hw3q4c-mnist-gan

November 22, 2023

MNIST GAN

[1]:
```
import torch
from torchvision import datasets
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import transforms
import numpy as np
import os
from matplotlib.pyplot import imsave
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

[2]:
```
# A transform to convert the images to tensor and normalize their RGB values
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize(mean=[0.5], std=[0.5])]
)

data = datasets.MNIST(root='../data/', train=True, transform=transform,␣
 ↪download=True) #Uses FashionMNIST dataset now

batch_size = 64
data_loader = DataLoader(dataset=data, batch_size=batch_size, shuffle=True,␣
 ↪drop_last=True)
```

```
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
../data/MNIST/raw/train-images-idx3-ubyte.gz

100%|      | 9912422/9912422 [00:00<00:00, 93556389.95it/s]

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


Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
../data/MNIST/raw/train-labels-idx1-ubyte.gz

100%|      | 28881/28881 [00:00<00:00, 87147981.17it/s]

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

```
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to
../data/MNIST/raw/t10k-images-idx3-ubyte.gz

100%|          | 1648877/1648877 [00:00<00:00, 45346539.27it/s]

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

```
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to
../data/MNIST/raw/t10k-labels-idx1-ubyte.gz

100%|          | 4542/4542 [00:00<00:00, 21214397.29it/s]

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

[3]:

```python
def to_onehot(x, num_classes=10):
    assert isinstance(x, int) or isinstance(x, (torch.LongTensor, torch.cuda.
 ↪LongTensor))
    if isinstance(x, int):
        c = torch.zeros(1, num_classes).long()
        c[0][x] = 1
    else:
        x = x.cpu()
        c = torch.LongTensor(x.size(0), num_classes)
        c.zero_()
        c.scatter_(1, x, 1) # dim, index, src value
    return c
```

[4]:

```python
to_onehot(3)
```

[4]:

```
tensor([[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]])
```

[5]:

```python
def get_sample_image(G, DEVICE, n_noise=100):
    img = np.zeros([280, 280])
    for j in range(10):
        c = torch.zeros([10, 10]).to(DEVICE)
        c[:, j] = 1
        z = torch.randn(10, n_noise).to(DEVICE)
        y_hat = G(z,c).view(10, 28, 28)
        result = y_hat.cpu().data.numpy()
        img[j*28:(j+1)*28] = np.concatenate([x for x in result], axis=-1)
    return img
```

0.1 Architecture

We now instantiate the generator and discriminator architectures. The generator takes a random noise vector and a one hot encoded label as input and produces an image. The discriminator takes an image and a one hot encoded label as input and produces a single value between 0 and 1. The discriminator is trained to output 1 for real images and 0 for fake images. The generator is trained to fool the discriminator by outputting images that look real.

I edited the number of classes from being included in the models, this will make the model a GAN instead of a Conditional GAN.

[6]:
```
class Generator(nn.Module):
    def __init__(self, input_size=100, num_classes=10, image_size=28*28):
        super(Generator, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(input_size, 128), # auxillary dimension for label; Got␣
↪rid of +num_classes
            nn.LeakyReLU(0.2),
            nn.Linear(128, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1024),
            nn.BatchNorm1d(1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, image_size),
            nn.Tanh()
        )

    def forward(self, x, c):
        x, c = x.view(x.size(0), -1), c.view(c.size(0), -1).float()
        v = x # v: [input, label] concatenated vector; Got rid of concatenation
        y_ = self.network(v)
        y_ = y_.view(x.size(0), 1, 28, 28)
        return y_
```

[7]:
```
class Discriminator(nn.Module):
    def __init__(self, input_size=28*28, num_classes=10, num_output=1):
        super(Discriminator, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(input_size, 512), #Got rid of +num_classes
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, num_output),
            nn.Sigmoid(),
```

```
        )

    def forward(self, x, c):
        x, c = x.view(x.size(0), -1), c.view(c.size(0), -1).float()
        v = x # v: [input, label] concatenated vector; Got rid of concatenation
        y_ = self.network(v)
        return y_
```

[8]:
```
MODEL_NAME = 'GAN' #Changed the name
DEVICE = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

D = Discriminator().to(DEVICE) # randomly intialized
G = Generator().to(DEVICE) # randomly initialized

max_epoch = 10
step = 0
n_noise = 100 # size of noise vector

criterion = nn.BCELoss()
D_opt = torch.optim.Adam(D.parameters(), lr=0.0002, betas=(0.5, 0.999))
G_opt = torch.optim.Adam(G.parameters(), lr=0.0002, betas=(0.5, 0.999))

# We will denote real images as 1s and fake images as 0s
# This is why we needed to drop the last batch of the data loader
all_ones = torch.ones([batch_size, 1]).to(DEVICE) # Discriminator label: real
all_zeros = torch.zeros([batch_size, 1]).to(DEVICE) # Discriminator Label: fake
```

[9]:
```
images, class_labels = next(iter(data_loader))
class_labels_encoded = class_labels.view(batch_size, 1)
class_labels_encoded = to_onehot(class_labels_encoded).to(DEVICE)
print(class_labels[:10])
print(class_labels_encoded[:10])
```

```
tensor([0, 7, 3, 0, 7, 4, 2, 8, 2, 8])
tensor([[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
        [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]], device='cuda:0')
```

[10]:
```
# a directory to save the generated images
if not os.path.exists('samples'):
```

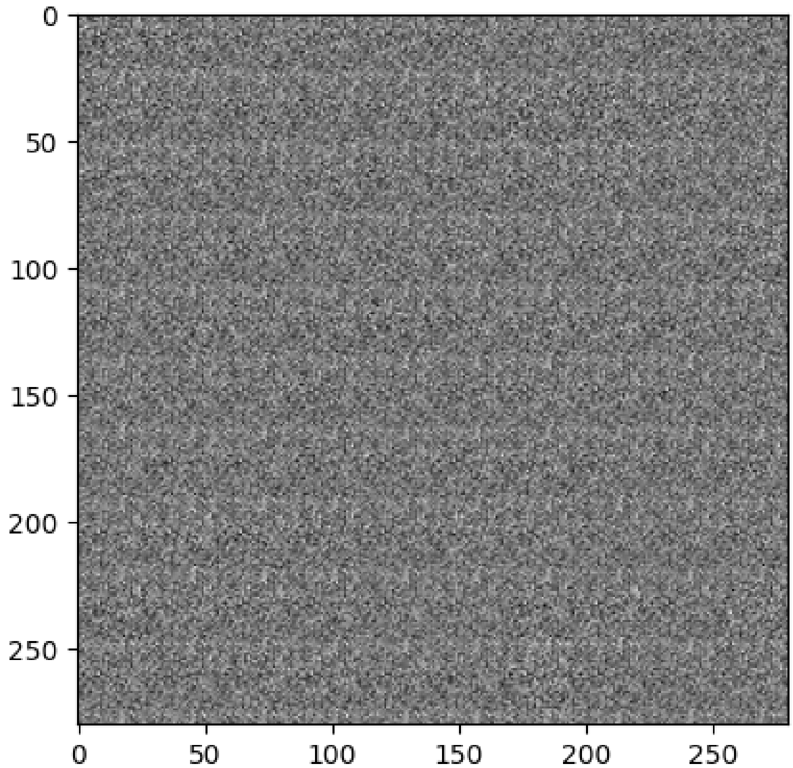
```
    os.makedirs('samples')

for epoch in range(max_epoch):
    for idx, (images, class_labels) in enumerate(data_loader):
        # Training Discriminator
        x = images.to(DEVICE)
        class_labels = class_labels.view(batch_size, 1) # add singleton␣
 ↪dimension so batch_size x 1
        class_labels = to_onehot(class_labels).to(DEVICE)
        x_outputs = D(x, class_labels) # input includes labels
        D_x_loss = criterion(x_outputs, all_ones) # Discriminator loss for real␣
 ↪images

        z = torch.randn(batch_size, n_noise).to(DEVICE)
        z_outputs = D(G(z, class_labels), class_labels) # input to both␣
 ↪generator and discriminator includes labels
        D_z_loss = criterion(z_outputs, all_zeros) # Discriminator loss for␣
 ↪fake images
        D_loss = D_x_loss + D_z_loss # Total Discriminator loss

        D.zero_grad()
        D_loss.backward()
        D_opt.step()

        # Training Generator
        z = torch.randn(batch_size, n_noise).to(DEVICE)
        z_outputs = D(G(z, class_labels), class_labels)
        G_loss = -1 * criterion(z_outputs, all_zeros) # Generator loss is␣
 ↪negative discriminator loss

        G.zero_grad()
        G_loss.backward()
        G_opt.step()

        if step % 500 == 0:
            print('Epoch: {}/{}, Step: {}, D Loss: {}, G Loss: {}'.
 ↪format(epoch, max_epoch, step, D_loss.item(), G_loss.item()))

        if step % 1000 == 0:
            G.eval()
            img = get_sample_image(G, DEVICE, n_noise)
            imsave('samples/{}_step{}.jpg'.format(MODEL_NAME, str(step).
 ↪zfill(3)), img, cmap='gray')
            G.train()
        step += 1
```

```
Epoch: 0/10, Step: 0, D Loss: 1.4321000576019287, G Loss: -0.6797271370887756
```

```
Epoch: 0/10, Step: 500, D Loss: 1.4028527736663818, G Loss: -0.6347811222076416
Epoch: 1/10, Step: 1000, D Loss: 1.1960117816925049, G Loss: -0.5185055136680603
Epoch: 1/10, Step: 1500, D Loss: 1.1932541131973267, G Loss:
-0.39768052101135254
Epoch: 2/10, Step: 2000, D Loss: 1.2234655618667603, G Loss: -0.2393479198217392
Epoch: 2/10, Step: 2500, D Loss: 1.078147053718567, G Loss: -0.6512055397033691
Epoch: 3/10, Step: 3000, D Loss: 1.2166550159454346, G Loss: -0.6759253740310669
Epoch: 3/10, Step: 3500, D Loss: 1.2387259006500244, G Loss: -0.6251146793365479
Epoch: 4/10, Step: 4000, D Loss: 1.2493551969528198, G Loss: -0.4827510714530945
Epoch: 4/10, Step: 4500, D Loss: 1.1359643936157227, G Loss: -0.5347049236297607
Epoch: 5/10, Step: 5000, D Loss: 1.3132450580596924, G Loss: -0.3203120827674866
Epoch: 5/10, Step: 5500, D Loss: 1.1416640281677246, G Loss: -0.522498369216919
Epoch: 6/10, Step: 6000, D Loss: 1.253251552581787, G Loss: -0.7048524618148804
Epoch: 6/10, Step: 6500, D Loss: 1.1745905876159668, G Loss: -0.5737121105194092
Epoch: 7/10, Step: 7000, D Loss: 1.3402128219604492, G Loss:
-0.46168065071105957
Epoch: 8/10, Step: 7500, D Loss: 1.2298719882965088, G Loss: -0.6179988384246826
Epoch: 8/10, Step: 8000, D Loss: 1.3035839796066284, G Loss:
-0.48102065920829773
Epoch: 9/10, Step: 8500, D Loss: 1.2523696422576904, G Loss: -0.7329338788986206
Epoch: 9/10, Step: 9000, D Loss: 1.4572618007659912, G Loss: -0.9165699481964111
```

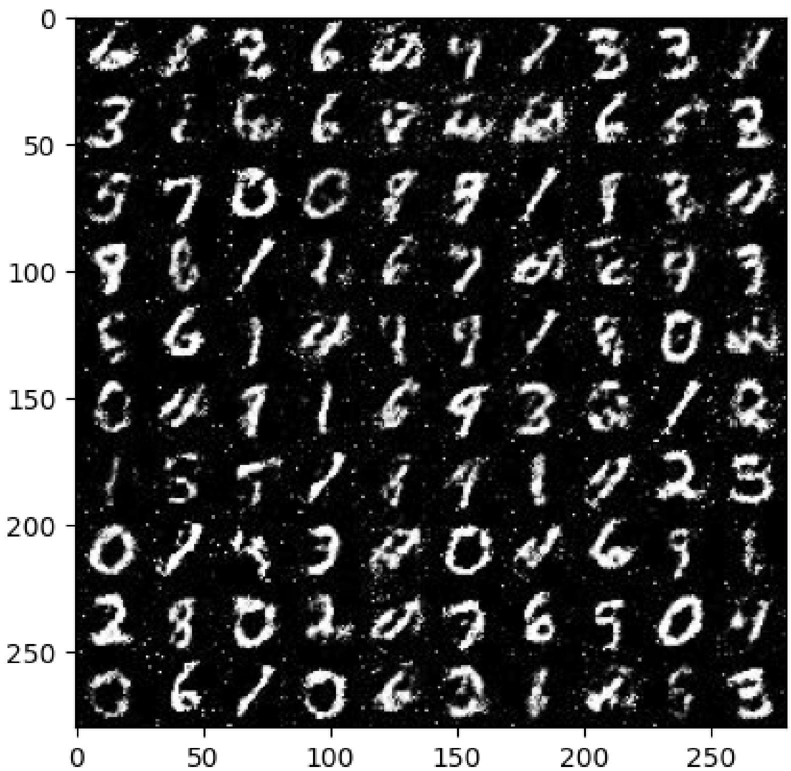
Now let's plot these images. At first, the generator just produces noise (as we expect).

[11]:
```python
img = mpimg.imread('samples/GAN_step000.jpg')
imgplot = plt.imshow(img)
plt.show()
```

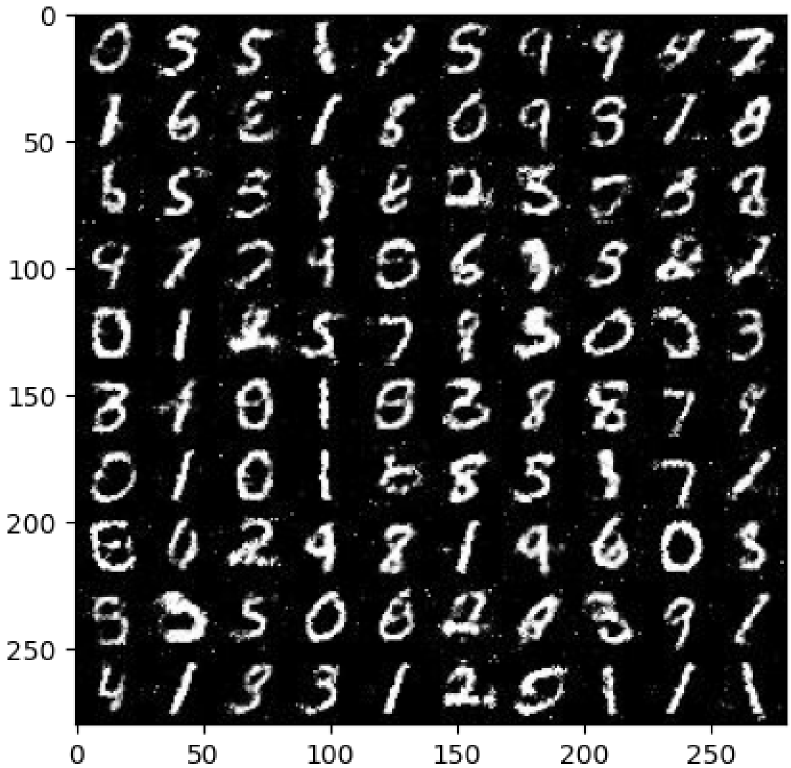


But then it gets better.

[12]:
```
img = mpimg.imread('samples/GAN_step5000.jpg')
imgplot = plt.imshow(img)
plt.show()
```



```
[13]: img = mpimg.imread('samples/GAN_step9000.jpg')
      imgplot = plt.imshow(img)
      plt.show()
```


0
50
100
150
200
250
0
50
100
150
200
250