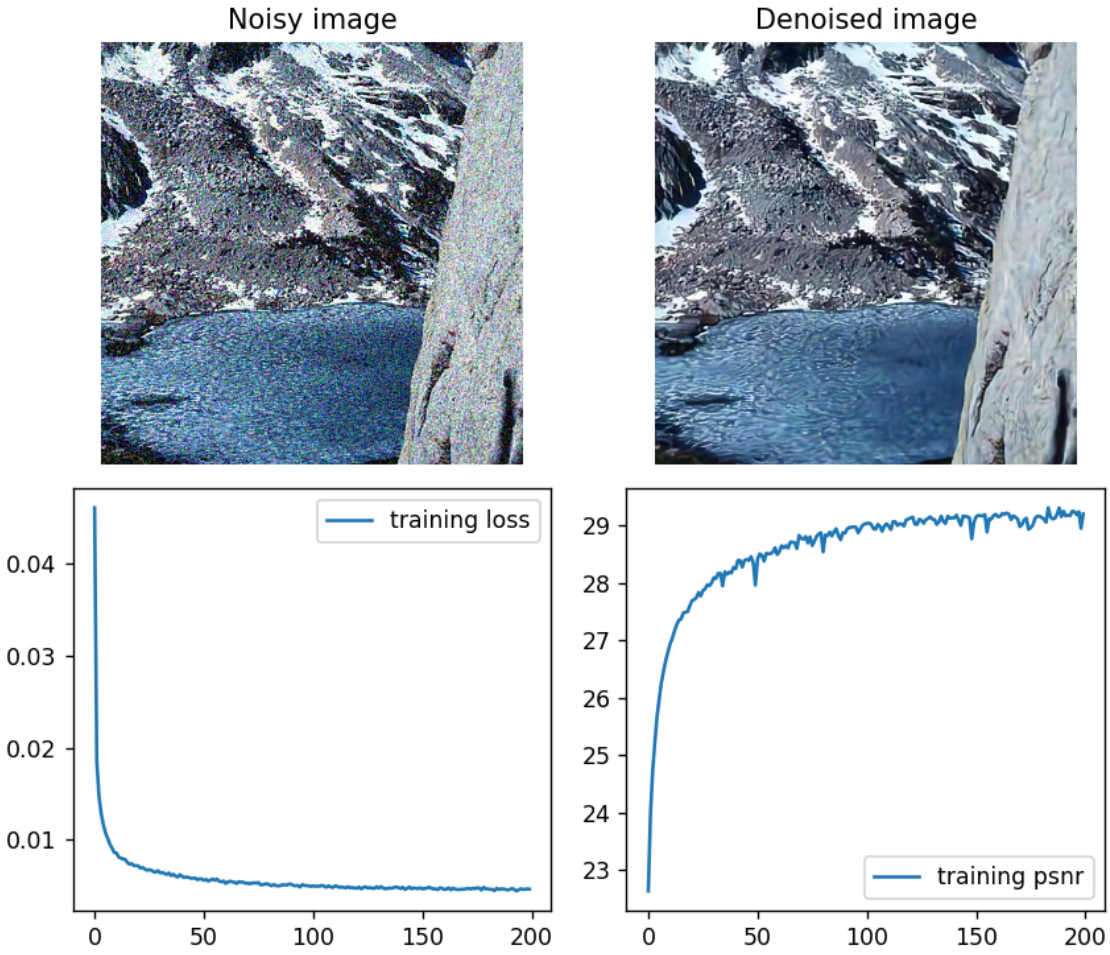
```
lr = 1e-3

net = DUDnCNN(6)

net = net.to(device)

adam = torch.optim.Adam(net.parameters(), lr=lr)

stats_manager = DenoisingStatsManager()

exp3 = nt.Experiment(net, train_set, test_set, adam, stats_manager,
output_dir="denoising3", batch_size=4 , perform_validation_during_training=True)
```

In [52]:

```
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][0]['loss'] for k in range(exp.epoch)],
    label="training loss")

    axes[1][1].plot([exp.history[k][1]['PSNR'] for k in range(exp.epoch)],
    label="training psnr")

    axes[1][0].legend()
    axes[1][1].legend()


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

In [53]:

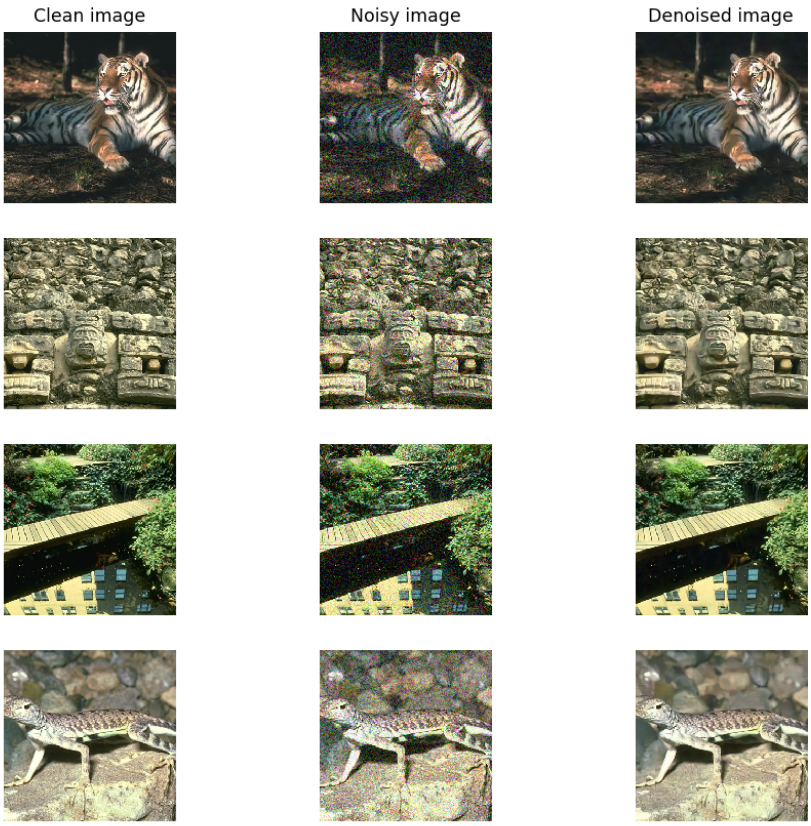
```
fig, axes = plt.subplots(ncols=2, nrows=2, figsize=(7,6))
exp3.run(num_epochs=200, plot=lambda exp: plot(exp, fig=fig, axes=axes, noisy=test_set[
73][0]))
```




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

In [54]:

```
fig, axes = plt.subplots(nrows=4, ncols=3, figsize =(9,8))

i = 0

for i in range(len(test_set)):

    if i == 4:
        break

    noisy, clean = test_set[i]
    myimshow(clean, ax =axes[i][0])
    axes[0][0].set_title('Clean image', fontsize = 10)

    myimshow(noisy, ax = axes[i][1])
    axes[0][1].set_title('Noisy image', fontsize = 10)

    with torch.no_grad():
        denoised = exp3.net(noisy[np.newaxis].to(exp3.net.device))[0]
    myimshow(denoised, ax = axes[i][2])
    axes[0][2].set_title('Denoised image' , fontsize = 10)
```




**Question 19**

Compare the validation performance of DnCNN, UDnCNN and DUDnCNN. Display results from a few samples of the testing set. Make sure your results are similar to the ones shown on Figure 1.

In [56]:

```
print("Evaluation of DnCNN : " , exp1.evaluate() )
```

```
Evaluation of DnCNN :  {'loss': 0.00530330971814692, 'PSNR': tensor(28.868
6, device='cuda:0')}
```

In [57]:

```
print("Evaluation of UDnCNN : " , exp2.evaluate() )
```

```
Evaluation of UDnCNN :  {'loss': 0.005919456835836172, 'PSNR': tensor(28.3
676, device='cuda:0')}
```

In [58]:

```
print("Evaluation of DUDnCNN : " , exp3.evaluate() )
```

```
Evaluation of DUDnCNN :  {'loss': 0.004875807603821158, 'PSNR': tensor(29.
2428, device='cuda:0')}
```

In [55]:

```
noisy = test_set[7][0]
clean = test_set[7][1]

with torch.no_grad():
    denoised1 = exp1.net(noisy[np.newaxis].to(exp1.net.device))[0]
    denoised2 = exp2.net(noisy[np.newaxis].to(exp2.net.device))[0]
    denoised3 = exp3.net(noisy[np.newaxis].to(exp3.net.device))[0]

fig, axes = plt.subplots(1,3, sharex='all' , sharey = 'all')


axes[0].set_title('DnCNN' , fontsize = 10)
myimshow(denoised1, axes[1])

axes[1].set_title('UDnCNN' , fontsize = 10)
myimshow(denoised2, axes[0])

axes[2].set_title('DUDnCNN' , fontsize = 10)
myimshow(denoised3, axes[2])


noisy = test_set[9][0]
clean = test_set[9][1]


with torch.no_grad():
    denoised1 = exp1.net(noisy[np.newaxis].to(exp1.net.device))[0]
    denoised2 = exp2.net(noisy[np.newaxis].to(exp2.net.device))[0]
    denoised3 = exp3.net(noisy[np.newaxis].to(exp3.net.device))[0]

fig, axes = plt.subplots(1,3, sharex='all' , sharey = 'all')


axes[0].set_title('DnCNN' , fontsize = 10)
myimshow(denoised1, axes[1])

axes[1].set_title('UDnCNN' , fontsize = 10)
myimshow(denoised2, axes[0])

axes[2].set_title('DUDnCNN' , fontsize = 10)
myimshow(denoised3, axes[2])


noisy = test_set[10][0]
clean = test_set[10][1]


with torch.no_grad():
    denoised1 = exp1.net(noisy[np.newaxis].to(exp1.net.device))[0]
    denoised2 = exp2.net(noisy[np.newaxis].to(exp2.net.device))[0]
    denoised3 = exp3.net(noisy[np.newaxis].to(exp3.net.device))[0]

fig, axes = plt.subplots(1,3, sharex='all' , sharey = 'all')


axes[0].set_title('DnCNN' , fontsize = 10)
myimshow(denoised1, axes[1])

axes[1].set_title('UDnCNN' , fontsize = 10)
```

```
myimshow(denoised2, axes[0])

axes[2].set_title('DUDnCNN' , fontsize = 10)
myimshow(denoised3, axes[2])\


noisy = test_set[5][0]
clean = test_set[5][1]


with torch.no_grad():
    denoised1 = exp1.net(noisy[np.newaxis].to(exp1.net.device))[0]
    denoised2 = exp2.net(noisy[np.newaxis].to(exp2.net.device))[0]
    denoised3 = exp3.net(noisy[np.newaxis].to(exp3.net.device))[0]

fig, axes = plt.subplots(1,3, sharex='all' , sharey = 'all')


axes[0].set_title('DnCNN' , fontsize = 10)
myimshow(denoised1, axes[1])

axes[1].set_title('UDnCNN' , fontsize = 10)
myimshow(denoised2, axes[0])

axes[2].set_title('DUDnCNN' , fontsize = 10)
myimshow(denoised3, axes[2])
```

DnCNN UDnCNN DUDnCNN



DnCNN UDnCNN DUDnCNN





Out[55]:

<matplotlib.image.AxesImage at 0x7fdbe9c04e10>

It is observed that the loss with

DnCNN network is = 0.00530

UDnCNN is = 0.00591

DUDnCNN is = 0.00487

The loss of UDnCNN is slighly more than DnCNN loss. However, there is no significant change. The loss decreases as we move from UDnCNN to DUDnCNN . Hence, amongst the 3 networks the DUDnCNN has the least loss.

It is also observed that PSNR with -

DnCNN network is = 28.868

UDnCNN is = 28.367

DUDnCNN is = 29.242

The PSNR of UDnCNN is slightly less than DnCNN psnr. There is no significant change in the psnrs of these 2 networks.

The PSNR icreases as we move from UDnCNN to DUDnCNN .

Hence, DUDnCNN has the highest value of PSNR amongst all three.

Hence, it is the best of the three.

**Question 20**

In [34]:

```
print(net)
```

```
DUDnCNN(
  (MSE): MSELoss()
  (conv): ModuleList(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2),
dilation=(2, 2))
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(4, 4),
dilation=(4, 4))
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(4, 4),
dilation=(4, 4))
    (5): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2),
dilation=(2, 2))
    (6): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): Conv2d(64, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (bn): ModuleList(
    (0): BatchNorm2d(64, eps=64, momentum=0.1, affine=True, track_running_
stats=True)
    (1): BatchNorm2d(64, eps=64, momentum=0.1, affine=True, track_running_
stats=True)
    (2): BatchNorm2d(64, eps=64, momentum=0.1, affine=True, track_running_
stats=True)
    (3): BatchNorm2d(64, eps=64, momentum=0.1, affine=True, track_running_
stats=True)
    (4): BatchNorm2d(64, eps=64, momentum=0.1, affine=True, track_running_
stats=True)
    (5): BatchNorm2d(64, eps=64, momentum=0.1, affine=True, track_running_
stats=True)
  )
)
```

In [33]:

```
for name, param in net.named_parameters():
    print(name, param.size(), param.dtype, param.requires_grad)
```

```
conv.0.weight torch.Size([64, 3, 3, 3]) torch.float32 True
conv.0.bias torch.Size([64]) torch.float32 True
conv.1.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.1.bias torch.Size([64]) torch.float32 True
conv.2.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.2.bias torch.Size([64]) torch.float32 True
conv.3.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.3.bias torch.Size([64]) torch.float32 True
conv.4.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.4.bias torch.Size([64]) torch.float32 True
conv.5.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.5.bias torch.Size([64]) torch.float32 True
conv.6.weight torch.Size([64, 64, 3, 3]) torch.float32 True
conv.6.bias torch.Size([64]) torch.float32 True
conv.7.weight torch.Size([3, 64, 3, 3]) torch.float32 True
conv.7.bias torch.Size([3]) torch.float32 True
bn.0.weight torch.Size([64]) torch.float32 True
bn.0.bias torch.Size([64]) torch.float32 True
bn.1.weight torch.Size([64]) torch.float32 True
bn.1.bias torch.Size([64]) torch.float32 True
bn.2.weight torch.Size([64]) torch.float32 True
bn.2.bias torch.Size([64]) torch.float32 True
bn.3.weight torch.Size([64]) torch.float32 True
bn.3.bias torch.Size([64]) torch.float32 True
bn.4.weight torch.Size([64]) torch.float32 True
bn.4.bias torch.Size([64]) torch.float32 True
bn.5.weight torch.Size([64]) torch.float32 True
bn.5.bias torch.Size([64]) torch.float32 True
```

**What is the number of parameters of DUDnCNN(D)?**

Ans) The number of parameters depend on -

1) The number of input channels (Input feature maps)

2) The number of output channels (Output feature maps)

3) The size of the kernel (Filter size)

4) Bias for each feature map

Let number of input channels be n

Let number of output channels be m

Let kernel size be (k * k )

Number of bias terms = Number of output channels = m

**FORMULA to calculate number of parameters**

$$= n * m * k * k + m$$

In case of the network given here,

1) For input convolutional layer

n = 3 , m = 64 , k = 3

No. of parameters = n *m* k * k + m

= 3 *64* 3 * 3 + 64

= 1728 + 64

= 1792

2) For convolutions with depth D

n = 64 , m = 64 , k = 3

No. of parameters for single convolutional layer = n *m* k * k + m

= 64 *64* 3 * 3 + 64

= 36864 + 64

= 36928

3) We consider batch normalization for each layer

Now, batch normalization has 2 parameters - alpha and beta

No. of output channels = 64

Total number of batch normalization parameters for a single layer

= 2 * 64

= 128

4) To calculate parameters for 'D' layers

Total no. of parameters for a single layer = no. of parameters + Batch normalization parameters

= 36928 + 128

= 37056

Total number of parameters for 'D' Layers

= D * no. of parameters for single layer

= D * 37056

= 37056D

5) For output convolutional layer

n = 64 , m = 3 , k = 3

no. of parameters = n *m* k * k + m

= 64 *3* 3 * 3 + 3

= 1728 + 3

= 1731

6) Total number of parameters for entire network ( DUDnCNN )

= 1792 + 37056D + 1731

= 37056D + 3523

$$Answer = 37056D + 3523$$

For D = 6 ,

$$= 37056 * 6 + 3523$$
$$= 225859$$

In [59]:

```
param_num = sum([p.numel() for p in exp1.net.parameters() if p.requires_grad])

print(param_num)
```

```
225859
```

**What is the receptive field of DUDnCNN(D)?**

The receptive field at the input is 1 .

The receptive field increases after each convolution by $2^{k-l+1}$ times the number needed to pad to preerve the spatial resolution .

k and l are the number of max pooling and un pooling placed before the convolution .

Here, we add a padding term in every convolution.

The receptive field for DnCNN is

= 1 + $2^{0-0+1}$ + $2^0$ $2^{0-0+1}$ + $2^1$ *(padding term)* $2^{0-0+1}$ + ..... $2^{D/2-1}$ + $2^{0-0+1}$ +

- $2^{D/2-1-1}$ $2^{0-0+1}$ + ..... + $2^{(D/2-1)-(D/2-1)}$ $2^{0-0+1}$ + 2

= ( 1 + 2 ) + ( 2 + $2^2$ + .....$2^{D/2}$ ) + ..... + ( $2^{(D/2)}$ + $2^{(D/2-1)}$ + ..... 2 ) + 2

Therefore , we get

= 5 + 2 * 2 (\frac{1 - $2^{D/2}$}{1 - 2})

= 5 + 4 ( $2^{D/2}$ - 1 )

if D = 6

= 5 + 4 ( $2^{6/2}$ - 1 )

= 5 + 4 ( $2^3$ - 1 )

= 5 + 4( 8 - 1 )

= 5 + 4 * 7

= 5 + 28

= 33

$$Receptive field = 33 * 33$$

1089 input pixels influence the output pixels

We conclude that DUDnCNN is the best of the three as it has the maximum psnr and minimum loss.

The receptive field remains the same in UDnCNN and DUDnCNN .

In case of UDnCnn , there is max pooling which leads to loss of information because it picks only the most important features and ignores few pixels. This is not considered good.

In DUDnCNN, dilation is used which preserves the receptive field, that is the number of input pixels that influence the output pixels and does not lose resolution. In dilation, there is no information loss . Hence, DUDnCNN performs the best.

In [ ]: