ECE 285 – MLIP – Assignment 4
Image Denoising with Deep CNNs

Written by Charles Deledalle and Anurag Paul on October 18, 2019.

In Assignments 1, 2, 3, we were focusing on classification. In Assignment 3, we introduced you to the data loading mechanisms of the package `torch.utils.data` and learning mechanisms of our homemade package `nntools` which we will continue to use here. In this assignment, we are going to consider a specific regression problem: image denoising. We will be using deep Convolutional Neural Networks (CNNs) with PyTorch, investigate DnCNN and U-net architectures.

We will be using images from the "Berkeley Segmentation Dataset and Benchmark" that are downloadable here https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/, and located on DSMLP here `/datasets/ee285f-public/bsds/`. This directory contains two sub-directories: `train` and `test`, which consist of 200 and 100 images, respectively, of either size 321×481 or 481×321. While we saw that thousand to millions of images were required for image classification, we can use a much smaller training set for image denoising. This is because denoising each pixel of an image can be seen as one regression problem. Hence, our training is in fact composed of $200 \times 321 \times 481 \approx 31$ million samples.

It is possible that DSMLP will be too busy to let you run everything from this assignment. We know what the results should look like, so we will only grade the code.

1 Getting started

First of all, connect to DSMLP (`dsmlp-login.ucsd.edu` server) and start a *pod* with enabled GPU/CUDA capabilities

```
$ launch-scipy-ml-gpu.sh
```

Next connect to your Jupyter Notebook from your web browser (refer to Assignment 0 for more details). Create a new notebook `assignment4.ipynb` and import

```
%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
```

Select the relevant device

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

For the following questions, please write your code and answers directly in your notebook. Organize your notebook with headings, markdown and code cells (following the numbering of the questions).

2 Creating noisy images of BSDS dataset with DataSet

Our goal is to use deep convolutional neural networks to learn the mapping $x_i \to y_i$ where x_i are noisy images (our data/observations) and y_i are clean images (our labels/ground-truth)[1]. We will consider the images of the BSDS dataset as our clean/ground-truth images: y_i. For each of them, we will generate noisy versions by adding white Gaussian noise: $x_i = y_i + w_i$ where w_i is an image where each pixel is an independent realization of a zero-mean Gaussian distribution with standard deviation $\sigma = 30$. Since images have different sizes, we will consider random crops of size of 180×180.

1. Create a variable `dataset_root_dir` and make it point to the BSDS dataset directory.

2. As we built a dataset class in Assignment 3, we will do that here again to create our `NoisyBSDSDataset`. Interpret and complete the following piece of code:

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

[1] Note that in regression literature, notation y is often used for noisy data and x for clean data, but we will use the machine learning notations where x is the observed data and y the prediction.

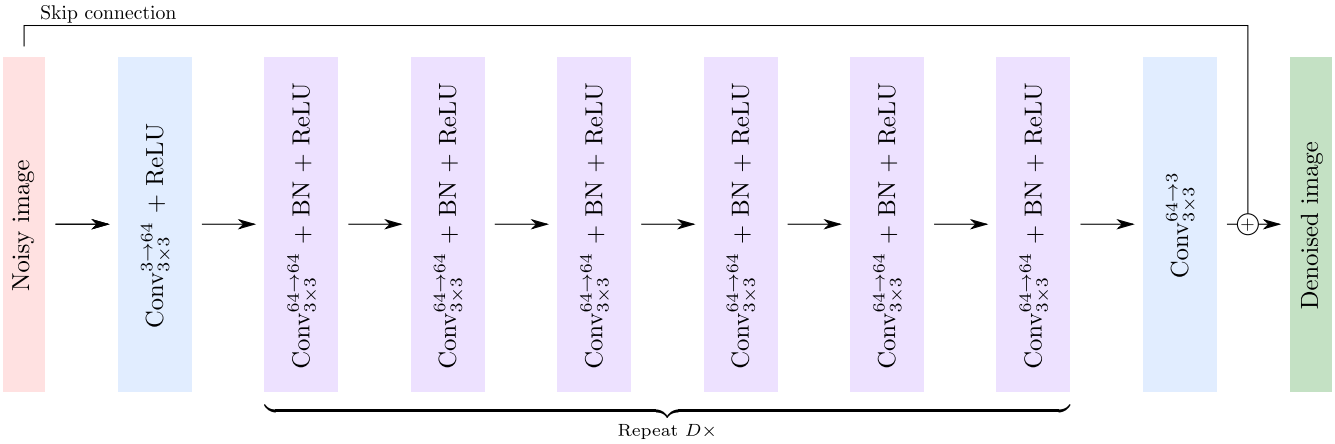
the method `__getitem__` should return two torch tensors of shape $(3, n_1, n_2)$ where $(n_1, n_2) =$ `image_size` and normalized to the range $[-1, 1]$.

3. Create `train_set` and `test_set` as instances of `NoisyBSDSDataset` for `mode='train'` and `mode='test'` respectively. Consider images of size 180×180 for training and 320×320 for testing. Sample the element with index 12 in the testing set. Store it in a variable `x`. Use `myimshow` function from Assignment 3 to display next to each other the clean and noisy version of this sample.

   Note that unlike the networks we investigate for classification, here, the networks will be fully convolutional. The architecture is then independent of the spatial dimension of the input. That is why we can use different spatial dimensions for training and validation.

3 DnCNN

In this assignment, we are going to investigate three different CNN architectures with variable depths. To make our code more versatile, easy to read and save precious coding time, all these architectures are going to inherit from the abstract class `NeuralNetwork` of the `nntools` package. The first architecture we will be considering is called DnCNN and can be described as:



Similar to VGG, it uses only 3×3 convolutions. Similar to ResNet, it uses a skip connection between the first and the last layer.

4. Similar to `NNClassifier` from Assignment 3, create a subclass `NNRegressor` that inherits from `NeuralNetwork` and implements the method `criterion` as being the MSE loss.

5. Copy, interpret and complete the following code implementing DnCNN:

```
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=?))
        # COMPLETE
        self.bn = nn.ModuleList()
        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))
        # COMPLETE
        y = self.conv[D+1](h) + x
        return y
```

Refer to PyTorch's documentation for detailed explanations about `nn.ModuleList` and `nn.BatchNorm2d`. Note that in order to preserve the spatial feature dimensions between each successive layer of the network, we will have to use zero-padding by a suitable number of pixels that you have to determine.

6. A very classical (but controversial) way to compare the quality of restoration techniques is to use the PSNR (Peak Signal-to-Noise-Ratio) defined for images ranging in $[-1, 1]$ as

$$\text{PSNR} = 10 \log_{10} \frac{4n}{\|y - d\|_2^2} \tag{1}$$

where d is the desired ideal image, y the estimate obtained from x and n the number of elements in the tensor. The PSNR measures in decibels (dB) the quality of the restoration: the higher the better. Similar to `ClassificationStatsManager` from Assignment 3, create the subclass `DenoisingStatsManager` that inherits from `StatsManager` and computes and averages PSNR between mini-batches (instead of accuracy).

7. Create a DnCNN network with $D = 6$ and transfer your network to GPU. Create an experiment `exp1` for DnCNN based on Adam optimizer with learning rate `1e-3`, using `DenoisingStatsManager`, mini batches of size 4, and storing the checkpoints in `denoising1`.

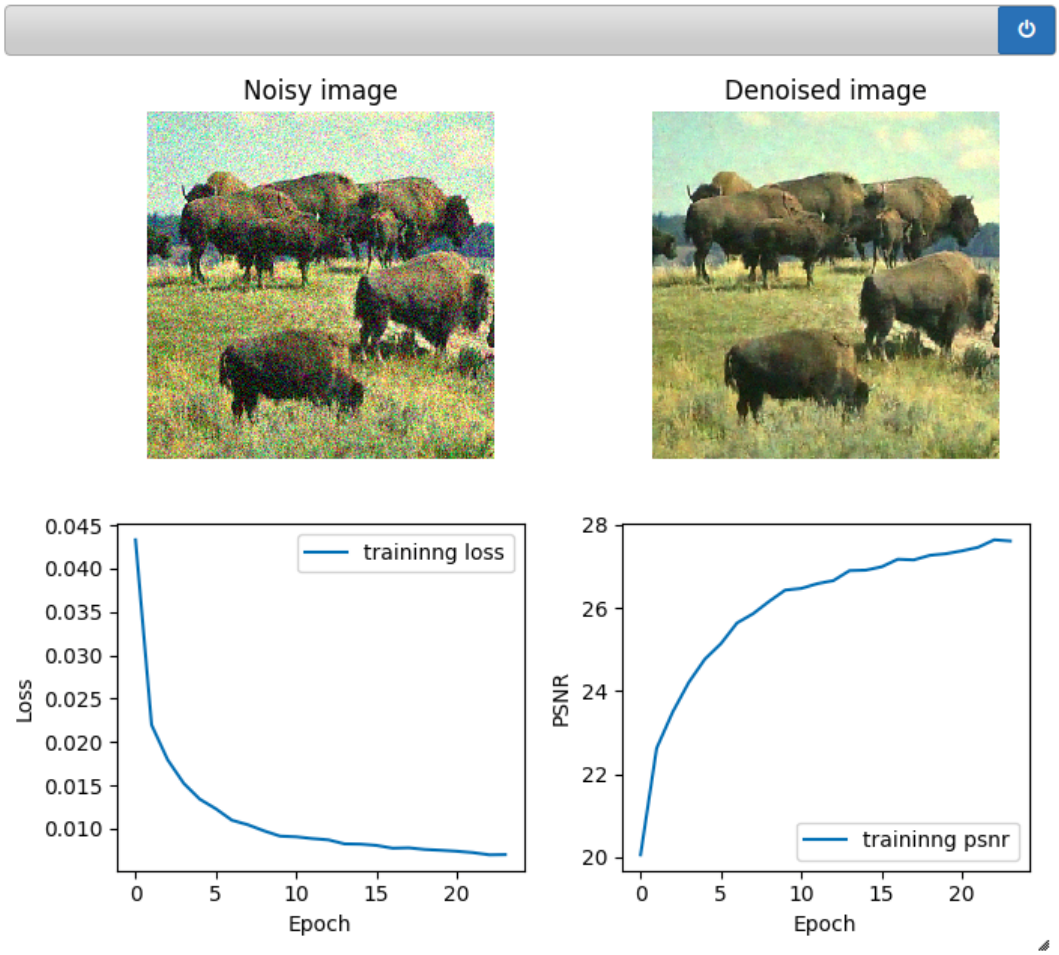
8. Run the experiment for 200 epochs after completing the following

```
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')
    # COMPLETE
    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=test_set[73][0]))
```

such that your function displays at every `visu_rate=2` epochs something similar to the results next page. If it does not so, interrupt it (`Esc+i+i`), check your code, delete the `output_dir`, and start again.

```
In [*]: 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 23
Epoch 24 (Time: 3.65s)
```

9. Once you have trained your network for 200 epochs (about 17 minutes), compare the noisy, clean and denoised results on a few images of the testing set. Evaluate the visual quality of your result. Do you see any artifacts or loss of information?

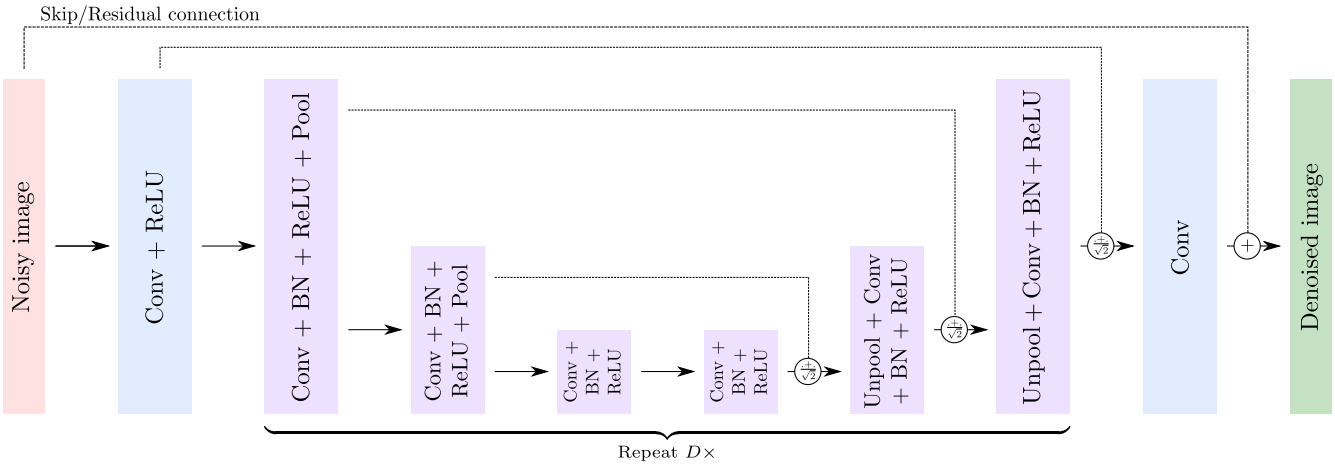
   Hint: use a `plt.subplots` created with the arguments `sharex='all'` and `sharey='all'`. This will allow you to zoom and navigate on the images in a synchronized way.

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

11. Denoising literature claims that for reducing Gaussian noise of standard deviation $\sigma = 30$ 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?

4 U-net like CNNs

As seen in class, pooling layers allows us to increase the receptive field without making the network deeper and slower. But pooling layers lose spatial resolution. In order to retrieve the spatial dimension we will have to use unpooling (like introduced for feature visualizing in ZFNet). Our architecture will resemble the so-called U-net architecture (that was originally introduced for image segmentation). Our network will consist of a contracting path and an expansive path, which gives it the U-shaped architecture. The contracting path is a typical convolutional network that consists of repeated application of convolutions, ReLU and max pooling operation. During the contraction, the spatial information is reduced. The expansive pathway reconstructs spatial information through a sequence of unpooling and convolutions (note that if we were using strided convolutions, we would have to use transpose convolutions), and combined high-resolution features from the contracting path. In the original U-net, features were combined by concatenating the channels of the tensors, but here we will consider their sum divided by $\sqrt{2}$ (this is to preserve the range of feature variations). The architecture is summarized in the following scheme:



12. Copy paste the class `DnCNN` into a new class `UDnCNN`. Modify it to implement the U-shaped DnCNN. Use `F.max_pool2d(..., return_indices=True)` and `F.max_unpool2d(..., output_size=...)` of size 2×2. You may need to create three lists to store the features, the spatial dimensions and the max pool indices.

    Hint: if the layer with index k is in the expansive pathway, its symmetrical layer in the contractive pathway has index $D - k + 1$.

13. Create a new experiment `exp2` for UDnCNN based on Adam optimizer with learning rate `1e-3`, using `DenoisingStatsManager`, mini batches of size 4, and storing the checkpoints in `denoising2`. Run your experiment for 200 epochs (about 15 minutes) using the same function `plot` as before.

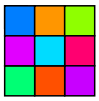
14. What is the number of parameters of UDnCNN(D)? What is the receptive field of UDnCNN(D)? Do you expect UDnCNN(D) to beat DnCNN(D)?

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

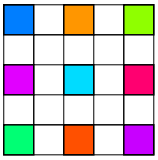
5 U-net like CNNs with dilated convolutions

Though pooling layers increase the receptive field, they lose information about exact locations. This is desired for classification, but for denoising this decreases performance. An alternative to pooling is to use

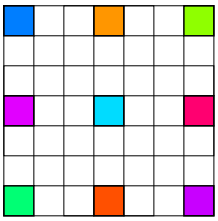
dilated convolutions (sometimes refer to the *à trous* algorithm, meaning *with holes*). Instead of increasing the receptive field by reducing the feature spatial dimensions by a factor 2 after each convolution, the filters are dilated by a factor 2. In order to maintain the same number of parameters, the dimensions are increased by injecting "holes" between each rows and columns of the filter. Please, refer to the following figure:



A standard 3×3 convolution kernel (`dilation=1`).



A 3×3 dilated kernel with `dilation=2`. In fact, a 5×5 kernel with holes of size 1.



A 3×3 dilated kernel with `dilation=3`. In fact, a 7×7 kernel with holes of size 2.

Next, instead of using unpooling to recover the original spatial dimensions, the filter dimensions are decreased by reducing the number of holes in the filters.

16. Copy paste the class `DnCNN` into a new class `DUDnCNN` and modify it to implement the Dilated U-shaped DnCNN. Each convolution placed after k pooling and l unpooling in the network, should be replaced by a dilated filter with $2^{k-l} - 1$ holes. This can be achieved with the `dilation` optional argument of `nn.Conv2d`. Make sure you set up the argument `padding` accordingly in order to maintain tensors with the same spatial dimension during the forward propagation.

17. In theory, dilated convolutions should not be slower than standard convolutions, but for some reasons they are when using the default PyTorch backend implementation. For this reason, add the two instructions `torch.backends.cudnn.benchmark=True` and `torch.backends.cudnn.benchmark=False`, before and after running any dilated convolutions. For more details, see the discussion here: https://github.com/pytorch/pytorch/issues/15054.

18. Create a new experiment for DUDnCNN based on Adam optimizer with learning rate `1e-3`, using `DenoisingStatsManager`, mini batches of size 4, and storing the checkpoints in `denoising3`. Run your experiment for 200 epochs (about 20 minutes) using the same function `plot` as before.

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.

20. What is the number of parameters of DUDnCNN(D)? What is the receptive field of DUDnCNN(D)? Conclude.

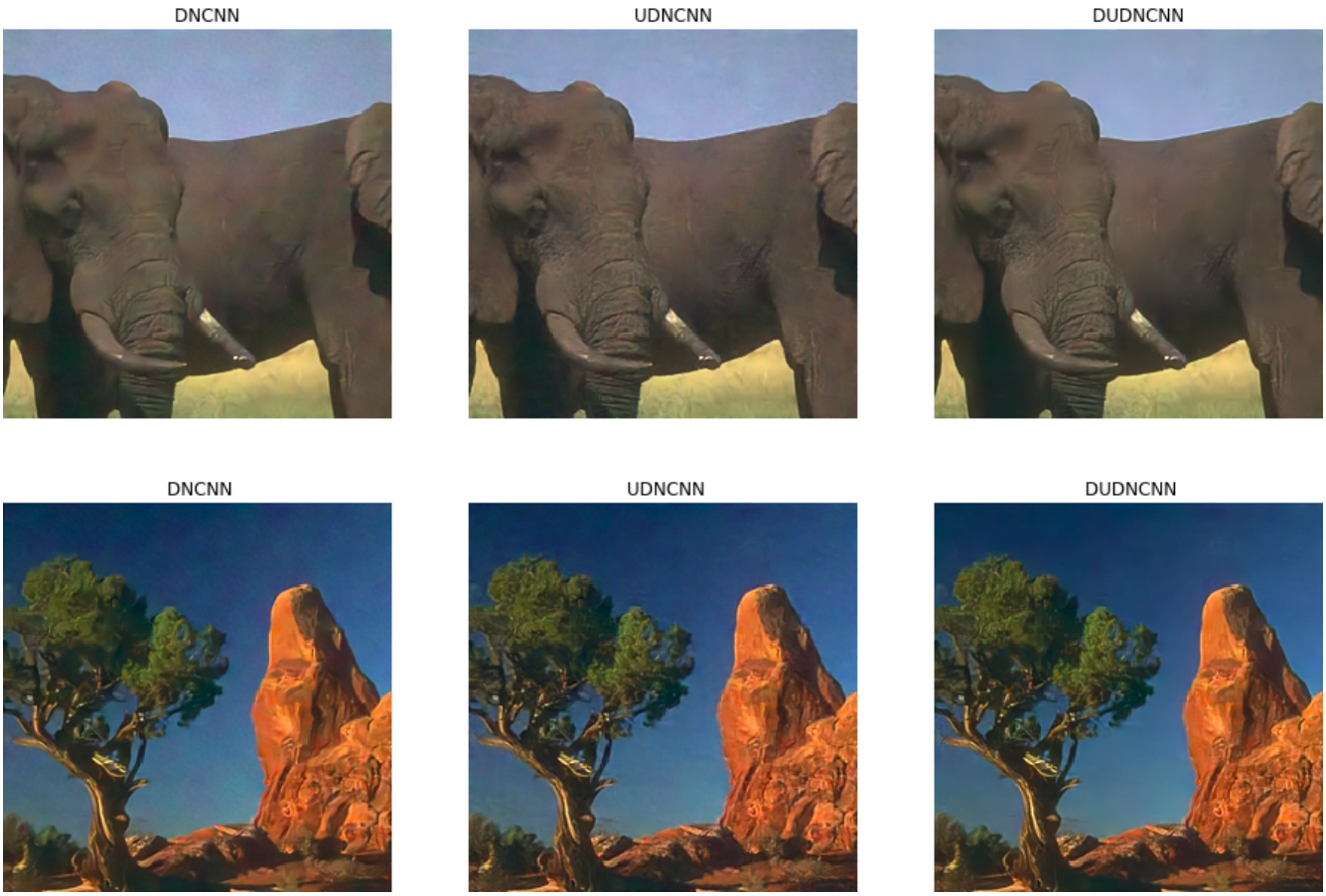


Figure 1: Comparative results between DnCNN, UDnCNN and DUDnCNN. All use 8 convolution layers and have the same number of learnable parameters.