MNIST-GAN

January 9, 2022

1 Another Simple GAN - MNIST Data set

In this notebook we are going to implement another simple GAN. We will use MNIST Dataset for this model and generate handwritten digits based on the dataset. the deep learning framework used for this task is PyTorch.

1.1 Importing and utility functions

[ ]:
```
import torch
from torch import nn
from tqdm.auto import tqdm
from torchvision import transforms
from torchvision.datasets import MNIST # Training dataset
from torchvision.utils import make_grid
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

def show_tensor_images(image_tensor, num_images=25, size=(1, 28, 28)):
    '''
    Function for visualizing images: Given a tensor of images, number of␣
↪images, and
    size per image, plots and prints the images in a uniform grid.
    '''
    image_unflat = image_tensor.detach().cpu().view(-1, *size)
    image_grid = make_grid(image_unflat[:num_images], nrow=5)
    plt.imshow(image_grid.permute(1, 2, 0).squeeze())
    plt.show()
```

1.2 Creating Generator

The generator class takes 3 inputs; 1. noise vector dimension, 2. image dimension, 3. inner dimension.

For this task we will use a 5-layered network. First 4 layers are similar, they will apply non-linear transformation until the output dimension is as similiar as the MNIST images (which is 28 * 28). The last layer of this network is different from the others. In this layer we don't need batch normlization or activation function, but it should be scaled. In the end we will implement forward function.

```
[ ]: class Generator(nn.Module):
         '''
             This is our Generator Class.
             Inputs ->
                 noise_dim = dimension of noise vector.
                 image_dim = dimension of images fitted for the dataset use,
                     mnist use 28 * 28 grayscale images, thus 28 * 28 * 1 = 784
                 hidden_dim = default value of inner dimension
         '''
         def __init__(self, noise_dim=10, image_dim=784, hidden_dim=128):
             super(Generator, self).__init__()
             self.block1 = nn.Sequential(
                 nn.Linear(noise_dim, hidden_dim),
                 nn.BatchNorm1d(hidden_dim),
                 nn.ReLU(inplace=True)
             )
             self.block2 = nn.Sequential(
                 nn.Linear(hidden_dim, hidden_dim * 2),
                 nn.BatchNorm1d(hidden_dim * 2),
                 nn.ReLU(inplace=True)
             )
             self.block3 = nn.Sequential(
                 nn.Linear(hidden_dim * 2, hidden_dim * 4),
                 nn.BatchNorm1d(hidden_dim * 4),
                 nn.ReLU(inplace=True)
             )
             self.block4 = nn.Sequential(
                 nn.Linear(hidden_dim * 4, hidden_dim * 8),
                 nn.BatchNorm1d(hidden_dim * 8),
                 nn.ReLU(inplace=True)
             )
             self.block5 = nn.Sequential(
                 nn.Linear(hidden_dim * 8, image_dim),
                 nn.Sigmoid()
             )

         def forward(self, noise):
             '''
                 Function for froward pass in our network.
                 takes noise as input and returns the image.
                 Input ->
                     noise = a noise tensor with dimension of (number_samples,␣
   ↪noise_dim)
             '''
             out = self.block1(noise)
             out = self.block2(out)
             out = self.block3(out)
```

```
        out = self.block4(out)
        out = self.block5(out)

        return out
```

1.3 Creating Noise

To be able to use generator, we need to be able to create noise vector. this noise vector plays an important role, with apropriate use of this concept we can make sure that our generated images are not from a class all the time. Since we are using batch approach, in each step we need to be able to generate multiple noise vectors, thus we will use a function to achieve this.

[ ]:
```
def get_noise(number_samples, noise_dim, device):
    '''
        Function for creating noise vector with given dimension
            (number_samples, noise_dim),
        Inputs ->
            number_samples = number of samples to geneare
            noise_dim = dimension of the noise vector
    '''
    return torch.randn(number_samples, noise_dim, device=device)
```

2 Creating Discriminator

In this section we are going to create the second component of the network, Discriminator. The Discriminator will take 2 inputs: 1. image dimension, 2. hidden dimension.

For this example we will use a 4-layered discriminator. It start with image tensor and transforms it until it returns a single number or in other words, 1-dimension tensor. this outputs simply classifies that whether the input image is original or generated, and finally in forward pass of this network, that takes an image tensor to be classified.

Have in mind that we don't need a sigmoid function after output layer since its already implemented in loss function.

[ ]:
```
class Discriminator(nn.Module):
    '''
        The Discriminator Class
        Inputs ->
            image_dim = dimension of images fitted for the dataset use,
                mnist use 28 * 28 grayscale images, thus 28 * 28 * 1 = 784
            hidden_dim = default value of inner dimension
    '''
    def __init__(self, image_dim=784, hidden_dim=128):
        super(Discriminator, self).__init__()
        self.block1 = nn.Sequential(
            nn.Linear(image_dim, hidden_dim * 4),
            nn.LeakyReLU(negative_slope=.2)
        )
        self.block2 = nn.Sequential(
            nn.Linear(hidden_dim * 4, hidden_dim * 2),
            nn.LeakyReLU(negative_slope=.2)
        )
        self.block3 = nn.Sequential(
            nn.Linear(hidden_dim * 2, hidden_dim),
            nn.LeakyReLU(negative_slope=.2)
        )
        self.block4 = nn.Linear(hidden_dim, 1)

    def forward(self, image):
        '''
            This function is for forward pass of the discriminator.
            Given an image tensor, it returs a 1-dimension tensor representing
            wheteher an image is original or generated.
            Inputs ->
                Flattened image tensor with dimension of (image_dim)
        '''
        out = self.block1(image)
        out = self.block2(out)
        out = self.block3(out)
        out = self.block4(out)
        return out
```

2.1 Training Phase

Before we start the training phase, we need to address some hyperparameters which is listed below:

- criterion: the loss function we want to use.
- n_epochs: the number of times we want to iterate through the entire dataset in training phase, epochs
- noise_dim: dimension of the noise vector
- display_step: in what frequency model show the generated images from trained generator
- batch_size: number of images for each batch (per forward/backward pass)
- lr: the learning rate
- device: the device type which we wants to perform training on

also in this step we will load the MNIST dataset as tensors using a data loader.

[ ]:
```
criterion = nn.BCEWithLogitsLoss()
n_epochs = 500
noise_dim = 64
display_step = 500
batch_size = 128
lr = .00001
device = 'cuda'
```

```
dataloader = DataLoader(
    MNIST('./data', download=True, transform=transforms.ToTensor()),
    batch_size=batch_size, shuffle=True)
```

Now, we can initialize! we need to initialize generator, discriminator and optimizers. Since each optimizer only takes the parameters of one particular model, we need 2 optimizers; one for generator and one for discriminator.

[ ]:
```
gen = Generator(noise_dim).to(device)
disc = Discriminator().to(device)
gen_opt = torch.optim.Adam(gen.parameters(), lr=lr)
disc_opt = torch.optim.Adam(disc.parameters(), lr=lr)
```

Before we train the GAN, we might need to calculate the discriminator's and generator's loss. this is the way how we, discriminator and generator will know how well they are performing and improve themselves.

IMPORTANT: generator is needed when want to calculate discriminator's loss, so we have to use .detach() on the generator result so we its parameters won't get updated in discriminator's backward pass.

[ ]:
```
def disc_loss(gen, disc, criterion, original, num_images, noise_dim, device):
    '''
        Return the loss of discriminator
        Inputs ->
            gen = generator model, this returns an image given noise
            disc = discriminator model, this returns prediction
            criterion = loss function, this is used for comparing␣
↪discriminator's
                prediction to the ground truth; generated = 0 and original = 1
            original = a batch of original images
            num_images = number of images generated should produce. also the␣
↪length
                of original images.
            noise_dim = dimension of noise vector
            device = the device type
        Returns ->
            disc_loss = loss value for current batch, torch scaler type
    '''
    noise_vec = get_noise(num_images, noise_dim, device)
    gen_image = gen(noise_vec).detach()
    gen_labels = torch.zeros(num_images, 1, device=device)
    gen_pred = disc(gen_image)
    gen_img_loss = criterion(gen_pred, gen_labels)
    org_labels = torch.ones(num_images, 1, device=device)
    org_pred = disc(original)
    org_img_loss = criterion(org_pred, org_labels)
    disc_loss = (org_img_loss + gen_img_loss) / 2
    return disc_loss
```

[ ]:
```
def gen_loss(gen, disc, criterion, num_images, noise_dim, device):
    '''
        Return the loss of generator
        Inputs ->
            gen = generator model, this returns an image given noise
            disc = discriminator model, this returns prediction
            criterion = loss function, this is used for comparing␣
↪discriminator's
                prediction to the ground truth; generated = 0 and original = 1
            num_images = number of images generated should produce. also the␣
↪length
                of original images.
            noise_dim = dimension of noise vector
            device = the device type
        Returns ->
            gen_loss = loss value for current batch, torch scaler type
    '''
    noise_vec = get_noise(num_images, noise_dim, device)
    gen_img = gen(noise_vec)
    gen_img_pred = disc(gen_img)
    gen_labels = torch.ones(num_images, 1, device=device)
    gen_loss = criterion(gen_img_pred, gen_labels)
    return gen_loss
```

2.2 Putting everything together

Finally we can put everything together! in each epoch, we will process the entire dataset in batches, and for each batch we need to update the discriminator and generator, with utilizing their loss. batches are sets of images that will be predicted on before the loss functions are calculated instead of calculating the loss function for each image. a loss greater than 1 is okay, since BCE can be any positive number for a sufficiently confident wrong guess.

Often in the beginning the discriminator will outperform the generator, because its job is easier. it's important that neither of them gets too good, since this might cause the entire model to stop learning. balancing the two models is actually very hard to do in a standatd GAN.

[ ]:
```
cur_step = 0
mean_generator_loss = 0
mean_discriminator_loss = 0
generator_loss = False
for epoch in range(n_epochs):

    # data loader return the batches
    for org, _ in tqdm(dataloader):
        cur_batch_size = len(org)
        # flattening batch of org images from dataset
        org = org.view(cur_batch_size, -1).to(device)

        ## updating discriminator ##
        # zeroing gradients before back prop
        disc_opt.zero_grad()
        discriminator_loss = disc_loss(gen, disc, criterion, org,␣
↪cur_batch_size, noise_dim, device)
        # update gradients
        discriminator_loss.backward(retain_graph=True)
        # update optimizer
        disc_opt.step()

        ## updating generator ##
        gen_opt.zero_grad()
        generator_loss = gen_loss(gen, disc, criterion, cur_batch_size,␣
↪noise_dim, device)
        generator_loss.backward(retain_graph=True)
        gen_opt.step()
        disc_opt.step()

        # calculating average discriminator and generator loss
        mean_discriminator_loss += discriminator_loss.item() / display_step
        mean_generator_loss += generator_loss.item() / display_step

        # visualization
        if cur_step % display_step == 0 and cur_step > 0:
            print(f'Epoch {epoch}, step {cur_step} -> generator loss:␣
↪{mean_generator_loss}, discriminator loss: {mean_discriminator_loss}')
            gen_noise = get_noise(cur_batch_size, noise_dim, device=device)
            generated = gen(gen_noise)
            show_tensor_images(generated)
            show_tensor_images(org)
            mean_generator_loss = 0
            mean_discriminator_loss = 0
        cur_step += 1
```

```
100%|     | 469/469 [00:16<00:00, 28.93it/s]
  6%|         | 30/469 [00:00<00:12, 34.29it/s]
```

```
Epoch 1, step 500 -> generator loss: 0.5948855748176572, discriminator loss:
0.5996641601324079
```

2.3 Saving the model

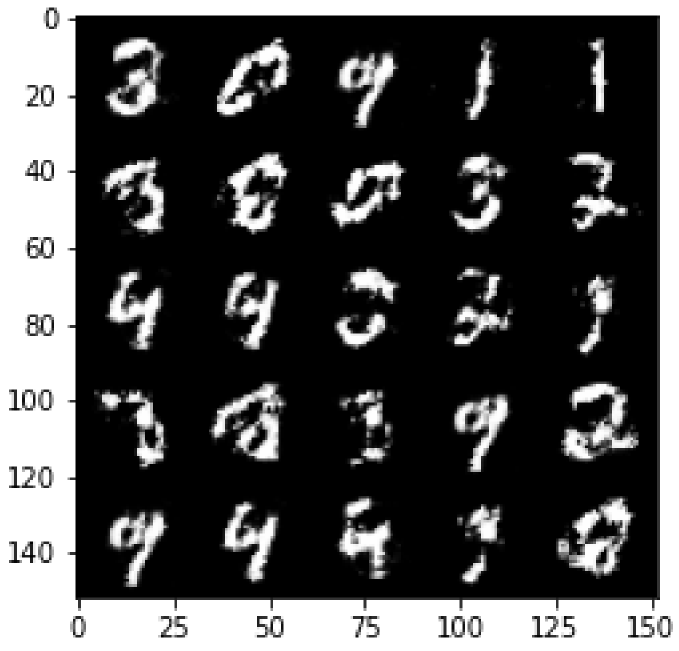
In this step we will save this trained model for further usage and avoid spending 160+ minutes for training it.

```
[ ]: torch.save(gen.state_dict(), 'models/MNIST-Gen-v1.0.pth')
     torch.save(disc.state_dict(), 'models/MNIST-Disc-v1.0.pth')
```

2.4 Checking the quality of GAN

Lets generated 25 samples of images based on MNIST dataset trained with a very simple and shallow GAN.

```
[ ]: test_noise = get_noise(25, 64, device='cuda')
     generated_test = gen(test_noise)
     show_tensor_images(generated_test)
```



---

3 Ramin F. - @SimplyRamin