Humanidades, Tecnología y Ciencia,
del Instituto Politécnico Nacional

Certificado de Reserva de Derechos al Uso Exclusivo del Título: 04-2010-0326124-14000-203 ISSN: 2007-1957

MODELO BAYESIANO PARA LA PREDICCIÓN DE RUIDO ITERATIVO EN LA GENERACIÓN DE IMÁGENES USANDO PYTHON

Israel Rivera Zárate
Instituto Politécnico Nacional-CIDETEC
irivera@ipn.mx

Miguel Hernández Bolaños
Instituto Politécnico Nacional-CIDETEC
mbolanos@ipn.mx

Patricia Pérez Romero
Instituto Politécnico Nacional-CIDETEC
promerop@ipn.mx

Abstract

Diffusion models works by destroying some input for example an image by gradually adding noise and then recovering the input from the noise in a backward process also called denoising, this is also called a Markov chain because it´s a sequence of stochastics events where each time step depends on the previous time step. A special property of the future models is that the latent states have the same dimensionality as the input. The task of the model can be described as predicting the noise that was added in each of the images that´s why the backward process is called parametrized. We use a neural network for it to generate new data one can simply perform the backward process from random noise and new data points are constructed.

Palabras clave: modelo probabilístico, modelo generador, ruido Gaussiano, proceso de difusión, autoencoder.

Como lo señalan J. Ho et al. (2020), en el ámbito del aprendizaje máquina los modelos generativos por difusión parten de la recopilación de un conjunto grande de datos *X* de entrenamiento. Se asume que los datos provienen de una distribución subyacente desconocida *q(X)*. Para obtener la distribución de los datos se creará un modelo, este modelo representa una distribución de probabilidad parametrizada $p_\theta(X)$. Se buscará ajustar los parámetros de la distribución del modelo para asegurar que se apegue de forma óptima a la

distribución de los datos de entrenamiento, lo cual se logra mediante muestreo de nuevos datos en la distribución de probabilidad $X \sim p_\theta(X)$. Si el modelo se realiza correctamente, aquellos elementos apegados a la distribución tendrán un valor de probabilidad alto, de lo contrario su probabilidad será mínima. Lo anterior corresponde con una conversión de un problema de modelado generativo en un problema de aprendizaje supervisado, lo cual se puede realizar añadiendo diferentes niveles de fidelidad T al conjunto de datos original quedando: $X_{0:T} = \{X_0, X_1, \ldots, X_{T-1}, X_T\}$. Donde X_0 corresponde con el dato original de entrenamiento siendo además el dato de mayor fidelidad. El dato de menor fidelidad será la muestra tomada directamente de $p(X_T)$. La predicción de una muestra de mayor fidelidad a partir de una de menor fidelidad se obtendrá mediante una regresión simple aplicando el teorema de Bayes: $p(X_{t-1} \mid X_t)$. Por lo tanto, se podrán generar datos mediante el muestreo de elementos de baja fidelidad $X_T \sim p(X_T)$ y luego de forma recursiva ir muestreando hacia niveles superiores de fidelidad $X_{T-1} \sim p(X_{t-1} \mid X_t)$. Para $t = T \ldots 1$ hasta generar X_0. Ver figura 1.

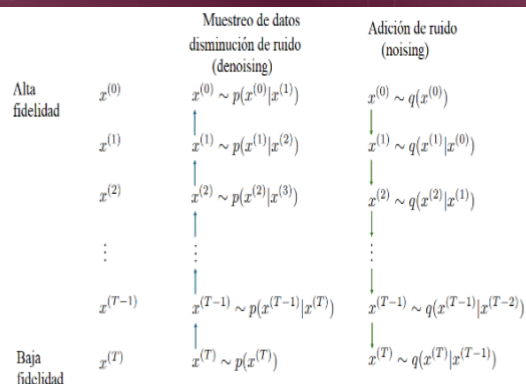

Figura 1. La generación de datos a diferentes niveles de fidelidad, Izquierda. La adición de ruido a diferentes niveles de fidelidad, Derecha. Elaboración propia.

I. Conceptos Introductorios

A. Proceso directo

De acuerdo con P. Dhariwal y A. Nichol (2021), un modelo de difusión se distingue por ser un modelo de variable latente en el que el posterior aproximado $q(X_{1:T} \mid X_0)$ llamado "proceso directo" o proceso de difusión q es capaz de producir variables latentes X_1 $hasta$ X_T, donde se va añadiendo pequeñas cantidades de ruido Gaussiano a intervalos regulares de tiempo t, con varianza $\beta_1, \ldots \beta_T$, (noising); hasta que la entrada sea destruida por completo (proceso Gausssiano autoregresivo de primer orden). Este proceso es usado para el entrenamiento del modelo. Ver figura 2.

Certificado de Reserva de Derechos al Uso Exclusivo del Título: 04-2010-0326124-14000-203 ISSN: 2007-1957

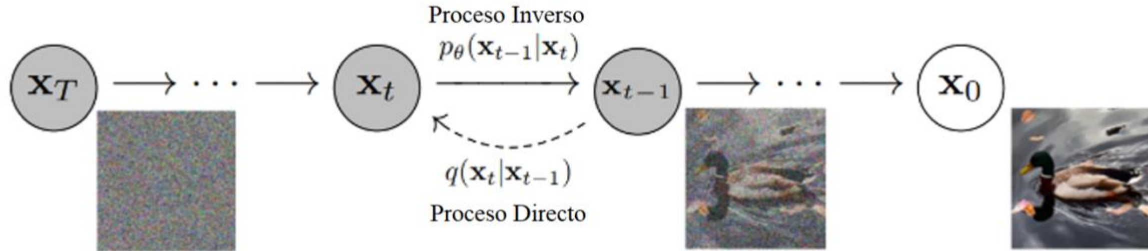

Figura 2. Gráfica dirigida del modelo probabilístico de difusión. Elaboración propia.

Se tiene entonces:

$$q(\mathbf{x}_{1:T}|\mathbf{x}_0) := \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1}),$$
$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) := \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \quad \ldots (1)$$

El proceso de adición de ruido definido en (2) permite muestrear aleatoriamente en algún paso arbitrario con $\alpha_t := 1 - \beta_t$ y $\bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$, por lo que se puede escribir:

$$q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)\mathbf{I})$$
$$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon \quad \ldots (2)$$

Donde $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. En este punto, 1 - $\bar{\alpha}_t$ establece la varianza del ruido para un intervalo de tiempo arbitrario. Empleando el teorema de Bayes se puede calcular el posterior $q(X_{t-1}|\ X_t,\ X_0)$, quedando definido como:

$$q(x_{t-1}|x_t, x_0) = \mathcal{N}(x_{t-1}; \tilde{\mu}(x_t, x_0), \tilde{\beta}_t\mathbf{I}) \quad (3)$$

B. Proceso inverso

Por su parte, J. Shi et al. (2022), señalan que en los modelos de difusión se toman muestras de una distribución creada a partir del “proceso inverso”, Esto es, se toma la muestra con ruido X_T y se van produciendo gradualmente las muestras con menos ruido $X_{T-1}, X_{T-2}, \ldots$ hasta alcanzar la muestra final X_0. Cada intervalo de tiempo t corresponde a cierto nivel de ruido, donde cada X_t puede ser considerada como una mezcla de la señal original X_0 con algún nivel de ruido ϵ. Un modelo de difusión aprende a producir una muestra con menor nivel de ruido X_{t-1} a partir de X_t (denoising). Se puede hacer una aproximación usando una red neuronal como sigue:

$$p_\theta(\mathbf{x}_{0:T}) := p(\mathbf{x}_T)\prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t),$$
$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) := \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t)) \quad \ldots (4)$$

C. Función de costo

Cabe indicar que B. Kawar et al. (2022), mencionan que para el entrenamiento del modelo se utilizará como función de costo la combinación de q y p que corresponden con un modelo variacional tipo autoencoder, donde: $L_{vlb} :=\ L_0 + L_1 + \cdots + L_{T-1} + L_T$, se tiene:

Certificado de Reserva de Derechos al Uso Exclusivo del Título: 04-2010-0326124-14000-203 ISSN: 2007-1957

$$\mathbb{E}_q\Big[\underbrace{D_{\mathrm{KL}}(q(\mathbf{x}_T|\mathbf{x}_0)\parallel p(\mathbf{x}_T))}_{L_T} + \sum_{t>1}\underbrace{D_{\mathrm{KL}}(q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)\parallel p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t))}_{L_{t-1}} \underbrace{-\log p_\theta(\mathbf{x}_0|\mathbf{x}_1)}_{L_0}\Big] \quad \ldots (5)$$

Para la implementación del algoritmo la expresión simplificada en términos de la función de expectación $E_{t,\,x_0,\epsilon}\,[\mathrm{L}_{t-1}]$ para estimar L_{vlb} queda como:

$$L_{\mathrm{simple}}(\theta) := \mathbb{E}_{t,\mathbf{x}_0,\boldsymbol{\epsilon}}\Big[\big\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)\big\|^2\Big] \quad (6)$$

II. Desarrollo

La implementación del modelo de difusión se desarrolló en Python (https://www.python.org, versión 3.6.1) con la ayuda de las librerías gratuitas PyTorch (https://pytorch.org, versión 2.0) y Tensorflow (https://www.tensorflow.org, versión 1.2.1), ampliamente aceptadas por la comunidad en inteligencia artificial. El entorno de desarrollo empleado ha sido el Notebook de Jupyter (http://jupyter.org, versión 5.0.0).

De acuerdo con O. Ronneberger et al. (2015), para el entrenamiento se utilizará el modelo de red U-NET, que es una red del tipo autoencoder. En la etapa del encoder se produce una disminución de la resolución espacial y se incrementa el número de canales. Por su parte, en la etapa del decoder se produce un incremento de la resolución espacial y una disminución en el número de los canales. Ver figura. U-NET es usada para modelar las transiciones del proceso inverso. Los canales transfieren características desde el encoder hacia el decoder. Para la difusión, la U-NET cuenta con dos entradas: imagen ruidosa e intervalo de tiempo y como salida la predicción del ruido ϵ_θ con θ como parámetros del modelo. Cabe señalar que para mejorar el desempeño se han añadido bloques de atención, así como capas de normalización. Ver figura 3.

En el presente trabajo se utilizó la base datos “ImageNet” que está disponible en el repositorio de Kaggle. La base de datos está compuesta por 1000 objetos contenidos en 1,281,167 imágenes de entrenamiento, 50,000 imágenes de validación y 100,000 imágenes de prueba.

III. Metodología

La metodología adoptada en el presente trabajo consiste en 3 pasos esenciales:

Paso 1: Creación del modelo (proceso directo).

Paso 2: Entrenamiento de la red (proceso inverso).

Paso 3. Generación de muestras.

Certificado de Reserva de Derechos al Uso Exclusivo del Título: 04-2010-0326124-14000-203 ISSN: 2007-1957

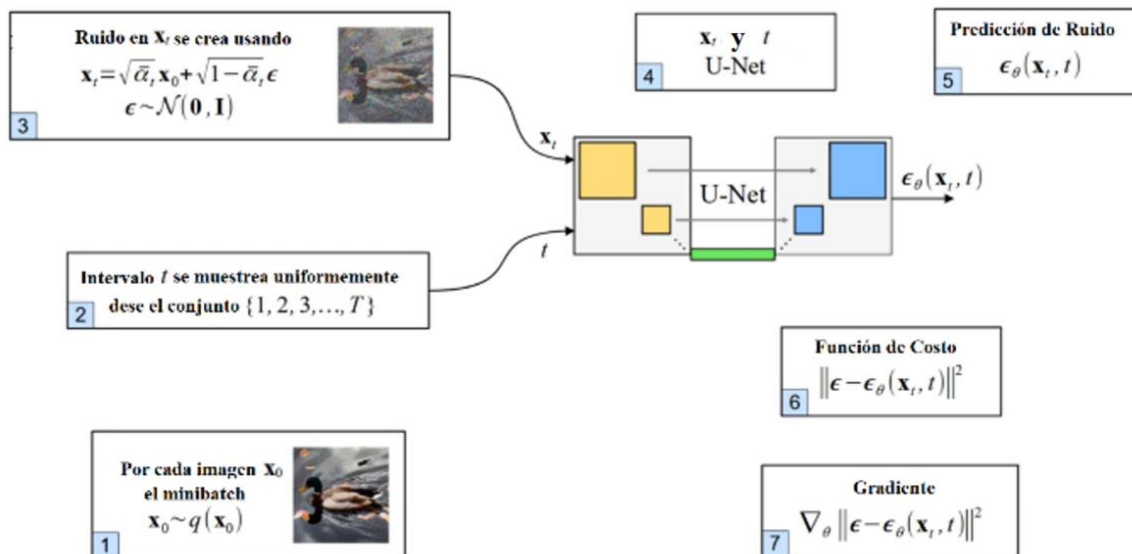

Figura 3. Implementación del modelo de difusión basado en la red U-Net. Elaboración propia.

A. Creación del modelo (proceso directo)

Se llevará a cabo la programación de las herramientas de difusión, como son la configuración de los parámetros del ruido Gaussiano, la función para imágenes con ruido, el muestreo de las imágenes, todo en una sola clase llamada: Diffusion. Los principales parámetros serán la cantidad de pasos de muestreo (noise_steps=1000), el extremo inferior y superior de la varianza $\beta_1 = 0.0001$ a $\beta_t = 0.02$), el tamaño de la imagen que será de 64x64. Así como los elementos necesarios para el entrenamiento. Ver figura 4.

```
class Diffusion:
    def __init__(self, noise_steps=1000, beta_start=1e-4, beta_end=0.02, img_size=256, device="cuda"):
        self.noise_steps = noise_steps
        self.beta_start = beta_start
        self.beta_end = beta_end
        self.img_size = img_size
        self.device = device

        self.beta = self.prepare_noise_schedule().to(device)
        self.alpha = 1. - self.beta
        self.alpha_hat = torch.cumprod(self.alpha, dim=0)
```

Figura 4. Fragmento de código de la etapa de difusión. Elaboración propia.

El proceso de adición de ruido definido en (4), permite muestrear aleatoriamente en algún paso arbitrario considerando los parámetros $\alpha_t := 1 - \beta_t$, $\bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$ y $\sqrt{1 - \bar{\alpha}_t}$. Se establece el número de pasos de tiempo en mil, igualmente se utilizaron los valores recomendados para la varianza, así como el uso de ruido Gaussiano lineal en vez del tipo cosenoidal. Así como se debe incluir adicionalmente el ruido aleatorio ε, (figura 5).

```
def prepare_noise_schedule(self):
    return torch.linspace(self.beta_start, self.beta_end, self.noise_steps)

def noise_images(self, x, t):
    sqrt_alpha_hat = torch.sqrt(self.alpha_hat[t])[:, None, None, None]
    sqrt_one_minus_alpha_hat = torch.sqrt(1 - self.alpha_hat[t])[:, None, None, None]
    Ɛ = torch.randn_like(x)
    return sqrt_alpha_hat * x + sqrt_one_minus_alpha_hat * Ɛ, Ɛ
```

Figura 5. Fragmento de código adición de ruido. Elaboración propia.

La función de muestreo creará las imágenes iniciales tomando muestras de una distribución normal usando torch.rand. A continuación, se ejecuta el ciclo que recorre los mil pasos de tiempo en orden inverso. Se inicia con el más alto hasta llegar al número 1. El primer paso en el ciclo se encarga de crear un tensor de longitud n con el paso de tiempo

Certificado de Reserva de Derechos al Uso Exclusivo del Título: 04-2010-0326124-14000-203 ISSN: 2007-1957

actual. Al final se realiza el escalamiento para que los valores de las imágenes estén en el rango de -1 y 1. La división por 2 es solo para devolver los valores en el rango de 0 y 1, que finalmente se multiplicarán por 255 para colocarlos en un rango adecuado de valores de pixeles. Ver figura 6.

```python
def sample(self, model, n):
    logging.info(f"Sampling {n} new images....")
    model.eval()
    with torch.no_grad():
        x = torch.randn((n, 3, self.img_size, self.img_size)).to(self.device)
        for i in tqdm(reversed(range(1, self.noise_steps)), position=0):
            t = (torch.ones(n) * i).long().to(self.device)
            predicted_noise = model(x, t)
            alpha = self.alpha[t][:, None, None, None]
            alpha_hat = self.alpha_hat[t][:, None, None, None]
            beta = self.beta[t][:, None, None, None]
            if i > 1:
                noise = torch.randn_like(x)
            else:
                noise = torch.zeros_like(x)
            x = 1 / torch.sqrt(alpha) * (x - ((1 - alpha) / (torch.sqrt(1 - alpha_hat))) * predicted_noise)
    model.train()
    x = (x.clamp(-1, 1) + 1) / 2
    x = (x * 255).type(torch.uint8)
    return x
```

Figura 6. Fragmento de código muestreo de datos. Elaboración propia.

B. Entrenamiento de la red (proceso inverso)

Se declara un objeto de la clase UNet donde se cuenta con partes distinguibles: el encoder y el decoder. Asimismo, se han definido 3 canales ya que se operará con imágenes a color. En la parte del encoder la función "Doubleconv" realiza las convoluciones de 3x3 en dos dimensiones, con padding de 1 y función de activación tipo RELU. En esta etapa se garantiza la reducción de dimensionalidad. Por otra parte, en el decoder, se aplica la función transpuesta 2d de la convolución, recibe un número variable de canales, se le aplica un número de filtros que ahora duplican la dimensionalidad, además de un stride de 2. Garantizando así el aumento de la dimensionalidad. Por último, la comunicación por los canales se lleva a cabo mediante una copia de las salidas de las capas, lo cual sucede para un tamaño de 64, 128 y 256 elementos. Ver figuras 7 y 8.

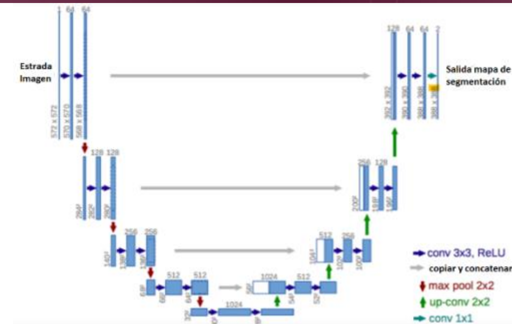

Figura 7. Modelo de red U-NET. Cada caja azul corresponde a un mapa de características multicanal. El número de canales aparece en la parte superior de cada caja. Las cajas blancas son copias de mapas de características. El tamaño x-y aparece en la parte inferior de cada caja. Las flechas denotan las diferentes operaciones. Red tomada de O. Ronneberger, P. Fischer and T. Brox (2015).

```python
class UNet(nn.Module):
    def __init__(self, c_in=3, c_out=3, time_dim=256, device="cuda"):
        super().__init__()
        self.device = device
        self.time_dim = time_dim
        self.inc = DoubleConv(c_in, 64)
        self.down1 = Down(64, 128)
        self.sa1 = SelfAttention(128, 32)
        self.down2 = Down(128, 256)
        self.sa2 = SelfAttention(256, 16)
        self.down3 = Down(256, 256)
        self.sa3 = SelfAttention(256, 8)

        self.bot1 = DoubleConv(256, 512)
        self.bot2 = DoubleConv(512, 512)
        self.bot3 = DoubleConv(512, 256)

        self.up1 = Up(512, 128)
        self.sa4 = SelfAttention(128, 16)
        self.up2 = Up(256, 64)
        self.sa5 = SelfAttention(64, 32)
        self.up3 = Up(128, 64)
        self.sa6 = SelfAttention(64, 64)
        self.outc = nn.Conv2d(64, c_out, kernel_size=1)
```

Figura 8. Fragmento de código arquitectura U-NET. Elaboración propia.

Para el entrenamiento, se realiza una instancia del modelo y se aplica el descenso del gradiente con base en la función de pérdida, para lo cual se emplea la función "BCEWithLogistsLoss" que aplicará la función de activación "sigmoid" a las salidas de la red (para que estén entre 0 y 1) y luego calcula la función "binary cross entropy". Al final del entrenamiento, se genera una nueva

muestra X_0 como resultado del proceso iterativo comenzando en el intervalo t en la imagen con ruido X_T. Ver figura 9.

```python
def train(args):
    setup_logging(args.run_name)
    device = args.device
    dataloader = get_data(args)
    model = UNet().to(device)
    optimizer = optim.AdamW(model.parameters(), lr=args.lr)
    mse = nn.MSELoss()
    diffusion = Diffusion(img_size=args.image_size, device=device)
    logger = SummaryWriter(os.path.join("runs", args.run_name))
    l = len(dataloader)

    for epoch in range(args.epochs):
        logging.info(f"Starting epoch {epoch}:")
        pbar = tqdm(dataloader)
        for i, (images, _) in enumerate(pbar):
            images = images.to(device)
            t = diffusion.sample_timesteps(images.shape[0]).to(device)
            x_t, noise = diffusion.noise_images(images, t)
            predicted_noise = model(x_t, t)
            loss = mse(noise, predicted_noise)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            pbar.set_postfix(MSE=loss.item())
            logger.add_scalar("MSE", loss.item(), global_step=epoch * l + i)

        sampled_images = diffusion.sample(model, n=images.shape[0])
        save_images(sampled_images, os.path.join("results", args.run_name, f"{epoch}.jpg"))
        torch.save(model.state_dict(), os.path.join("models", args.run_name, f"ckpt.pt"))
```

Figura 9. Fragmento de código entrenamiento del modelo. Elaboración propia.

C. Generación de muestras

1. Interpolación

Como lo señalan B. Kawar et al. (2022), se pueden interpolar imágenes fuente $X_0, X_0' \sim q(X_0)$ en el espacio latente, usando q como un decodificador estocástico, $X_t, X_t' \sim q(X_t \mid X_0)$, luego decodificar la latente interpolada linearmente $\overline{X_t} = (1 - \lambda)\, X_0 + \lambda X_0'$ hacia el espacio de las imágenes mediante el proceso inverso, $\overline{X}_0 \sim p(X_0 \mid \overline{X_t})$. El proceso inverso ayuda a remover elementos no deseados en las versiones linealmente interpoladas con algún grado de corrupción. El ruido se fija a diferentes valores de λ, de forma que X_t y X_t' tienen el mismo valor. El proceso inverso produce reconstrucciones de gran calidad e interpolaciones con variaciones sutiles en pose, color de piel, estilo de peinado, expresión y fondo.

2. Escalamiento de datos

Se ha asumido que los datos de la imagen constan de números enteros {0,1,…,255} escalados linealmente a [-1, 1]. Esto asegura que la red neuronal del proceso inverso opere con entradas escaladas consistentemente con el priori normal estándar $p(X_T)$. Para obtener valores discretos del logaritmo de la verosimilitud, se estableció el último término del proceso inverso en el decodificador del proceso inverso derivado de la Gaussiana $\mathcal{N}(X_0\,; \mu_\theta(X_1, \mathbf{1}), \boldsymbol{\sigma}_{\mathbf{1}}^{\mathbf{2}}\mathbf{I})$:

$$p_\theta(\mathbf{x}_0|\mathbf{x}_1) = \prod_{i=1}^{D} \int_{\delta_-(x_0^i)}^{\delta_+(x_0^i)} \mathcal{N}(x; \mu_\theta^i(\mathbf{x}_1, 1), \sigma_1^2)\, dx$$
$$\delta_+(x) = \begin{cases} \infty & \text{if } x = 1 \\ x + \frac{1}{255} & \text{if } x < 1 \end{cases} \qquad \delta_-(x) = \begin{cases} -\infty & \text{if } x = -1 \\ x - \frac{1}{255} & \text{if } x > -1 \end{cases} \quad \ldots(7)$$

Donde D es la dimensionalidad de los datos e i es el superíndice que indica la extracción de una coordenada. Ver figuras 10 y 11.

```python
#Interpolating between faces - Linear interpolation
##########################################################################

from numpy import linspace

# Function to generate random latent points
#Same as defined above, re-defining for convenience.
def generate_latent_points(latent_dim, n_samples, n_classes=10):
    # generate points in the latent space
    x_input = randn(latent_dim * n_samples)
    z_input = x_input.reshape(n_samples, latent_dim) #Reshape to be
    return z_input

# Interpolation between two points in latent space
def interpolate_points(p1, p2, n_steps=10):
    # interpolate between points (e.g., between 0 and 1 if you divde
    ratios = linspace(0, 1, num=n_steps)
    # linear interpolation of vectors based on the above interpolati
    vectors = list()
    for ratio in ratios:
        v = (1.0 - ratio) * p1 + ratio * p2
        vectors.append(v)
    return asarray(vectors)

# create a plot of generated images
def plot_generated(examples, n):
```

Figura 10. Fragmento de código interpolación lineal. Elaboración propia.

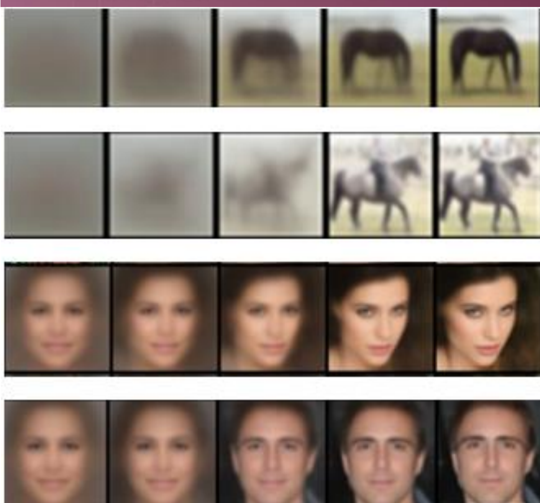

Figura 11. Interpolación y Escalamiento de los datos aplicado al proceso de eliminación de ruido. Elaboración propia.

IV. Pruebas y Resultados

Con el fin de verificar la calidad y alcance del modelo propuesto, se realizó el cálculo y comparación de las métricas de calidad entre el modelo propuesto en el presente trabajo y los modelos propuestos en P. Dhariwal and A. Nichol (2021). En estos modelos se utilizó la base de datos de ImageNet en resoluciones de 128x128, 256x256 y 512x512, respectivamente. Los modelos propuestos son:

a) **StyleGAN:** Red Generativa Antagónica principalmente usada para rostros propuesta por Nvidia en 2018. La universidad de Washington utilizó esta red para crear rostros falsos en el evento ¿Which Face is Real? Consta de 26.2 M de parámetros entrenables. Cuenta con 8 capas de redes para mapeo y 18 capas para síntesis. Además, utiliza módulos AdaIN que se encargan de llevar a cabo la normalización de instancia adaptiva.

b) **ADM:** Modelo de difusión por ablación (Ablated Difussion Model) basado en la red U-Net, por lo cual utiliza una arquitectura tipo autoencoder con una serie de capas convolucionales, donde se lleva a cabo una primera etapa de contracción y una posterior de dilatación, la información espacial se reduce en tanto que la información de características aumenta.

c) **DDPM:** modelo probabilístico de difusión por eliminación de ruido (Denoising Diffusion Probabilistic Model). Establece un proceso iterativo de eliminación de ruido basado en variables latentes conduciendo a muestras de alta fidelidad con suficientes pasos de eliminación de ruido. Son una combinación de redes ResNet con capas de Atención multi cabeza.

Métricas de calidad:

A. Distancia de inicio de Fréchet (FID)

De acuerdo con Yu, Yu & Zhang, Weibin & Deng, Yun (2021). El FID calcula las activaciones (vectores de características) que se usan para estimar la media y la covarianza de los datos generados y de los datos reales. Las activaciones se combinan como un modelo Gaussiano multivariado. La distancia de Fréchet entre los Gaussianos se utiliza para cuantificar la calidad de las muestras generadas (Ecuación 8). El FID más bajo significa distancias más pequeñas entre los datos sintéticos y los reales. Ver figura 12.

$$FID(r,g) = \|\mu_r - \mu_g\|_2^2 + Tr(COV_r + COV_g - 2\sqrt{COV_r\, COV_g}) \quad \ldots (8)$$

```
def calculate_fid(act1, act2):
      # media y covarianza
      mu1, sigma1 = act1.mean(axis=0), cov(act1, rowvar=False)
      mu2, sigma2 = act2.mean(axis=0), cov(act2, rowvar=False)
      # suma de las diferencias de las medias al cuadrado
      ssdiff = numpy.sum((mu1 - mu2)**2.0)
      # raíz cuadrada del producto de covarianzas
      covmean = sqrtm(sigma1.dot(sigma2))
      # tomar la parte real para números complejos
      if iscomplexobj(covmean):
            covmean = covmean.real
      # calcular FID
      fid = ssdiff + trace(sigma1 + sigma2 - 2.0 * covmean)
      return fid
```

Certificado de Reserva de Derechos al Uso Exclusivo del Título: 04-2010-0326124-14000-203 ISSN: 2007-1957

Figura 12. Fragmento de código para el cálculo de la FID. Elaboración propia.

B. Precisión

Como lo señala Vizcaíno Salazar G. (2002) la precisión mide la fracción de predicciones que el modelo estimó como positivos y que corresponden realmente con casos positivos. Es una medida indirecta de la calidad de los clasificadores.

$$precisión = \frac{TP}{TP+FP} \ldots \text{ (9)}$$

C. Recuperación

La recuperación (recall) mide la proporción de casos verdaderos positivos. Es utilizada para saber cuántos valores positivos son correctamente clasificados.

$$recuperación = \frac{TP}{TP+FN} \ldots \text{ (10)}$$

Donde: Verdaderos positivos (TP): imágenes que han sido correctamente clasificadas como reales. Falsos negativos (FN): imágenes han sido clasificadas erróneamente como falsas. Verdaderos negativos (TN): imágenes que han sido correctamente clasificadas como falsas. Falsos positivos (FP): imágenes clasificadas erróneamente como reales.

La tabla 1 muestra los resultados obtenidos por las métricas de evaluación de los modelos reportados por P. Dhariwal and A. Nichol (2021). Se establece una comparación en la calidad de las muestras con modelos generativos de última aparición para cada tarea. ADM se refiere al modelo de difusión por ablación y ADM-G además utiliza la guía del clasificador. Los modelos de difusión apoyados en la base de datos LSUN se muestrearon utilizando 1000 pasos. Los modelos de difusión apoyados en la base de datos ImageNet son muestreados usando 250 pasos.

Tabla 1. Métricas en los modelos más destacados. Reportados en P. Dhariwal and A. Nichol (2021).

Model	FID	sFID	Prec	Rec
LSUN Bedrooms 256×256				
DCTransformer	6.40	6.66	0.44	**0.56**
DDPM	4.89	9.07	0.60	0.45
IDDPM	4.24	8.21	0.62	0.46
StyleGAN	2.35	6.62	0.59	0.48
ADM (dropout)	**1.90**	**5.59**	**0.66**	0.51
LSUN Horses 256×256				
StyleGAN2	3.84	6.46	0.63	0.48
ADM	2.95	**5.94**	0.69	**0.55**
ADM (dropout)	**2.57**	6.81	**0.71**	**0.55**
LSUN Cats 256×256				
DDPM	17.1	12.4	0.53	0.48
StyleGAN2	7.25	**6.33**	0.58	0.43
ADM (dropout)	**5.57**	6.69	**0.63**	**0.52**
ImageNet 64×64				
BigGAN-deep	4.06	3.96	**0.79**	0.48
IDDPM	2.92	**3.79**	0.74	0.62
ADM	2.61	**3.77**	0.73	0.63
ADM (dropout)	**2.07**	4.29	0.74	**0.63**

Model	FID	sFID	Prec	Rec
ImageNet 128×128				
BigGAN-deep	6.02	7.18	**0.86**	0.35
LOGAN	3.36			
ADM	5.91	**5.09**	0.70	**0.65**
ADM-G (25 steps)	5.98	7.04	0.78	0.51
ADM-G	**2.97**	**5.09**	0.78	0.59
ImageNet 256×256				
DCTransformer	36.51	8.24	0.36	**0.67**
VQ-VAE-2	31.11	17.38	0.36	0.57
IDDPM	12.26	5.42	0.70	0.62
SR3	11.30			
BigGAN-deep	6.95	7.36	**0.87**	0.28
ADM	10.94	6.02	0.69	0.63
ADM-G (25 steps)	5.44	5.32	0.81	0.49
ADM-G	**4.59**	**5.25**	0.82	0.52
ImageNet 512×512				
BigGAN-deep	8.43	8.13	**0.88**	0.29
ADM	23.24	10.19	0.73	**0.60**
ADM-G (25 steps)	8.41	9.67	0.83	0.47
ADM-G	**7.72**	**6.57**	0.87	0.42

Asimismo, en la tabla 2 se muestran los resultados obtenidos por el modelo propuesto en el presente trabajo.

Certificado de Reserva de Derechos al Uso Exclusivo del Título: 04-2010-0326124-14000-203 ISSN: 2007-1957

Tabla 2. Resultados de las métricas obtenidas por el modelo propuesto.

Model	FID	Prec	Rec
ImageNet 512×512			
U-Net	7.70	0.85	0.29
ImageNet 256×256			
U-Net	4.5	0.85	0.28
ImageNet 128×128			
U-Net	2.95	0.85	0.30
ImageNet 64×64			
U-Net	2.00	0.75	0.40

Finalmente, en la figura 13 se puede observar las gráficas de evolución de la Función de Pérdida de las diferentes arquitecturas.

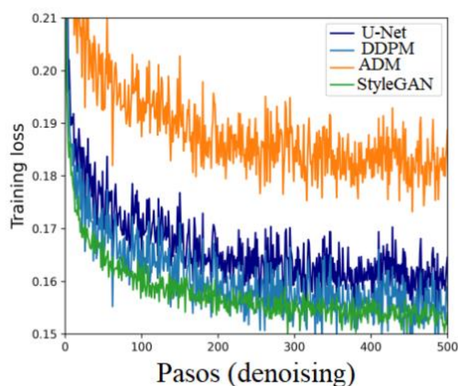

Figura 13. Gráfica de la evolución de la Función de Pérdida de las diferentes arquitecturas. Elaboración propia.

V. Conclusiones

Las redes generativas GAN ocupan actualmente el primer lugar en tareas de generación de imágenes sintéticas. Sin embargo, en el presente trabajo se pudo mostrar que el modelo propuesto de difusión por eliminación de ruido en forma iterativa pudo obtener un nivel de calidad igual o superior a los modelos generativos bien conocidos del estado del arte. Ver Tablas 1 y 2 respectivamente. En el presente trabajo, el modelo propuesto basado en la arquitectura de la red U-Net logró alcanzar unas métricas con valores cercanos al 10% de los valores más competitivos de las redes GAN. Del mismo modo, se pudo apreciar que el número de pasos requeridos para la generación de imágenes con una calidad aceptable se alcanzó cerca de los 500 pasos o intervalos temporales en el proceso de eliminación de ruido (denoising). Ver figura 13. Se puede considerar que las diferencias entre los modelos de difusión y las redes generadoras GAN se debe al menos a que éstas últimas han sido profundamente exploradas y se han refinado ampliamente, sin embargo, en este proceso han intercambiado la diversidad por la fidelidad, produciendo así muestras de alta calidad, pero sin lograr cubrir toda la distribución de datos. Con el modelo por difusión se buscó cubrir estas condiciones mejorando la arquitectura mediante la U-Net y luego diseñando un esquema para intercambiar diversidad por fidelidad.

VI. Referencias

Base de Datos utilizada: “ImageNet” de Kaggle, https://www.image-net.org/. (Última consulta 19-08-2024).

B. Kawar, M. Elad, S. Ermon and J. Song (2022). Denoising diffusion restoration models. Proc. DGM4HSD.

J. Ho, A. Jain and P. Abbeel (2020). Denoising diffusion probabilistic models. Proc. Int. Conf. Neural Inf. Process. Syst., pp. 6840-6851.

J. Shi, C. Wu, J. Liang, X. Liu and N. Duan (2022). DiVAE: Photorealistic images synthesis with denoising diffusion decoder.

O. Ronneberger, P. Fischer and T. Brox (2015). U-Net: Convolutional networks for biomedical image segmentation. Proc. 18th Int. Conf. Med. Image Comput. Comput.-Assist. Interv., pp. 234-241.

P. Dhariwal and A. Nichol (2021). Diffusion models beat GANs on image synthesis. Proc. Int. Conf. Neural Inf. Process. Syst., pp. 8780-8794.

Vizcaíno Salazar G. (2002). Sensibilidad y Especificidad. Medicina basada en la evidencia y análisis de diseños de investigación clínica. Maracaibo, Venezuela: EDILUZ; 57-71.

Yu, Yu & Zhang, Weibin & Deng, Yun (2021). Frechet Inception Distance (FID)for Evaluating GANs.