شبکه‌های مولد تخاصمی

برای پیاده‌سازی شبکه‌های مولد تخاصمی از PyTorch استفاده می‌کنیم.

حال خط به خط فایل نوت‌بوک شرح داده می‌شود.

[32]:
```
import os
import torch
import torchvision
import torch.nn as nn
from torchvision.transforms import ToTensor, Normalize, Compose
from torchvision.datasets import MNIST
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader
from IPython.display import Image
from torchvision.utils import save_image
%matplotlib inline
```

Download MNIST

[9]:
```
mnist = MNIST(root='data',
              train=True,
              download=True,
              transform=Compose([ToTensor(), Normalize(mean=(0.5,), std=(0.5,))]))
```

این دو سلول برای ایمپورت کردن لایبرری‌های مهم و هم‌چنین دانلود دیتاست MNIST به کار می‌روند.

Image features

[14]:
```
img, label = mnist[0]
print('Label: ', label)
print(img[:,0:5,0:5])
torch.min(img), torch.max(img)
```

```
Label:  5
tensor([[[-1., -1., -1., -1., -1.],
         [-1., -1., -1., -1., -1.],
         [-1., -1., -1., -1., -1.],
         [-1., -1., -1., -1., -1.],
         [-1., -1., -1., -1., -1.]]])
```

[14]:
```
(tensor(-1.), tensor(1.))
```

[15]:
```
img.shape
```

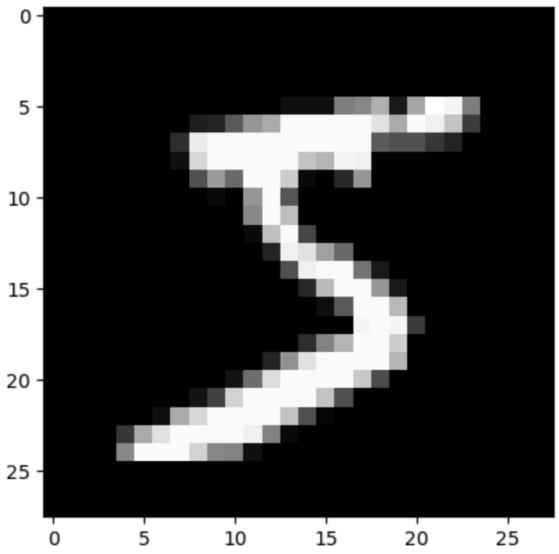
[15]:
```
torch.Size([1, 28, 28])
```

در مرحله‌ی بعد به بررسی یکی از داده‌های دیتاست MNIST می‌پردازیم، همانطور که مشاهده می‌کنید، لیبل اولین داده‌ی این دیتاست ۵ است و ماکسیمم و مینیمم عنصر این عکس ۱ و -۱ است و این بدین معنی است که مقدار یک پیکسلدر بازه‌ی -۱ تا ۱ قرار دارد. حال بدین ترتیب برای اینکه بتوانیم یک عکس را پلات کنیم باید آن را از بازه‌ی -۱ تا ۱ به بازه‌ی ۰ تا ۱ شیفت دهیم و برای اینکار تابع denorm را تعریف می‌کنیم:

Plot a digit

[17]:
```
denorm = lambda x: ((x + 1) / 2).clamp(0, 1)

img_norm = denorm(img)
plt.imshow(img_norm[0], cmap='gray')
print('Label:', label)
```

```
Label: 5
```



[34]:
```
batch_size = 100
data_loader = DataLoader(mnist, batch_size, shuffle=True)
```

حال پس از مشاهده‌ی یک داده یک دیتالودر می‌سازیم که داده‌ها را در بچ‌های ۱۰۰ تایی و به صورت شافل به ما می‌دهد.

Discriminator Network

[20]:
```
image_size = 784
hidden_size = 256

D = nn.Sequential(
    nn.Linear(image_size, hidden_size),
    nn.LeakyReLU(0.2),
    nn.Linear(hidden_size, hidden_size),
    nn.LeakyReLU(0.2),
    nn.Linear(hidden_size, 1),
    nn.Sigmoid())
```

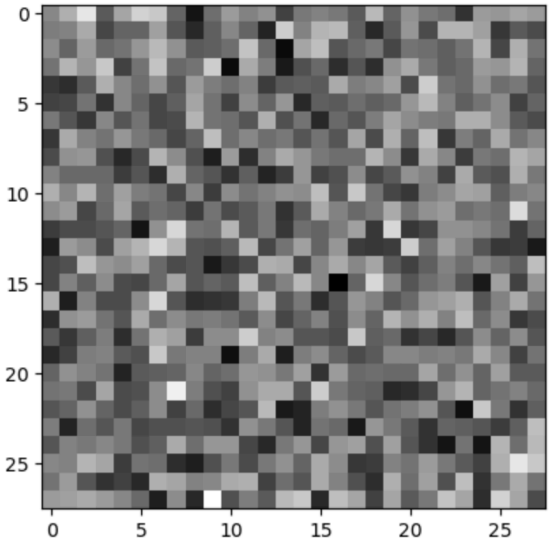
Generator Network

[21]:
```
latent_size = 64

G = nn.Sequential(
    nn.Linear(latent_size, hidden_size),
    nn.ReLU(),
    nn.Linear(hidden_size, hidden_size),
    nn.ReLU(),
    nn.Linear(hidden_size, image_size),
    nn.Tanh())
```

مرحله‌ی بعد طراحی معماری شبکه‌های Discriminator و Generator است. شبکه‌ی Discriminator همانطور که در ویدیو نیز اشاره شد همانند کارآگاه عمل می‌کند و هدف آن تشخیص عکس فیک از عکس واقعی است. در حقیقت ورودی آن یک عکس 28 در 28 یا 784 در 1 و خروجی آن نیز ۱ نود از ۰ تا ۱ است. هم‌چنین این بین نیز ۲ لایه مخفی ۲۵۶ نودی وجود دارد. شبکه‌ی جنریتور نیز یک سید ۶۴تایی دریافت کرده و خروجی آن 784 است و در این بین ۲ لایه‌ی مخفی ۲۵۶ تایی وجود دارد.

Now, let's randomly generate an image by our generator

[22]:
```
y = G(torch.randn(2, latent_size))
gen_imgs = denorm(y.reshape((-1, 28,28)).detach())
plt.imshow(gen_imgs[0], cmap='gray');
```



این یک تصویر خروجی از جنریتور قبل از آموزش است.

Define Discriminator Optimizer

[25]:
```
criterion = nn.BCELoss()
d_optimizer = torch.optim.Adam(D.parameters(), lr=0.0001)
```

[26]:
```
def reset_grad():
    d_optimizer.zero_grad()
    g_optimizer.zero_grad()

def train_discriminator(images):
    real_labels = torch.ones(batch_size, 1)
    fake_labels = torch.zeros(batch_size, 1)

    outputs = D(images)
    d_loss_real = criterion(outputs, real_labels)
    real_score = outputs

    z = torch.randn(batch_size, latent_size)
    fake_images = G(z)
    outputs = D(fake_images)
    d_loss_fake = criterion(outputs, fake_labels)
    fake_score = outputs

    d_loss = d_loss_real + d_loss_fake
    reset_grad()
    d_loss.backward()
    d_optimizer.step()

    return d_loss, real_score, fake_score
```

حال گام بعدی تعریف توابع مربوط به Optimize کردن Discriminator است. بدین ترتیب در تابع train_discriminator لیبل‌های واقعی با ۱ و لیبل‌های فیک با ۰ نمایش داده می‌شوند. سپس عکس‌ها به شبکه‌ی Discriminator داده می‌شوند و لاس آن‌ها با خروجی ۱ سنجیده می‌شود. حال جنریتور مقداری عکس تولید می‌کند و لاس آن‌ها در شبکه‌ی Discriminator با خروجی ۰ شنجیده می‌شود و در نهایت لاس‌ها با هم جمع زده و مینیمم می‌شوند.

Define Generator Optimizer

[27]:
```
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.0001)
```

[28]:
```
def train_generator():
    z = torch.randn(batch_size, latent_size)
    fake_images = G(z)
    labels = torch.ones(batch_size, 1)
    g_loss = criterion(D(fake_images), labels)

    reset_grad()
    g_loss.backward()
    g_optimizer.step()
    return g_loss, fake_images
```

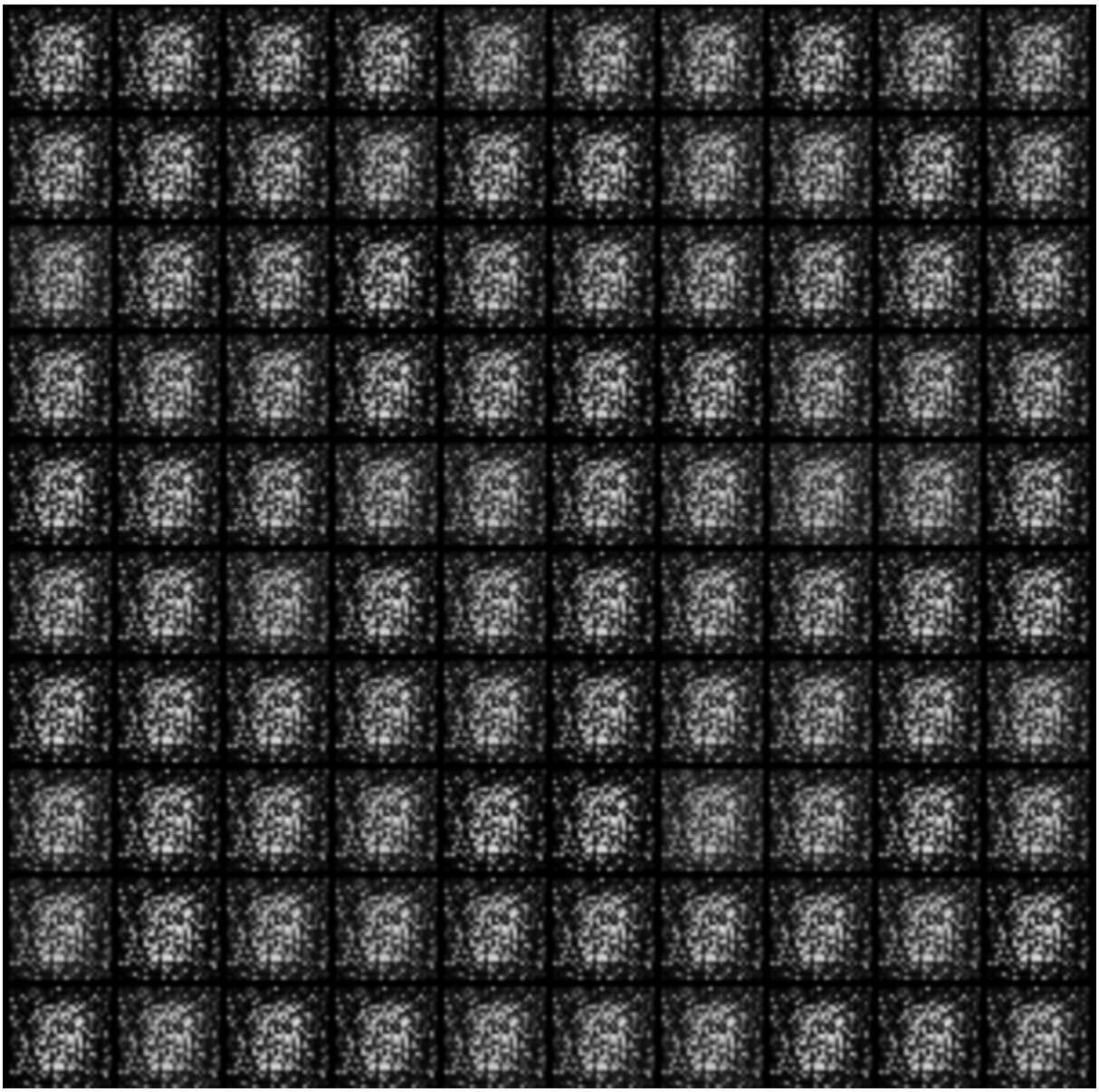
برای اپتیمایز کردن جنریتور نیز جنریتور مقداری عکس تولید کرده و لاس خروجی Discriminator با ۱ سنجیده و مینیمم می‌شود.

Training

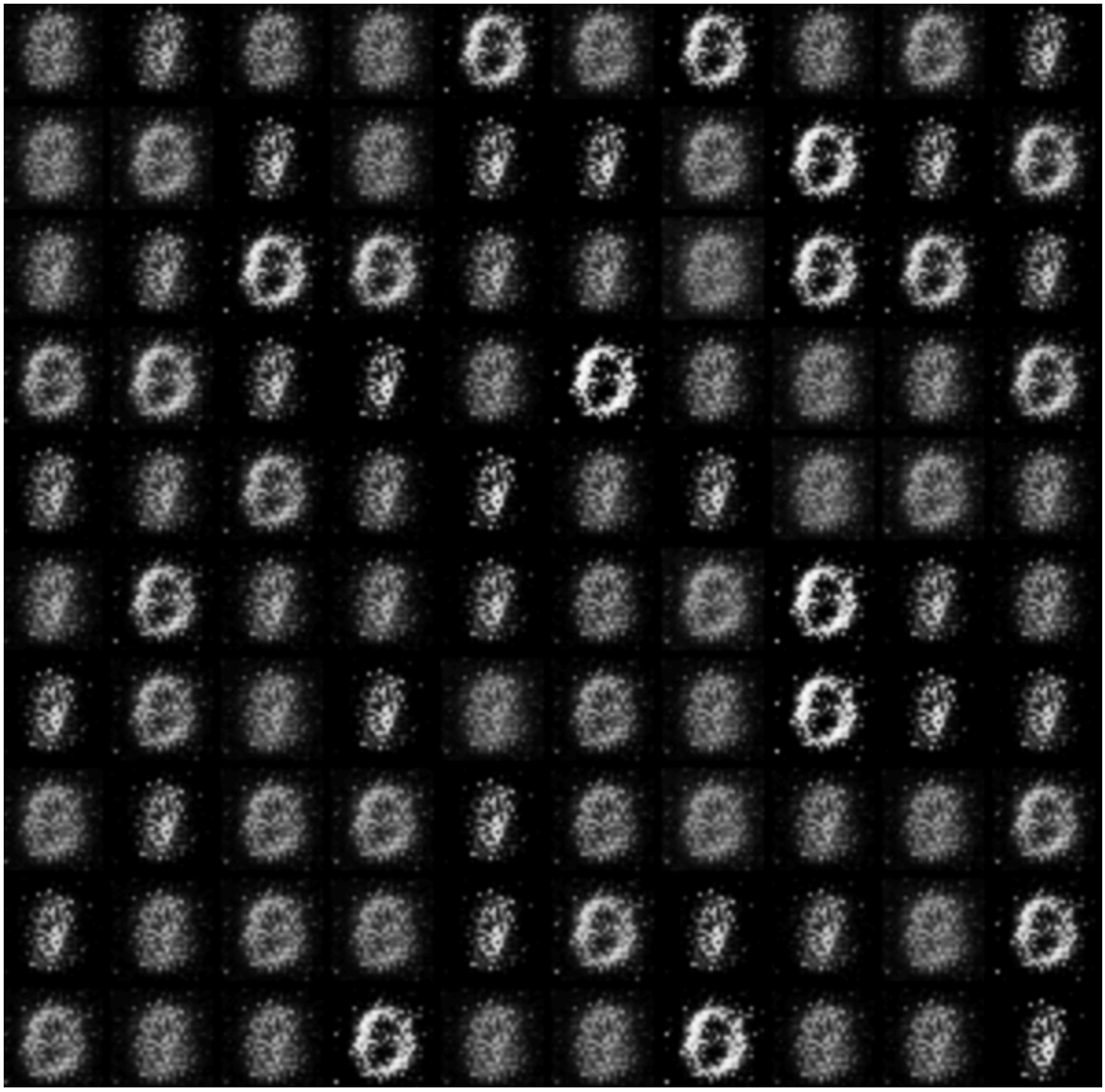
[*]:
```
%%time

num_epochs = 300
total_step = len(data_loader)
d_losses, g_losses, real_scores, fake_scores = [], [], [], []

for epoch in range(num_epochs):
    for i, (images, _) in enumerate(data_loader):
        images = images.reshape(batch_size, -1)

        d_loss, real_score, fake_score = train_discriminator(images)
        g_loss, fake_images = train_generator()

        if (i+1) % 200 == 0:
            d_losses.append(d_loss.item())
            g_losses.append(g_loss.item())
            real_scores.append(real_score.mean().item())
            fake_scores.append(fake_score.mean().item())
            print('Epoch [{}/{}], Step [{}/{}], d_loss: {:.4f}, g_loss: {:.4f}, D(x): {:.2f}, D(G(z)): {:.2f
                  .format(epoch, num_epochs, i+1, total_step, d_loss.item(), g_loss.item(),
                          real_score.mean().item(), fake_score.mean().item()))

    save_fake_images(epoch+1)
```

حال به ازای ۳۰۰ ایپاک فرآیند آموزش را شروع می‌کنیم و به ازای هر ۲۰۰ عکس مقادیر لاس و D(G(z)) را گزارش می‌کنیم. تصاویر زیر آموزش شبکه را بعد از تعدادی ایپاک نمایش می‌دهند:

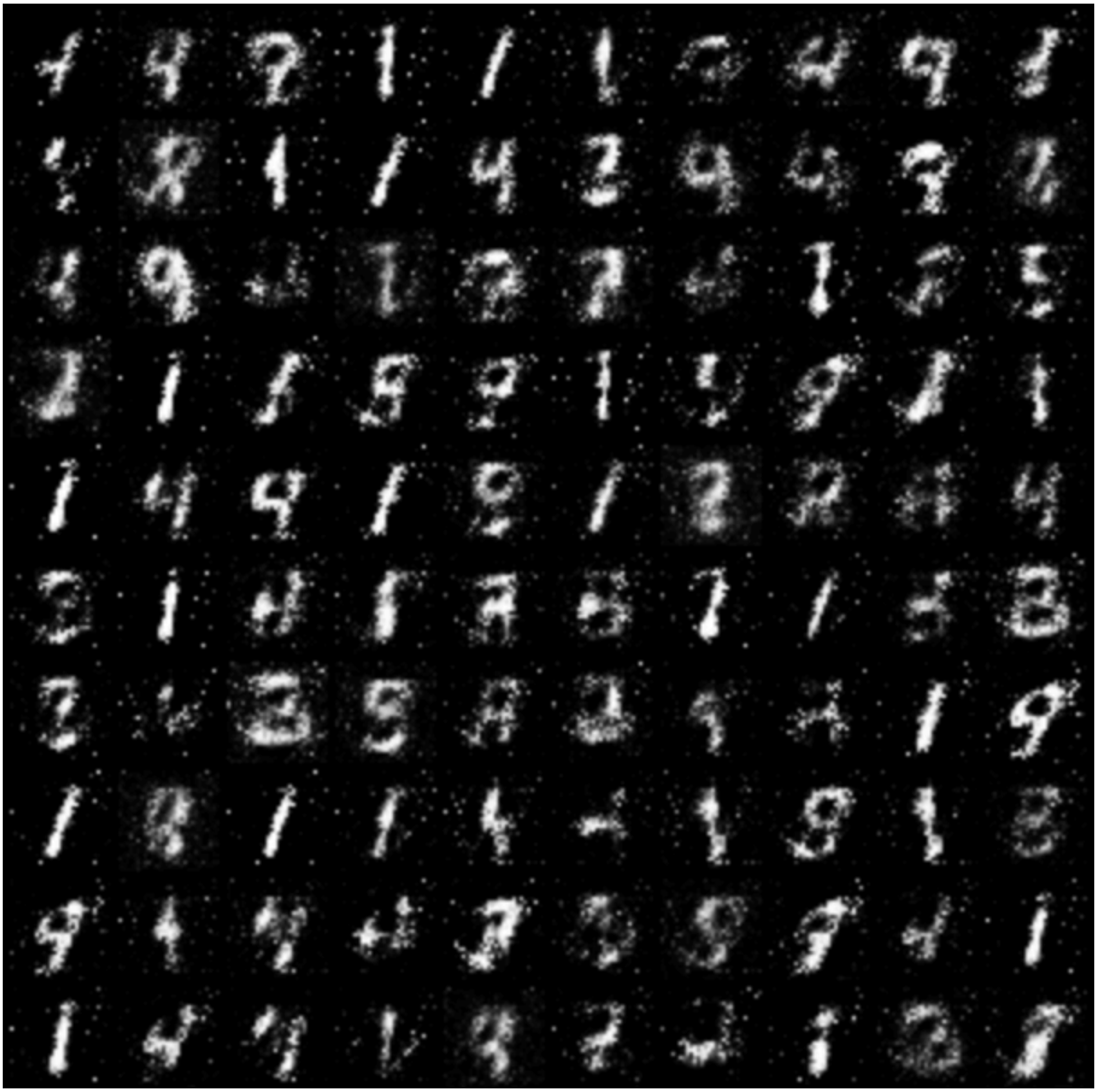
۱ ایپاک:



۱۰ ایپاک:



۵۰ ایپاک:



۱۵۰ ایپاک:

۲۵۰ اپاک: