Lab 1: Variational Autoencoders and Generative Adversarial Networks

Course: Generative AI for computer vision
Level: M2 (Master's 2)
Duration: 4 hours

Objective

In this lab, you will:

1. Implement a **Variational Autoencoder (VAE)** to learn how to reconstruct images by sampling from a latent space.
2. Understand how the components of a VAE can be adapted to conceptualize and implement a **Generative Adversarial Network (GAN)**.
3. Compare the roles of VAEs and GANs in generative modeling.
4. Reflect on the strengths and weaknesses of each approach.

Background

Variational Autoencoders (VAEs)

A **VAE** is a generative model that reconstructs input data by encoding it into a latent space and then decoding it back. The latent space is a compressed representation that captures the most important features of the input.

The key elements of a VAE:

- **Encoder:** Compresses input data into a latent distribution (mean and variance).
- **Latent Space:** The encoded representation space.
- **Reparameterization Trick:** Allows gradients to flow through the stochastic latent space.
- **Decoder:** Reconstructs the input from the latent space.

Generative Adversarial Networks (GANs)

A **GAN** is a generative model consisting of two components:

- **Generator:** Produces fake data from random noise (latent space).
- **Discriminator:** Distinguishes between real and fake data.

GANs are trained using an adversarial process where the generator tries to fool the discriminator, and the discriminator tries to detect fake data.

Reversing VAEs into GANs

In a conceptual sense:

- A **VAE decoder** can be seen as a **GAN generator**.
- The **GAN discriminator** replaces the **VAE encoder** by determining the quality of generated samples instead of encoding input data.

Part 1: Implementing a Variational Autoencoder (VAE)

Instructions

1. **Download the MNIST dataset** of handwritten digits. (as help code provided below for this first session)
2. Implement the following components in TensorFlow/Keras:
    - **Encoder:** Compresses images into a latent space of dimension latent_dim.
    - **Reparameterization Trick:** Samples latent variables from a Gaussian distribution.
    - **Decoder:** Reconstructs images from the latent space.
3. Define the **VAE loss function**:
    - Reconstruction Loss (binary cross-entropy).
    - KL Divergence Loss (regularizes the latent space).
4. Train the VAE and visualize the reconstructed images.

Questions

1. Why do we use the **reparameterization trick in VAEs**?
2. How does the **KL divergence loss** affect the latent space?
3. How does changing the latent space dimension (latent_dim) impact the reconstruction quality?

Part 2: From VAE to GAN

1. **Conceptual Discussion**:
    - Explain how the VAE decoder can be used as a GAN generator.
    - Discuss the differences between the VAE encoder and the GAN discriminator.
2. **Implement a GAN** using the VAE decoder as the generator (with no training step).

Deliverables

1. **Code Implementation** of the VAE and GAN.
2. **Answers to the Questions**.
3. **Visualizations** of reconstructed (VAE) and generated (GAN) images.

```
# For pedagogic purpose, the code is sequential here, it should be of form : function
definitions and archtitecture in different files.

import tensorflow as tf
from tensorflow.keras import layers, Model
import numpy as np
import matplotlib.pyplot as plt

# Load MNIST dataset
(x_train, _), (x_test, _) = tf.keras.datasets.mnist.load_data()
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
latent_dim = 2  # Latent space dimension

# Encoder
class Encoder(Model):
    def __init__(self, latent_dim):
        super(Encoder, self).__init__()
        self.flatten = layers.Flatten()
        self.dense1 = layers.Dense(256, activation="relu")
        self.mean = layers.Dense(latent_dim)
        self.log_var = layers.Dense(latent_dim)

    def call(self, x):
        x = self.flatten(x)
        x = self.dense1(x)
        mean = self.mean(x)
        log_var = self.log_var(x)
        return mean, log_var

# Reparameterization trick
def sample_z(mean, log_var):
    epsilon = tf.random.normal(shape=tf.shape(mean))
    return mean + tf.exp(0.5 * log_var) * epsilon

# Decoder
class Decoder(Model):
    def __init__(self):
        super(Decoder, self).__init__()
        self.dense1 = layers.Dense(256, activation="relu")
        self.dense2 = layers.Dense(28 * 28, activation="sigmoid")
        self.reshape = layers.Reshape((28, 28, 1))
```

```
    def call(self, z):
        z = self.dense1(z)
        z = self.dense2(z)
        return self.reshape(z)

# VAE
class VAE(Model):
    def __init__(self, encoder, decoder):
        super(VAE, self).__init__()
        self.encoder = encoder
        self.decoder = decoder

    def call(self, x):
        mean, log_var = self.encoder(x)
        z = sample_z(mean, log_var)
        reconstruction = self.decoder(z)
        return reconstruction, mean, log_var

encoder = Encoder(latent_dim)
decoder = Decoder()
vae = VAE(encoder, decoder)

# Loss Function
def vae_loss(x, reconstruction, mean, log_var):
    reconstruction_loss = tf.reduce_mean(
        tf.keras.losses.binary_crossentropy(x, reconstruction)
    )
    reconstruction_loss *= 28 * 28
    kl_divergence = -0.5 * tf.reduce_sum(1 + log_var - tf.square(mean) - tf.exp(log_var))
    return reconstruction_loss + kl_divergence

# Optimizer
optimizer = tf.keras.optimizers.Adam()

# Training Loop
@tf.function
def train_step(x):
    with tf.GradientTape() as tape:
        reconstruction, mean, log_var = vae(x)
        loss = vae_loss(x, reconstruction, mean, log_var)
    gradients = tape.gradient(loss, vae.trainable_variables)
    optimizer.apply_gradients(zip(gradients, vae.trainable_variables))
    return loss

# Training
```

```
epochs = 20
batch_size = 128
train_dataset =
tf.data.Dataset.from_tensor_slices(x_train).shuffle(60000).batch(batch_size)
for epoch in range(epochs):
    for step, x_batch in enumerate(train_dataset):
        loss = train_step(x_batch)
    print(f"Epoch {epoch + 1}, Loss: {loss.numpy()}")
```