1 Denoising Autoencoder

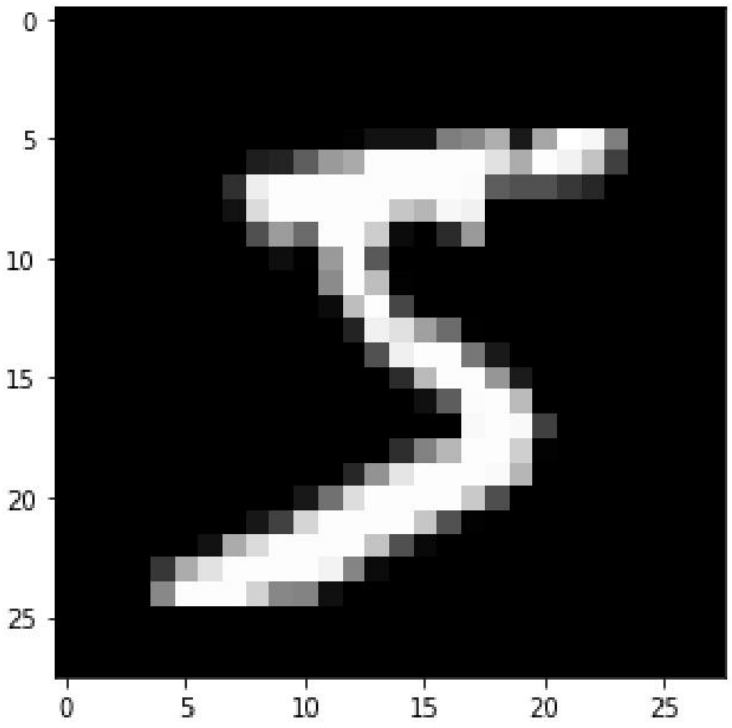
In [2]:
```
import torch
import numpy as np
from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
# convert data to torch.FloatTensor
transform = transforms.ToTensor()

# load the training and test datasets
train_data = datasets.MNIST(root='data', train=True,
                                    download=True, transform=transform)
test_data = datasets.MNIST(root='data', train=False,
                                   download=True, transform=transform)

# Create training and test dataloaders
num_workers = 0
# how many samples per batch to load
batch_size = 20

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, num_worker
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, num_workers=
```

1.0.1 Visualize the Data

In [3]:
```
import matplotlib.pyplot as plt
%matplotlib inline

# obtain one batch of training images
dataiter = iter(train_loader)
images, labels = dataiter.next()
images = images.numpy()

# get one image from the batch
img = np.squeeze(images[0])
```

```
        fig = plt.figure(figsize = (5,5))
        ax = fig.add_subplot(111)
        ax.imshow(img, cmap='gray')
```

```
Out[3]: <matplotlib.image.AxesImage at 0x7f3e0bf4d5c0>
```



---

2 Denoising

Structure Downsampling: 3 convolutional layers with a maxpooling layer after each Upsampling: 3 convolutional transpose layers.

```
In [4]: import torch.nn as nn
        import torch.nn.functional as F

        # define the NN architecture
        class ConvDenoiser(nn.Module):
            def __init__(self):
                super(ConvDenoiser, self).__init__()
```

```
            ## encoder layers ##
            self.conv1 = nn.Conv2d(1,8,3, padding =1)
            self.conv2 = nn.Conv2d(8,16,3, padding =1)
            self.conv3 = nn.Conv2d(16,4,3, padding =1)
            self.maxpool = nn.MaxPool2d(2,2)
            self.maxpool2 = nn.MaxPool2d(3,1,padding=1)
            ## decoder layers ##
            ## a kernel of 2 and a stride of 2 will increase the spatial dims by 2
            self.t_conv = nn.ConvTranspose2d(4, 32, 2, stride=2)
            self.t_conv2 = nn.ConvTranspose2d(32, 16, 2, stride=2)
            self.t_conv3 = nn.ConvTranspose2d(16, 4, 3, stride=1, padding=1)
            self.t_conv4 = nn.ConvTranspose2d(4, 1, 3, stride=1, padding=1)

        def forward(self, x):
            ## encode ##
            x = F.relu(self.conv1(x))
            x = self.maxpool(x)
            x = F.relu(self.conv2(x))
            x = self.maxpool(x)
            x = F.relu(self.conv3(x))
            x = self.maxpool2(x)

            ## decode ##
            x = F.relu(self.t_conv(x))
            x = F.relu(self.t_conv2(x))
            x = F.relu(self.t_conv3(x))
            x = F.sigmoid(self.t_conv4(x))
            return x

    # initialize the NN
    model = ConvDenoiser()
    print(model)
```

```
ConvDenoiser(
  (conv1): Conv2d(1, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(16, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (maxpool2): MaxPool2d(kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=False)
  (t_conv): ConvTranspose2d(4, 32, kernel_size=(2, 2), stride=(2, 2))
  (t_conv2): ConvTranspose2d(32, 16, kernel_size=(2, 2), stride=(2, 2))
  (t_conv3): ConvTranspose2d(16, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (t_conv4): ConvTranspose2d(4, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
)
```

2.1 Training

Mean square error instead of cross entropy loss as we are comparing pixel values not probabilities. Random noise is added to the image and the network optimizes on the differences in pixel values between noisy and non noisy images

```
In [5]: # specify loss function
        criterion = nn.MSELoss()

        # specify loss function
        optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

In [6]: n_epochs = 30

        noise_factor=0.5

        for epoch in range(1, n_epochs+1):
            train_loss = 0.0


            for data in train_loader:
                # _ stands in for labels, here
                # no need to flatten images
                images, _ = data

                ## add random noise to the input images
                noisy_imgs = images + noise_factor * torch.randn(*images.shape)
                # Clip the images to be between 0 and 1
                noisy_imgs = np.clip(noisy_imgs, 0., 1.)

                optimizer.zero_grad()
                outputs = model(noisy_imgs)
                # calculate the loss
                # the "target" is still the original, not-noisy images
                loss = criterion(outputs, images)
                loss.backward()
                optimizer.step()
                # update running training loss
                train_loss += loss.item()*images.size(0)

            # print avg training statistics
            train_loss = train_loss/len(train_loader)
            print('Epoch: {} \tTraining Loss: {:.6f}'.format(
                epoch,
                train_loss
                ))

Epoch: 1         Training Loss: 1.301858
Epoch: 2         Training Loss: 0.811017
```

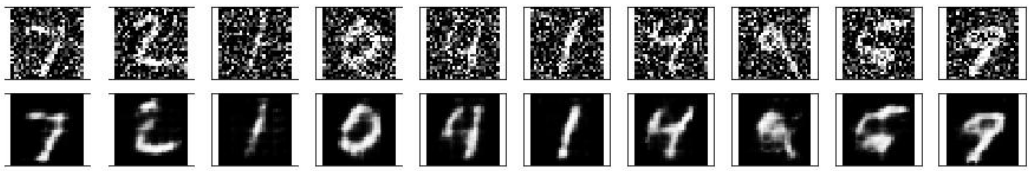
```
Epoch: 3         Training Loss: 0.677806
Epoch: 4         Training Loss: 0.620712
Epoch: 5         Training Loss: 0.595749
Epoch: 6         Training Loss: 0.579604
Epoch: 7         Training Loss: 0.567612
Epoch: 8         Training Loss: 0.559670
Epoch: 9         Training Loss: 0.551749
Epoch: 10         Training Loss: 0.547900
Epoch: 11         Training Loss: 0.542482
Epoch: 12         Training Loss: 0.539663
Epoch: 13         Training Loss: 0.536300
Epoch: 14         Training Loss: 0.533579
Epoch: 15         Training Loss: 0.529686
Epoch: 16         Training Loss: 0.527560
Epoch: 17         Training Loss: 0.525995
Epoch: 18         Training Loss: 0.523648
Epoch: 19         Training Loss: 0.522672
Epoch: 20         Training Loss: 0.520861
Epoch: 21         Training Loss: 0.518459
Epoch: 22         Training Loss: 0.517344
Epoch: 23         Training Loss: 0.517253
Epoch: 24         Training Loss: 0.515363
Epoch: 25         Training Loss: 0.514180
Epoch: 26         Training Loss: 0.512922
Epoch: 27         Training Loss: 0.512262
Epoch: 28         Training Loss: 0.511432
Epoch: 29         Training Loss: 0.510891
Epoch: 30         Training Loss: 0.508956
```

2.2 Checking out the results

Noise is added to image in the test set. The model does a reasonably good job of removing noise from the

```
In [7]: # obtain one batch of test images
        dataiter = iter(test_loader)
        images, labels = dataiter.next()

        # add noise to the test images
        noisy_imgs = images + noise_factor * torch.randn(*images.shape)
        noisy_imgs = np.clip(noisy_imgs, 0., 1.)

        # get sample outputs
        output = model(noisy_imgs)
        # prep images for display
        noisy_imgs = noisy_imgs.numpy()
```

```
# output is resized into a batch of iages
output = output.view(batch_size, 1, 28, 28)
# use detach when it's an output that requires_grad
output = output.detach().numpy()

# plot the first ten input images and then reconstructed images
fig, axes = plt.subplots(nrows=2, ncols=10, sharex=True, sharey=True, figsize=(25,4))

# input images on top row, reconstructions on bottom
for noisy_imgs, row in zip([noisy_imgs, output], axes):
    for img, ax in zip(noisy_imgs, row):
        ax.imshow(np.squeeze(img), cmap='gray')
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
```



In [ ]:

In [ ]:

In [ ]:

In [ ]: