gans

# example code

|  |
| --- |
| import torch  import torch.nn as nn  import torch.nn.functional as F  from torchvision import datasets  import torchvision.transforms as transforms  from torch.utils.data import DataLoader  from torchvision.transforms.functional import to\_pil\_image  import matplotlib.pylab as plt  %matplotlib inline  import os  import numpy as np  import time  device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')  import torch  import torchvision  print(torch.\_\_version\_\_)  print(torchvision.\_\_version\_\_)  # 데이터 경로 지정  path2data = './data'  os.makedirs(path2data, exist\_ok=True) # 폴더 생성  # MNIST dataset 불러오기  train\_ds = datasets.MNIST(path2data, train=True, transform=transforms.Compose([transforms.ToTensor(),  transforms.Normalize([0.5],[0.5])]), download=True)  # 샘플 이미지 확인  img, label = train\_ds[0]  plt.imshow(to\_pil\_image(img),cmap='gray')  # 데이터 로더 생성  train\_dl = DataLoader(train\_ds, batch\_size=32, shuffle=True)  # check  for x, y in train\_dl:      print(x.shape, y.shape)      break  # generator: noise를 입력받아 이미지를 생성합니다.  class Generator(nn.Module):      def \_\_init\_\_(self, params):          super().\_\_init\_\_()          self.nz = params['nz'] # 입력 노이즈 벡터 수, 100          self.img\_size = params['img\_size'] # 이미지 크기, 1x28x28          self.model = nn.Sequential(              \*self.\_fc\_layer(self.nz, 128, normalize=False),              \*self.\_fc\_layer(128,256),              \*self.\_fc\_layer(256,512),              \*self.\_fc\_layer(512,1024),              nn.Linear(1024,int(np.prod(self.img\_size))),              nn.Tanh()          )      def forward(self, z):          img = self.model(z)          img = img.view(img.size(0), \*self.img\_size)          return img      # fc layer      def \_fc\_layer(self, in\_channels, out\_channels, normalize=True):          layers = []          layers.append(nn.Linear(in\_channels, out\_channels)) # fc layer          if normalize:              layers.append(nn.BatchNorm1d(out\_channels, 0.8)) # BN          layers.append(nn.LeakyReLU(0.2)) # LeakyReLU          return layers  # check  params = {'nz':100,            'img\_size':(1,28,28)}  x = torch.randn(16,100).to(device) # random noise  model\_gen = Generator(params).to(device)  output = model\_gen(x) # noise를 입력받아 이미지 생성  print(output.shape)  # discriminator: 진짜 이미지와 가짜 이미지를 분류합니다.  class Discriminator(nn.Module):      def \_\_init\_\_(self,params):          super().\_\_init\_\_()          self.img\_size = params['img\_size'] # 이미지 크기, 1x28x28          self.model = nn.Sequential(              nn.Linear(int(np.prod(self.img\_size)), 512),              nn.LeakyReLU(0.2),              nn.Linear(512,256),              nn.LeakyReLU(0.2),              nn.Linear(256,1),              nn.Sigmoid()          )      def forward(self, x):          x = x.view(x.size(0),-1)          x = self.model(x)          return x  # check  x = torch.randn(16,1,28,28).to(device)  model\_dis = Discriminator(params).to(device)  output = model\_dis(x)  print(output.shape)  # 가중치 초기화  def initialize\_weights(model):      classname = model.\_\_class\_\_.\_\_name\_\_      # fc layer      if classname.find('Linear') != -1:          nn.init.normal\_(model.weight.data, 0.0, 0.02)          nn.init.constant\_(model.bias.data, 0)      # batchnorm      elif classname.find('BatchNorm') != -1:          nn.init.normal\_(model.weight.data, 1.0, 0.02)          nn.init.constant\_(model.bias.data, 0)  # 가중치 초기화 적용  model\_gen.apply(initialize\_weights);  model\_dis.apply(initialize\_weights);  # 손실 함수  loss\_func = nn.BCELoss()  from torch import optim  # 최적화 파라미터  lr = 2e-4  beta1 = 0.5  opt\_dis = optim.Adam(model\_dis.parameters(),lr=lr,betas=(beta1,0.999))  opt\_gen = optim.Adam(model\_gen.parameters(),lr=lr,betas=(beta1,0.999))  real\_label = 1.  fake\_label = 0.  nz = params['nz']  num\_epochs = 100  loss\_history={'gen':[],                'dis':[]}  batch\_count = 0  start\_time = time.time()  model\_dis.train()  model\_gen.train()  for epoch in range(num\_epochs):      for xb, yb in train\_dl:          ba\_si = xb.size(0)          xb = xb.to(device)          yb\_real = torch.Tensor(ba\_si,1).fill\_(1.0).to(device)          yb\_fake = torch.Tensor(ba\_si,1).fill\_(0.0).to(device)          # Generator          model\_gen.zero\_grad()          noise = torch.randn(ba\_si,nz, device=device) # 노이즈 생성          out\_gen = model\_gen(noise) # 가짜 이미지 생성          out\_dis = model\_dis(out\_gen) # 가짜 이미지 판별          loss\_gen = loss\_func(out\_dis, yb\_real)          loss\_gen.backward()          opt\_gen.step()          # Discriminator          model\_dis.zero\_grad()          out\_real = model\_dis(xb) # 진짜 이미지 판별          out\_fake = model\_dis(out\_gen.detach()) # 가짜 이미지 판별          loss\_real = loss\_func(out\_real, yb\_real)          loss\_fake = loss\_func(out\_fake, yb\_fake)          loss\_dis = (loss\_real + loss\_fake) / 2          loss\_dis.backward()          opt\_dis.step()          loss\_history['gen'].append(loss\_gen.item())          loss\_history['dis'].append(loss\_dis.item())          batch\_count += 1          if batch\_count % 1000 == 0:              print('Epoch: %.0f, G\_Loss: %.6f, D\_Loss: %.6f, time: %.2f min' %(epoch, loss\_gen.item(), loss\_dis.item(), (time.time()-start\_time)/60))  # 가중치 저장  path2models = './models/'  os.makedirs(path2models, exist\_ok=True)  path2weights\_gen = os.path.join(path2models, 'weights\_gen.pt')  path2weights\_dis = os.path.join(path2models, 'weights\_dis.pt')  torch.save(model\_gen.state\_dict(), path2weights\_gen)  torch.save(model\_dis.state\_dict(), path2weights\_dis)  # 가중치 불러오기  weights = torch.load(path2weights\_gen)  model\_gen.load\_state\_dict(weights)  # evaluation mode  model\_gen.eval()  # fake image 생성  with torch.no\_grad():      fixed\_noise = torch.randn(16, 100, device=device)      img\_fake = model\_gen(fixed\_noise).detach().cpu()  print(img\_fake.shape)  # 가짜 이미지 시각화  plt.figure(figsize=(10,10))  for ii in range(16):      plt.subplot(4,4,ii+1)      plt.imshow(to\_pil\_image(0.5\*img\_fake[ii]+0.5),cmap='gray')      plt.axis('off') |

# testing result

|  |
| --- |
| Epoch: 0, G\_Loss: 1.225349, D\_Loss: 0.325917, time: 0.25 min  Epoch: 1, G\_Loss: 0.158426, D\_Loss: 1.034579, time: 0.50 min  Epoch: 1, G\_Loss: 1.455488, D\_Loss: 0.248437, time: 0.76 min  Epoch: 2, G\_Loss: 2.177262, D\_Loss: 0.273030, time: 1.01 min  Epoch: 2, G\_Loss: 2.182041, D\_Loss: 0.468778, time: 1.27 min  Epoch: 3, G\_Loss: 2.937927, D\_Loss: 0.167037, time: 1.52 min  Epoch: 3, G\_Loss: 2.943258, D\_Loss: 0.126765, time: 1.77 min  Epoch: 4, G\_Loss: 2.143854, D\_Loss: 0.196634, time: 2.03 min  Epoch: 4, G\_Loss: 2.523314, D\_Loss: 0.229086, time: 2.28 min  Epoch: 5, G\_Loss: 1.598965, D\_Loss: 0.190893, time: 2.54 min  Epoch: 5, G\_Loss: 2.490905, D\_Loss: 0.128028, time: 2.79 min  Epoch: 6, G\_Loss: 2.946451, D\_Loss: 0.189892, time: 3.05 min  Epoch: 6, G\_Loss: 2.453724, D\_Loss: 0.220313, time: 3.30 min  Epoch: 7, G\_Loss: 2.650912, D\_Loss: 0.430259, time: 3.56 min  Epoch: 7, G\_Loss: 3.881546, D\_Loss: 0.307121, time: 3.81 min  Epoch: 8, G\_Loss: 1.882840, D\_Loss: 0.157760, time: 4.07 min  Epoch: 9, G\_Loss: 2.844269, D\_Loss: 0.200138, time: 4.32 min  Epoch: 9, G\_Loss: 2.262634, D\_Loss: 0.103063, time: 4.58 min  Epoch: 10, G\_Loss: 1.091042, D\_Loss: 0.318176, time: 4.83 min  Epoch: 10, G\_Loss: 3.054717, D\_Loss: 0.157054, time: 5.09 min  Epoch: 11, G\_Loss: 1.134031, D\_Loss: 0.240978, time: 5.34 min  Epoch: 11, G\_Loss: 4.131405, D\_Loss: 0.188712, time: 5.60 min  Epoch: 12, G\_Loss: 0.948373, D\_Loss: 0.420420, time: 5.85 min  Epoch: 12, G\_Loss: 4.673799, D\_Loss: 0.494476, time: 6.11 min  Epoch: 13, G\_Loss: 4.125015, D\_Loss: 0.262952, time: 6.37 min  Epoch: 97, G\_Loss: 2.139873, D\_Loss: 0.195012, time: 46.50 min Epoch: 98, G\_Loss: 2.071497, D\_Loss: 0.243359, time: 46.75 min Epoch: 98, G\_Loss: 2.840386, D\_Loss: 0.238688, time: 47.00 min Epoch: 99, G\_Loss: 1.283836, D\_Loss: 0.342422, time: 47.26 min Epoch: 99, G\_Loss: 3.590829, D\_Loss: 0.248883, time: 47.52 min |