Generating Images with VAEs and GANs

- In this assignment, you will implement two models: a Variational Auto Encoder (VAE) and a Generative Adversarial Network (GAN), for the task of image generation.
- You will train these models on Fashion-MNIST, a synthetic dataset including the grayscale images of shoes, clothes, hats, etc.
- For each task, we only provide abstract instructions. You have the freedom to implement the model from scratch and customize helper functions for your convenience.
- We will primarily grade your work based on the quality of generated images by visualization.
- The GAN models usually need long-term training to produce high quality images. If you need GPUs to accelerate your experiment, please consider using Google's colab which provides free GPU runtime.

Vartional Auto Encoder (VAE)

- A VAE is an image generation model following the design of an auto-encoder, consisting of an encoder and a decoder trained by minimizing the reconstruction error. Unlike conventional Autoencoders, in which the bottleneck is constructed as reducing the capacity of the latent block, VAEs parameterize the latent space with a prior distribution, thereby providing a statistical interpretation of the reconstruction process.
- A VAE model optimizes the Evidence Lower Bound (ELBO) as follows:
$$P(x) \geq E_{q_\theta(z|x)}[q_\phi(x|z)] + D_{KL}\left(q_\theta\left(z|x\right)||p\left(z\right)\right)$$
where q_θ, q_ϕ are instantiated as encoder and decoder, respectively. $p(z)$ is the prior distribution, which is usually chosen as Normal distribution $\mathcal{N}(0, 1)$
- After training VAE, you are expected to generate two diversified samples (function privided), which should hold similar content as the input.

```
"""
Import PyTorch libraries.
"""
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
from torchvision.utils import save_image
import matplotlib.pyplot as plt
import numpy as np
import random
```

```
"""
Enable CUDA if the GPU is available
"""
if torch.cuda.is_available():
  device = torch.device('cuda')
else:
  device = torch.device('cpu')


"""
Define the dataloader for the Fashion MNIST dataset.
"""
train_set = torchvision.datasets.FashionMNIST("./data", download=True, transform=
                                                transforms.Compose([transforms.ToTens
test_set = torchvision.datasets.FashionMNIST("./data", download=True, train=False, tra
                                               transforms.Compose([transforms.ToTenso
train_loader = torch.utils.data.DataLoader(train_set,
                                           batch_size=100)
test_loader = torch.utils.data.DataLoader(test_set,
                                          batch_size=100)
def output_label(label):
    output_mapping = {
                 0: "T-shirt/Top",
                 1: "Trouser",
                 2: "Pullover",
                 3: "Dress",
                 4: "Coat",
                 5: "Sandal",
                 6: "Shirt",
                 7: "Sneaker",
                 8: "Bag",
                 9: "Ankle Boot"
                 }
    input = (label.item() if type(label) == torch.Tensor else label)
    return output_mapping[input]
image, label = next(iter(train_set))
plt.imshow(image.squeeze(), cmap="gray")
print(output_label(label))
print(label)
print(image.shape)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-ima
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-ima
```

26422272/? [00:04<00:00, 9060244.39it/s]

```
Extracting ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ./data/FashionMI

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-lab
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-lab
```

29696/? [00:00<00:00, 39850.35it/s]

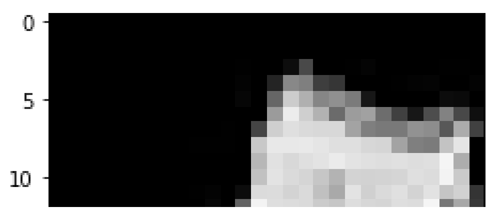
```
Extracting ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ./data/FashionMI

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-imag
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-imag
```

4422656/? [00:02<00:00, 3215173.46it/s]

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

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labe
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labe
```

6144/? [00:00<00:00, 155221.08it/s]

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

Ankle Boot
9
torch.Size([1, 28, 28])
```



Define VAE Model (TODO) [20 points]

- Define encoder, decoder, and reparameterazation module
- Implement the forward pass of the VAE model

```
"""
A Convolutional Variational Autoencoder
"""
class VAE(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_layer = nn.Linear(784, 392)
        self.linear_down_m = nn.Linear(392, 20)
        self.linear_down_v = nn.Linear(392, 20)
        self.linear_up1 = nn.Linear(20, 392)
        self.linear_up2 = nn.Linear(392, 784)
        self.sigmoid_activation = nn.Sigmoid()

    def encode(self, x):
```

```
        x = F.relu(self.input_layer(x))
        mean = self.linear_down_m(x)
        logvar = self.linear_down_v(x)
        return mean, logvar

    def reparameterize(self, mean, var):
        logvar = var.exp()
        sample = torch.FloatTensor(mean.size()).normal_().to(device)
        out = sample.mul(logvar).add(mean)
        return out

    def decode(self, sample):
        x = F.relu(self.linear_up1(sample))
        out = self.sigmoid_activation(self.linear_up2(x))
        return out

    def forward(self, x):
        mean, logvar = self.encode(x)
        sample = self.reparameterize(mean, logvar)
        decoded = self.decode(sample)
        return decoded, mean, logvar
```

Complete the Training Loop (TODO) [15 points]

- Implement the complete training loop for the VAE model

```
"""
Initialize Hyperparameters
"""
batch_size = 128
learning_rate = 1e-3
num_epochs = 20

"""
Initialize the network and the Adam optimizer
"""
model = VAE().to(device)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

"""
Training the network for a given number of epochs
"""
for epoch in range(num_epochs):
    model.train()
    train_loss1 = 0
    train_loss2 = 0
    for _, data in enumerate(train_loader):
        image, _ = data
        image = image.view(image.size(0), -1).to(device)
        optimizer.zero_grad()
        out, mean, logvar = model(image)
```

```
        out, mean, logvar = model(image)
        image_loss = nn.MSELoss(reduction='sum')(out, image)
        kloss = -0.5 * torch.sum(1 + logvar - mean.pow(2) - logvar.exp())
        loss = image_loss + kloss
        loss.backward()
        optimizer.step()
        train_loss1 += image_loss
        train_loss2 += kloss

    print('Epoch {}/{}, loss {:.4f}'.format(epoch, num_epochs, (train_loss1+train_loss
```

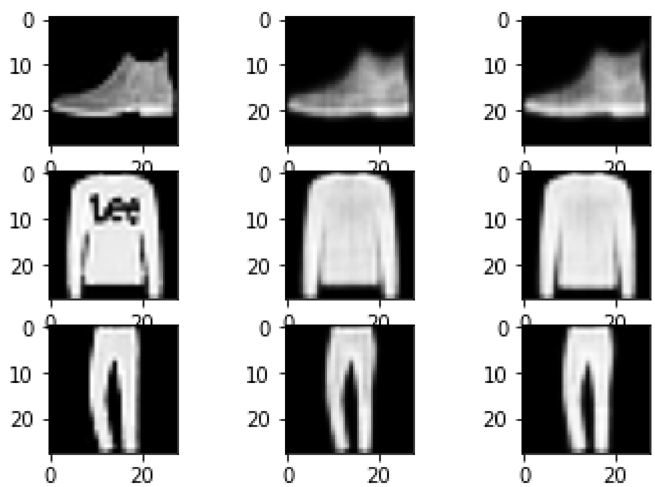
```
Epoch 0/20, loss 35.6633
Epoch 1/20, loss 24.6583
Epoch 2/20, loss 22.6981
Epoch 3/20, loss 21.8235
Epoch 4/20, loss 21.3217
Epoch 5/20, loss 20.9933
Epoch 6/20, loss 20.7431
Epoch 7/20, loss 20.5362
Epoch 8/20, loss 20.3614
Epoch 9/20, loss 20.2149
Epoch 10/20, loss 20.1003
Epoch 11/20, loss 19.9962
Epoch 12/20, loss 19.9068
Epoch 13/20, loss 19.8332
Epoch 14/20, loss 19.7591
Epoch 15/20, loss 19.6899
Epoch 16/20, loss 19.6402
Epoch 17/20, loss 19.5820
Epoch 18/20, loss 19.5392
Epoch 19/20, loss 19.4910
```

Visualizing the output

- Display the VAE output by running multiple samples of the latent space.
- You can modify this function accordingly to fit your implementation

```
"""
The following part takes two images from test loader to feed into the VAE.
Both the original image and generated image(s) from the distribution are shown.
"""

import matplotlib.pyplot as plt
import numpy as np

model.eval()
with torch.no_grad():
    imgs, _ = list(test_loader)[0]
    imgs = imgs.to(device)
    new_img = imgs.view(imgs.size(0), -1)
```

```
fig, ax = plt.subplots(3,3)
ax = ax.reshape(-1)
for i in range(3):
  for j in range(3):
    if j == 0:
      # input image
      img_i = np.transpose(imgs[i].cpu().numpy(), [1,2,0])
      ax[i*3+j].imshow(np.squeeze(img_i), cmap = 'gray')
    else:
      # vae generation results
      out, _, _ = model(new_img)
      out = out.cpu().numpy()[i].reshape(28,28)
      ax[i*3+j].imshow(out, cmap = 'gray')
plt.show()
```



Generative Adversarial Network (GAN)

- A GAN is a generation model trained to convert the samples from prior distribution z to the target domains x, e.g., images or text, in an unsupervised fashion.
- A GAN consists of a generator (G) and discriminator (D) model, where G is trained to produce realistic samples of the target domain, while the D learns to identified the samples from generator and the real domain, and serve as the supervision to optimize the G.
- GAN training typically follows two iterative steps:

  1. $\max_D \log(D(x)) + \log(1 - D(G(z)))$
  2. $\max_G \log(D(G(z)))$

Define GAN Model (TODO) [20 points]

- Define the generator and discriminator for the GAN model

```
start_size = 64
```

```
image_size = 784
int_size = 256
batch_size = 100

"""
Define the generator
"""
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.gen_seq = nn.Sequential(
            nn.Linear(start_size, int_size),
            nn.ReLU(),
            nn.Linear(int_size, int_size),
            nn.ReLU(),
            nn.Linear(int_size, image_size),
            nn.Tanh()
        )
    def forward(self, input):
        output = self.gen_seq(input)
        return output

"""
Define the discriminator
"""
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.disc_seq = nn.Sequential(
            nn.Linear(image_size, int_size),
            nn.LeakyReLU(0.2),
            nn.Linear(int_size, int_size),
            nn.LeakyReLU(0.2),
            nn.Linear(int_size, 1),
            nn.Sigmoid()
        )

    def forward(self, input):
        output = self.disc_seq(input)
        return output
```

Complete the Training Loop (TODO) [15 points]

- Implement the complete training loop for the GAN model
- You can monitor the loss curve between generator and discriminator to tune the hyperparamters

- The loss of generator and discriminator converge toward some non-zero values at the end.

```
"""
Initialize Hyperparameters
"""
learning_rate = 0.0002
num_epochs = 300
"""
Initialize the network and the Adam optimizer
"""
generator = Generator().to(device)
discriminator = Discriminator().to(device)
loss_func = nn.BCELoss()
disc_optimizer = torch.optim.Adam(discriminator.parameters(), lr=learning_rate)
gen_optimizer = torch.optim.Adam(generator.parameters(), lr=learning_rate)

disc_losses = []
gen_losses = []
DofXs = []
GofDofXs = []

for epoch in range(num_epochs):
    for i, (images, label) in enumerate(train_loader):
        images = images.reshape(batch_size, -1).to(device)

        #discriminator
        rlabels = torch.ones(batch_size, 1).to(device)
        flabels = torch.zeros(batch_size, 1).to(device)
        out = discriminator(images)
        discriminator_rloss = loss_func(out, rlabels)
        DofX = out

        z = torch.randn(batch_size, start_size).to(device)
        fakes = generator(z)
        out = discriminator(fakes)
        discriminator_floss = loss_func(out, flabels)
        GofDofX = out

        discriminator_loss = discriminator_rloss + discriminator_floss

        disc_optimizer.zero_grad()
        gen_optimizer.zero_grad()
        discriminator_loss.backward()
        disc_optimizer.step()

        #generator
        z = torch.randn(batch_size, start_size).to(device)
        fakes = generator(z)
        zlabels = torch.ones(batch_size, 1).to(device)
        generator_loss = loss_func(discriminator(fakes), zlabels)
```

```
    disc_optimizer.zero_grad()
    gen_optimizer.zero_grad()
    generator_loss.backward()
    gen_optimizer.step()

    if (i+1) % len(train_loader) == 0:
        disc_losses.append(discriminator_loss.item())
        gen_losses.append(generator_loss.item())
        DofXs.append(DofX.mean().item())
        GofDofXs.append(GofDofX.mean().item())
        print('Epoch {}/{}, disc_loss: {:.4f}, gen_loss: {:.4f}, D(x): {:.4f}, D(
              .format(epoch, num_epochs, discriminator_loss.item(), generator_loss
```

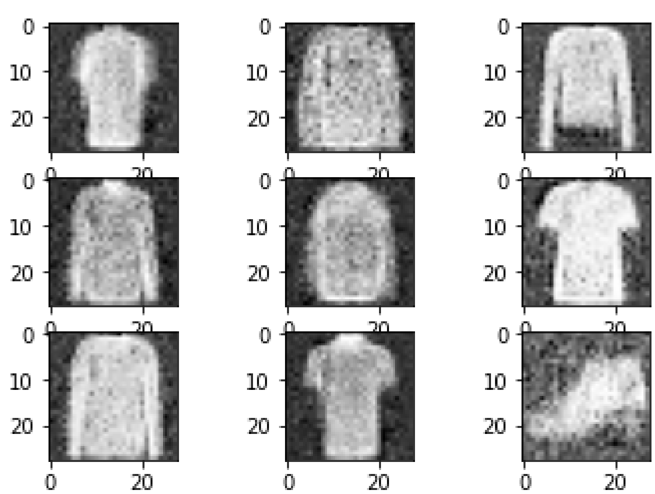
```
Epoch 240/300, disc_loss: 0.9709, gen_loss: 1.5238, D(x): 0.6672, D(G(x)): 0.265
Epoch 241/300, disc_loss: 0.9592, gen_loss: 1.7260, D(x): 0.6908, D(G(x)): 0.301
Epoch 242/300, disc_loss: 0.8626, gen_loss: 1.7123, D(x): 0.7225, D(G(x)): 0.273
Epoch 243/300, disc_loss: 0.7730, gen_loss: 2.1976, D(x): 0.7064, D(G(x)): 0.212
Epoch 244/300, disc_loss: 1.0898, gen_loss: 2.1009, D(x): 0.6341, D(G(x)): 0.248
Epoch 245/300, disc_loss: 0.8978, gen_loss: 1.8393, D(x): 0.6734, D(G(x)): 0.192
Epoch 246/300, disc_loss: 0.9226, gen_loss: 1.8197, D(x): 0.7173, D(G(x)): 0.304
Epoch 247/300, disc_loss: 0.9041, gen_loss: 1.6426, D(x): 0.6764, D(G(x)): 0.252
Epoch 248/300, disc_loss: 1.1963, gen_loss: 2.0043, D(x): 0.6264, D(G(x)): 0.274
Epoch 249/300, disc_loss: 0.9203, gen_loss: 2.1373, D(x): 0.6891, D(G(x)): 0.250
Epoch 250/300, disc_loss: 0.8779, gen_loss: 1.6488, D(x): 0.7002, D(G(x)): 0.279
Epoch 251/300, disc_loss: 0.9717, gen_loss: 1.9046, D(x): 0.6728, D(G(x)): 0.268
Epoch 252/300, disc_loss: 0.8281, gen_loss: 1.9360, D(x): 0.7560, D(G(x)): 0.302
Epoch 253/300, disc_loss: 0.7616, gen_loss: 1.9395, D(x): 0.7432, D(G(x)): 0.242
Epoch 254/300, disc_loss: 1.1080, gen_loss: 1.4316, D(x): 0.6817, D(G(x)): 0.324
Epoch 255/300, disc_loss: 0.7856, gen_loss: 1.9725, D(x): 0.6817, D(G(x)): 0.164
Epoch 256/300, disc_loss: 1.0327, gen_loss: 1.6056, D(x): 0.6816, D(G(x)): 0.308
Epoch 257/300, disc_loss: 0.7766, gen_loss: 1.5415, D(x): 0.7382, D(G(x)): 0.283
Epoch 258/300, disc_loss: 0.7791, gen_loss: 1.7538, D(x): 0.7779, D(G(x)): 0.291
Epoch 259/300, disc_loss: 1.0993, gen_loss: 1.6785, D(x): 0.7320, D(G(x)): 0.354
Epoch 260/300, disc_loss: 0.7157, gen_loss: 2.2125, D(x): 0.7440, D(G(x)): 0.221
Epoch 261/300, disc_loss: 0.9472, gen_loss: 2.3232, D(x): 0.6360, D(G(x)): 0.222
Epoch 262/300, disc_loss: 0.6461, gen_loss: 1.9910, D(x): 0.7452, D(G(x)): 0.197
Epoch 263/300, disc_loss: 0.8125, gen_loss: 1.7770, D(x): 0.7097, D(G(x)): 0.226
Epoch 264/300, disc_loss: 0.9019, gen_loss: 1.8847, D(x): 0.6926, D(G(x)): 0.233

Epoch 265/300, disc_loss: 0.8331, gen_loss: 1.8891, D(x): 0.7316, D(G(x)): 0.252
Epoch 266/300, disc_loss: 0.7838, gen_loss: 1.7998, D(x): 0.7455, D(G(x)): 0.259
Epoch 267/300, disc_loss: 0.9543, gen_loss: 1.7060, D(x): 0.6724, D(G(x)): 0.249
Epoch 268/300, disc_loss: 0.8830, gen_loss: 2.1059, D(x): 0.6941, D(G(x)): 0.245
Epoch 269/300, disc_loss: 0.8783, gen_loss: 1.8287, D(x): 0.6860, D(G(x)): 0.230
Epoch 270/300, disc_loss: 1.0270, gen_loss: 1.3826, D(x): 0.7199, D(G(x)): 0.355
Epoch 271/300, disc_loss: 0.8477, gen_loss: 2.0885, D(x): 0.7403, D(G(x)): 0.280
Epoch 272/300, disc_loss: 0.7030, gen_loss: 1.8038, D(x): 0.7415, D(G(x)): 0.219
Epoch 273/300, disc_loss: 0.9698, gen_loss: 1.8164, D(x): 0.6632, D(G(x)): 0.253
Epoch 274/300, disc_loss: 0.8587, gen_loss: 1.7720, D(x): 0.7488, D(G(x)): 0.330
Epoch 275/300, disc_loss: 0.8598, gen_loss: 1.6185, D(x): 0.6891, D(G(x)): 0.224
Epoch 276/300, disc_loss: 0.9355, gen_loss: 1.8444, D(x): 0.6949, D(G(x)): 0.268
Epoch 277/300, disc_loss: 1.0768, gen_loss: 1.8741, D(x): 0.7328, D(G(x)): 0.387
Epoch 278/300, disc_loss: 0.8365, gen_loss: 1.7646, D(x): 0.6982, D(G(x)): 0.260
Epoch 279/300, disc_loss: 0.8664, gen_loss: 1.6122, D(x): 0.6776, D(G(x)): 0.226
Epoch 280/300, disc_loss: 0.9077, gen_loss: 1.5876, D(x): 0.7140, D(G(x)): 0.294
```

```
Epoch 281/300, disc_loss: 0.8642, gen_loss: 1.8355, D(x): 0.7247, D(G(x)): 0.276
Epoch 282/300, disc_loss: 0.8788, gen_loss: 1.6858, D(x): 0.7614, D(G(x)): 0.328
Epoch 283/300, disc_loss: 0.9690, gen_loss: 1.7742, D(x): 0.7249, D(G(x)): 0.343
Epoch 284/300, disc_loss: 0.8060, gen_loss: 2.0578, D(x): 0.7346, D(G(x)): 0.257
Epoch 285/300, disc_loss: 1.0217, gen_loss: 1.5658, D(x): 0.7010, D(G(x)): 0.342
Epoch 286/300, disc_loss: 0.9670, gen_loss: 1.4514, D(x): 0.7311, D(G(x)): 0.344
Epoch 287/300, disc_loss: 0.9032, gen_loss: 1.6142, D(x): 0.7897, D(G(x)): 0.380
Epoch 288/300, disc_loss: 0.9412, gen_loss: 1.7596, D(x): 0.7114, D(G(x)): 0.301
Epoch 289/300, disc_loss: 0.7955, gen_loss: 1.7024, D(x): 0.8027, D(G(x)): 0.329
Epoch 290/300, disc_loss: 0.6479, gen_loss: 1.8811, D(x): 0.7693, D(G(x)): 0.225
Epoch 291/300, disc_loss: 0.8711, gen_loss: 1.7604, D(x): 0.7001, D(G(x)): 0.285
Epoch 292/300, disc_loss: 0.7225, gen_loss: 1.9267, D(x): 0.7543, D(G(x)): 0.256
Epoch 293/300, disc_loss: 0.9898, gen_loss: 1.4098, D(x): 0.7146, D(G(x)): 0.331
Epoch 294/300, disc_loss: 0.8634, gen_loss: 1.6264, D(x): 0.7213, D(G(x)): 0.293
Epoch 295/300, disc_loss: 1.0210, gen_loss: 1.6315, D(x): 0.7244, D(G(x)): 0.350
Epoch 296/300, disc_loss: 0.9259, gen_loss: 1.9647, D(x): 0.7298, D(G(x)): 0.324
Epoch 297/300, disc_loss: 0.9731, gen_loss: 1.4594, D(x): 0.6752, D(G(x)): 0.274
```

Visualizing the output

- Display the GAN output by running multiple samples of the latent space.
- You can modify this function accordingly to fit your implementation

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(3,3)
ax = ax.reshape(-1)
for i in range(9):
  z = torch.randn(1, latent_size).to(device)
  fake = generator(z)
  fake = fake.reshape((-1, 28,28)).squeeze(0).detach()
  ax[i].imshow(fake.cpu().data.numpy(), cmap = 'gray')
plt.show()
```



3s completed at 12:18 AM