HOMEWORK 6

>>Hemanth Sridhar Nakshatri<<
>>nakshatri@wisc.edu, 9085807346<<

Instructions: Use this latex file as a template to develop your homework. Submit your homework on time as a single pdf file. Please wrap your code and upload to a public GitHub repo, then attach the link below the instructions so that we can access it. Answers to the questions that are not within the pdf are not accepted. This includes external links or answers attached to the code implementation. Late submissions may not be accepted. You can choose any programming language (i.e. python, R, or MATLAB). Please check Piazza for updates about the homework. It is ok to share the results of the experiments and compare them with each other.

The implementation codes and results can be found in my github repository here https://github.com/hemanth-nakshatri/ECE760-Machine-Learning/tree/main/hw/hw6. In case HW6 is somehow inaccesssible, all Homeworks can be found at https://github.com/hemanth-nakshatri/ECE760-Machine-Learning/.

***NOTE:** The Jupyter notebook with code has been attached to the end of the PDF as well for reference.*

1 Implementation: GAN (50 pts)

In this part, you are expected to implement GAN with MNIST dataset. We have provided a base jupyter notebook (gan-base.ipynb) for you to start with, which provides a model setup and training configurations to train GAN with MNIST dataset.

(a) Implement training loop and report learning curves and generated images in epoch 1, 50, 100. Note that drawing learning curves and visualization of images are already implemented in provided jupyter notebook. (20 pts)

Procedure 1 Training GAN, modified from Goodfellow et al. (2014)

Input: m: real data batch size, n_z: fake data batch size
Output: Discriminator D, Generator G
for number of training iterations **do**
# Training discriminator
Sample minibatch of n_z noise samples $\{z^{(1)}, z^{(2)}, \cdots, z^{(n_z)}\}$ from noise prior $p_g(z)$
Sample minibatch of $\{x^{(1)}, x^{(2)}, \cdots, x^{(m)}\}$
Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d}\Big(\frac{1}{m}\sum_{i=1}^{m}\log D(x^{(i)}) + \frac{1}{n_z}\sum_{i=1}^{n_z}\log(1 - D(G(z^{(i)})))\Big)$$

Training generator
Sample minibatch of n_z noise samples $\{z^{(1)}, z^{(2)}, \cdots, z^{(n_z)}\}$ from noise prior $p_g(z)$
Update the generator by ascending its stochastic gradient:

$$\nabla_{\theta_g}\frac{1}{n_z}\sum_{i=1}^{n_z}\log D(G(z^{(i)}))\Big)$$

end for
# The gradient-based updates can use any standard gradient-based learning rule. In the base code, we are using Adam optimizer (Kingma and Ba, 2014)

The learning curve and the generated samples for Q1a are shown in Fig. 1 and Fig. 2.

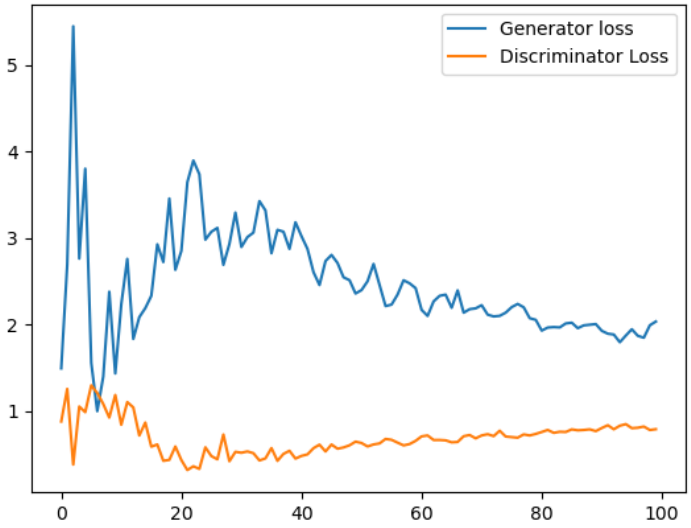


Figure 1: Learning curve for Q1a. Basic GAN

(a) epoch 1 (b) epoch 50 (c) epoch 100

Figure 2: Generated images by G for Question 1a.

(b) Replace the generator update rule as the original one in the slide,
"Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{n_z} \sum_{i=1}^{n_z} \log(1 - D(G(z^{(i)})))$$

" , and report learning curves and generated images in epoch 1, 50, 100. Compare the result with (a). Note that it may not work. If training does not work, explain why it doesn't work.
You may find this helpful: https://jonathan-hui.medium.com/gan-what-is-wrong-with-the-gan-cost-function-6f594162ce01 (10 pts)

Here, instead of ascending the gradient, we are asked to descend the gradient with the provided gradient/loss function. With gradient ascent, we are trying to increase the probability that the discriminator is fooled. But with descent, we are trying to decrease the chances of the discriminator being wrong.

This leads to the generator learning to produce images that the discriminator detect them as fakes easily resulting in a stable equilibrium. Thus generator keeps producing similar images and discriminator classifies it as fake easily. Due to this, no learning happens as seen in Fig. 3 and Fig. 4.

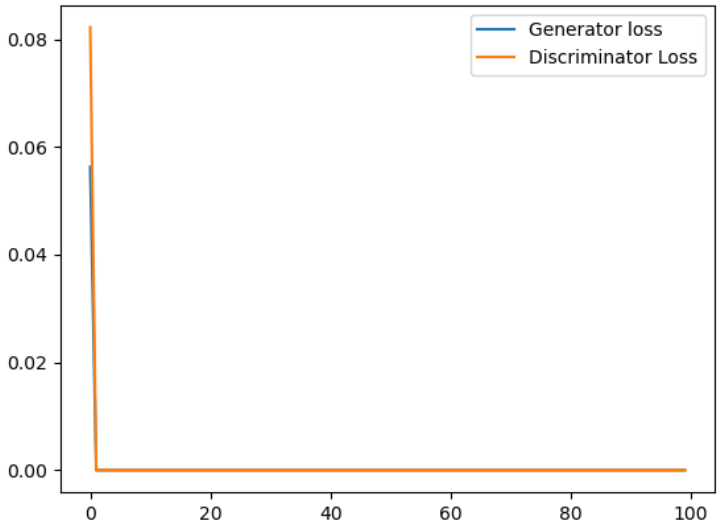
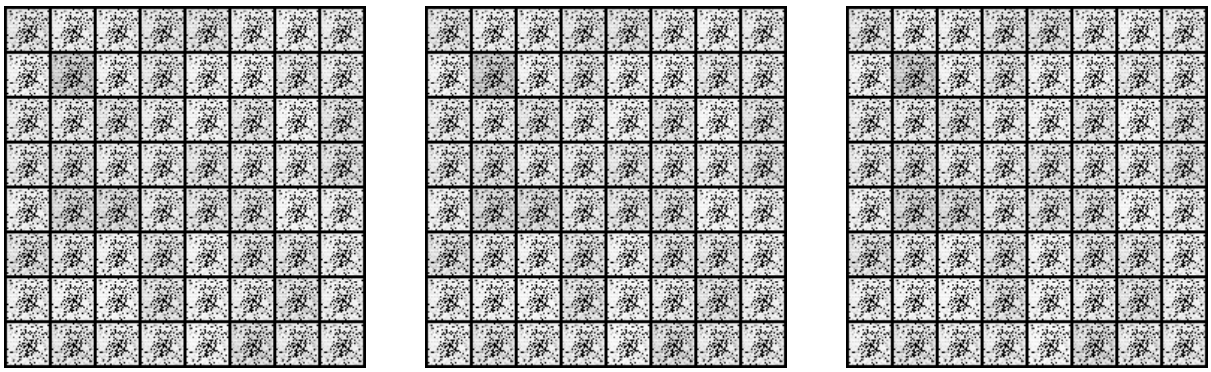


Figure 3: Learning curve for Q1b. Basic GAN



(a) epoch 1 (b) epoch 50 (c) epoch 100

Figure 4: Generated images by G for Question 1b.

(c) Except the method that we used in (a), how can we improve training for GAN? Implement that and report your setup, learning curves, and generated images in epoch 1, 50, 100. This question is an open-ended question and you can choose whichever method you want. (20 pts)

The hyperparameter β_1 was changed from its default *0.9* to *0.5* and β_2 was kept at default *0.999*. This controls the decay rate. This was inspired by the various papers like DCGAN, SCGAN etc that use this kind of implementation.

The learning curve and the generated samples for Q1c are shown in Fig. 5 and Fig. 6.

Note that many other improvements might be possible to be used in conjunction with this.

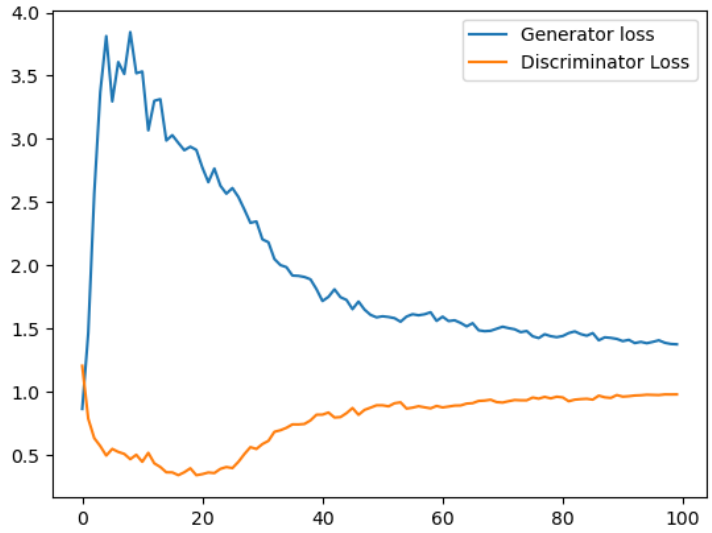
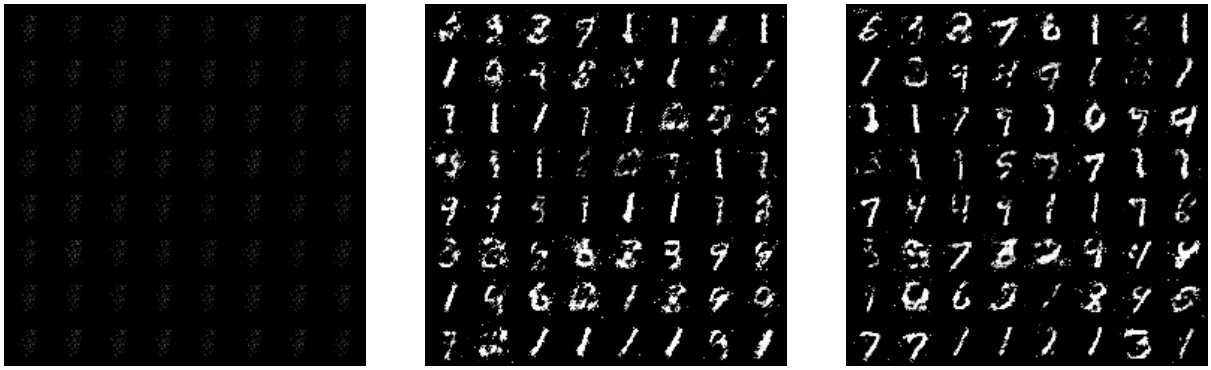


Figure 5: Learning curve for Q1c. Basic GAN



(a) epoch 1 (b) epoch 50 (c) epoch 100

Figure 6: Generated images by G for Question 1c.

2 Directed Graphical Model [25 points]

Consider the directed graphical model (aka Bayesian network) in Figure 7.

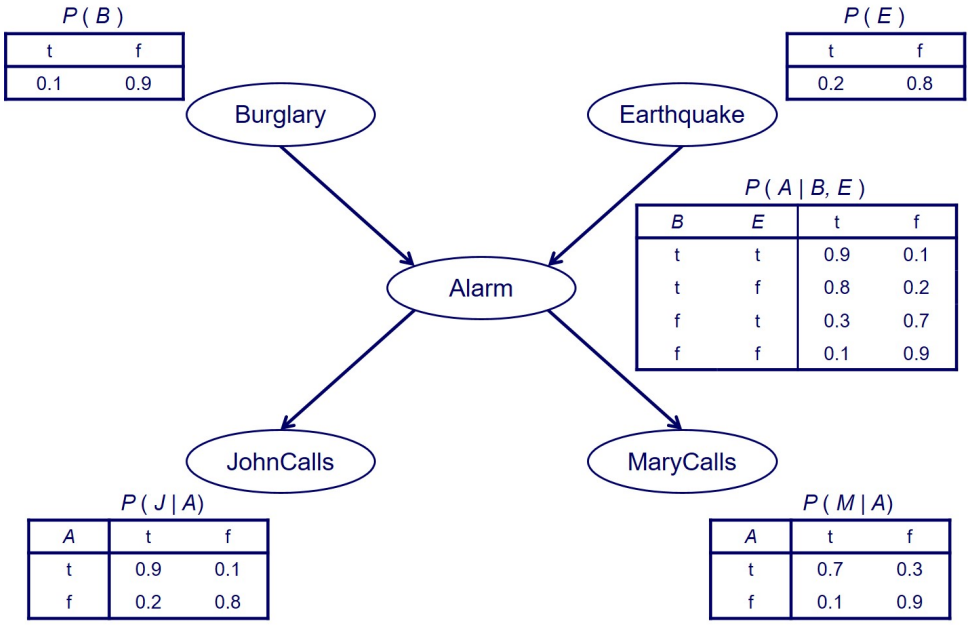


Figure 7: A Bayesian Network example.

Compute $P(B = t \mid E = f, J = t, M = t)$ and $P(B = t \mid E = t, J = t, M = t)$. (10 points for each) These are the conditional probabilities of a burglar in your house (yikes!) when both of your neighbors John and Mary call you and say they hear an alarm in your house, but without or with an earthquake also going on in that area (what a busy day), respectively.

$$P(B = t \mid E = f, J = t, M = t) = \frac{P(B = t, E = f, J = t, M = t)}{P(E = f, J = t, M = t)}$$

$$\begin{aligned} P(B = t, E = f, J = t, M = t) &= \sum_A P(j = t \mid A)P(M = t \mid A)P(A \mid B = t, E = f)P(B = t)P(E = f) \\ &= (0.9 \times 0.7 \times 0.8 \times 0.1 \times 0.8) + (0.2 \times 0.1 \times 0.2 \times 0.1 \times 0.8) \\ &= 0.04032 + 0.00032 \\ &= 0.04064 \end{aligned}$$

$$\begin{aligned} P(E = f, J = t, M = t) &= \sum_A \sum_B P(j = t \mid A)P(M = t \mid A)P(A \mid B, E = f)P(B)P(E = f) \\ &= 0.04064 + (0.9 \times 0.7 \times 0.1 \times 0.9 \times 0.8) + (0.2 \times 0.1 \times 0.9 \times 0.9 \times 0.8) \\ &= 0.04064 + 0.04536 + 0.01296 \\ &= 0.09896 \end{aligned}$$

Thus,

$$P(B = t \mid E = f, J = t, M = t) = \frac{0.04064}{0.09896} = \mathbf{0.410670978}$$

Now,

$$P(B = t \mid E = t, J = t, M = t) = \frac{P(B = t, E = t, J = t, M = t)}{P(E = t, J = t, M = t)}$$

$$\begin{aligned} P(B = t, E = t, J = t, M = t) &= \sum_A P(j = t \mid A)P(M = t \mid A)P(A \mid B = t, E = t)P(B = t)P(E = t) \\ &= (0.9 \times 0.7 \times 0.9 \times 0.1 \times 0.2) + (0.2 \times 0.1 \times 0.1 \times 0.1 \times 0.2) \\ &= 0.01134 + .00004 \\ &= 0.01138 \end{aligned}$$

$$\begin{aligned}P(E=t,J=t,M=t) &= \sum_A \sum_B P(j=t \mid A)P(M=t \mid A)P(A \mid B,E=t)P(B)P(E=t) \\ &= 0.01138 + (0.9 \times 0.7 \times 0.3 \times 0.9 \times 0.2) + (0.2 \times 0.1 \times 0.7 \times 0.9 \times 0.2) \\ &= 0.01138 + 0.03402 + 0.00252 \\ &= 0.04792\end{aligned}$$

Hence,

$$P(B=t \mid E=t,J=t,M=t) = \frac{0.01138}{0.04792} = \mathbf{0.237479132}$$

3 Chow-Liu Algorithm [25 pts]

Suppose we wish to construct a directed graphical model for 3 features X, Y, and Z using the Chow-Liu algorithm. We are given data from 100 independent experiments where each feature is binary and takes value T or F. Below is a table summarizing the observations of the experiment:

X	Y	Z	Count
T	T	T	36
T	T	F	4
T	F	T	2
T	F	F	8
F	T	T	9
F	T	F	1
F	F	T	8
F	F	F	32

1. Compute the mutual information $I(X,Y)$ based on the frequencies observed in the data. (5 pts)

$$\begin{aligned}I(X,Y) &= \sum_{x\in\mathcal{X}} \sum_{y\in\mathcal{Y}} P(x,y)\log_2 \frac{P(x,y)}{P(x)P(y)} \\ &= 0.4\log_2 \frac{0.4}{(0.5)(0.5)} + 0.1\log_2 \frac{0.1}{(0.5)(0.5)} + 0.1\log_2 \frac{0.1}{(0.5)(0.5)} + 0.4\log_2 \frac{0.4}{(0.5)(0.5)} \\ &= 0.2712 - 0.1322 - 0.1322 + 0.2712 \\ &= \mathbf{0.2781}\end{aligned}$$

2. Compute the mutual information $I(X,Z)$ based on the frequencies observed in the data. (5 pts)

$$\begin{aligned}I(X,Z) &= \sum_{x\in\mathcal{X}} \sum_{z\in\mathcal{Z}} P(x,z)\log_2 \frac{P(x,z)}{P(x)P(z)} \\ &= 0.38\log_2 \frac{0.38}{(0.5)(0.55)} + 0.12\log_2 \frac{0.12}{(0.5)(0.45)} + 0.17\log_2 \frac{0.17}{(0.5)(0.55)} + 0.33\log_2 \frac{0.33}{(0.5)(0.45)} \\ &= 0.1773 - 0.1088 - 0.1180 + 0.1823 \\ &= \mathbf{0.1328}\end{aligned}$$

3. Compute the mutual information $I(Z,Y)$ based on the frequencies observed in the data. (5 pts)

$$\begin{aligned}I(Z,Y) &= \sum_{z\in\mathcal{Z}} \sum_{y\in\mathcal{Y}} P(z,y)\log_2 \frac{P(z,y)}{P(z)P(y)} \\ &= 0.45\log_2 \frac{0.45}{(0.55)(0.5)} + 0.1\log_2 \frac{0.1}{(0.55)(0.5)} + 0.05\log_2 \frac{0.05}{(0.45)(0.5)} + 0.4\log_2 \frac{0.4}{(0.45)(0.5)} \\ &= 0.3197 - 0.1459 - 0.1085 + 0.3320 \\ &= \mathbf{0.3973}\end{aligned}$$

4. Which undirected edges will be selected by the Chow-Liu algorithm as the maximum spanning tree? (5 pts)
   Edges (Z, Y) and (X, Y) would be selected.

5. Root your tree at node X, assign directions to the selected edges. (5 pts)
   $X \rightarrow Y \rightarrow Z$

References

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. *Advances in neural information processing systems*, 27.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

gan-base-code

November 20, 2023

[ ]:
```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torch.optim as optim
import torchvision.datasets as datasets
import imageio
import numpy as np
import matplotlib
from torchvision.utils import make_grid, save_image
from torch.utils.data import DataLoader
from matplotlib import pyplot as plt
from tqdm import tqdm
```

1 Define learning parameters

[ ]:
```
# learning parameters
batch_size = 512
epochs = 100
sample_size = 64 # fixed sample size for generator
nz = 128 # latent vector size
k = 1 # number of steps to apply to the discriminator
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

2 Prepare training dataset

[ ]:
```
transform = transforms.Compose([
                                transforms.ToTensor(),
                                transforms.Normalize((0.5,),(0.5,)),
])
to_pil_image = transforms.ToPILImage()

# Make input, output folders
!mkdir -p input
!mkdir -p outputs

# Load train data
```

```
train_data = datasets.MNIST(
    root='input/data',
    train=True,
    download=True,
    transform=transform
)
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
```

```
A subdirectory or file -p already exists.
Error occurred while processing: -p.
A subdirectory or file input already exists.
Error occurred while processing: input.
A subdirectory or file -p already exists.
Error occurred while processing: -p.
A subdirectory or file outputs already exists.
Error occurred while processing: outputs.
```

3 Generator

[ ]:
```
class Generator(nn.Module):
    def __init__(self, nz):
        super(Generator, self).__init__()
        self.nz = nz
        self.main = nn.Sequential(
            nn.Linear(self.nz, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, 784),
            nn.Tanh(),
        )
    def forward(self, x):
        return self.main(x).view(-1, 1, 28, 28)
```

4 Discriminator

[ ]:
```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.n_input = 784
        self.main = nn.Sequential(
            nn.Linear(self.n_input, 1024),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
```

```
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )
    def forward(self, x):
        x = x.view(-1, 784)
        return self.main(x)
```

[ ]:
```
generator = Generator(nz).to(device)
discriminator = Discriminator().to(device)
print('##### GENERATOR #####')
print(generator)
print('######################')
print('\n##### DISCRIMINATOR #####')
print(discriminator)
print('######################')
```

```
##### GENERATOR #####
Generator(
  (main): Sequential(
    (0): Linear(in_features=128, out_features=256, bias=True)
    (1): LeakyReLU(negative_slope=0.2)
    (2): Linear(in_features=256, out_features=512, bias=True)
    (3): LeakyReLU(negative_slope=0.2)
    (4): Linear(in_features=512, out_features=1024, bias=True)
    (5): LeakyReLU(negative_slope=0.2)
    (6): Linear(in_features=1024, out_features=784, bias=True)
    (7): Tanh()
  )
)
######################

##### DISCRIMINATOR #####
Discriminator(
  (main): Sequential(
    (0): Linear(in_features=784, out_features=1024, bias=True)
    (1): LeakyReLU(negative_slope=0.2)
    (2): Dropout(p=0.3, inplace=False)
    (3): Linear(in_features=1024, out_features=512, bias=True)
    (4): LeakyReLU(negative_slope=0.2)
    (5): Dropout(p=0.3, inplace=False)
    (6): Linear(in_features=512, out_features=256, bias=True)
    (7): LeakyReLU(negative_slope=0.2)
```

```
    (8): Dropout(p=0.3, inplace=False)
    (9): Linear(in_features=256, out_features=1, bias=True)
    (10): Sigmoid()
  )
)
######################
```

5 Tools for training

[ ]:
```
# optimizers
optim_g = optim.Adam(generator.parameters(), lr=0.0002,betas=(0.5,0.999))
optim_d = optim.Adam(discriminator.parameters(), lr=0.0002,betas=(0.5,0.999))
```

[ ]:
```
# loss function
criterion = nn.BCELoss() # Binary Cross Entropy loss
```

[ ]:
```
losses_g = [] # to store generator loss after each epoch
losses_d = [] # to store discriminator loss after each epoch
images = [] # to store images generatd by the generator
```

[ ]:
```
# to create real labels (1s)
def label_real(size):
    data = torch.ones(size, 1)
    return data.to(device)
# to create fake labels (0s)
def label_fake(size):
    data = torch.zeros(size, 1)
    return data.to(device)
```

[ ]:
```
# function to create the noise vector
def create_noise(sample_size, nz):
    return torch.randn(sample_size, nz).to(device)
```

[ ]:
```
# to save the images generated by the generator
def save_generator_image(image, path):
    save_image(image, path)
```

[ ]:
```
# create the noise vector - fixed to track how GAN is trained.
noise = create_noise(sample_size, nz)
```

6 Q. Write training loop

[ ]:
```
torch.manual_seed(7777)

def generator_loss(output, true_label):
    ############ YOUR CODE HERE ##########
```

```
    # return criterion(output, true_label)
    return criterion(output, true_label)


    ######################################

def discriminator_loss(output, true_label):
    ############ YOUR CODE HERE ##########
    return criterion(output, true_label)


    ######################################


for epoch in range(epochs):
    loss_g = 0.0
    loss_d = 0.0
    for bi, data in tqdm(enumerate(train_loader), total=int(len(train_data)/
 ↪train_loader.batch_size)):
        ############ YOUR CODE HERE ##########
        x = data[0]

        ### UPDATE DISCRIMINATOR ###
        optim_d.zero_grad() # initialize gradient

        # real pass
        real_output = discriminator(x)
        d_loss_real = discriminator_loss(real_output, label_real(real_output.
 ↪size()[0]))
        d_loss_real.backward()

        # fake pass
        fake_data = generator(create_noise(sample_size, nz))
        fake_output = discriminator(fake_data)
        d_loss_fake = discriminator_loss(fake_output, label_fake(sample_size))
        d_loss_fake.backward()

        # loss_D = (d_loss_real + d_loss_fake) / 2
        # loss_D.backward()
        optim_d.step()

        # loss_d += loss_D.detach().numpy()
        loss_d += d_loss_real.item() + d_loss_fake.item()

        ### UPDATE GENERATOR ###
        optim_g.zero_grad() # initialize gradient
```

```
        generated_data = generator(create_noise(sample_size, nz))
        generated_output = discriminator(generated_data)
        loss_G = generator_loss(generated_output, label_real(sample_size)) # we␣
↪want to trick the discriminator on this pass

        loss_G.backward()
        optim_g.step()

        # loss_g += loss_G.detach().numpy()
        loss_g += loss_G.item()

        ######################################


    # create the final fake image for the epoch
    generated_img = generator(noise).cpu().detach()

    # make the images as grid
    generated_img = make_grid(generated_img)

    # visualize generated images
    # if (epoch + 1) % 5 == 0:
    #     plt.imshow(generated_img.permute(1, 2, 0))
    #     plt.title(f'epoch {epoch+1}')
    #     plt.axis('off')
    #     plt.show()

    # save the generated torch tensor models to disk
    save_generator_image(generated_img, f"outputs/gen_img{epoch+1}.png")
    images.append(generated_img)
    epoch_loss_g = loss_g / bi # total generator loss for the epoch
    epoch_loss_d = loss_d / bi # total discriminator loss for the epoch
    losses_g.append(epoch_loss_g)
    losses_d.append(epoch_loss_d)

    print(f"Epoch {epoch+1} of {epochs}")
    print(f"Generator loss: {epoch_loss_g:.8f}, Discriminator loss:␣
↪{epoch_loss_d:.8f}")
```

```
118it [00:21,  5.44it/s]

Epoch 1 of 100
Generator loss: 0.89233431, Discriminator loss: 1.21344539

118it [00:22,  5.18it/s]

Epoch 2 of 100
Generator loss: 1.51508693, Discriminator loss: 0.80018493

118it [00:28,  4.12it/s]
```

```
Epoch 3 of 100
Generator loss: 2.57623989, Discriminator loss: 0.60748687

118it [00:28,  4.08it/s]

Epoch 4 of 100
Generator loss: 3.10700087, Discriminator loss: 0.56760097

118it [00:24,  4.75it/s]

Epoch 5 of 100
Generator loss: 3.38156862, Discriminator loss: 0.63911728

118it [00:26,  4.49it/s]

Epoch 6 of 100
Generator loss: 3.39346600, Discriminator loss: 0.57983313

118it [00:24,  4.88it/s]

Epoch 7 of 100
Generator loss: 3.40839900, Discriminator loss: 0.57603893

118it [00:24,  4.76it/s]

Epoch 8 of 100
Generator loss: 3.16496424, Discriminator loss: 0.62582696

118it [00:26,  4.53it/s]

Epoch 9 of 100
Generator loss: 3.13270225, Discriminator loss: 0.60029280

118it [00:22,  5.32it/s]

Epoch 10 of 100
Generator loss: 3.72435599, Discriminator loss: 0.42099568

118it [00:22,  5.32it/s]

Epoch 11 of 100
Generator loss: 3.72008466, Discriminator loss: 0.48486457

118it [00:22,  5.36it/s]

Epoch 12 of 100
Generator loss: 3.34511882, Discriminator loss: 0.45234268

118it [00:21,  5.37it/s]

Epoch 13 of 100
Generator loss: 3.37860875, Discriminator loss: 0.48890289

118it [00:22,  5.35it/s]

Epoch 14 of 100
Generator loss: 3.03140165, Discriminator loss: 0.55070614

118it [00:22,  5.36it/s]
```

```
Epoch 15 of 100
Generator loss: 3.09468642, Discriminator loss: 0.46424356

118it [00:21,  5.40it/s]

Epoch 16 of 100
Generator loss: 3.15281580, Discriminator loss: 0.36138126

118it [00:21,  5.40it/s]

Epoch 17 of 100
Generator loss: 2.99566414, Discriminator loss: 0.35367088

118it [00:22,  5.36it/s]

Epoch 18 of 100
Generator loss: 2.98920418, Discriminator loss: 0.36099387

118it [00:22,  5.35it/s]

Epoch 19 of 100
Generator loss: 3.03096411, Discriminator loss: 0.36841094

118it [00:23,  5.03it/s]

Epoch 20 of 100
Generator loss: 2.94286393, Discriminator loss: 0.33076372

118it [00:24,  4.85it/s]

Epoch 21 of 100
Generator loss: 2.98633715, Discriminator loss: 0.34647985

118it [00:22,  5.22it/s]

Epoch 22 of 100
Generator loss: 2.83965758, Discriminator loss: 0.40250333

118it [00:22,  5.25it/s]

Epoch 23 of 100
Generator loss: 2.71199918, Discriminator loss: 0.42603354

118it [00:22,  5.27it/s]

Epoch 24 of 100
Generator loss: 2.58012214, Discriminator loss: 0.46551912

118it [00:25,  4.55it/s]

Epoch 25 of 100
Generator loss: 2.45036890, Discriminator loss: 0.50648787

118it [00:23,  4.95it/s]

Epoch 26 of 100
Generator loss: 2.41153055, Discriminator loss: 0.52529246

118it [00:22,  5.24it/s]
```

```
Epoch 27 of 100
Generator loss: 2.36411245, Discriminator loss: 0.52114117

118it [00:22,  5.25it/s]

Epoch 28 of 100
Generator loss: 2.29258940, Discriminator loss: 0.61361109

118it [00:25,  4.55it/s]

Epoch 29 of 100
Generator loss: 2.22686949, Discriminator loss: 0.60302456

118it [00:23,  5.07it/s]

Epoch 30 of 100
Generator loss: 2.13979986, Discriminator loss: 0.66952474

118it [00:22,  5.27it/s]

Epoch 31 of 100
Generator loss: 2.05574288, Discriminator loss: 0.69716041

118it [00:22,  5.22it/s]

Epoch 32 of 100
Generator loss: 1.99202017, Discriminator loss: 0.70046395

118it [00:22,  5.28it/s]

Epoch 33 of 100
Generator loss: 2.02642933, Discriminator loss: 0.70523105

118it [00:24,  4.86it/s]

Epoch 34 of 100
Generator loss: 2.04642163, Discriminator loss: 0.69000542

118it [00:22,  5.31it/s]

Epoch 35 of 100
Generator loss: 2.05895000, Discriminator loss: 0.70150603

118it [00:22,  5.33it/s]

Epoch 36 of 100
Generator loss: 2.00070626, Discriminator loss: 0.72214012

118it [00:22,  5.35it/s]

Epoch 37 of 100
Generator loss: 2.07180620, Discriminator loss: 0.67736482

118it [00:22,  5.25it/s]

Epoch 38 of 100
Generator loss: 1.99152974, Discriminator loss: 0.71399385

118it [00:21,  5.39it/s]
```

```
Epoch 39 of 100
Generator loss: 1.95386147, Discriminator loss: 0.72824327

118it [00:21,  5.39it/s]

Epoch 40 of 100
Generator loss: 1.93761626, Discriminator loss: 0.71283562

118it [00:22,  5.21it/s]

Epoch 41 of 100
Generator loss: 1.94291955, Discriminator loss: 0.71932507

118it [00:21,  5.40it/s]

Epoch 42 of 100
Generator loss: 1.92840281, Discriminator loss: 0.71938275

118it [00:22,  5.34it/s]

Epoch 43 of 100
Generator loss: 1.90660114, Discriminator loss: 0.71734198

118it [00:21,  5.38it/s]

Epoch 44 of 100
Generator loss: 1.77763409, Discriminator loss: 0.77622649

118it [00:21,  5.40it/s]

Epoch 45 of 100
Generator loss: 1.75386907, Discriminator loss: 0.79112470

118it [00:22,  5.35it/s]

Epoch 46 of 100
Generator loss: 1.81372272, Discriminator loss: 0.77026190

118it [00:21,  5.40it/s]

Epoch 47 of 100
Generator loss: 1.74393080, Discriminator loss: 0.80526546

118it [00:21,  5.37it/s]

Epoch 48 of 100
Generator loss: 1.66839369, Discriminator loss: 0.81753920

118it [00:21,  5.37it/s]

Epoch 49 of 100
Generator loss: 1.71521196, Discriminator loss: 0.82347169

118it [00:25,  4.60it/s]

Epoch 50 of 100
Generator loss: 1.68646187, Discriminator loss: 0.82383290

118it [00:26,  4.41it/s]
```

```
Epoch 51 of 100
Generator loss: 1.65603752, Discriminator loss: 0.84499193

118it [00:25,  4.54it/s]

Epoch 52 of 100
Generator loss: 1.62368114, Discriminator loss: 0.85223950

118it [00:25,  4.67it/s]

Epoch 53 of 100
Generator loss: 1.55585768, Discriminator loss: 0.89431892

118it [00:25,  4.65it/s]

Epoch 54 of 100
Generator loss: 1.59321513, Discriminator loss: 0.88365322

118it [00:25,  4.67it/s]

Epoch 55 of 100
Generator loss: 1.60118935, Discriminator loss: 0.89448089

118it [00:25,  4.67it/s]

Epoch 56 of 100
Generator loss: 1.57526695, Discriminator loss: 0.88796601

118it [00:25,  4.69it/s]

Epoch 57 of 100
Generator loss: 1.62836117, Discriminator loss: 0.86869409

118it [00:29,  4.03it/s]

Epoch 58 of 100
Generator loss: 1.56424481, Discriminator loss: 0.90864033

118it [00:27,  4.28it/s]

Epoch 59 of 100
Generator loss: 1.58526596, Discriminator loss: 0.87613403

118it [00:28,  4.17it/s]

Epoch 60 of 100
Generator loss: 1.52161574, Discriminator loss: 0.91021602

118it [00:26,  4.38it/s]

Epoch 61 of 100
Generator loss: 1.51836001, Discriminator loss: 0.91460216

118it [00:25,  4.67it/s]

Epoch 62 of 100
Generator loss: 1.52771683, Discriminator loss: 0.91601816

118it [00:24,  4.84it/s]
```

```
Epoch 63 of 100
Generator loss: 1.55184626, Discriminator loss: 0.90790288

118it [00:23,  4.99it/s]

Epoch 64 of 100
Generator loss: 1.51838174, Discriminator loss: 0.91162333

118it [00:25,  4.67it/s]

Epoch 65 of 100
Generator loss: 1.54813432, Discriminator loss: 0.90119194

118it [00:26,  4.42it/s]

Epoch 66 of 100
Generator loss: 1.59452512, Discriminator loss: 0.89543752

118it [00:24,  4.73it/s]

Epoch 67 of 100
Generator loss: 1.55837267, Discriminator loss: 0.90915580

118it [00:24,  4.83it/s]

Epoch 68 of 100
Generator loss: 1.53145211, Discriminator loss: 0.91264296

118it [00:24,  4.86it/s]

Epoch 69 of 100
Generator loss: 1.52301974, Discriminator loss: 0.91983768

118it [00:24,  4.87it/s]

Epoch 70 of 100
Generator loss: 1.54421294, Discriminator loss: 0.89149651

118it [00:26,  4.48it/s]

Epoch 71 of 100
Generator loss: 1.57269332, Discriminator loss: 0.89512495

118it [00:27,  4.27it/s]

Epoch 72 of 100
Generator loss: 1.53277545, Discriminator loss: 0.91472891

118it [00:31,  3.76it/s]

Epoch 73 of 100
Generator loss: 1.48078786, Discriminator loss: 0.93550708

118it [00:25,  4.68it/s]

Epoch 74 of 100
Generator loss: 1.49389724, Discriminator loss: 0.92044977

118it [00:25,  4.61it/s]
```

```
Epoch 75 of 100
Generator loss: 1.50201782, Discriminator loss: 0.93656620

118it [00:24,  4.73it/s]

Epoch 76 of 100
Generator loss: 1.49686191, Discriminator loss: 0.92345280

118it [00:25,  4.63it/s]

Epoch 77 of 100
Generator loss: 1.48998355, Discriminator loss: 0.93137390

118it [00:25,  4.57it/s]

Epoch 78 of 100
Generator loss: 1.46482457, Discriminator loss: 0.95430436

118it [00:29,  4.04it/s]

Epoch 79 of 100
Generator loss: 1.44150949, Discriminator loss: 0.95308710

118it [00:27,  4.24it/s]

Epoch 80 of 100
Generator loss: 1.46326796, Discriminator loss: 0.92909782

118it [00:24,  4.75it/s]

Epoch 81 of 100
Generator loss: 1.46234879, Discriminator loss: 0.94469263

118it [00:25,  4.59it/s]

Epoch 82 of 100
Generator loss: 1.46724195, Discriminator loss: 0.93962512

118it [00:23,  5.02it/s]

Epoch 83 of 100
Generator loss: 1.42198136, Discriminator loss: 0.96211639

118it [00:23,  4.98it/s]

Epoch 84 of 100
Generator loss: 1.42722088, Discriminator loss: 0.95936511

118it [00:25,  4.55it/s]

Epoch 85 of 100
Generator loss: 1.40960123, Discriminator loss: 0.96557617

118it [00:24,  4.74it/s]

Epoch 86 of 100
Generator loss: 1.41691896, Discriminator loss: 0.97293112

118it [00:23,  4.95it/s]
```

```
Epoch 87 of 100
Generator loss: 1.37999171, Discriminator loss: 0.97815017

118it [00:23,  5.00it/s]

Epoch 88 of 100
Generator loss: 1.43002717, Discriminator loss: 0.95671900

118it [00:23,  5.10it/s]

Epoch 89 of 100
Generator loss: 1.39904070, Discriminator loss: 0.97419283

118it [00:27,  4.36it/s]

Epoch 90 of 100
Generator loss: 1.41083335, Discriminator loss: 0.96994813

118it [00:25,  4.66it/s]

Epoch 91 of 100
Generator loss: 1.41515271, Discriminator loss: 0.96993881

118it [00:23,  5.02it/s]

Epoch 92 of 100
Generator loss: 1.39114712, Discriminator loss: 0.97714430

118it [00:23,  4.97it/s]

Epoch 93 of 100
Generator loss: 1.37084920, Discriminator loss: 0.97657584

118it [00:23,  5.02it/s]

Epoch 94 of 100
Generator loss: 1.38915571, Discriminator loss: 0.98580016

118it [00:23,  5.02it/s]

Epoch 95 of 100
Generator loss: 1.37325205, Discriminator loss: 0.98559688

118it [00:22,  5.20it/s]

Epoch 96 of 100
Generator loss: 1.36980436, Discriminator loss: 0.99146896

118it [00:22,  5.27it/s]

Epoch 97 of 100
Generator loss: 1.38177992, Discriminator loss: 0.98015804

118it [00:23,  5.00it/s]

Epoch 98 of 100
Generator loss: 1.38730154, Discriminator loss: 0.98014543

118it [00:23,  4.93it/s]
```

```
Epoch 99 of 100
Generator loss: 1.38026452, Discriminator loss: 0.98306672

118it [00:24,  4.89it/s]

Epoch 100 of 100
Generator loss: 1.36594954, Discriminator loss: 1.00223202
```

[ ]:
```python
print('DONE TRAINING')
torch.save(generator.state_dict(), 'outputs/generator.pth')
```

```
DONE TRAINING
```

[ ]:
```python
# save the generated images as GIF file
imgs = [np.array(to_pil_image(img)) for img in images]
imageio.mimsave('outputs/generator_images.gif', imgs)

np.savetxt("Generator loss.txt", np.array(losses_g))
np.savetxt("Discriminator loss.txt", np.array(losses_d))
```

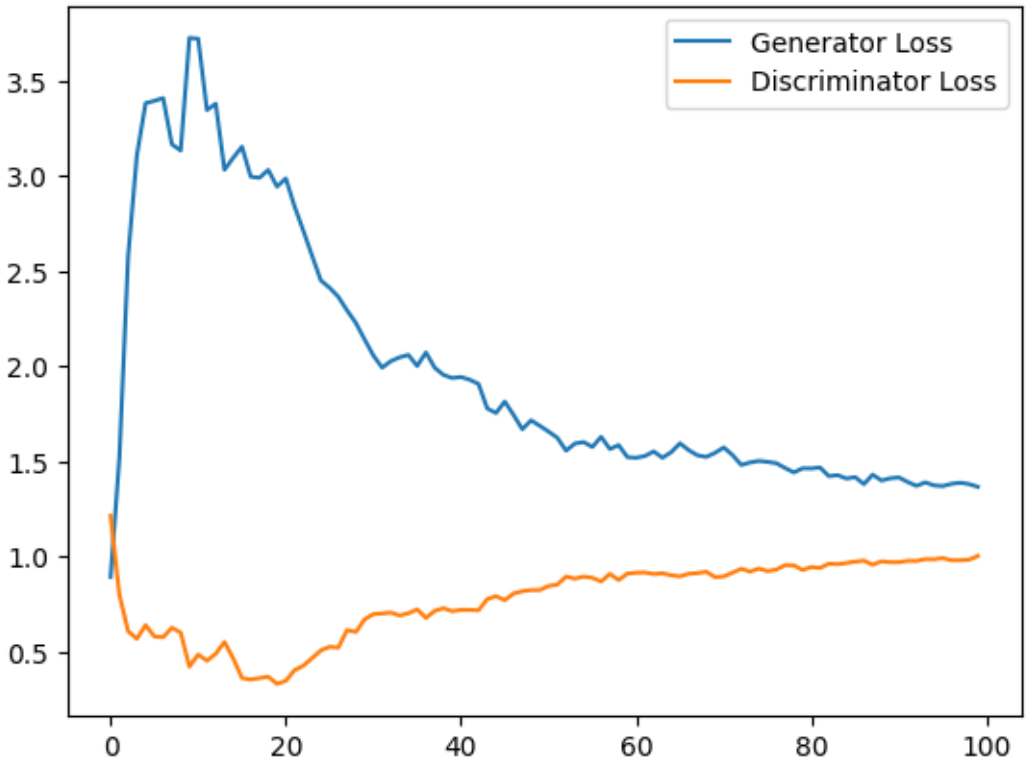
[ ]:
```python
# # plot and save the generator and discriminator loss
# plt.figure()
# plt.plot(losses_g, label='Generator loss')
# plt.plot(losses_d, label='Discriminator Loss')
# plt.legend()
# plt.savefig('outputs/loss.png')
```

[ ]:
```python
import numpy as np
import matplotlib.pyplot as plt

disc = np.loadtxt("./Discriminator loss.txt")
gen = np.loadtxt("./Generator loss.txt")

plt.plot(gen, label='Generator Loss')
plt.plot(disc, label='Discriminator Loss')
plt.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x1a42ef57e80>
```


Generator Loss
Discriminator Loss
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
20
40
60
80
100

[ ]: