Denoising_Autoencoder_Exercise

May 15, 2020

1 Denoising Autoencoder

Sticking with the MNIST dataset, let's add noise to our data and see if we can define and train an autoencoder to *de*-noise the images.

Let's get started by importing our libraries and getting the dataset.

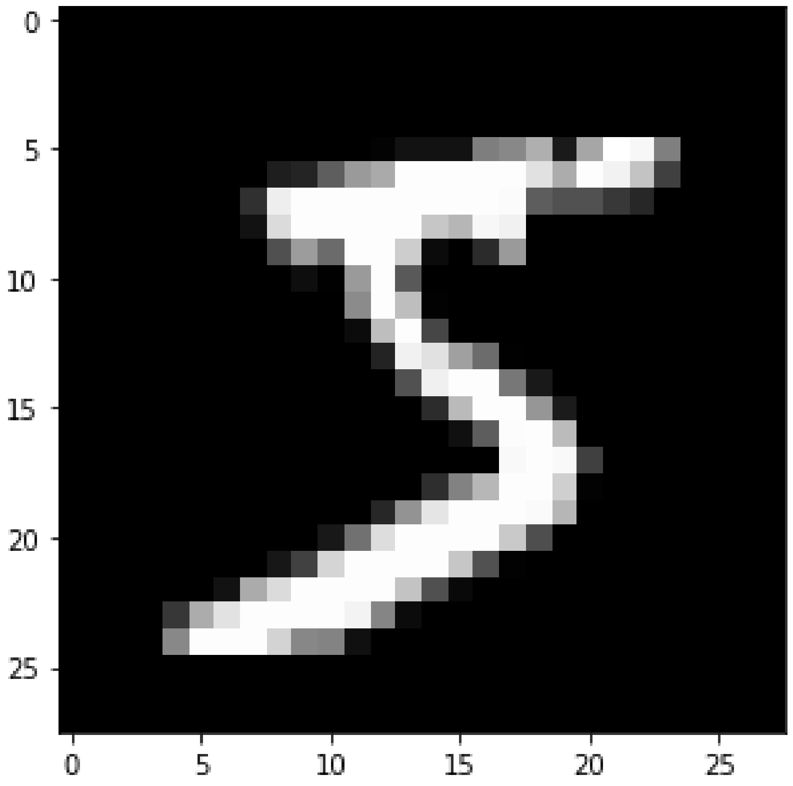
```
In [1]: import torch
        import numpy as np
        from torchvision import datasets
        import torchvision.transforms as transforms

        # 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 [2]: 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[2]: <matplotlib.image.AxesImage at 0x7f958d5bb6d8>
```



---

2 Denoising

As I've mentioned before, autoencoders like the ones you've built so far aren't too useful in practive. However, they can be used to denoise images quite successfully just by training the network on noisy images. We can create the noisy images ourselves by adding Gaussian noise to the training images, then clipping the values to be between 0 and 1.

We'll use noisy images as input and the original, clean images as targets.

Below is an example of some of the noisy images I generated and the associated, denoised images.

Since this is a harder problem for the network, we'll want to use *deeper* convolutional layers here; layers with more feature maps. You might also consider adding additional layers. I suggest starting with a depth of 32 for the convolutional layers in the encoder, and the same depths going backward through the decoder.

TODO: Build the network for the denoising autoencoder. Add deeper and/or additional layers compared to the model above.

```
In [3]: 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, 32, 3, padding=1)
                # conv layer (depth from 32 --> 16), 3x3 kernels
                self.conv2 = nn.Conv2d(32, 16, 3, padding=1)
                # conv layer (depth from 16 --> 8), 3x3 kernels
                self.conv3 = nn.Conv2d(16, 8, 3, padding=1)
                # pooling layer to reduce x-y dims by two; kernel and stride of 2
                self.pool = nn.MaxPool2d(2, 2)

                ## decoder layers ##
                # transpose layer, a kernel of 2 and a stride of 2 will increase the spatial dim
                self.t_conv1 = nn.ConvTranspose2d(8, 8, 3, stride=2)  # kernel_size=3 to get to
                # two more transpose layers with a kernel of 2
                self.t_conv2 = nn.ConvTranspose2d(8, 16, 2, stride=2)
                self.t_conv3 = nn.ConvTranspose2d(16, 32, 2, stride=2)
                # one, final, normal conv layer to decrease the depth
                self.conv_out = nn.Conv2d(32, 1, 3, padding=1)


            def forward(self, x):
                ## encode ##
                x = F.relu(self.conv1(x))
                x = self.pool(x)
                # add second hidden layer
                x = F.relu(self.conv2(x))
                x = self.pool(x)
                # add third hidden layer
                x = F.relu(self.conv3(x))
                x = self.pool(x)  # compressed representation

                ## decode ##
        # add transpose conv layers, with relu activation function
        x = F.relu(self.t_conv1(x))
        x = F.relu(self.t_conv2(x))
        x = F.relu(self.t_conv3(x))
        # transpose again, output should have a sigmoid applied
        x = F.sigmoid(self.conv_out(x))

        return x

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

```
ConvDenoiser(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(16, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (t_conv1): ConvTranspose2d(8, 8, kernel_size=(3, 3), stride=(2, 2))
  (t_conv2): ConvTranspose2d(8, 16, kernel_size=(2, 2), stride=(2, 2))
  (t_conv3): ConvTranspose2d(16, 32, kernel_size=(2, 2), stride=(2, 2))
  (conv_out): Conv2d(32, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
)
```

2.1 Training

We are only concerned with the training images, which we can get from the `train_loader`.

> In this case, we are actually **adding some noise** to these images and we'll feed these `noisy_imgs` to our model. The model will produce reconstructed images based on the noisy input. But, we want it to produce *normal* un-noisy images, and so, when we calculate the loss, we will still compare the reconstructed outputs to the original images!

Because we're comparing pixel values in input and output images, it will be best to use a loss that is meant for a regression task. Regression is all about comparing quantities rather than probabilistic values. So, in this case, I'll use `MSELoss`. And compare output images and input images as follows:

```
loss = criterion(outputs, images)
```

```
In [4]: # specify loss function
        criterion = nn.MSELoss()

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

```
In [5]: # number of epochs to train the model
        n_epochs = 15

        # for adding noise to images
        noise_factor=0.5

        for epoch in range(1, n_epochs+1):
            # monitor training loss
            train_loss = 0.0

            ###################
            # train the model #
            ###################
            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.)

                # clear the gradients of all optimized variables
                optimizer.zero_grad()
                ## forward pass: compute predicted outputs by passing *noisy* images to the mode
                outputs = model(noisy_imgs)
                # calculate the loss
                # the "target" is still the original, not-noisy images
                loss = criterion(outputs, images)
                # backward pass: compute gradient of the loss with respect to model parameters
                loss.backward()
                # perform a single optimization step (parameter update)
                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
                ))
```

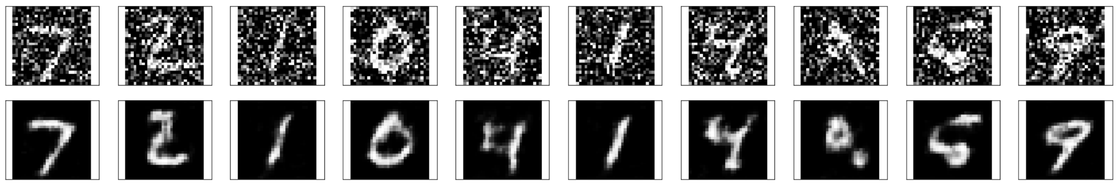
```
Epoch: 1         Training Loss: 0.972297
Epoch: 2         Training Loss: 0.693028
Epoch: 3         Training Loss: 0.641522
Epoch: 4         Training Loss: 0.604615
```

```
Epoch: 5         Training Loss: 0.578457
Epoch: 6         Training Loss: 0.559070
Epoch: 7         Training Loss: 0.543198
Epoch: 8         Training Loss: 0.530441
Epoch: 9         Training Loss: 0.521129
Epoch: 10         Training Loss: 0.513526
Epoch: 11         Training Loss: 0.506963
Epoch: 12         Training Loss: 0.502208
Epoch: 13         Training Loss: 0.498162
Epoch: 14         Training Loss: 0.493779
Epoch: 15         Training Loss: 0.491073
```

2.2 Checking out the results

Here I'm adding noise to the test images and passing them through the autoencoder. It does a suprising great job of removing the noise, even though it's sometimes difficult to tell what the original number is.

```
In [6]: # 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 [ ]:
```