Bachelor's Degree in Mathematics

Bachelor's Thesis

# Generative adversarial networks applied to Science

*Miguel Tajada Ferrer*

supervised by

Dr. Luis Martín Moreno

Faculty of Sciences. University of Zaragoza
Department of Condensed Matter
Academic Year 2022-2023

# Contents

# 1 Code for Section 4.1

```
from Bio import SeqIO
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LeakyReLU
from tensorflow.keras.optimizers.legacy import Adam

#Subroutines
from plot_loss_evolution import plot_loss
import ADN_vectores as adn_vect

# %%
from Bio import SeqIO

# Cargar las secuencias de ADN
records = list(SeqIO.parse("sequences.fasta", "fasta"))

# Imprimir el número de secuencias
print("El archivo contiene", len(records), "secuencias de ADN.")

# Calculate and print the length of each DNA sequence
for record in records:
    print("La secuencia contiene", len(record.seq), "nucleótidos (letras)")

# IDEA DE LA GAN
#
# Tomamos la secuencia del COVID que tiene 29903 letras y la cortamos en pedazos de 60.
Esos serán los datos de entrenamiento. Buscamos que el generador sea capaz de generar
otras secuencias de 60.



# Cargar las secuencias de ADN
records = list(SeqIO.parse("sequences.fasta", "fasta"))

# Preprocesar las secuencias para usarlas en la GAN
training_data = np.array([adn_vect.seq_to_vec(str(record.seq)[:60]) for record in records])
training_data = training_data.reshape(training_data.shape[0], -1)

input_dim = 100
seq_size = 60 * 4

def create_generator():
    model = Sequential()
    model.add(Dense(256, input_dim=input_dim))
```

```
    model.add(LeakyReLU(0.2))
    model.add(Dense(512))
    model.add(LeakyReLU(0.2))
    model.add(Dense(seq_size, activation='tanh'))
    return model

def create_discriminator():
    model = Sequential()
    model.add(Dense(512, input_dim=seq_size))
    model.add(LeakyReLU(0.2))
    model.add(Dense(256))
    model.add(LeakyReLU(0.2))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer=Adam())
    return model

# Construir y compilar la GAN
generator = create_generator()
discriminator = create_discriminator()
discriminator.trainable = False
gan = Sequential([generator, discriminator])
gan.compile(loss='binary_crossentropy', optimizer=Adam())



# Lista para guardar las pérdidas
discriminator_losses = []
generator_losses = []


# Entrenar la GAN
epochs = 5000
batch_size = 64

for epoch in range(epochs):
    # Generar ruido aleatorio
    noise = np.random.normal(0, 1, size=(batch_size, input_dim))

    # Generar secuencias de ADN
    generated_seqs = generator.predict(noise)

    # Obtener secuencias de ADN reales
    real_seqs_indices = np.random.randint(0, training_data.shape[0], size=batch_size)
    real_seqs = training_data[real_seqs_indices]
```

```
    # Combinar las secuencias reales y generadas
    all_seqs = np.concatenate([real_seqs, generated_seqs])

    # Etiquetas para las secuencias reales y generadas
    labels = np.zeros(2*batch_size)
    labels[:batch_size] = 0.9 # Usar etiquetas suaves para las secuencias reales

    # Entrenar el discriminador
    discriminator.trainable = True
    d_loss = discriminator.train_on_batch(all_seqs, labels)

    # Entrenar el generador
    noise = np.random.normal(0, 1, size=(batch_size, input_dim))
    labels = np.ones(batch_size) # Las etiquetas
    discriminator.trainable = False
    g_loss = gan.train_on_batch(noise, labels)

    # Guardar las pérdidas
    discriminator_losses.append(d_loss)
    generator_losses.append(g_loss)




#Save model

import os

# Create 'models' directory if it doesn't exist
if not os.path.exists('./models'):
    os.makedirs('./models')

# Save the model
gan.save('./models/covid_1000_epochs.keras')


# %%
import matplotlib.pyplot as plt

# Crear el rango de épocas
epochs = range(1, len(discriminator_losses) + 1)

plt.figure(figsize=(12,6))

# Gráfica de la pérdida del discriminador
```

```
plt.plot(epochs, discriminator_losses, label='Discriminator Loss')

# Gráfica de la pérdida del generador
plt.plot(epochs, generator_losses, label='Generator Loss')

plt.title('Generator and Discriminator Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()


# %%
generator.summary()

# %%
discriminator.summary()

# %%
# %%
# Generar un nuevo vector de ruido aleatorio
noise = np.random.normal(0, 1, size=(1, input_dim))

# Usar el generador para crear una nueva secuencia de ADN a partir del ruido
generated_seq = generator.predict(noise)

def vec_to_seq(vec):
    # Crear un diccionario para mapear vectores a nucleótidos
    vec_to_nuc = {0: 'A', 1: 'C', 2: 'G', 3: 'T'}

    # Redondear los valores del vector para que coincidan con los del diccionario
    vec_rounded = np.argmax(vec.reshape(-1, 4), axis=1)

    seq = [vec_to_nuc[i] for i in vec_rounded]
    return ''.join(seq)

# Convertir la secuencia generada a texto
generated_seq_text = vec_to_seq(generated_seq[0])

print("Generated DNA sequence: ", generated_seq_text)
```

## 2 Code for section 4.2.

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from subroutines_lmm_gan import create_random_gaussian_image
import pickle

gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)

# %%
# Create a vector with 28x28 images
num_images = 6000  # You can change this to generate more or fewer images
dataset0 = [create_random_gaussian_image() for _ in range(num_images)]
dataset = np.array(dataset0).reshape([6000,32,32,1])

open_file = open('dataset0.pkl', "wb")
pickle.dump(dataset0, open_file)
open_file.close()

# %%
# Display the generated images
for i in range(10):
    plt.subplot(2, 5, i+1)
    plt.imshow(dataset0[i], cmap='gray')
    plt.axis('off')

plt.show()

# %%
discriminator = keras.Sequential(
    [
      keras.Input(shape=(32, 32, 1)),
      layers.Conv2D(32, kernel_size=4, strides=2, padding="same"),
      layers.LeakyReLU(alpha=0.2),
      layers.Conv2D(64, kernel_size=4, strides=2, padding="same"),
      layers.LeakyReLU(alpha=0.2),
      layers.Conv2D(128, kernel_size=4, strides=2, padding="same"),
      layers.LeakyReLU(alpha=0.2),
      layers.Flatten(),
      layers.Dropout(0.2),
```

```
      layers.Dense(1, activation="sigmoid"),
     ],
     name="discriminator",
)

# %%
latent_dim = 128
generator = keras.Sequential(
    [
     keras.Input(shape=(latent_dim,)),
     layers.Dense(8 * 8 * 128),
     layers.Reshape((8, 8, 128)),
     layers.Conv2DTranspose(128, kernel_size=4, strides=2, padding="same"),
     layers.LeakyReLU(alpha=0.2),
     layers.Conv2DTranspose(256, kernel_size=4, strides=2, padding="same"),
     layers.LeakyReLU(alpha=0.2),
     layers.Conv2D(1, kernel_size=5, padding="same", activation="sigmoid"),

    ],
    name="generator",
)


# %%
discriminator.summary()


# %%
generator.summary()

# %%
class GAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super().__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
        self.d_loss_metric = keras.metrics.Mean(name="d_loss")
        self.g_loss_metric = keras.metrics.Mean(name="g_loss")

    def compile(self, d_optimizer, g_optimizer, loss_fn):
        super(GAN, self).compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.loss_fn = loss_fn
```

```
    @property
    def metrics(self):
        return [self.d_loss_metric, self.g_loss_metric]

    def train_step(self, real_images):
        batch_size = tf.shape(real_images)[0]
        random_latent_vectors = tf.random.normal(
            shape=(batch_size, self.latent_dim))
        generated_images = self.generator(random_latent_vectors)
        combined_images = tf.concat([generated_images, real_images], axis=0)
        labels = tf.concat(
            [tf.ones((batch_size, 1)), tf.zeros((batch_size, 1))],
            axis=0
        )
        labels += 0.05 * tf.random.uniform(tf.shape(labels))
        with tf.GradientTape() as tape:
            predictions = self.discriminator(combined_images)
            d_loss = self.loss_fn(labels, predictions)
        grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
        self.d_optimizer.apply_gradients(
            zip(grads, self.discriminator.trainable_weights)
        )

        random_latent_vectors = tf.random.normal(
            shape=(batch_size, self.latent_dim))
        misleading_labels = tf.zeros((batch_size, 1))

        with tf.GradientTape() as tape:
            predictions = self.discriminator(
                self.generator(random_latent_vectors))
            g_loss = self.loss_fn(misleading_labels, predictions)
        grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.g_optimizer.apply_gradients(
            zip(grads, self.generator.trainable_weights))
        self.d_loss_metric.update_state(d_loss)
        self.g_loss_metric.update_state(g_loss)
        return {"d_loss": self.d_loss_metric.result(),
                "g_loss": self.g_loss_metric.result()}

# %%
class GANMonitor(keras.callbacks.Callback):
    def __init__(self, num_img=1, latent_dim=128):
        self.num_img = num_img
        self.latent_dim = latent_dim
```

```
    def on_epoch_end(self, epoch, logs=None):
        random_latent_vectors = tf.random.normal(
            shape=(self.num_img, self.latent_dim))
        generated_images = self.model.generator(random_latent_vectors)
        generated_images *= 255
        generated_images.numpy()
        for i in range(self.num_img):
            img = keras.utils.array_to_img(generated_images[i])
            img.save(f"generated_img_{epoch:03d}_{i}.png")

# %%
epochs = 300
gan = GAN(discriminator=discriminator, generator=generator,
          latent_dim=latent_dim)

# %%
gan.compile(
    d_optimizer=keras.optimizers.legacy.Adam(learning_rate=0.0001),
    g_optimizer=keras.optimizers.legacy.Adam(learning_rate=0.0001),
    loss_fn=keras.losses.BinaryCrossentropy(),
)

gan.fit(
    dataset, epochs=epochs,
    callbacks=[GANMonitor(num_img=1, latent_dim=latent_dim)]
)

# %%
# Extract the loss history
d_loss_history = gan.history.history['d_loss']
g_loss_history = gan.history.history['g_loss']

# Plot the losses
plt.figure(figsize=(12, 6))
plt.plot(d_loss_history, label='Discriminator Loss')
plt.plot(g_loss_history, label='Generator Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Generator vs Discriminator Loss')
plt.legend()
plt.grid(True)
plt.show()


# %%
```

```
generator.save('generator.h5')
discriminator.save('discriminator.h5')

# %%
# Define the number of images to generate
num_images_to_generate = 3

# Generate random latent vectors
latent_dim = 128  # Assuming this is the same latent dimension you've used before
random_latent_vectors = tf.random.normal(shape=(num_images_to_generate, latent_dim))

# Generate images using the generator
generated_images = generator.predict(random_latent_vectors)

# Visualize the generated images
plt.figure(figsize=(10, 5))
for i in range(num_images_to_generate):
    plt.subplot(1, 3, i+1)
    plt.imshow(generated_images[i, :, :, 0], cmap='gray')
    plt.axis('off')
plt.tight_layout()
plt.show()



# %%
import tensorflow as tf
from tensorflow.keras.models import load_model
import matplotlib.pyplot as plt

# Load the generator model
generator = load_model('generator.h5')

# Define the number of images to generate
num_images_to_generate = 3

# Generate random latent vectors
latent_dim = 128  # Assuming this is the same latent dimension you've used before
random_latent_vectors = tf.random.normal(shape=(num_images_to_generate, latent_dim))

# Generate images using the generator
generated_images = generator.predict(random_latent_vectors)

# Visualize each generated image in an independent figure
for i in range(num_images_to_generate):
    plt.figure()  # This creates a new figure
```

```
    plt.imshow(generated_images[i, :, :, 0], cmap='gray')
    plt.axis('off')
    plt.show()
```

# 3 Code for section 4.3.

```
import numpy as np
from tensorflow import keras
import matplotlib.pyplot as plt
import tensorflow as tf
from keras import layers
tf.compat.v1.disable_eager_execution()  # gp loss won't work with eager
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)


# Wasserstein GANs

def rectangular_array(n=9):
    """ Return x,y coordinates for rectangular array with n^2 stations. """
    n0 = (n - 1) / 2
    return (np.mgrid[0:n, 0:n].astype(float) - n0)


def generator_model(latent_size):
    latent = layers.Input(shape=(latent_size,), name="noise")
    z = layers.Dense(9 * 9 * latent_size)(latent)
    z = layers.Reshape((9, 9, latent_size))(z)
    z = layers.Conv2D(1, (3, 3), activation = 'sigmoid', padding='same',
    kernel_initializer='he_normal')(z)
    return keras.models.Model(latent, z, name="generator")

latent_size = 128
g = generator_model(latent_size)
g.summary()


def critic_model():
    image = layers.Input(shape=(9,9,1), name="images")
    x = layers.Conv2D(64, (3, 3), padding='same', kernel_initializer='he_normal',
    input_shape=(9, 9, 1))(image)
    x = layers.LeakyReLU()(x)
    x = layers.GlobalMaxPooling2D()(x)
    x = layers.Dense(100)(x)
    x = layers.LeakyReLU()(x)
    x = layers.Dense(1)(x)
    return keras.models.Model(image, x, name="critic")
```

```
critic = critic_model()
critic.summary()


def make_trainable(model, trainable):
    ''' Freezes/unfreezes the weights in the given model '''
    for layer in model.layers:
        # print(type(layer))
        if type(layer) is layers.BatchNormalization:
            layer.trainable = True
        else:
            layer.trainable = trainable



# Freeze the critic during the generator training and unfreeze
the generator during the generator training

# %%
make_trainable(critic, False)
make_trainable(g, True) # This is in principal not needed here

# %%
gen_input = g.inputs
generator_training = keras.models.Model(gen_input, critic(g(gen_input)))
generator_training.summary()

keras.utils.plot_model(generator_training, show_shapes=True)


import tensorflow.keras.backend as K

def wasserstein_loss(y_true, y_pred):
    """Calculates the Wasserstein loss - critic maximises the distance between its output
    for real and generated samples.
    To achieve this generated samples have the label -1 and real samples the label 1. """
    return K.mean(y_true * y_pred)

def generator_compilation(eta_generator):
    return generator_training.compile(keras.optimizers.legacy.Adam(eta_generator,
    beta_1=0.5, beta_2=0.9, decay=0.0), loss=[wasserstein_loss])


# %% [markdown]
# Gradient penalty
```

```
# To obtain the Wasserstein distance, we have to use the gradient penalty
to enforce the Lipschitz constraint.
# Therefore, we need to design a layer that samples on straight lines between real
and fake samples

# %%
BATCH_SIZE = 64

class UniformLineSampler(tf.keras.layers.Layer):
    def __init__(self, batch_size):
        super().__init__()
        self.batch_size = batch_size

    def call(self, inputs, **kwargs):
        weights = K.random_uniform((self.batch_size, 1, 1, 1))
        return(weights * inputs[0]) + ((1 - weights) * inputs[1])

    def compute_output_shape(self, input_shape):
        return input_shape[0]

# %% [markdown]
# We design the pipeline of the critic training by inserting generated
(use generator + noise directly to circumvent expensive prediction step)
and real samples into the sampling layer and additionally feeding
generated and real samples into the critic.

# %%
make_trainable(critic, True)  # unfreeze the critic during the critic training
make_trainable(g, False)  # freeze the generator during the critic training

g_out = g(g.inputs)
critic_out_fake_samples = critic(g_out)
critic_out_data_samples = critic(critic.inputs)
averaged_batch = UniformLineSampler(BATCH_SIZE)([g_out, critic.inputs[0]])
averaged_batch_out = critic(averaged_batch)

critic_training = keras.models.Model(inputs=[g.inputs, critic.inputs],
                    outputs=[critic_out_fake_samples, critic_out_data_samples,
                    averaged_batch_out])

# Let us visualize this "computational graph". The critic outputs will be used
# for the Wasserstein loss and the UniformLineSampler output for the gradient
# penalty.

critic_training.summary()
```

```
# %% [markdown]
# We now design the gradient penalty as proposed by in https://arxiv.org/abs/1704.00028

# %%
from functools import partial

def gradient_penalty_loss(y_true, y_pred, averaged_batch, penalty_weight):
    """Calculates the gradient penalty.
    The 1-Lipschitz constraint of improved WGANs is enforced by adding a
    term that penalizes a gradient norm in the critic unequal to 1."""
    gradients = K.gradients(y_pred, averaged_batch)
    gradients_sqr_sum = K.sum(K.square(gradients)[0], axis=(1, 2, 3))
    gradient_penalty = penalty_weight * K.square(1 - K.sqrt(gradients_sqr_sum))
    return K.mean(gradient_penalty)


gradient_penalty = partial(gradient_penalty_loss,
averaged_batch=averaged_batch, penalty_weight=10)  # construct the gradient penalty
gradient_penalty.__name__ = 'gradient_penalty'

# %%
def critic_compilation(critic_eta):
    return critic_training.compile(keras.optimizers.legacy.Adam(critic_eta, beta_1=0.5,
    beta_2=0.9, decay=0.0), loss=[wasserstein_loss,
    wasserstein_loss, gradient_penalty])

# The labels for the training are (Remember we used as for the Wasserstein loss
# a simple multiplication to add a sign.)

positive_y = np.ones(BATCH_SIZE)
negative_y = -positive_y
dummy = np.zeros(BATCH_SIZE)  # keras throws an error when calculating
a loss without having a label -> needed for using the gradient penalty loss


# %%
def plot_image(images, i):
    fig = plt.figure(figsize=(4, 4))
    plt.imshow(images[i,:,:], cmap='gray')
    plt.axis('off')

def magnetization_array(array0):
    accumulated_m = 0
    array = 2* array0 - 1
```

```
    for i in range(array.shape[0]):
       accumulated_m = accumulated_m + np.abs(array[i,:,:,0].sum()) / array.shape[1]**2
    return accumulated_m / array.shape[0]


# %% [markdown]
# GRID SEARCH
#
# Importante: mi criterio para elegir la configuración de parámetros va a
ser la que de menor critic_loss. No me va a importar la loss
del discriminador (critic).
#
# Nota. Tal vez el critic_eta se pueda dejar fijo.

# %%
# Define a range of epochs
EPOCHS = range(1, 3)  # This will loop through epochs 1 and 2

# Define a set of learning rates for the critic
critic_learning_rates = [0.00001, 0.000001, 0.0000001]

# Define a set of learning rates for the generator
generator_learning_rates = [0.001, 0.0001, 0.00001]

best_hyperparameters = {}  # Store the best combination for each
temperature (gen_loss, critic_loss, epoch, critic_eta, generator_eta)

temperatures=np.arange(0.1, 4., 0.1) #Define temperatures

critic_iterations = 1

# %% [markdown]
# Define a function that does the training for each temperature

# %%
def train_gan_for_temperatures(temperatures, EPOCHS):
    real_mags = []
    pred_mags = []
    generator_loss = []
    critic_loss = []
    with open('ising_9_9_t'+str(temperature) + '.npy', 'rb') as f:
        ss = np.load(f)

    shower_maps = ss
    shower_maps = (shower_maps + 1) / 2
```

```
nsamples = len(shower_maps)
permutation = np.random.permutation(nsamples)
shower_maps = shower_maps[permutation,:,:,:]

iterations_per_epoch = nsamples // (critic_iterations * BATCH_SIZE)
iters = 0
for epoch in range(EPOCHS):
    print("Epoch:", epoch)

     for iteration in range(iterations_per_epoch):

        for j in range(critic_iterations):
             noise_batch = np.random.randn(BATCH_SIZE, latent_size)

            shower_batch = shower_maps[BATCH_SIZE*(j+iteration):BATCH_SIZE*
            (j+iteration+1)]
             critic_loss=critic_training.
             train_on_batch([noise_batch, shower_batch],
             [negative_y, positive_y, dummy])

            noise_batch = np.random.randn(BATCH_SIZE, latent_size)

            generator_loss =generator_training.train_on_batch([noise_batch],
            [positive_y]) #modification
            iters += 1

            if iters % 100 == 1:
                print("Iteration:", iters)
                print("Critic loss:", critic_loss)
                print("Generator loss:", generator_loss)

            # Check if this combination gives a lower generator loss for
            the current temperature
            if temperature not in best_hyperparameters or
            generator_loss < best_hyperparameters[temperature][0]:

                best_hyperparameters[temperature] =
                (generator_loss, critic_loss, epoch, critic_eta, generator_eta,
                keras.models.clone_model(g))
                #Guarda también una copia del modelo entero
```

```
# %%
```

```
import itertools
import json
import os

# Ruta del archivo donde se guardará el estado
STATE_FILE = 'hyperparam_search_state.json'

# Función para guardar el estado actual
def save_state(temperature, epoch, critic_eta, generator_eta):
    with open(STATE_FILE, 'w') as f:
        json.dump({
            'temperature': temperature,
            'epoch': epoch,
            'critic_eta': critic_eta,
            'generator_eta': generator_eta
        }, f)

# Función para cargar el estado guardado
def load_state():
    if os.path.exists(STATE_FILE):
        with open(STATE_FILE, 'r') as f:
            return json.load(f)
    return None

# Cargar el estado si existe
state = load_state()

# Si existe un estado guardado, continuar desde ese punto
if state:
    # Use numpy's where() method for numpy arrays
    start_idx_temp = np.where(temperatures == state['temperature'])[0][0]

    # Convert the range object to a list before using index()
    start_idx_epochs = list(EPOCHS).index(state['epoch'])

    start_idx_critic_eta = critic_learning_rates.index(state['critic_eta'])
    start_idx_generator_eta = generator_learning_rates.index(state['generator_eta'])
else:
    start_idx_temp = start_idx_epochs = start_idx_critic_eta = start_idx_generator_eta = 0
```

```
# %%
```

```
# Iterar sobre las combinaciones de hiperparámetros
for temperature, epochs, critic_eta, generator_eta in itertools.product(
        temperatures[start_idx_temp:],
        EPOCHS[start_idx_epochs:],
        critic_learning_rates[start_idx_critic_eta:],
        generator_learning_rates[start_idx_generator_eta:]):

    print("Temperature:", temperature)
    print("Epochs:", epochs)
    print("Critic Learning Rate:", critic_eta)
    print("Generator Learning Rate:", generator_eta)

    # Guardar el estado antes de comenzar con esta combinación
    save_state(temperature, epochs, critic_eta, generator_eta)

    generator_compilation(generator_eta)
    critic_compilation(critic_eta)
    train_gan_for_temperatures(temperature, epochs)

# Eliminar el archivo de estado una vez finalizado
if os.path.exists(STATE_FILE):
    os.remove(STATE_FILE)



# %%
# Guardar los modelos del generador en el disco
for temperature, values in best_hyperparameters.items():
    model = values[-1]  # Get the generator model
    model.save(f'generator_model_temperature_{temperature}.h5')

# %% [markdown]
# PLOT RESULTS

# %%
# Extract critic loss and generator loss from the dictionary
critic_loss_data = np.array([values[1] for values in best_hyperparameters.values()])
generator_loss_data = np.array([values[0] for values in best_hyperparameters.values()])



# Plot generator loss
plt.figure(figsize=(10, 5))
plt.plot(np.arange(len(generator_loss_data)), generator_loss_data, color='red',
markersize=12, label=r'Total')
```

```
plt.legend(loc='upper right')
plt.xlabel(r'Iterations')
plt.ylabel(r'Loss')
plt.title("Generator Loss")
plt.show()

# %% [markdown]
# PREDICTIONS

# %%
# 1. Generar el vector de ruido
noise_vector = np.random.randn(100000, latent_size)

# Número de temperaturas
num_temperatures = len(temperatures)

# Inicializar un array para almacenar las predicciones
predicted_gen_outputs = np.zeros((num_temperatures * 100000, 9, 9))

# Iterar a través de todas las temperaturas en best_hyperparameters
for idx, temperature in enumerate(temperatures):
    # Cargar el modelo del generador correspondiente a la temperatura actual
    model_path = f'generator_model_temperature_{temperature}.h5'
    generator = keras.models.load_model(model_path, compile=False)

    # Hacer una predicción con ese modelo
    prediction = generator.predict_on_batch(noise_vector) #prediction

    # Almacenar la predicción en el array
    start_idx = idx * 100000
    end_idx = start_idx + 100000
    predicted_gen_outputs[start_idx:end_idx] = prediction
```