assignment004

December 1, 2019

1 A53299801 - SIDDARTH MEENAKSHI SUNDARAM

[1]:
```
%matplotlib inline
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
```

[2]:
```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
```

```
cuda
```

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

[4]:
```
#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')
        # random crop
        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, i+self.image_size[0], j+self.image_size[1]])
        transform = tv.transforms.Compose([
            # convert it to a tensor
            tv.transforms.ToTensor(),
            # normalize it to the range [-1, 1]
            tv.transforms.Normalize((.5, .5, .5), (.5, .5, .5))
            ])
        clean = transform(clean)

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

[5]:
```
print(tv.__version__)
```

```
0.4.2
```

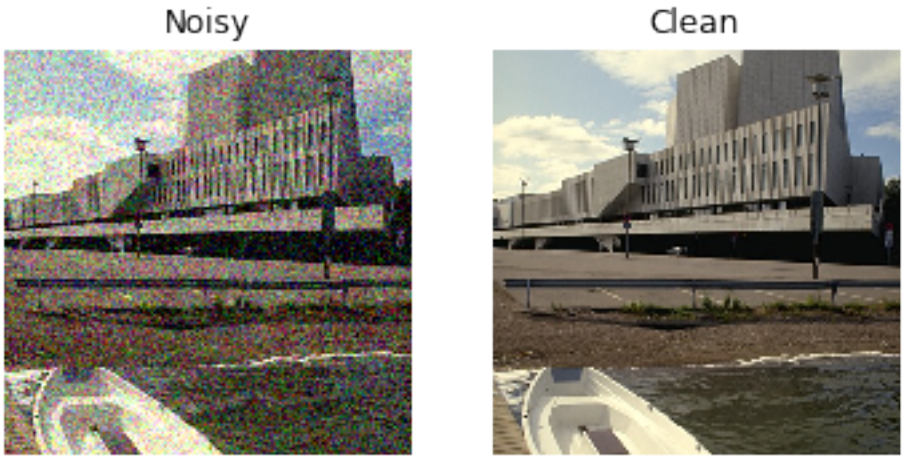
[6]:
```
#3
train_set = NoisyBSDSDataset(dataset_root_dir, mode='train', image_size=(180,␣
↪180))
test_set = NoisyBSDSDataset(dataset_root_dir, mode='test', image_size=(320,␣
↪320))
```

[7]:
```
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
```

[8]:
```
x = test_set[12]
fig, axes = plt.subplots(ncols=2)
myimshow(x[0], ax=axes[0])
axes[0].set_title('Noisy')
myimshow(x[1], ax=axes[1])
```

```
axes[1].set_title('Clean')
print(f'image size is {x[0].shape}.')
```

```
image size is torch.Size([3, 320, 320]).
```



[9]:
```
#4
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)
```

[14]:
```
#5
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.conv.extend([nn.Conv2d(C, C, 3, padding=1) for _ in range(D)])
        self.conv.append(nn.Conv2d(C, 3, 3, padding=1))

        self.bn = nn.ModuleList()
        for k in range(D):
            self.bn.append(nn.BatchNorm2d(C, C))
```

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

[15]:
```
#6
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 = x.shape[0] * x.shape[1] * x.shape[2] * x.shape[3]
        self.running_psnr += 10*torch.log10(4*n/(torch.norm(y-d)**2))

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

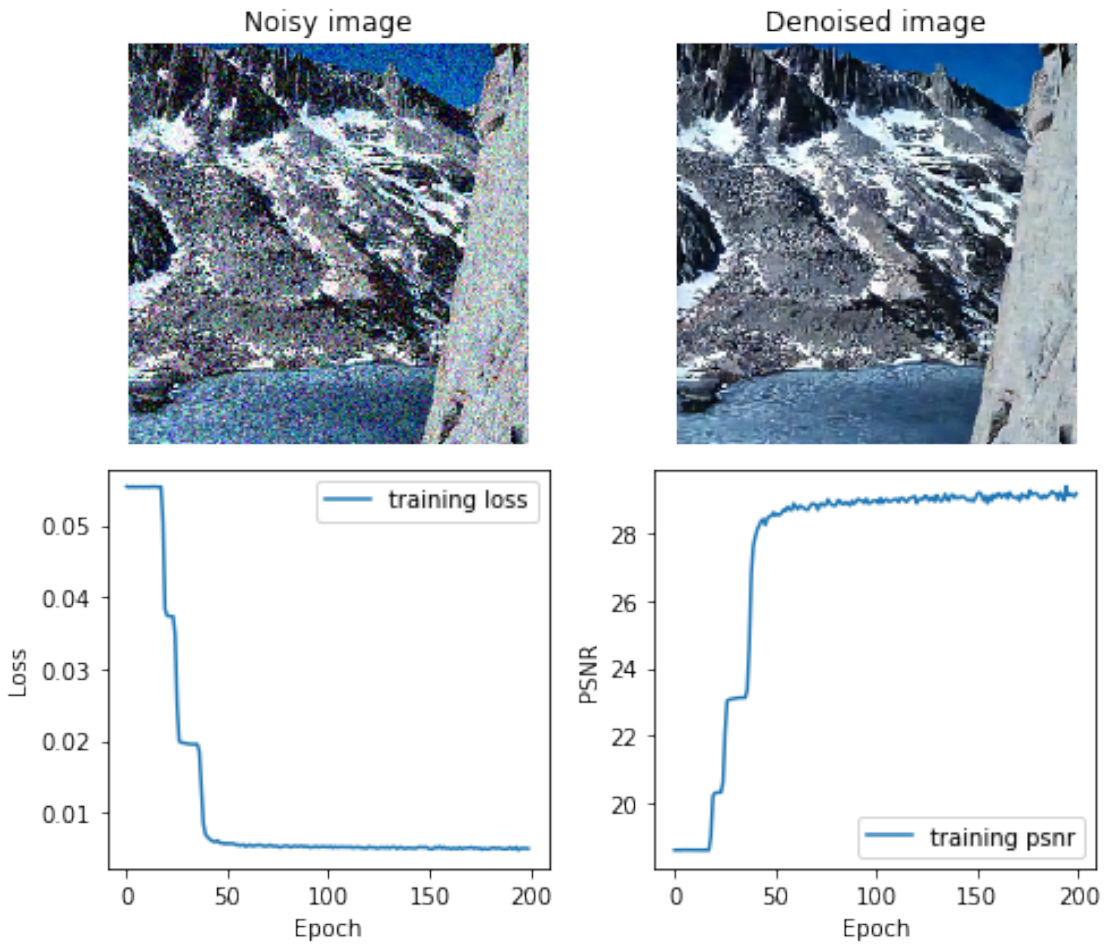
[16]:
```
#7
lr = 1e-3
net = DnCNN(6).to(device)
adam = torch.optim.Adam(net.parameters(), lr=lr)
stats_manager = DenoisingStatsManager()
exp1 = nt.Experiment(net, train_set, test_set, adam, stats_manager,␣
 ↪batch_size=4,
                     output_dir="denoising1", perform_validation_during_training=True)
```

[17]:
```
#8
def plot(exp, fig, axes, noisy, visu_rate=2):
    if exp.epoch % visu_rate != 0:
        return
    with torch.no_grad():
        denoised = exp.net(noisy[None].to(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][0].set_ylabel('Loss')
    axes[1][0].set_xlabel('Epoch')
    axes[1][0].legend()

    axes[1][1].plot([exp.history[k][0]['PSNR'] for k in range(exp.epoch)],␣
↪label='training psnr')
    axes[1][1].set_ylabel('PSNR')
    axes[1][1].set_xlabel('Epoch')
    axes[1][1].legend()

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

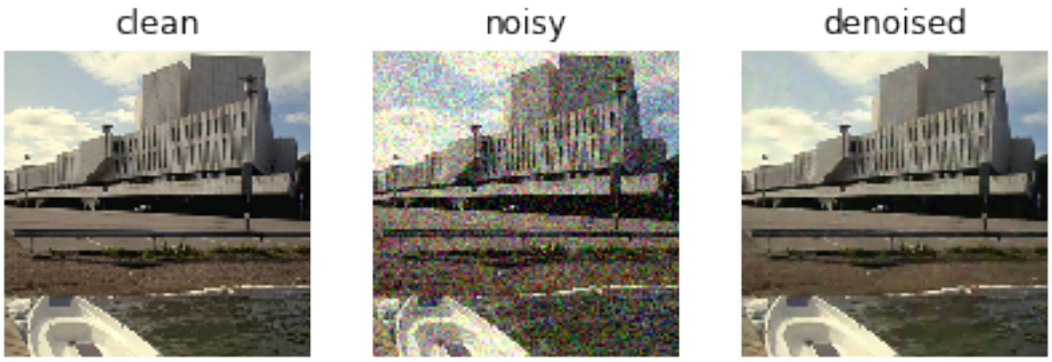
[37]:
```
fig, axes = plt.subplots(ncols=2, nrows=2, figsize=(7, 6))
exp1.run(num_epochs=200, plot=lambda exp: plot(exp, fig=fig, axes=axes,
                                               noisy=test_set[73][0]))
```

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



[19]:
```
#9
img = []
model = exp1.net.to(device)
titles = ['clean', 'noisy', 'denoised']

x, clean = test_set[12]
x = x.unsqueeze(0).to(device)
img.append(clean)
img.append(x[0])

model.eval()
with torch.no_grad():
    y = model.forward(x)
img.append(y[0])

fig, axes = plt.subplots(ncols=3, figsize=(7,6), sharex='all', sharey='all')
for i in range(len(img)):
```

```
    myimshow(img[i], ax=axes[i])
    axes[i].set_title(f'{titles[i]}')
```



We can see that the denoised image is not exactly the same as the clean image. There is some loss of information. If we zoom in, we can see in the denoised image that the white lines on the road have been partially lost. But the denoised image is almost same as the clean image. This can be inferred from PSNR too that some data is lost.

[20]:
```
#10
for name, param in model.named_parameters():
    print(name, param.size(), param.requires_grad)
```

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

```
bn.3.weight torch.Size([64]) True
bn.3.bias torch.Size([64]) True
bn.4.weight torch.Size([64]) True
bn.4.bias torch.Size([64]) True
bn.5.weight torch.Size([64]) True
bn.5.bias torch.Size([64]) True
```

Q10

Number of parameters of DnCNN(D):

The first layer has 64 x 3 x 3 x 3 + 64 parameters = 1792 parameters. The middle D convolution layers have 64 x 64 x 3 x 3 x D + 64 x D parameters = 36928 x D parameters. The last convolution layer has 3 x 64 x 3 x 3 + 3 parameters = 1731 parameters. Bias, Weights of BN - parameters = 64 x 2 x D Hence, there are total 3523 + 37056 x D parameters.

Number of Parameters of DnCNN(D)= 3523 + 37056 x D

In case of D=6, Number of parameters : 3523 + 37056 x D = 225859 parameters.

Receptive field of DnCNN(D):

The receptive field of the input layer is 1. Equation to compute the receptive field is $2^{k-l+1} \times$ *padding size* while preserving spatial resolution, where k and l are the number of the pooling and unpooling layers respectively, There are no pooling and unpooling layers, k=0, l=0, each layer increases the dimensions of receptive field by $2^{0-0+1} = 2$.

The receptive field of DnCNN with depth of d should be (5 + 2d) x (5 + 2d)

In case of D=6, Receptive field : $17x17$

Q11

We need a pixel to be influenced by 33 x 33 pixels. Since receptive field is $(1 + 2 \times (D + 2)) \times (1 + 2 \times (D + 2))$.

$1 + 2 \times (D + 2) = 33.$

$\implies D = 14.$

For the number of parameters, it would be $3523 + 37056 \times 14 = 522307$.

As the number of parameters increases, the computation time increases.

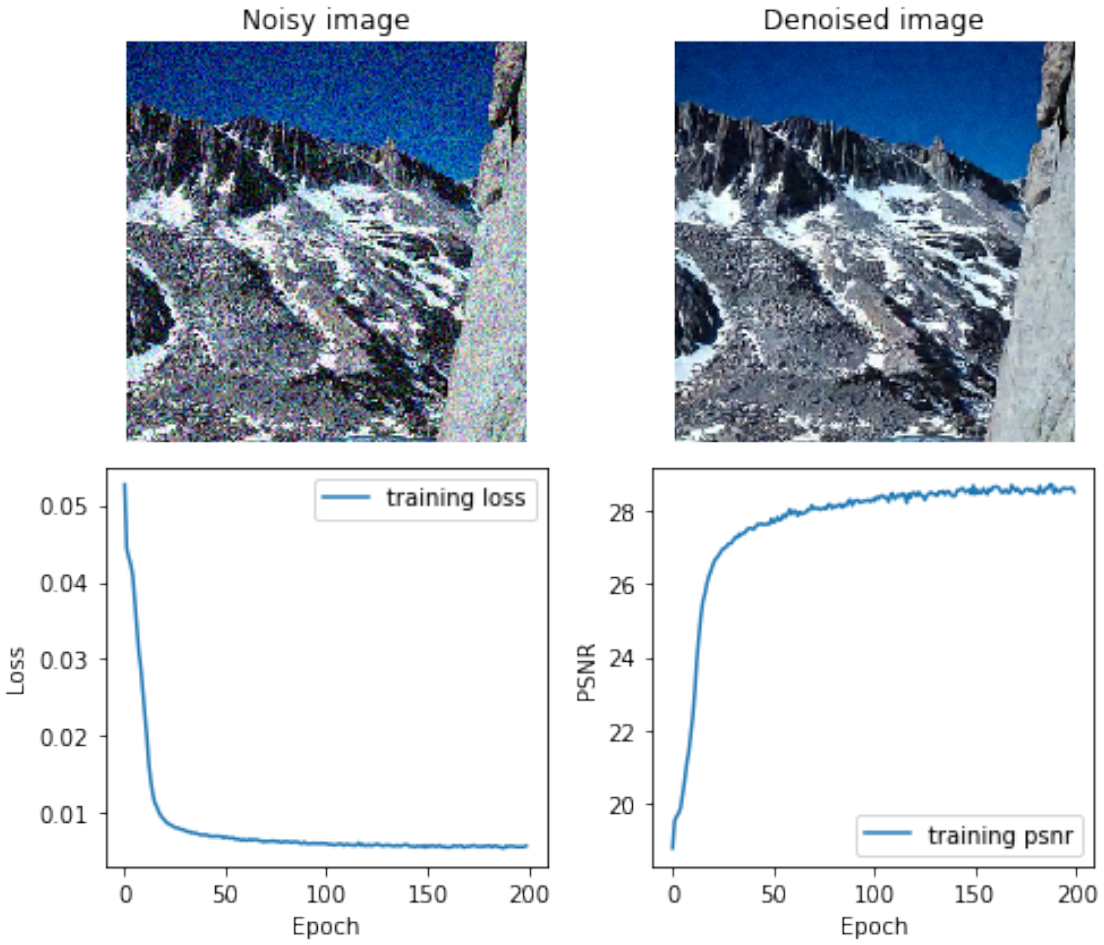
[21]:
```
#12
class UDnCNN(NNRegressor):

    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))
        self.conv.extend([nn.Conv2d(C, C, 3, padding=1) for _ in range(D)])
        self.conv.append(nn.Conv2d(C, 3, 3, padding=1))
```

```
        self.bn = nn.ModuleList()
        for k in range(D):
            self.bn.append(nn.BatchNorm2d(C, C))

    def forward(self, x):
        D = self.D
        h = F.relu(self.conv[0](x))
        features = []
        maxpool_idx = []
        spatial_dim = []
        for i in range(D//2-1):
            spatial_dim.append(h.shape)
            h, idx = F.max_pool2d(F.relu(self.bn[i](self.conv[i+1](h))),
                                  kernel_size=(2,2), return_indices=True)
            features.append(h)
            maxpool_idx.append(idx)
        for i in range(D//2-1, D//2+1):
            h = F.relu(self.bn[i](self.conv[i+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+features[-j])/np.sqrt(2)))),
                               maxpool_idx[-j], kernel_size=(2,2),␣
↪output_size=spatial_dim[-j])
        y = self.conv[D+1](h) + x
        return y
```

[22]:
```
#13
lr = 1e-3
net = UDnCNN(6).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)
```

[23]:
```
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
```



[24]:
```
#14
for name, param in exp2.net.named_parameters():
    print(name, param.size(), param.requires_grad)
```

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

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

Pooling and unpooling operations do not have learnable parameters

The number of parameters of UDnCNN is the same as that of DnCNN = **3523 + 37056 x D**

The receptive field of UDnCNN would be $(\sum_{i=1}^{D/2} 2^{i+1} + 5) \times (\sum_{i=1}^{D/2} 2^{i+1} + 5)$.

When D=6, the receptive field would be $(4 + 8 + 16 + 5) \times (4 + 8 + 16 + 5) = 33 \times 33$

UDnCNN would not beat DnCNN because UDnCNN uses pooling which might lose some information. Based on the training PSNR and loss we can check that DnCNN is slightly better than UDnCNN.

[25]:
```
#15
#DnCNN
exp1.evaluate()
```

[25]:
```
{'loss': 0.005239203460514545, 'PSNR': tensor(28.9295, device='cuda:0')}
```

[26]:
```
# UDnCNN
exp2.evaluate()
```

[26]:
```
{'loss': 0.005947012938559055, 'PSNR': tensor(28.3512, device='cuda:0')}
```

We can see that DnCNN has better performance than UDnCNN as expected. The PSNR of DnCNN is slightly greater than UDnCNN. This can be attributed to loss of information due to usage of pooling layer in UDnCNN.

[27]:
```
#16
#17
class DUDnCNN(NNRegressor):

    def __init__(self, D, C=64):
        super(DUDnCNN, self).__init__()
        self.D = D
```

```
        # compute k(max_pool) and l(max_unpool)
        k = [0]
        k.extend([i for i in range(D//2)])
        k.extend([k[-1] for _ in range(D//2, D+1)])
        l = [0 for _ in range(D//2+1)]
        l.extend([i for i in range(D+1-(D//2+1))])
        l.append(l[-1])

        # holes and dilations for convolution layers
        holes = [2**(kl[0]-kl[1])-1 for kl in zip(k,l)]
        dilations = [i+1 for i in holes]

        # convolution layers
        self.conv = nn.ModuleList()
        self.conv.append(nn.Conv2d(3, C, 3, padding=dilations[0],␣
↪dilation=dilations[0]))
        self.conv.extend([nn.Conv2d(C, C, 3, padding=dilations[i+1],␣
↪dilation=dilations[i+1]) for i in range(D)])
        self.conv.append(nn.Conv2d(C, 3, 3, padding=dilations[-1],␣
↪dilation=dilations[-1]))



        # batch normalization
        self.bn = nn.ModuleList()
        self.bn.extend([nn.BatchNorm2d(C, C) for _ in range(D)])


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

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

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

        for i in range(D//2 + 1, D):
            j = i - (D//2 + 1) + 1
```

```
            torch.backends.cudnn.benchmark = True
            h = self.conv[i+1]((h + features[-j]) / np.sqrt(2))
            torch.backends.cudnn.benchmark = False
            h = F.relu(self.bn[i](h))

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

[28]:
```
#18
lr = 1e-3
net = DUDnCNN(6).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)
```

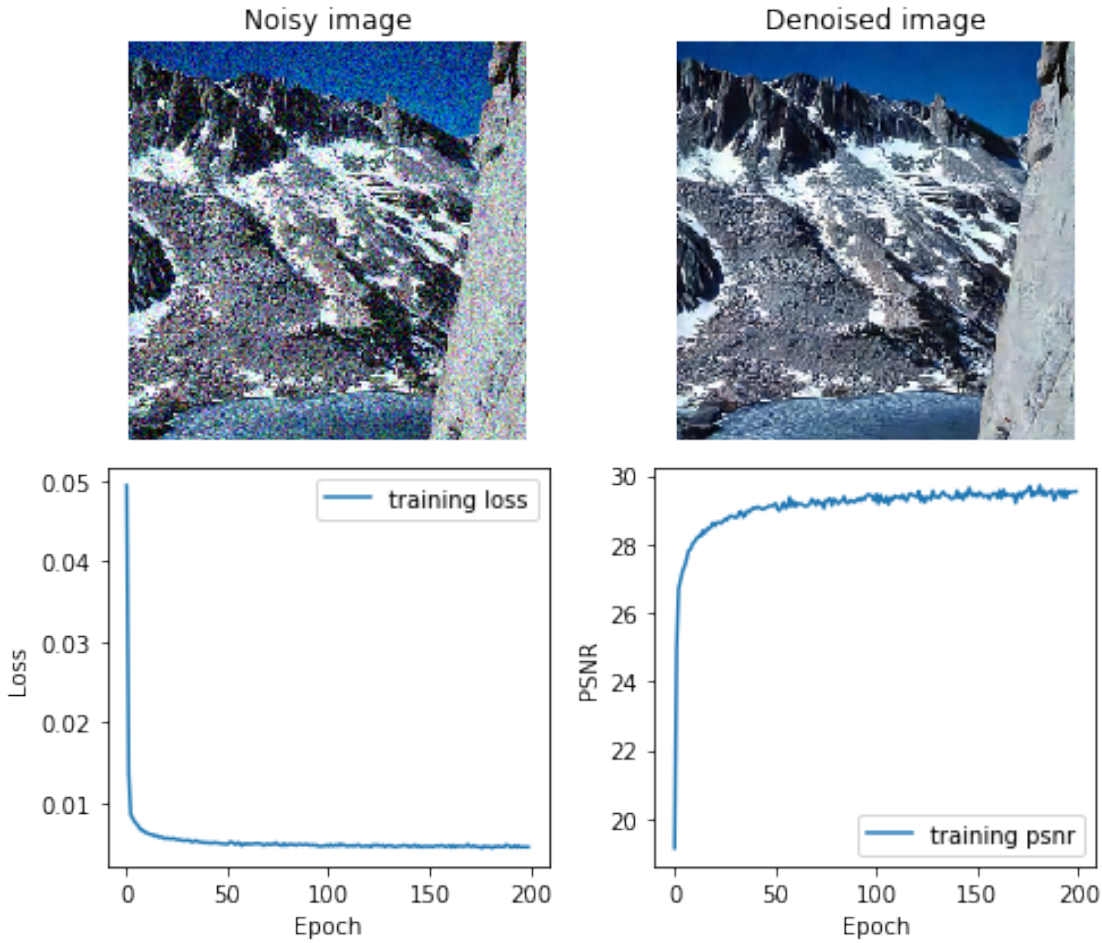
[29]:
```
exp3
```

[29]:
```
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)
  )
))
TrainSet(NoisyBSDSDataset(mode=train, image_size=(180, 180), sigma=30))
ValSet(NoisyBSDSDataset(mode=test, image_size=(320, 320), sigma=30))
Optimizer(Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    eps: 1e-08
    lr: 0.001
    weight_decay: 0
))
StatsManager(DenoisingStatsManager)
BatchSize(4)
PerformValidationDuringTraining(True)
```

[30]:
```
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
```



[31]:
```
# DnCNN
exp1.evaluate()
```

[31]: {'loss': 0.005247166259214282, 'PSNR': tensor(28.9161, device='cuda:0')}

[32]:
```
# UDnCNN
exp2.evaluate()
```

[32]: {'loss': 0.005941951163113118, 'PSNR': tensor(28.3598, device='cuda:0')}

[33]:
```
# DUDnCNN
exp3.evaluate()
```

[33]: {'loss': 0.004997735042124986, 'PSNR': tensor(29.1296, device='cuda:0')}

[34]:
```
#19
num = 3
```

```
img = []
nets = [exp1.net, exp2.net, exp3.net]
titles = ['DnCNN', 'UDnCNN', 'DUDnCNN']

for i in range(num):
    x, _ = test_set[7*i+7]
    x = x.unsqueeze(0).to(device)
    img.append(x)

fig, axes = plt.subplots(nrows=num, ncols=3, figsize=(9,6), sharex='all',␣
↪sharey='all')
for i in range(num):
    for j in range(len(nets)):
        model = nets[j].to(device)
        model.eval()
        with torch.no_grad():
            y = model.forward(img[i])

        myimshow(y[0], ax=axes[i][j])
        axes[i][j].set_title(f'{titles[j]}')
```

[35]:
```
exp3.net
```

[35]:
```
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)
  )
)
```

[36]:
```
#20
for name, param in exp3.net.named_parameters():
    print(name, param.size(), param.requires_grad)
```

```
conv.0.weight torch.Size([64, 3, 3, 3]) True
conv.0.bias torch.Size([64]) True
conv.1.weight torch.Size([64, 64, 3, 3]) True
conv.1.bias torch.Size([64]) True
conv.2.weight torch.Size([64, 64, 3, 3]) True
conv.2.bias torch.Size([64]) True
conv.3.weight torch.Size([64, 64, 3, 3]) True
conv.3.bias torch.Size([64]) True
conv.4.weight torch.Size([64, 64, 3, 3]) True
```

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

Number of parameters:

The number of parameters of DUDnCNN is the same as that of UDnCNN and DnCNN because dilated convolution does not have extra parameters, compared with original convolution. So, **the number of parameters of DUDnCNN(D) is 3523 + 37056 x D**

For D=6, Number of parameters = 225859

Receptive field:

Since the padding size is not the same for each layer as the UDnCNN,

The receptive field for DUDnCNN(D):

$(\sum_{i=1}^{D/2} 2^{i+1} + 5) \times (\sum_{i=1}^{D/2} 2^{i+1} + 5)$

For D=6 , its receptive field is $33x33$