Denoising with a learned DNN prior

We consider the classical image or data denoising problem, where the goal is to remove zero-mean white Gaussian noise from a given image or data point. In more detail, our goal is to obtain an estimate of a vector $y_0 \in \mathbb{R}^n$ from the noisy observation

$$y = y_0 + \eta,$$

where η is zero-mean Gaussian noise with covariance matrix σ^2/nI, and y_0 lies in the range of the generator, i.e., $y_0 = G(x_0)$.

We consider the following two-step denoising algorithm:

1. Obtaine an estimate $\hat{x}$ of the latent representation by minimizing the empirical loss
$$f(x) = \|G(x) - y_0\|_2^2$$
using gradient descent.
2. Obtain an estimate of the image as $\hat{y} = G(\hat{x})$.

We learn G by training an encoder-decoder network (i.e., an autoencoder) and taking G as the decoder.

In [1]:
```
import torch
import torch.nn as nn
import torch.utils as utils
from torch.autograd import Variable
import torchvision.datasets as dset
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
%matplotlib inline
import torch.nn.functional as F

import random
import numpy as np
import collections
```

Get data

In [2]:
```
mnist_train = dset.MNIST("./", train=True, transform=transforms.ToTensor(), target_transform=None, download=True)
mnist_test  = dset.MNIST("./", train=False, transform=transforms.ToTensor(), target_transform=None, download=True)

batch_size = 1
train_set = [ex for ex in torch.utils.data.DataLoader(dataset=mnist_train,batch_size=batch_size,shuffle=True)]
test_set = [ex for ex in torch.utils.data.DataLoader(dataset=mnist_test,batch_size=batch_size,shuffle=True)]

# construct training and test set only consisting of twos
def extract_nu(dset,nu):
    eset = []
    for image,label in train_set:
        if label.numpy() == nu:
            eset.append((image,label))
    return eset
train_twos = extract_nu(train_set,2)
test_twos = extract_nu(test_set,2)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./MNIST/raw/train-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 503: Service Unavailable

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to ./MNIST/raw/train-images-idx3-ubyte.gz


Extracting ./MNIST/raw/train-images-idx3-ubyte.gz to ./MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 503: Service Unavailable

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz to ./MNIST/raw/train-labels-idx1-ubyte.gz


Extracting ./MNIST/raw/train-labels-idx1-ubyte.gz to ./MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 503: Service Unavailable

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz to ./MNIST/raw/t10k-images-idx3-ubyte.gz


Extracting ./MNIST/raw/t10k-images-idx3-ubyte.gz to ./MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 503: Service Unavailable

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz to ./MNIST/raw/t10k-labels-idx1-ubyte.gz


Extracting ./MNIST/raw/t10k-labels-idx1-ubyte.gz to ./MNIST/raw

Processing...
Done!

/Users/murong/.conda/envs/pythonProject/lib/python3.8/site-packages/torchvision/datasets/mnist.py:502: UserWarning: The given NumP
y array is not writeable, and PyTorch does not support non-writeable tensors. This means you can write to the underlying (supposed
ly non-writeable) NumPy array using the tensor. You may want to copy the array to protect its data or make it writeable before con
verting it to a tensor. This type of warning will be suppressed for the rest of this program. (Triggered internally at  /Users/dis
tiller/project/conda/conda-bld/pytorch_1616554845587/work/torch/csrc/utils/tensor_numpy.cpp:143.)
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
```

Specification of the autoencoder

Below, specify the encoder and decoder parameterized by lengthbottleneck:

- Encoder: Two-layer linear neural network consisting of linear layer + relu + linear layer + relu. The first linear layer takes the 784 input neurons, maps it to 400 neurons, and the second layer maps the 400 neurons to lengthbottleneck neurons.
- Decoder: linear layer + relu + linear layer + relu; lengthbottleneck neurons -> 400 neurons -> 784 neurons

In [3]:
```
class Encoder(nn.Module):
    def __init__(self,lenbottleneck = 20):
        super(Encoder,self).__init__()
        self.fc1 = nn.Linear(28*28, 20*20)
        self.fc2 = nn.Linear(20*20,lenbottleneck)
    def forward(self,x):
        x = x.view(-1, 28*28)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return x

class Decoder(nn.Module):
    def __init__(self,lenbottleneck = 20):
        super(Decoder,self).__init__()
        self.fc1 = nn.Linear(lenbottleneck,20*20)
        self.fc2 = nn.Linear(20*20,28*28)
    def forward(self,x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = x.view(28,28)
        return x
```

In [4]:
```
def train(encoder,decoder,trainset,learning_rate = 0.001,epoch = 10):
    parameters = list(encoder.parameters()) + list(decoder.parameters())
    loss_func = nn.MSELoss() # mean square loss after the decoder
    print("learning_rate: ",learning_rate)
    optimizer = torch.optim.SGD(parameters, lr=learning_rate)
    ctr = 0
    for i in range(epoch):
        print("at epoch ", i + 1 , "/", epoch)
        for (image,label) in trainset:
            ctr += 1
            image = Variable(image)
            optimizer.zero_grad()
            output = encoder(image)
            output = decoder(output)
            loss = loss_func(output,image)
            loss.backward()
            optimizer.step()
    print("trained in ",ctr," iterations")
    return encoder, decoder
```

Training autoencoders

In [5]:
```
# train autoencoder for all digits
K = 10
encoder10 = Encoder(K)
decoder10 = Decoder(K)
encoder,decoder = train(encoder10,decoder10,train_set[:10000],1.0)
```

```
learning_rate:  1.0
at epoch  1 / 10

/Users/murong/.conda/envs/pythonProject/lib/python3.8/site-packages/torch/nn/modules/loss.py:528: UserWarning: Using a target size
(torch.Size([1, 1, 28, 28])) that is different to the input size (torch.Size([28, 28])). This will likely lead to incorrect result
s due to broadcasting. Please ensure they have the same size.
  return F.mse_loss(input, target, reduction=self.reduction)

at epoch  2 / 10
at epoch  3 / 10
at epoch  4 / 10
at epoch  5 / 10
at epoch  6 / 10
at epoch  7 / 10
at epoch  8 / 10
at epoch  9 / 10
at epoch  10 / 10
trained in  100000  iterations
```

In [6]:
```
# train autoencoder for all digits
K = 20
encoder20 = Encoder(K)
decoder20 = Decoder(K)
encoder20,decoder20 = train(encoder20,decoder20,train_set[:10000],1.0)
```

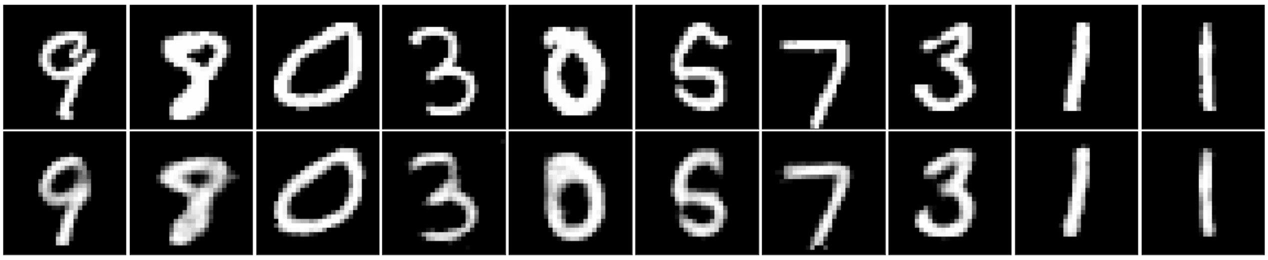
```
learning_rate:  1.0
at epoch  1 / 10
at epoch  2 / 10
at epoch  3 / 10
at epoch  4 / 10
at epoch  5 / 10
at epoch  6 / 10
at epoch  7 / 10
at epoch  8 / 10
at epoch  9 / 10
at epoch  10 / 10
trained in  100000  iterations
```

Check output of autoencoder: print a few input/output pairs

In [7]:
```
# given two lists of images as np-arrays, plot them as a row
def plot_images(top,bottom):
    fig, axes = plt.subplots(nrows=2, ncols=10, sharex=True, sharey=True, figsize=(20,4))
    for images, row in zip([top, bottom], axes):
        for img, ax in zip(images, row):
            ax.imshow(img, cmap='Greys_r')
            ax.get_xaxis().set_visible(False)
            ax.get_yaxis().set_visible(False)
    fig.tight_layout(pad=0.1)
    return fig
```

In [8]:
```
in_imgs = []
out_imgs = []
numprint = 10
for i,(img,label) in enumerate(test_set):
    if i >= numprint:
        break
    out_img = decoder20(encoder20(Variable(img)))
    in_imgs += [img[0][0].numpy()] # img is 1x1x28x28 tensor
    out_img.data.clamp_(0, 1)
    out_imgs += [out_img.data.numpy()]

fig = plot_images(in_imgs,out_imgs)
```



Task 1

Denoising with a trained decoder

Write the function for denoising below that denoises a noisy image given a generator learned earlier that maps a K-dimensional input space to an image.

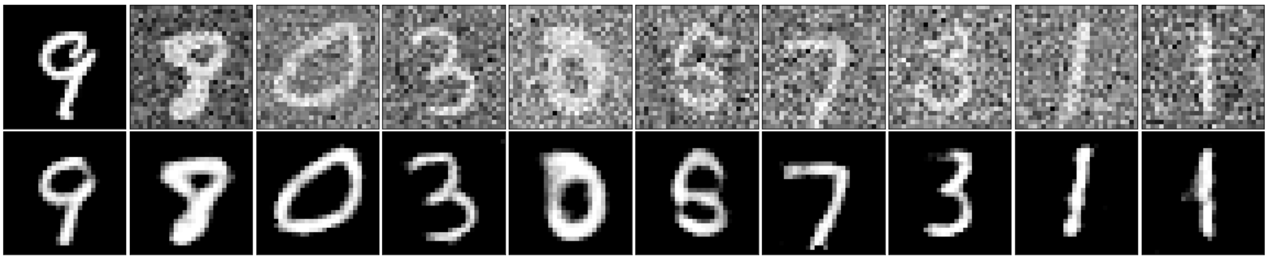
In [9]:
```
# denoise by recovering estimating a latent representation and passing that through the decoder
def denoise(net,noisy_image,K):
    # Step 1: Estimate the latent representation z
    z = torch.rand(K, requires_grad=True, device='cpu')
    learning_rate = 1.0
    loss_func = nn.MSELoss()
    optimizer = torch.optim.SGD([z], lr=learning_rate)
    net.training = False
    ite = 0
    diff = 1e4
    loss_prev = 1e4
    tol = 1e-6
    while diff > tol:
        ite += 1
        optimizer.zero_grad()
        decoded = net(z)
        loss = loss_func(decoded, noisy_image)
        loss.backward()
        optimizer.step()
        diff = abs(loss_prev - loss)
        loss_prev = loss
    print("trained in ", ite, " iterations to find the latent representation")
    # Step 2: Decode the latent estimation
    recovered_img = net(z)
    return recovered_img
```

Visualize denoising performance

In [12]:
```
K = 20
Sigmas = [0.3*i for i in range(10)] # noise variances
noisy_imgs = []
rec_imgs = []
for (img,label),sigma in zip(test_set[:len(Sigmas)],Sigmas):
    img = img[0][0]
    noise = np.sqrt(sigma)*torch.norm(img)/np.sqrt(28*28)*torch.randn(28, 28)
    noisy_img = img + noise
    rec_img = denoise(decoder20,Variable(noisy_img),K)
    rec_img.data.clamp_(0, 1)
    noisy_imgs += [noisy_img.numpy()]
    rec_imgs += [rec_img.data.numpy()]

# plot and save to file
fig = plot_images(noisy_imgs,rec_imgs)
fig.savefig("denoising_ex_0.1.png")
```

```
trained in  1645  iterations to find the latent representation
trained in  2095  iterations to find the latent representation
trained in  1164  iterations to find the latent representation
trained in  2431  iterations to find the latent representation
trained in  2182  iterations to find the latent representation
trained in  1055  iterations to find the latent representation
trained in  1430  iterations to find the latent representation
trained in  708  iterations to find the latent representation
trained in  986  iterations to find the latent representation
trained in  2324  iterations to find the latent representation
```



Task 2

Compare empirically which network (the one with K=10, or the one with K=20). Your choice how to do this comparison.

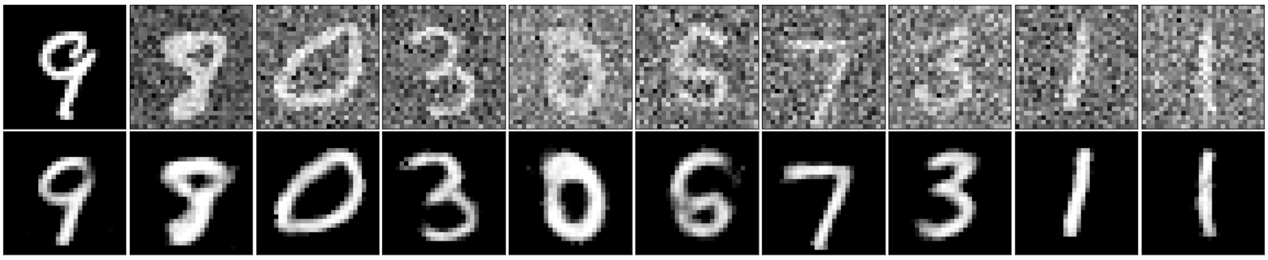
In [13]:
```
def check_denosing_performance(clean_imgs, rec_imgs):
    """
    Use average MSE of reconstructed images as the performance criterion.
    """
    mse = []
    for i in range(len(clean_imgs)):
        mse.append(np.square(np.subtract(clean_imgs[i], rec_imgs[i])).mean())
    return np.mean(mse)
```

In [18]:
```
K_10 = 10
K_20 = 20
Sigmas = [0.3 * i for i in range(10)]  # noise variances
noisy_imgs = []
clean_imgs = []
rec_imgs_10 = []
rec_imgs_20 = []
for (img, label), sigma in zip(test_set[:len(Sigmas)], Sigmas):
    img = img[0][0]
    noise = np.sqrt(sigma) * torch.norm(img) / np.sqrt(28 * 28) * torch.randn(28, 28)
    noisy_img = img + noise
    rec_img_10 = denoise(decoder10, Variable(noisy_img), K_10)
    rec_img_20 = denoise(decoder20, Variable(noisy_img), K_20)
    rec_img_10.data.clamp_(0, 1)
    rec_img_20.data.clamp_(0, 1)
    noisy_imgs += [noisy_img.numpy()]
    clean_imgs += [img.numpy()]
    rec_imgs_10 += [rec_img_10.data.numpy()]
    rec_imgs_20 += [rec_img_20.data.numpy()]
```

```
trained in  1227  iterations to find the latent representation
trained in  1530  iterations to find the latent representation
trained in  1620  iterations to find the latent representation
trained in  1924  iterations to find the latent representation
trained in  696  iterations to find the latent representation
trained in  1244  iterations to find the latent representation
trained in  1545  iterations to find the latent representation
trained in  2441  iterations to find the latent representation
trained in  1816  iterations to find the latent representation
trained in  1695  iterations to find the latent representation
trained in  1429  iterations to find the latent representation
trained in  872  iterations to find the latent representation
trained in  1065  iterations to find the latent representation
trained in  1025  iterations to find the latent representation
trained in  1948  iterations to find the latent representation
trained in  1273  iterations to find the latent representation
trained in  1139  iterations to find the latent representation
trained in  2014  iterations to find the latent representation
trained in  1709  iterations to find the latent representation
trained in  1290  iterations to find the latent representation
```

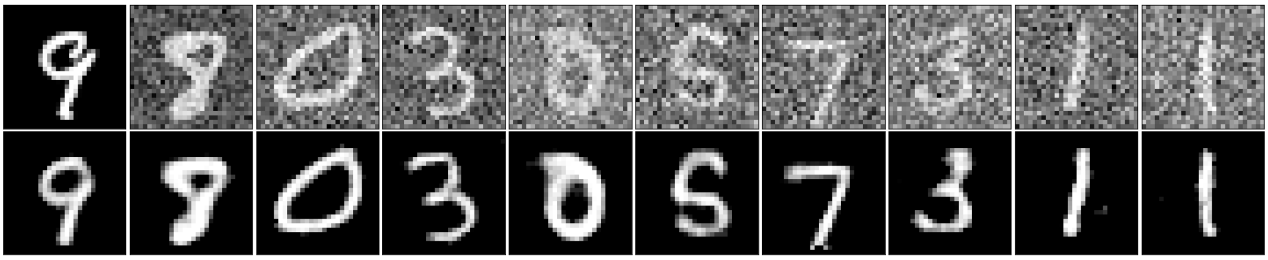
In [19]:
```
# plot and save to file
print('Reconstruction performance of using network_10: ')
fig_10 = plot_images(noisy_imgs, rec_imgs_10)
fig_10.savefig("decoder10.png")
```

```
Reconstruction performance of using network_10:
```



In [20]:
```
print('Reconstruction performance of using network_20: ')
fig_20 = plot_images(noisy_imgs, rec_imgs_20)
fig_20.savefig("decoder20.png")
```

```
Reconstruction performance of using network_20:
```



In [21]:
```
mse_10 = check_denoising_performance(clean_imgs, rec_imgs_10)
mse_20 = check_denoising_performance(clean_imgs, rec_imgs_20)
print('The average MSE of network_10 is ', mse_10)
print('The average MSE of network_20 is ', mse_20)
```

```
The average MSE of network_10 is  0.015664935
The average MSE of network_20 is  0.012662632
```

Because the average reconstruction MSE of network_20 is smaller than network_10, using the latent dimension K=20 is better for this denoising task.