**Лабораторна робота №4: GAN. Фрактали**

**Виконав студент ФІТ 4-5**

**Тимошенко Іван**

**Звіт:**

1. Генеративно-зворотні мережі (GAN) для датасету Fashion MNIST:

Завдання: Розроблено GAN для створення реалістичних зображень одягу.

Реалізація: Використано TensorFlow і Keras для побудови моделі. Датасет Fashion MNIST використовується для навчання.

Результати: Виведено графіки функцій втрат та зображення згенерованого одягу.

1. Крижинка Коха:

Завдання: Реалізовано алгоритм побудови крижинки Коха з рекурсивною можливістю.

Реалізація: Використано рекурсію та векторну графіку для створення фрактальної форми.

Результати: Виведено зображення крижинки Коха з обраним рівнем рекурсії.

1. Килим Серпінського:

Завдання: Побудовано килим Серпінського.

Реалізація: Використано геометричні принципи для створення фрактальної структури.

Результати: Виведено зображення килима Серпінського.

**Висновок**

Завершено розробку трьох завдань у сфері генеративних моделей та фрактальної геометрії. Реалізовано генеративно-зворотні мережі для створення зображень одягу на основі датасету Fashion MNIST, а також побудовано крижинку Коха та килим Серпінського як приклади фрактальних структур. Виконання цих завдань дозволило продемонструвати використання різних алгоритмів та методів у машинному навчанні та комп'ютерній графіці.

**КОД**

**1.**

import tensorflow as tf

from tensorflow.keras import layers, models

import numpy as np

import matplotlib.pyplot as plt

# Завантаження датасету

(x\_train, \_), (\_, \_) = tf.keras.datasets.fashion\_mnist.load\_data()

# Нормалізація даних

x\_train = x\_train.astype('float32') / 255

x\_train = np.expand\_dims(x\_train, axis=-1) # Розширення зображень до третього виміру

# Завдання розмірності вхідного шуму для генератора

latent\_dim = 100

# Створення генератора

generator = models.Sequential([

layers.Dense(7 \* 7 \* 128, input\_dim=latent\_dim),

layers.LeakyReLU(alpha=0.2),

layers.Reshape((7, 7, 128)),

layers.Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same'),

layers.LeakyReLU(alpha=0.2),

layers.Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same'),

layers.LeakyReLU(alpha=0.2),

layers.Conv2D(1, (7, 7), activation='sigmoid', padding='same')

])

# Створення дискримінатора

discriminator = models.Sequential([

layers.Conv2D(64, (3, 3), strides=(2, 2), padding='same', input\_shape=[28, 28, 1]),

layers.LeakyReLU(alpha=0.2),

layers.Dropout(0.4),

layers.Conv2D(64, (3, 3), strides=(2, 2), padding='same'),

layers.LeakyReLU(alpha=0.2),

layers.Dropout(0.4),

layers.Flatten(),

layers.Dense(1, activation='sigmoid')

])

# З'єднання генератора та дискримінатора в єдину модель GAN

discriminator.compile(optimizer='adam', loss='binary\_crossentropy')

discriminator.trainable = False

gan\_input = tf.keras.Input(shape=(latent\_dim,))

fake\_image = generator(gan\_input)

gan\_output = discriminator(fake\_image)

gan = models.Model(gan\_input, gan\_output)

gan.compile(optimizer='adam', loss='binary\_crossentropy')

# Навчання моделі GAN

batch\_size = 128

iterations = 10000

for step in range(iterations):

noise = np.random.normal(0, 1, size=(batch\_size, latent\_dim))

generated\_images = generator.predict(noise)

real\_images = x\_train[np.random.randint(0, x\_train.shape[0], size=batch\_size)]

x\_combined = np.concatenate([real\_images, generated\_images]) # Виправлено

y\_combined = np.concatenate([np.ones((batch\_size, 1)), np.zeros((batch\_size, 1))])

d\_loss = discriminator.train\_on\_batch(x\_combined, y\_combined)

noise = np.random.normal(0, 1, size=(batch\_size, latent\_dim))

y\_mislabeled = np.ones((batch\_size, 1))

g\_loss = gan.train\_on\_batch(noise, y\_mislabeled)

if step % 100 == 0:

print(f"Step {step}/{iterations} | Discriminator Loss: {d\_loss} | Generator Loss: {g\_loss}")

# Візуалізація результату

n = 10 # кількість зображень для візуалізації

plt.figure(figsize=(20, 4))

for i in range(n):

ax = plt.subplot(2, n, i + 1)

plt.imshow(generated\_images[i].reshape(28, 28), cmap='gray')

plt.axis('off')

plt.show()

**2.**

import numpy as np

import matplotlib.pyplot as plt

def \_koch\_snowflake\_complex(order):

if order == 0:

return [complex(0, 0), complex(1, 0)]

else:

points = \_koch\_snowflake\_complex(order - 1)

new\_points = []

for i in range(len(points) - 1):

p1, p2 = points[i], points[i + 1]

one\_third = (p2 - p1) / 3

two\_third = 2 \* (p2 - p1) / 3

new\_points.extend([p1, p1 + one\_third, p1 + one\_third + one\_third \* np.exp(np.pi \* 1j / 3), p1 + two\_third])

new\_points.append(points[-1])

return new\_points

def koch\_snowflake(order):

points = \_koch\_snowflake\_complex(order)

x = np.real(points)

y = np.imag(points)

plt.figure(figsize=(8, 8))

plt.plot(x, y)

plt.title(f"Koch Snowflake (Order {order})")

plt.axis('equal')

plt.show()

# Використання функції для побудови крижинки Коха з рівнем рекурсії 3

koch\_snowflake(order=3)

**3.**

import numpy as np

import matplotlib.pyplot as plt

def sierpinski\_carpet(order, size):

# Перевірка базового випадку

if order == 0:

return np.ones((size, size))

# Зменшення розміру зображення в 3 рази

small\_square = sierpinski\_carpet(order - 1, size // 3)

# Створення пустої карти

carpet = np.zeros((size, size))

# Розташування маленьких квадратів всередині великого

for i in range(3):

for j in range(3):

if not (i == 1 and j == 1):

carpet[i \* size // 3: (i + 1) \* size // 3, j \* size // 3: (j + 1) \* size // 3] = small\_square

return carpet

# Визначення порядку і розміру килиму

order = 5

size = 243 # 3 \*\* 5

# Створення килиму Серпінського

carpet = sierpinski\_carpet(order, size)

# Відображення килиму Серпінського

plt.figure(figsize=(6, 6))

plt.imshow(carpet, cmap='binary', interpolation='none')

plt.axis('off')

plt.show()

**Скріни**



