ДОДАТОК А

Вихідний код програми для досвідчених експериментів

```
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np
import matplotlib.pyplot as plt
import os
import time

# Load the MNIST dataset
(train_images, train_labels), (test_images,
                               test_labels) = tf.keras.datasets.mnist.load_data()
# Normalize the images to [-1, 1]
train_images = (train_images - 127.5) / 127.5
train_images = np.expand_dims(train_images, axis=-1)
BUFFER_SIZE = 60000
BATCH_SIZE = 256
NUM_CLASSES = 10
noise_dim = 100

train_dataset = tf.data.Dataset.from_tensor_slices(
    (train_images, train_labels)).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)


def make_generator_model():
    noise_input = layers.Input(shape=(noise_dim,))
    label_input = layers.Input(shape=(NUM_CLASSES,))
    combined_input = layers.concatenate([noise_input, label_input])

    x = layers.Dense(7*7*256, use_bias=False)(combined_input)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU()(x)

    x = layers.Reshape((7, 7, 256))(x)
    x = layers.Conv2DTranspose(128, (5, 5), strides=(
        1, 1), padding='same', use_bias=False)(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU()(x)

    x = layers.Conv2DTranspose(64, (5, 5), strides=(
        2, 2), padding='same', use_bias=False)(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU()(x)

    x = layers.Conv2DTranspose(1, (5, 5), strides=(
        2, 2), padding='same', use_bias=False, activation='tanh')(x)

    return tf.keras.Model([noise_input, label_input], x)


def make_discriminator_model():
    image_input = layers.Input(shape=(28, 28, 1))
    label_input = layers.Input(shape=(NUM_CLASSES,))
    label_embedding = layers.Dense(28 * 28)(label_input)
    label_embedding = layers.Reshape((28, 28, 1))(label_embedding)

    combined_input = layers.concatenate([image_input, label_embedding])

    x = layers.Conv2D(64, (5, 5), strides=(
        2, 2), padding='same')(combined_input)
    x = layers.LeakyReLU()(x)
    x = layers.Dropout(0.3)(x)

    x = layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same')(x)
    x = layers.LeakyReLU()(x)
    x = layers.Dropout(0.3)(x)

    x = layers.Flatten()(x)
    x = layers.Dense(1)(x)

    return tf.keras.Model([image_input, label_input], x)

generator_uniform = make_generator_model()
generator_normal = make_generator_model()
generator_exponential = make_generator_model()
```

```
discriminator = make_discriminator_model()

cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)


def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss


def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)


generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

# Initialize the optimizers with the variables
generator_optimizer.build(generator_uniform.trainable_variables +
                          generator_normal.trainable_variables                                    +
generator_exponential.trainable_variables)
discriminator_optimizer.build(discriminator.trainable_variables)

EPOCHS = 50


@tf.function
def train_step(images, labels, generator, noise_fn):
    batch_size = tf.shape(images)[0]
    noise = noise_fn([batch_size, noise_dim])
    labels_one_hot = tf.one_hot(labels, NUM_CLASSES)

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator([noise, labels_one_hot], training=True)

        real_output = discriminator([images, labels_one_hot], training=True)
        fake_output = discriminator(
            [generated_images, labels_one_hot], training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(
        gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(
        disc_loss, discriminator.trainable_variables)

    generator_optimizer.apply_gradients(
        zip(gradients_of_generator, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(
        zip(gradients_of_discriminator, discriminator.trainable_variables))

    return gen_loss, disc_loss


def  train(dataset,  epochs,  generator,  noise_fn,  generator_name,  gen_losses,  disc_losses,
save_interval=5):
    for epoch in range(epochs):
        start = time.time()

        epoch_gen_loss = []
        epoch_disc_loss = []

        for image_batch, label_batch in dataset:
            gen_loss, disc_loss = train_step(
                image_batch, label_batch, generator, noise_fn)
            epoch_gen_loss.append(gen_loss)
            epoch_disc_loss.append(disc_loss)

        gen_losses.append(np.mean(epoch_gen_loss))
        disc_losses.append(np.mean(epoch_disc_loss))

        if (epoch + 1) % save_interval == 0:
            generator.save_weights(
                f'{generator_name}_generator_epoch_{epoch+1}.h5')
            discriminator.save_weights(
                f'{generator_name}_discriminator_epoch_{epoch+1}.h5')
```

```
        print(f'Time for epoch {epoch + 1} is {time.time() - start:.2f} sec, '
              f'Gen Loss: {gen_losses[-1]:.4f}, Disc Loss: {disc_losses[-1]:.4f}')


def load_model(generator, discriminator, epoch, generator_name):
    generator.load_weights(f'{generator_name}_generator_epoch_{epoch}.h5')
    discriminator.load_weights(
        f'{generator_name}_discriminator_epoch_{epoch}.h5')


def exponential_noise(shape, rate=1.0):
    uniform_noise = tf.random.uniform(shape, minval=0, maxval=1)
    return -tf.math.log(1.0 - uniform_noise) / rate


def generate_and_save_images(model, epoch, noise, labels, num_images, generator_name):
    predictions = model([noise, labels], training=False)

    fig = plt.figure(figsize=(4, 4))

    for i in range(num_images):
        plt.subplot(4, 4, i+1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
        plt.axis('off')

    plt.savefig(f'{generator_name}_image_at_epoch_{epoch:04d}.png')
    plt.show()


# Number of images to generate
num_images_to_generate = 16

# Desired label for the generated images (for example, generating digit '5')
desired_label = 5
desired_labels = tf.one_hot(
    [desired_label] * num_images_to_generate, NUM_CLASSES)

# Check if model weights already exist
uniform_weights_exist = os.path.exists(
    f'generator_uniform_generator_epoch_{EPOCHS}.h5')
normal_weights_exist = os.path.exists(
    f'generator_normal_generator_epoch_{EPOCHS}.h5')
exponential_weights_exist = os.path.exists(
    f'generator_exponential_generator_epoch_{EPOCHS}.h5')

# Lists to store loss values for each generator
gen_losses_uniform = []
disc_losses_uniform = []
gen_losses_normal = []
disc_losses_normal = []
gen_losses_exponential = []
disc_losses_exponential = []

# Train or load generator with uniform noise
if not uniform_weights_exist:
    print("Training the generator with uniform noise...")
    train(train_dataset, EPOCHS, generator_uniform, tf.random.uniform,
          'generator_uniform', gen_losses_uniform, disc_losses_uniform)
else:
    print("Loading the pre-trained generator with uniform noise...")
    load_model(generator_uniform, discriminator, EPOCHS, 'generator_uniform')

# Train or load generator with normal noise
if not normal_weights_exist:
    print("Training the generator with normal noise...")
    train(train_dataset, EPOCHS, generator_normal, tf.random.normal,
          'generator_normal', gen_losses_normal, disc_losses_normal)
else:
    print("Loading the pre-trained generator with normal noise...")
    load_model(generator_normal, discriminator, EPOCHS, 'generator_normal')

# Train or load generator with exponential noise
if not exponential_weights_exist:
    print("Training the generator with exponential noise...")
    train(train dataset, EPOCHS, generator exponential, exponential noise,
          'generator_exponential', gen_losses_exponential, disc_losses_exponential)
else:
    print("Loading the pre-trained generator with exponential noise...")
```

```
    load_model(generator_exponential, discriminator,
               EPOCHS, 'generator_exponential')

# Generate seed for the specified number of images
seed_uniform = tf.random.uniform([num_images_to_generate, noise_dim])
seed_normal = tf.random.normal([num_images_to_generate, noise_dim])
seed_exponential = exponential_noise([num_images_to_generate, noise_dim])

# Choose which generator to use for generating images (e.g., uniform, normal, or exponential)
# 'generator_uniform', 'generator_normal', or 'generator_exponential'
chosen_generator_name = 'generator_uniform'
chosen_generator = {
    'generator_uniform': generator_uniform,
    'generator_normal': generator_normal,
    'generator_exponential': generator_exponential
}[chosen_generator_name]
chosen_seed = {
    'generator_uniform': seed_normal,
    'generator_normal': seed_normal,
    'generator_exponential': seed_normal
}[chosen_generator_name]

# Generate and save images
generate_and_save_images(chosen_generator, EPOCHS, chosen_seed,
                         desired_labels, num_images_to_generate, chosen_generator_name)

# Plotting the losses for uniform noise
plt.figure(figsize=(10, 5))
plt.plot(gen_losses_uniform, label='Generator Loss - Uniform')
plt.plot(disc_losses_uniform, label='Discriminator Loss - Uniform')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.savefig('generator_uniform_losses.png')
plt.show()

# Plotting the losses for normal noise
plt.figure(figsize=(10, 5))
plt.plot(gen_losses_normal, label='Generator Loss - Normal')
plt.plot(disc_losses_normal, label='Discriminator Loss - Normal')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.savefig('generator_normal_losses.png')
plt.show()

# Plotting the losses for exponential noise
plt.figure(figsize=(10, 5))
plt.plot(gen_losses_exponential, label='Generator Loss - Exponential')
plt.plot(disc_losses_exponential, label='Discriminator Loss - Exponential')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.savefig('generator_exponential_losses.png')
plt.show()
```

ДОДАТОК Б

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	103 с.
	Інші документи	
2.	Презентаційні матеріали	__ плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Білоконь В.А.			Формування альернативних підходів до генерації шумів у GAN-мережах	Шифр групи	Код напр./спец.
Перевірив		Рябова Н.В.				СШІм-22-2	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Філатов В.О.					