TP1 Advanced Generative Adversarial Networks (GANs)

Part 1: CNN-based GAN

Question 1: What is Transpose Convolution and why do we use it Generator?

La convolution transposée, également connue sous le nom de déconvolution ou de convolution à pas fractionnaire, est une opération qui effectue l'inverse d'une convolution régulière. Elle est utilisée dans le générateur d'un GAN pour suréchantillonner le vecteur de bruit d'entrée afin de générer des images de plus haute résolution. L'opération de convolution transposée applique un noyau apprenable à la carte de caractéristiques d'entrée, ce qui augmente effectivement les dimensions spatiales des données. Cela permet au générateur de transformer des cartes de caractéristiques de basse résolution en images de haute résolution en apprenant le processus de suréchantillonnage de l'espace latent à l'espace de sortie.

Dans le code, la convolution transposée est utilisée dans le générateur comme suit :

```
layers.Conv2DTranspose(128, kernel_size=4, strides=2, padding="same")
layers.Conv2DTranspose(64, kernel_size=4, strides=2, padding="same")
layers.Conv2DTranspose(1, kernel_size=7, activation="sigmoid", padding="same")
```

Question 2: What are LeakyReLU and sigmoid and why do we use them?

LeakyReLU est une fonction d'activation qui permet un petit gradient négatif lorsque l'entrée est négative, contrairement à ReLU qui a un gradient nul pour les valeurs négatives. Cela aide à prévenir le problème de la disparition du gradient et à accélérer la convergence du modèle. LeakyReLU est couramment utilisé dans les réseaux de neurones convolutifs pour introduire une non-linéarité et favoriser la stabilité de l'entraînement.

Sigmoid est une fonction d'activation qui normalise les sorties entre 0 et 1. Elle est souvent utilisée dans la dernière couche du générateur pour produire des images avec des valeurs de pixel comprises entre 0 et 1.

Dans le code, LeakyReLU et sigmoid sont utilisés comme suit :

```
layers.LeakyReLU(alpha=0.2)
layers.Conv2DTranspose(1, kernel_size=7, activation="sigmoid", padding="same")
```

Ces fonctions d'activation sont utilisées pour améliorer la stabilité et la performance du modèle GAN.

Question 3: Read the code and comment every line.

```
import tensorflow as tf  # Importation de la bibliothèque TensorFlow
from tensorflow.keras import layers  # Importation des couches de Keras
import numpy as np  # Importation de NumPy pour les opérations numériques
import matplotlib.pyplot as plt  # Importation de Matplotlib pour la visualisation

# Chargement du dataset MNIST
(x_train, _), (_, _) = tf.keras.datasets.mnist.load_data()
x_train = x_train.astype("float32") / 255.0  # Normalisation des images entre 0 et
1
x_train = np.expand_dims(x_train, -1)  # Ajout d'une dimension pour les canaux

latent_dim = 100  # Dimension du vecteur latent

# Générateur CNN
def build_generator():
    model = tf.keras.Sequential([
```

```
        layers.Dense(7 * 7 * 256, input_dim=latent_dim),  # Couche dense pour
transformer le vecteur latent
        layers.Reshape((7, 7, 256)),  # Remodelage en une carte de caractéristiques
7x7 avec 256 canaux
        layers.Conv2DTranspose(128, kernel_size=4, strides=2, padding="same"),  #
Convolution transposée pour augmenter la résolution
        layers.LeakyReLU(alpha=0.2),  # Activation LeakyReLU
        layers.Conv2DTranspose(64, kernel_size=4, strides=2, padding="same"),  #
Convolution transposée pour augmenter la résolution
        layers.LeakyReLU(alpha=0.2),  # Activation LeakyReLU
        layers.Conv2DTranspose(1, kernel_size=7, activation="sigmoid",
padding="same")  # Convolution transposée pour générer l'image finale
    ])
    return model

# Discriminateur CNN
def build_discriminator():
    model = tf.keras.Sequential([
        layers.Conv2D(64, kernel_size=4, strides=2, padding="same", input_shape=
(28, 28, 1)),  # Convolution pour extraire les caractéristiques
        layers.LeakyReLU(alpha=0.2),  # Activation LeakyReLU
        layers.Conv2D(128, kernel_size=4, strides=2, padding="same"),  #
Convolution pour extraire les caractéristiques
        layers.LeakyReLU(alpha=0.2),  # Activation LeakyReLU
        layers.Flatten(),  # Aplatissement des cartes de caractéristiques
        layers.Dense(1, activation="sigmoid")  # Couche dense pour produire une
probabilité de validité
    ])
    return model

# Modèle GAN
generator = build_generator()  # Construction du générateur
discriminator = build_discriminator()  # Construction du discriminateur
discriminator.compile(optimizer=tf.keras.optimizers.Adam(0.0002),
loss="binary_crossentropy", metrics=["accuracy"])  # Compilation du discriminateur
discriminator.trainable = False  # Fixation des poids du discriminateur pendant
l'entraînement du GAN

gan_input = layers.Input(shape=(latent_dim,))  # Entrée du GAN
gan_output = discriminator(generator(gan_input))  # Sortie du GAN
gan = tf.keras.Model(gan_input, gan_output)  # Construction du modèle GAN
gan.compile(optimizer=tf.keras.optimizers.Adam(0.0002), loss="binary_crossentropy")
# Compilation du GAN

# Entraînement du GAN
def train_gan(generator, discriminator, gan, epochs, batch_size=128):
    for epoch in range(epochs):
        for _ in range(batch_size):
            # Entraînement du discriminateur
            noise = tf.random.normal([batch_size, latent_dim])  # Génération de
bruit aléatoire
            fake_images = generator.predict(noise)  # Génération d'images factices
            real_images = x_train[np.random.randint(0, x_train.shape[0],
batch_size)]  # Sélection d'images réelles

            real_labels = tf.ones((batch_size, 1))  # Étiquettes pour les images
réelles
```

```
            fake_labels = tf.zeros((batch_size, 1))  # Étiquettes pour les images factices

            d_loss_real = discriminator.train_on_batch(real_images, real_labels)  # Entraînement sur les images réelles
            d_loss_fake = discriminator.train_on_batch(fake_images, fake_labels)  # Entraînement sur les images factices

            # Entraînement du générateur
            misleading_labels = tf.ones((batch_size, 1))  # Étiquettes trompeuses pour le générateur
            g_loss = gan.train_on_batch(noise, misleading_labels)  # Entraînement du générateur

        print(f"Epoch {epoch + 1}/{epochs}, D Loss: {d_loss_real[0] + d_loss_fake[0]}, G Loss: {g_loss}")  # Affichage des pertes

train_gan(generator, discriminator, gan, epochs=10)  # Entraînement du GAN

# Génération et visualisation des images
def generate_images(generator, n_images):
    noise = tf.random.normal([n_images, latent_dim])  # Génération de bruit aléatoire
    generated_images = generator.predict(noise)  # Génération d'images
    fig, axes = plt.subplots(1, n_images, figsize=(20, 4))  # Création de la figure pour la visualisation
    for i, img in enumerate(generated_images):
        axes[i].imshow(img.squeeze(), cmap="gray")  # Affichage de l'image
        axes[i].axis("off")  # Suppression des axes
    plt.show()  # Affichage de la figure
    plt.savefig('generated_images.png')  # Sauvegarde des images générées

generate_images(generator, 10)  # Génération et visualisation de 10 images
```