Universidad Autónoma de Yucatán

Facultad de Matemáticas

Licenciado en Ciencias de la Computación

Inteligencia Artificial

P R O Y E C T O F I N A L

Doc. Alejandro Pasos Ruiz

Est. Julian Emiliano Ortiz Rivero — Est. Angel

Mariel Osalde Salazar — Est. Miguel

Omar Solis Rodriguez

Crear una red neuronal adversarial para generar generar imágenes de un concepto específico.

Escribir un reporte detallando los ajustes y problemas principales.

Mérida, Yucatán, Enero - Mayo 2025

Índice general

Capítulo 1

Preprocesamiento de Imágenes

1.1. Contexto del Estudio

El preprocesamiento de imágenes representa un paso importante en el análisis visual automatizado, especialmente al trabajar con conjuntos de datos heterogéneos. Adaptar las imágenes a un formato y tamaño uniformes facilita su procesamiento posterior por algoritmos de visión por computadora o modelos de aprendizaje automático. Mantener la calidad visual durante esta etapa resulta esencial para evitar que las transformaciones afecten la interpretación de los datos [1, 2, 8, 10].

El método implementado realiza un redimensionado proporcional de cada imagen, preservando la relación de aspecto original con el fin de evitar distorsiones. Cuando la imagen no alcanza las dimensiones deseadas, se añade relleno (padding) para completar el tamaño. Esto garantiza que todas las imágenes posean un tamaño homogéneo, en este caso de 320×320 píxeles, lo cual asegura consistencia en el conjunto de datos y contribuye a mejorar la eficiencia y precisión de los modelos que utilizan estas imágenes como entrada [1, 10].

1.2. Importación de Bibliotecas

Las bibliotecas `os`, `cv2` y `tqdm` fueron utilizadas para automatizar el procesamiento masivo de imágenes en este estudio. `Os` permitió gestionar la estructura de directorios,

creando carpetas de salida y listando archivos con extensiones específicas, lo que facilitó el acceso y organización de los datos [2, 8, 10]. `Cv2` brindó funciones para la lectura, redimensionamiento manteniendo la relación de aspecto y adición de padding en imágenes, asegurando que todas tuvieran un tamaño uniforme, necesario para el análisis posterior.

```
import os
import cv2
from tqdm import tqdm
```

La biblioteca `tqdm` proporcionó una barra de progreso visual que permitió monitorear el avance del procesamiento sobre grandes volúmenes de datos, optimizando la supervisión durante la ejecución de las tareas [1, 2, 8, 10].

1.3. Función de redimensionamiento y padding

La función `resize_and_pad` se diseñó para redimensionar imágenes manteniendo la relación de aspecto original a fin de evitar distorsiones durante el procesamiento. La incorporación de padding ocurre cuando las dimensiones de la imagen redimensionada no coinciden con el tamaño objetivo, garantizando que todas las imágenes presenten dimensiones uniformes [1, 2, 8].

```
def resize_and_pad(img, target_size=(320, 320), pad_color=(0, 0, 0)):
    """
    Redimensiona la imagen manteniendo la relación de aspecto
    y añade padding si es necesario.

    Args:
        img: Imagen de entrada (OpenCV format)
        target_size: Tamaño deseado (ancho, alto)
        pad_color: Color del padding (B, G, R)

    Returns:
        Imagen redimensionada con padding
```

```
    """
    h, w = img.shape[:2]
    target_w, target_h = target_size

    ratio = min(target_w / w, target_h / h)
    new_w, new_h = int(w * ratio), int(h * ratio)

    resized = cv2.resize(
        img, (new_w, new_h), interpolation=cv2.INTER_AREA
    )

    padded = cv2.copyMakeBorder(
        resized,
        pad_h,
        target_h - new_h - pad_h,
        pad_w,
        target_w - new_w - pad_w,
        cv2.BORDER_CONSTANT,
        value=pad_color
    )

    return padded
```

El proceso comienza calculando la escala mínima necesaria para ajustar la imagen al tamaño deseado sin deformarla. Seguidamente, se realiza la redimensión mediante interpolación de área, para luego añadir un borde con un color definido que completa el tamaño objetivo; esta operación se aplica de manera simétrica en los bordes horizontales y verticales [1, 2, 8, 10].

1.4. Función para procesar imágenes en carpeta

La función `process_folder` procesa todas las imágenes contenidas en una carpeta determinada y guarda las imágenes resultantes en otra carpeta; esta operación no solo automatiza el flujo de trabajo en el preprocesamiento visual, sino que también permite definir el tamaño deseado para las imágenes procesadas, asegurando la uniformidad

necesaria para etapas posteriores en el análisis, como la extracción de características o el entrenamiento de modelos de aprendizaje automático, donde la consistencia en las dimensiones resulta esencial para evitar errores y mejorar el rendimiento [1, 2, 8].

```
def process_folder(input_folder, output_folder, target_size=(320, 320)):
    os.makedirs(
        output_folder, exist_ok=True
    )

    valid_extensions = (
        '.jpg', '.jpeg', '.png', '.bmp', '.tiff'
    )
    image_files = [
        f for f in os.listdir(input_folder)
        if f.lower().endswith(valid_extensions)
    ]

    for filename in tqdm(
        image_files, desc="Procesando imágenes"
    ):
        try:
            img_path = os.path.join(
                input_folder, filename
            )
            img = cv2.imread(img_path)

            if img is None:
                print(
                    f"\nNo se pudo leer {filename}. Saltando..."
                )
                continue

            processed_img = resize_and_pad(
                img, target_size
            )
```

```
            output_path = os.path.join(
                output_folder, filename
            )
            cv2.imwrite(output_path, processed_img)

        except Exception as e:
            print(
                f"\nError procesando {filename}: {str(e)}"
            )
```

Este procedimiento inicia creando la carpeta de salida en caso de que esta no exista, para luego generar una lista con las imágenes válidas según sus extensiones. Durante el procesamiento, cada imagen se lee, redimensiona y añade padding mediante la función resize_and_pad, para finalmente guardar el resultado en la carpeta destino. En caso de errores durante la lectura o procesamiento, se registran mensajes informativos que permiten identificar y saltar archivos problemáticos [1, 2, 8, 10].

1.5. Bloque principal de ejecución

El bloque `if __name__ == "__main__"` configura las rutas de las carpetas de entrada y salida para el procesamiento de imágenes. Esta sección permite ejecutar la función `process_folder` con los parámetros definidos, facilitando la automatización del procesamiento desde un script independiente [1, 2, 8].

```
if __name__ == "__main__":
    input_folder = "C:\\Users\\mikem\\Downloads\\Mango\\normal"
    output_folder = "C:\\Users\\mikem\\Downloads\\Mango\\db_320x320"

    print(
        f"Procesando imágenes de {input_folder}..."
    )
    process_folder(input_folder, output_folder)
    print(
        f"\n¡Proceso completado! Imágenes guardadas en {output_folder}"
    )
```

En esta sección se especifican las rutas absolutas de las carpetas que contienen las imágenes originales y el destino para las imágenes procesadas. Durante la ejecución, se muestra información en consola para indicar el inicio y la finalización del proceso, lo que ayuda en el seguimiento del estado de la operación [1, 2, 8, 10].

Capítulo 2

Generación de Imágenes Mediante Redes Generativas Antagónicas

2.1. Contexto del Estudio

Las redes generativas antagónicas (GAN) constituyen una técnica innovadora para la creación de imágenes sintéticas a partir de un esquema adversarial en el que intervienen dos redes neuronales: el generador, encargado de producir las imágenes, y el discriminador, responsable de evaluarlas con base en su autenticidad. Estas redes han sido aplicadas en ámbitos diversos, destacando su uso en agricultura inteligente para la simulación de escenarios complejos [4, 11]. La generación de imágenes bajo condiciones meteorológicas variadas ha ampliado las capacidades de simulación en estudios climáticos [3], mientras que la conversión de patrones isométricos ha mejorado la versatilidad en síntesis gráfica [5].

El éxito en la generación de imágenes con alto grado de realismo mediante GAN depende de la selección precisa de arquitecturas, funciones de pérdida y estrategias de regularización que garanticen la estabilidad durante el entrenamiento. Esto cobra especial relevancia en aplicaciones que demandan calidad visual rigurosa, como ocurre en el campo médico, donde la generación de imágenes para diagnóstico requiere fidelidad extrema [9]. Se han desarrollado herramientas virtuales que integran redes neuronales convolucionales con GAN para simular procesos específicos, lo que incrementa la aplicabilidad de esta tecnología [6, 7].

2.2. Descripción del Proceso de Generación

2.2.1. Instalación de Dependencias

Antes de ejecutar el modelo, es necesario instalar las bibliotecas requeridas; este paso garantiza la compatibilidad del entorno con las versiones esperadas de cada paquete, reduciendo posibles conflictos durante la carga o ejecución del código. A continuación, se muestran los comandos utilizados para instalar las dependencias principales:

```
!pip install numpy==1.26 tensorflow==2.14 \
tensorflow-addons==0.22.0 matplotlib keras==2.14

!pip install tensorflow matplotlib \
keras numpy
```

Una vez instaladas las dependencias, se procede a importar las bibliotecas necesarias para el procesamiento de imágenes, la construcción del modelo y la visualización de resultados. Las importaciones se organizan según su funcionalidad, facilitando la lectura y el mantenimiento del código:

```
import os
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.preprocessing.image import \
    load_img, img_to_array
import matplotlib.pyplot as plt
```

Este bloque inicializa el entorno de trabajo, permitiendo acceder a funciones esenciales para la manipulación de tensores, la construcción de redes neuronales y la carga de imágenes desde el sistema de archivos local, todo dentro del marco de trabajo proporcionado por TensorFlow y Keras.

2.2.2. Parámetros de Configuración del Modelo

En este bloque se definen las variables globales que controlan el comportamiento del modelo generativo, así como las rutas necesarias para el acceso a los datos y la escritura de resultados. Estos parámetros permiten ajustar fácilmente la resolución de las imágenes, la dimensión del espacio latente, el tamaño del lote de entrenamiento, la duración del proceso y la frecuencia con la que se guardan las salidas generadas:

```
# ================== CONFIG ==================
IMAGE_SIZE = 64
LATENT_DIM = 100
BATCH_SIZE = 64
EPOCHS = 30000
SAVE_INTERVAL = 500

DATA_DIR = '/content/drive/MyDrive/db_mango/' \
           'mango/db_320x320'  # <-- CAMBIA ESTO A TU CARPETA

OUTPUT_DIR = '/content/mangos_generados'
os.makedirs(OUTPUT_DIR, exist_ok=True)
```

El parámetro `DATA_DIR` debe apuntar a la carpeta que contiene las imágenes originales utilizadas como conjunto de entrenamiento; por su parte, `OUTPUT_DIR` define el destino donde se almacenarán las imágenes generadas periódicamente, y se crea automáticamente si no existe. Esta sección facilita la reutilización del código en diferentes entornos o experimentos, al centralizar todos los valores críticos de configuración.

2.2.3. Función para Cargar Imágenes desde Carpeta

Esta función se encarga de cargar las imágenes desde una carpeta especificada, ajustando su tamaño y normalizándolas para facilitar su uso en el entrenamiento del modelo generativo. El proceso incluye la conversión a arreglos numéricos y la normalización en el rango $[-1, 1]$, lo cual es común en redes neuronales para mejorar la estabilidad del entrenamiento.

```
# ================== CARGAR IMÁGENES ==================
def load_images_from_folder(folder, size=(IMAGE_SIZE, IMAGE_SIZE)):
    images = []
    for filename in os.listdir(folder):
        path = os.path.join(folder, filename)
        try:
            img = load_img(path, target_size=size)  # RGB por defecto
            img = img_to_array(img)
            img = (img / 127.5) - 1.0  # Normalizar a [-1, 1]
            images.append(img)
        except:
            continue
    return np.array(images)
```

Esta implementación maneja errores silenciosamente, saltando archivos que no puedan ser cargados o procesados, lo que permite mayor robustez en entornos con datos heterogéneos o corruptos.

2.2.4. Definición del Generador

El generador es una red neuronal secuencial que transforma un vector latente en una imagen sintética con las características deseadas. Utiliza capas densas, normalización por lotes, activaciones LeakyReLU y convoluciones transpuestas para incrementar la resolución desde un espacio reducido hasta la imagen final.

```
# ================== DEFINIR GENERADOR ==================
def build_generator():
    model = tf.keras.Sequential([
        layers.Dense(8 * 8 * 256, use_bias=False, \
                     input_shape=(LATENT_DIM,)),
        layers.BatchNormalization(),
        layers.LeakyReLU(),
        layers.Reshape((8, 8, 256)),

        layers.Conv2DTranspose(128, 5, strides=2, \
```

```
                            padding='same', use_bias=False),
        layers.BatchNormalization(),
        layers.LeakyReLU(),

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

        layers.Conv2DTranspose(3, 5, strides=2, \
                            padding='same', activation='tanh', \
                            use_bias=False)
    ])
    return model
```

Este diseño permite generar imágenes en formato RGB normalizadas en el rango $[-1, 1]$, adecuadas para ser evaluadas y refinadas por la red discriminadora en el entrenamiento adversarial.

2.2.5. Definición del Discriminador

El discriminador es una red convolucional que evalúa la autenticidad de las imágenes, diferenciando entre las reales y las generadas. Su arquitectura incluye capas convolucionales con activaciones LeakyReLU y capas de dropout para evitar sobreajuste, finalizando con una capa densa que produce la puntuación de realismo.

```
# ================== DEFINIR DISCRIMINADOR ==================
def build_discriminator():
    model = tf.keras.Sequential([
        layers.Conv2D(64, 5, strides=2, padding='same', \
                      input_shape=[64, 64, 3]),
        layers.LeakyReLU(),
        layers.Dropout(0.3),

        layers.Conv2D(128, 5, strides=2, padding='same'),
```

```
        layers.LeakyReLU(),
        layers.Dropout(0.3),

        layers.Conv2D(256, 5, strides=2, padding='same'),
        layers.LeakyReLU(),
        layers.Dropout(0.3),

        layers.Flatten(),
        layers.Dense(1)
    ])
    return model
```

Esta configuración busca balancear la capacidad del discriminador para distinguir imágenes reales de las generadas, a la vez que mantiene la generalización para no sobreajustar al conjunto de entrenamiento.

2.2.6. Función para Guardar Imágenes Generadas

Esta función convierte las imágenes generadas por el modelo, que están normalizadas en el rango $[-1, 1]$, de vuelta al rango estándar $[0, 255]$ y las guarda en disco en formato PNG. Se nombran según la época de entrenamiento y un índice, permitiendo un seguimiento cronológico de la evolución del generador.

```
# ================== GUARDAR IMÁGENES ==================
def save_images(images, epoch):
    images = (images * 127.5 + 127.5).numpy() \
             .astype(np.uint8)
    for i, img in enumerate(images):
        plt.imsave(f"{OUTPUT_DIR}/epoch{epoch}_img{i}.png", img)
```

De esta forma, se facilita la evaluación visual periódica del progreso del modelo a lo largo del entrenamiento, almacenando las imágenes en la carpeta configurada previamente.

2.2.7. Función de Entrenamiento y Bloque Principal

El procedimiento comienza con la carga seguida de la normalización de las imágenes desde el directorio configurado; luego, se crea el conjunto de datos aplicando aleatorización junto con segmentación en lotes. La función de entrenamiento incorpora un paso que, mediante diferenciación automática, calcula y aplica los gradientes para optimizar ambos modelos de manera simultánea, fomentando así la generación progresiva de imágenes realistas.

```
# ================== ENTRENAMIENTO ==================
def train():
    print("Cargando imágenes...")
    images = load_images_from_folder(DATA_DIR)
    dataset = tf.data.Dataset.from_tensor_slices(images) \
                .shuffle(1000).batch(BATCH_SIZE)
    print(f"{len(images)} imágenes cargadas.")

    generator = build_generator()
    discriminator = build_discriminator()

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

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

    @tf.function
    def train_step(images):
        noise = tf.random.normal([BATCH_SIZE, LATENT_DIM])

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

            real_output = discriminator(images, training=True)
            fake_output = discriminator(generated_images, training=True)
```

```
            gen_loss = cross_entropy(tf.ones_like(fake_output), \
                                     fake_output)
            disc_loss = cross_entropy(tf.ones_like(real_output), \
                                     real_output) + \
                        cross_entropy(tf.zeros_like(fake_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

    print("Iniciando entrenamiento...")
    for epoch in range(EPOCHS):
        for image_batch in dataset:
            gen_loss, disc_loss = train_step(image_batch)

        if (epoch + 1) % SAVE_INTERVAL == 0:
            print(f"Epoch {epoch+1}, Gen Loss: {gen_loss.numpy():.4f}, \
Disc Loss: {disc_loss.numpy():.4f}")
            noise = tf.random.normal([16, LATENT_DIM])
            generated_images = generator(noise, training=False)
            generated_images = (generated_images + 1) / 2
            save_images(generated_images, epoch + 1)

    print("Entrenamiento finalizado.")

if __name__ == "__main__":
    train()
```

El ciclo de entrenamiento se ejecuta por el número de épocas definido, procesando cada lote del conjunto de datos y actualizando los modelos. Periódicamente, se informa el progreso mediante pérdidas de generador y discriminador, mientras que las imágenes generadas se almacenan para seguimiento visual. Este mecanismo asegura una evolución constante y controlada del modelo, permitiendo evaluar el avance y ajustar parámetros si es necesario.

2.3. Ajustes Realizados

2.3.1. Cambio en la resolución y canales de salida

Se modifico el generador para soportar imágenes de mayor resolución (64x64 pixeles) y con 3 canales de color para soportar imágenes en RGB. Esto implico aumentar el tamaño inicial de la capa "Densez ajustar el número de filtros en las capas desconvolucionales para que el modelo pueda generar las imágenes más detalladas y a color.

2.3.2. Modificación del discriminador

Se modificó el discriminador para soportar mayor complejidad, es decir, para trabajar con imágenes más grandes y en color. Se le añadieron más capas convolucionales y se incrementó la cantidad de filtros, alcanzando los 256, lo que le permite capturar características más complejas de las imágenes reales.Ademas el tamaño de entrada se ajusto de 28x28x1 a 64x64x3 pues la resolución de 64x64 pixeles mostró mejores resultados en el menor tiempo.

2.3.3. Cambio en los parámetros

Se ajustaron los parámetros para reducir el tamaño de lote de 255 a 64, se aumentaron las épocas a 1500, se estableció un intervalo de resultados cada 500 épocas. Pues al trabajar con un dataset no estandarizado resultaría ser más complejo que con el original de MNIST.

2.3.4. Uso de dataset personalizado

Al utilizar nuestro propio dataset, se incorporó una nueva función para leer imágenes del disco, redimensionarlas, convertirlas a matrices NumPy y normalizarlas.

2.3.5. Función de guardado

implementa una función que guarda únicamente las mejores imágenes generadas, basándose en las puntuaciones asignadas por el discriminador. Esto permite conservar solo los resultados más realistas durante el entrenamiento, facilitando el seguimiento de la evolución del modelo y evitando guardar ejemplos poco útiles o fallidos.

2.3.6. Más capas convolucionales

Se añadió una capa convolucional adicional y más capas Dropout en el discriminador, lo cual ayuda a mejorar la capacidad del modelo para distinguir imágenes reales de las generadas, al tiempo que se reduce el riesgo de sobre ajuste.

2.4. Problemas Encontrados

2.4.1. Datos personalizados

Durante el entrenamiento del modelo GAN con el conjunto de imágenes de mangos, se presentaron diversos problemas que afectaron la estabilidad del proceso. A diferencia de datasets estandarizados como MNIST, el conjunto de datos utilizado mostró una alta variabilidad en aspectos como color, iluminación y forma de los mangos, lo cual dificultó la convergencia del modelo. Se evidenció un desequilibrio entre el generador y el discriminador, derivando en episodios de mode collapse, donde el generador producía imágenes repetitivas y poco variadas. Además, la mayor resolución de las imágenes y su naturaleza en color (RGB) incrementaron considerablemente la demanda computacional, ralentizando el entrenamiento. También se detectó que algunas imágenes del dataset presentaban ruido o inconsistencias visuales que afectaron

negativamente el desempeño del discriminador.

2.5. Evaluación de Resultados

Al finalizar el proceso de entrenamiento, se generaron imágenes sintéticas de mangos que variaron en calidad y nivel de realismo a lo largo de las épocas. Se observó una mejora progresiva en la nitidez, el color y la forma de las imágenes generadas conforme avanzaba el entrenamientoa, particularmente a partir de las mil épocas. En etapas iniciales, los resultados eran borrosos o poco definidos, pero con el tiempo, el generador logró producir representaciones visualmente más coherentes con las imágenes reales del conjunto de datos. No obstante, también se identificaron signos de colapso de modo en ciertos momentos del entrenamiento, donde el generador tendía a producir imágenes repetitivas. Las mejores muestras generadas fueron seleccionadas utilizando el discriminador como criterio de evaluación, lo que permitió preservar únicamente aquellas imágenes que lograron engañar de forma más efectiva al modelo discriminativo. En conjunto, los resultados reflejan tanto el potencial del modelo como las áreas que requieren ajustes adicionales para mejorar la diversidad y fidelidad de las imágenes sintéticas.

2.6. Discusión

Los resultados obtenidos durante el entrenamiento de la red generativa antagónica reflejan tanto los logros como las limitaciones del modelo en su aplicación al dominio de imágenes de mangos. A lo largo de las épocas, fue evidente una mejora en la capacidad del generador para sintetizar imágenes que imitan visualmente las características de las muestras reales. Sin embargo, también se manifestó un fenómeno recurrente de colapso de modo, donde el generador producía imágenes similares entre sí, lo cual evidencia una falta de diversidad en la salida del modelo. Esta situación indica que, si bien el generador fue capaz de aprender ciertas representaciones dominantes del conjunto de datos, no logró capturar completamente la variabilidad presente en la base original. Además, se identificaron fluctuaciones en las pérdidas del generador y del discriminador, lo cual sugiere que la competencia entre ambos modelos no siempre se mantuvo en equilibrio, dificultando una convergencia estable. Estos hallazgos

destacan la necesidad de explorar estrategias de mejora, como el ajuste fino de hiperparámetros, la implementación de técnicas de regularización o la incorporación de arquitecturas más avanzadas, para lograr una generación más robusta y variada de imágenes sintéticas.

Figura 2.1: Representación generada de un mango en baja resolución, donde predominan tonalidades verdes y amarillas distribuidas de forma irregular. La forma del fruto no es completamente reconocible.

Figura 2.2: Imagen sintética que sugiere la presencia de un mango a partir del patrón cromático, con colores rojizos y anaranjados. La estructura general aparece difusa.

Figura 2.3: Visualización de un mango generado artificialmente, donde se distingue un contorno ovalado en tonos cálidos. El nivel de detalle es limitado debido a la baja resolución.

Bibliografía

[1] Jesús Bobadilla. *Machine learning y deep learning: usando Python, Scikit y Keras*. Ediciones de la U, 2021.

[2] José Manuel Ortega Candel. *Big data, machine learning y data science en python*. Ra-Ma Editorial, 2022.

[3] Antonio Cuadrado Cobo et al. Generación de imágenes simulando distintas condiciones meteorológicas mediante el uso de redes generativas antagónicas (gans). *Revista Ibérica de Sistemas e Tecnologias de Informação*, 2021.

[4] Jesús Miguel Angel Rodríguez Mendoza, Guillermo Ronquillo Lomeli, and Angel Iván García. Aplicaciones de redes generativas antagónicas en agricultura inteligente. *La Mecatrónica en México*, page 95, 2020.

[5] Diego Navarro Mateu, Oriol Carrasco, and Pedro Cortés Nieves. Conversión de patrones en isométricas a través de redes generativas antagónicas (gans). *Revista Ibérica de Sistemas e Tecnologias de Informação*, 2022.

[6] Marina Palma. Probador virtual basado en redes neuronales convolucionales (cnn) y redes generativas antagónicas (gan). Master's thesis, Revista Ibérica de Sistemas e Tecnologias de Informação, 2023.

[7] Hubner Janampa Patilla, Edgar Gutiérrez Gómez, Adolfo Quispe Arroyo, and Roly Auccatoma Tinco. Generación imágenes de la cerámica wari mediante redes generativas antagónicas (gan). *Revista Ibérica de Sistemas e Tecnologias de Informação*, 20(E74):484–496, 2024.

[8] Why Python. Python. *Python releases for windows*, 24, 2021.

[9] Cesar Rioja-García, Mariko Nakano-Miyatake, Oswaldo Ulises Juarez-Sandoval, Keiji Yanai, Gibran Benítez-García, et al. Generación de imágenes médicas en

el área de la retinopatía diabética con ayuda de redes generativas antagónicas. *Pädi Boletín Científico de Ciencias Básicas e Ingenierías del ICBI*, 11(22):95–102, 2024.

[10] Cristian L Vidal-Silva, Aurora Sánchez-Ortiz, Jorge Serrano, and José M Rubio. Experiencia académica en desarrollo rápido de sistemas de información web con python y django. *Formación universitaria*, 14(5):85–94, 2021.

[11] Sergio Yovine, Franz Mayr, and Ramiro Visca Zanoni. Informe final del proyecto: Anonimización de datos basada en redes generativas antagónicas. *Revista Ibérica de Sistemas e Tecnologias de Informação*, 2021.