

Modelos Generativos Profundos

UN ENFOQUE DESDE LA
CREATIVIDAD
COMPUTACIONAL



EBM III

Clase 18

Dra. Wendy Aguilar

Modelos Basados en Energía

¿Qué aprende el modelo?

- Aprende una función de energía $E_\theta(x)$ que mide cuán probable o plausible es una imagen x .
- Asigna:
 - Datos reales → energía baja
 - Datos falsos → energía alta
- En lugar de aprender un generador o un decodificador, el EBM **modela directamente la distribución de los datos** como:
$$p_\theta(x) = \frac{e^{-E_\theta(x)}}{Z_\theta}$$
- En otras palabras: **aprende el relieve del paisaje de energía** del espacio de imágenes.

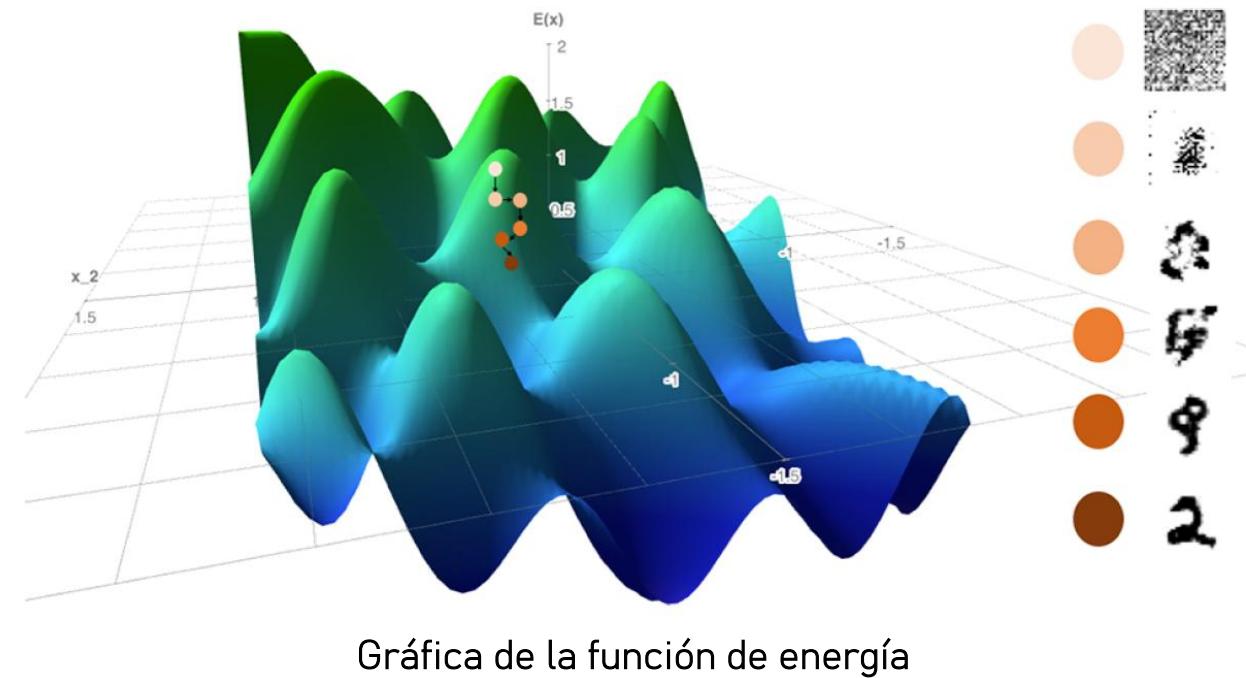
Arquitectura de la red que aprende la función de energía

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 32, 32, 1)	0
conv2d (Conv2D)	(None, 16, 16, 16)	416
conv2d_1 (Conv2D)	(None, 8, 8, 32)	4,640
conv2d_2 (Conv2D)	(None, 4, 4, 64)	18,496
conv2d_3 (Conv2D)	(None, 2, 2, 64)	36,928
flatten (Flatten)	(None, 256)	0
dense (Dense)	(None, 64)	16,448
dense_1 (Dense)	(None, 1)	65

Modelos Basados en Energía

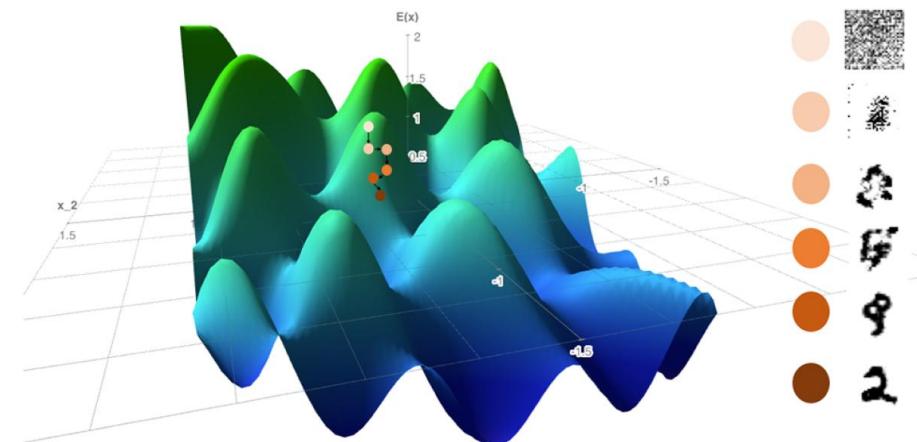
¿Cómo aprende?

- Usa Contrastive Divergence (CD) para comparar ejemplos reales vs. falsos:
 - Minimiza $E_\theta(x_{real})$
 - Maximiza $E_\theta(x_{fake})$
- Los ejemplos falsos provienen de un **buffer de muestras** que se actualiza continuamente.
- Este proceso hace que el modelo “esculpa” su paisaje de energía, formando **valles en torno a los datos reales**.
- La red ajusta sus parámetros para que las imágenes del dataset caigan en regiones de baja energía.



El EBM no genera imágenes porque tenga un generador, sino porque **aprende el relieve energético del mundo visual** y sabe **caminar dentro de él** para encontrar ejemplos plausibles.

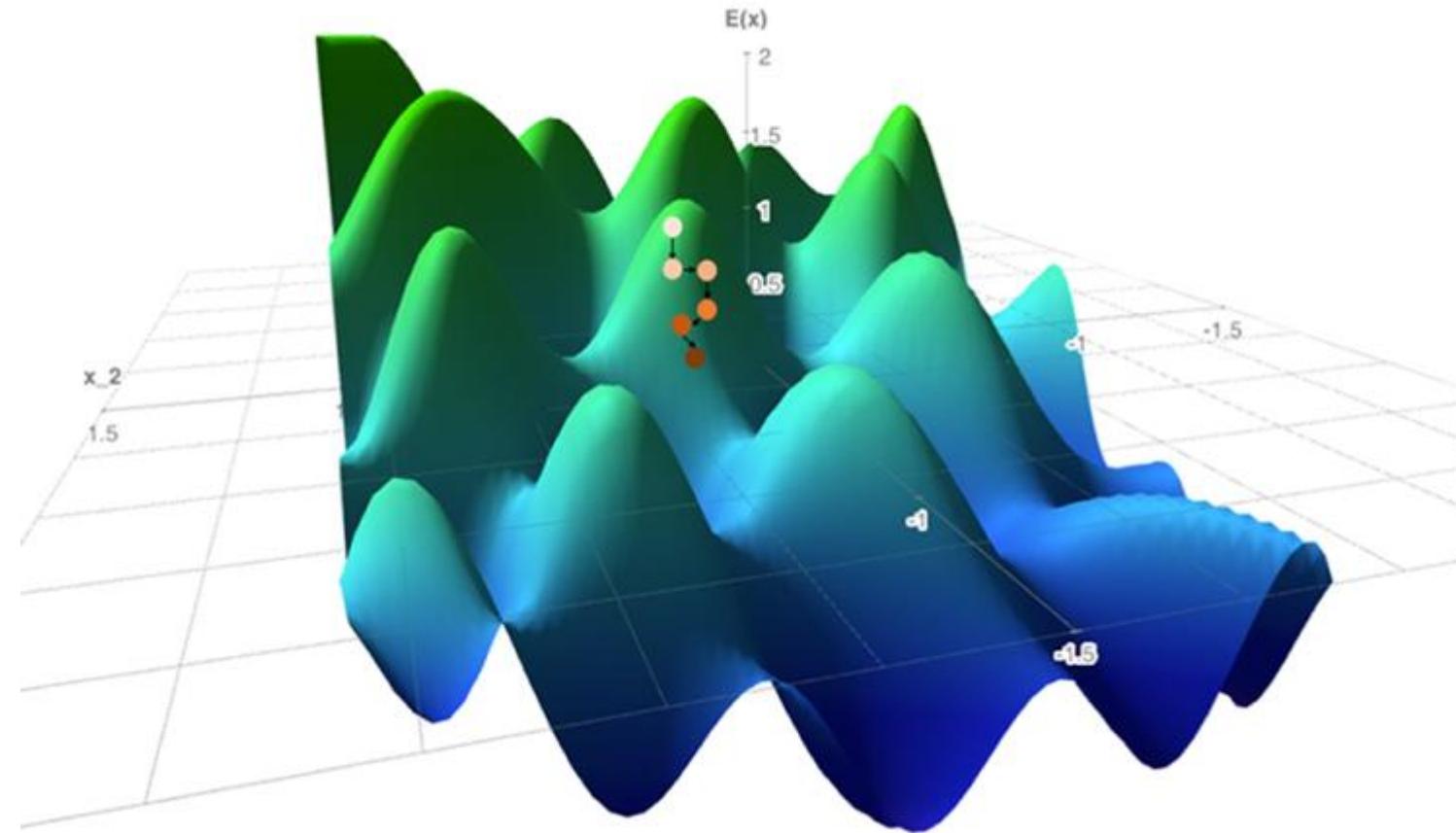
Por tanto, el poder generativo del modelo surge de **cómo esculpe su paisaje energético**, no de una decodificación latente.





En una gráfica de la energía en 3D como esta

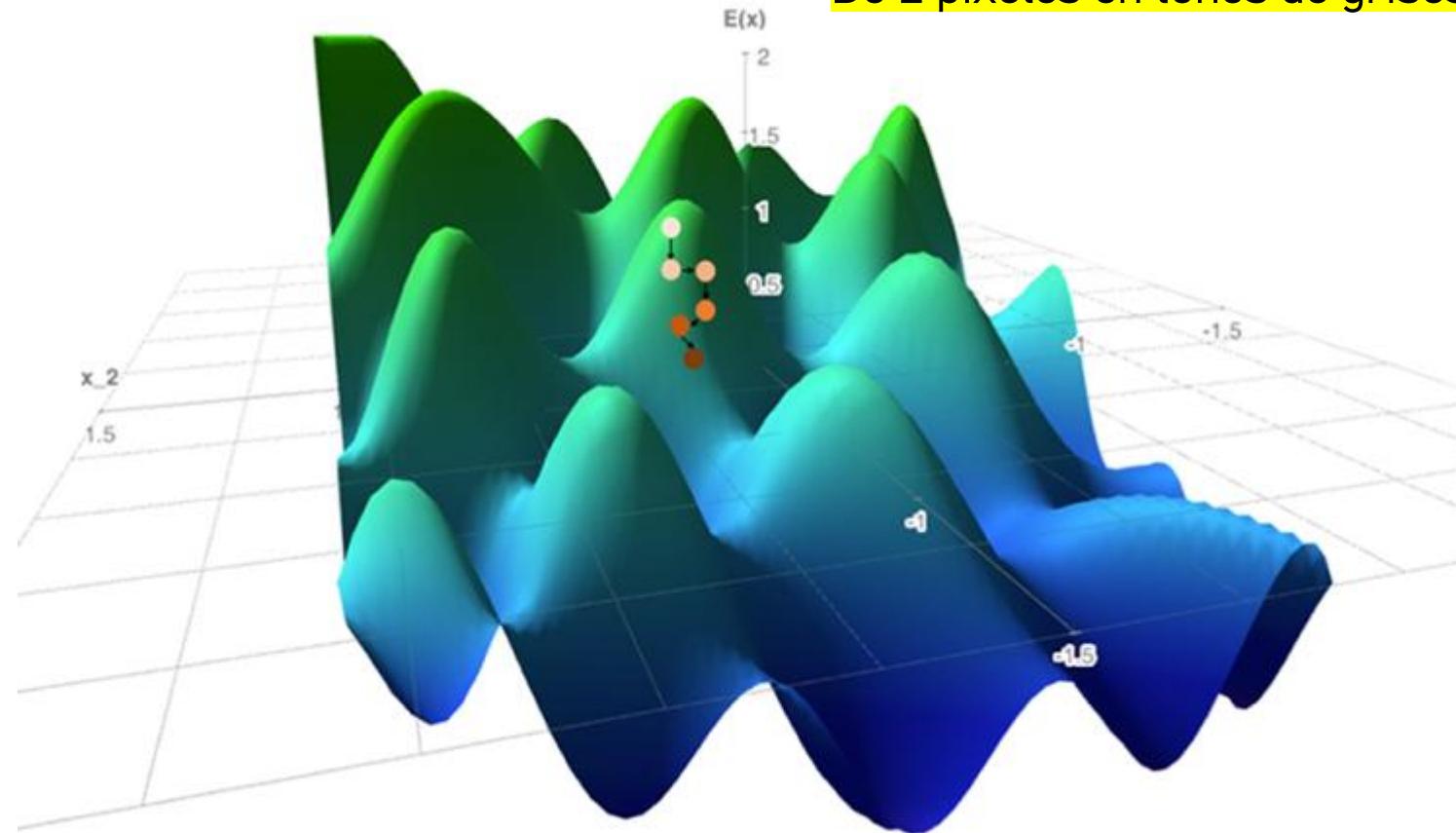
¿de qué tamaño serían las imágenes que ese modelo está representando y por qué?



En una gráfica de la energía en 3D como esta

¿de qué tamaño serían las imágenes que ese modelo está representando y por qué?

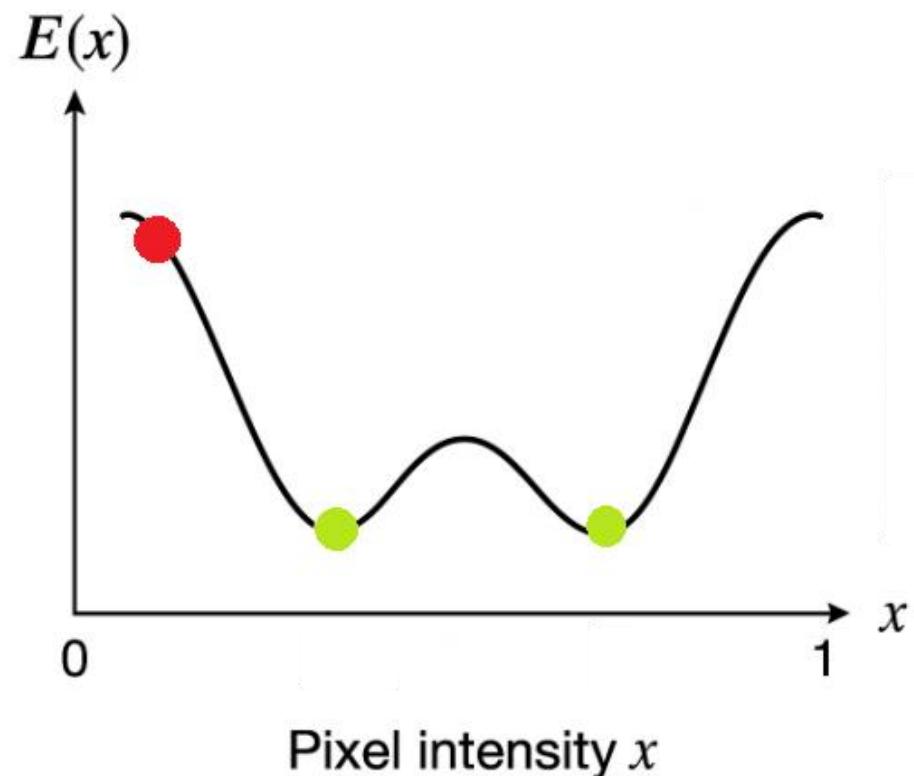
De 2 pixeles en tonos de grises





¿Cómo se vería el paisaje de energía si quisieramos generar imágenes de un pixel en tonos de grises?

¿Cómo se vería el paisaje de energía si quisiéramos generar imágenes de un pixel en tonos de grises?





¿En qué dimensión estaría el paisaje de energía para las imágenes de MNIST de 32x32 pixeles en escala de grises?

¿En qué dimensión estaría el paisaje de energía para las imágenes de MNIST de 32x32 pixeles en escala de grises?

$$E : \mathbb{R}^{1024} \rightarrow \mathbb{R}$$



¿En qué dimensión estaría el paisaje de energía para las imágenes de CIFAR10 de 32x32 pixeles a color?

¿En qué dimensión estaría el paisaje de energía para las imágenes de CIFAR10 de 32x32 pixeles a color?

$$E : \mathbb{R}^{3072} \rightarrow \mathbb{R}$$

Ejercicio de tarea



Enviar a

wendysan@hotmail.com antes
de la clase del miércoles 22
de octubre

- Crear una **libreta de Google Colab** donde implementarás paso a paso el **modelo basado en energía** visto en clase.
- Deberás **completar el código mostrado durante la sesión** con los fragmentos necesarios para que el modelo pueda **entrenarse correctamente** sobre el conjunto de datos de imágenes MNIST.

ebm.ipynb



Generación de nuevas muestras a partir del EBM

[actividad_de_tarea_EBMs.pdf](#)



Actividad:

Generación de imágenes con un Energy-Based Model (EBM)

Objetivo

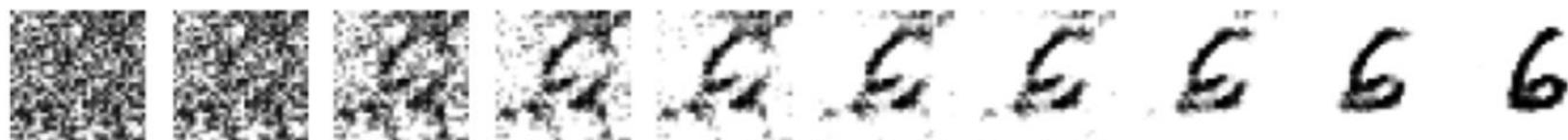
Explorar cómo un modelo basado en energía (EBM) genera nuevas imágenes a partir de ruido aleatorio, y analizar cómo la calidad y diversidad de las muestras varía con el número de épocas de entrenamiento.

Esta actividad les permitirá:

- Comprender que el EBM no genera a partir de una distribución gaussiana fija, sino de su propia **distribución de energía** aprendida.
- Observar empíricamente cómo el entrenamiento moldea el paisaje de energía, haciendo que el modelo produzca imágenes más plausibles a medida que aprende.

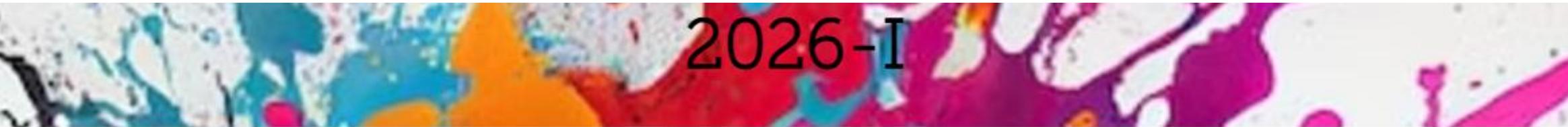


Muestras generadas del modelo entrenado por 50 épocas



Step 0 1 3 5 10 30 50 100 300 999





2026-I

Inicio

Presentacion

Temario

Clases

Proyectos de profundización

Contacto

Clase 17

22 de octubre, 2025

Modelos basados en energía II

Recursos

- [actividad de tarea EBMs.pdf](#)
- [ebm.ipynb](#)
- [ebm entrenado diferentes épocas.zip](#)

Evolución de los Modelos Basados en Energía

1985	Boltzmann Machine	Primer EBM; red no dirigida con unidades visibles y ocultas; muestreo Gibbs; muy costosa de entrenar.
1986	Restricted Boltzmann Machine	Estructura bipartita (sin conexiones intra-capa); entrenamiento más estable; base de las <i>Deep Belief Networks</i> .
2006	Deep Belief Networks	Apilamiento de RBMs; entrenamiento capa a capa; preludio de los <i>Deep EBMs</i> .
2008–2014	Score Matching y EBMs continuos	Transición del muestreo binario a continuo; introducción del <i>denoising score matching</i> .
2015–2018	Reaparición de los EBMs en deep learning	Surgen intentos de entrenar EBMs con redes profundas, pero inestables.
→ 2019	Implicit Generation and Modeling with Energy-Based Models	EBM profundo moderno con entrenamiento estable mediante contrastive divergence + Langevin dynamics . <ul style="list-style-type: none">◆ Muestra que los EBMs pueden generar imágenes de alta dimensión (CIFAR-10, ImageNet-32) sin usar redes explícitas generativas (sin decoder).◆ Este paper revitaliza el interés en EBMs dentro de la IA generativa moderna.
2020–2023	Score-based EBMs / Diffusion Models	<ul style="list-style-type: none">• Aparecen los Denoising Diffusion Probabilistic Models (DDPMs).• Se demuestra que los modelos de difusión aprenden el gradiente del logaritmo de probabilidad (<i>score</i>), equivalente a un campo de energía.• Estos modelos se convierten en la base de los sistemas generativos más avanzados: DALL-E 2 y Stable Diffusion.



Modelos de difusión

Clase 18

Dra. Wendy Aguilar

Modelos Generativos Profundos

UN ENFOQUE DESDE LA
CREATIVIDAD
COMPUTACIONAL

Modelos de Difusión con Eliminación de Ruido (DDM)

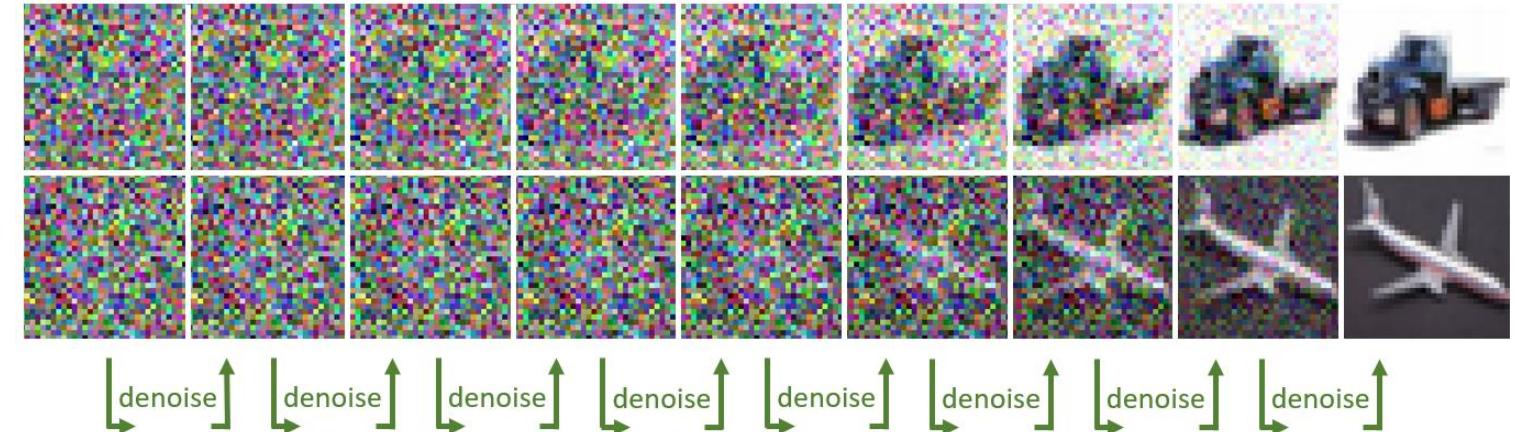
Denoising Diffusion Models (DDM)
2020

La idea es muy sencilla:

Entrenamos un modelo de aprendizaje profundo para **eliminar el ruido** de una **imagen** a lo largo de una **serie de pasos muy pequeños**.

Si comenzamos desde **ruido puro y aleatorio**, en teoría, deberíamos poder aplicar repetidamente el modelo **hasta obtener una imagen que parezca provenir del conjunto de entrenamiento**.

¡Lo sorprendente es que este concepto tan simple funciona excepcionalmente bien en la práctica!



Oxford 102 Flowers Dataset

ddm.ipynb


Ejecutar celdas:

0. Parámetros

1. Preparación de los datos

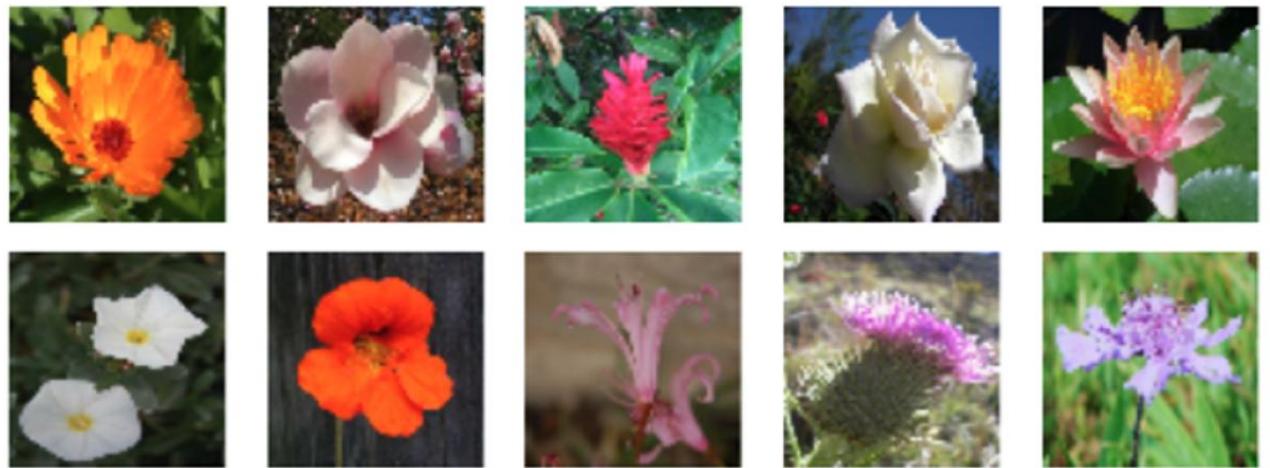
 1.1 Carga de credenciales de Kaggle

 1.2 Configuración del acceso a Kaggle

 1.3 Descarga y extracción del dataset

 1.4 Carga del dataset como conjunto de entrenamiento

 1.5 Preprocesamiento del dataset



- 102 clases (diferentes especies de flores)
- 8,189 imágenes

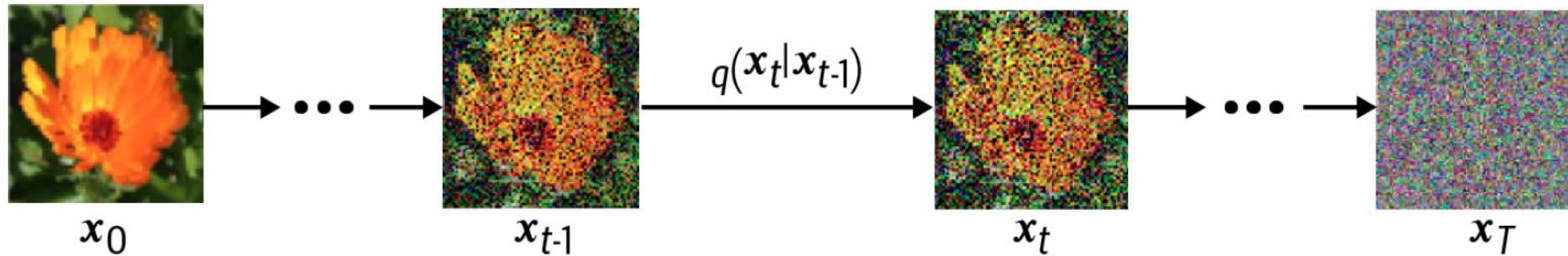
Nota: Si tienes acceso a una GPU puedes poner a ejecutar todo el notebook. **Aproximadamente 1 hora**
Solo necesitas ajustar la primera celda a tu estructura de directorios:

```
base_dir = "/content/drive/MyDrive/Colab Notebooks/CursolAGenerativa/notebooks/Clase17_EBM"
```

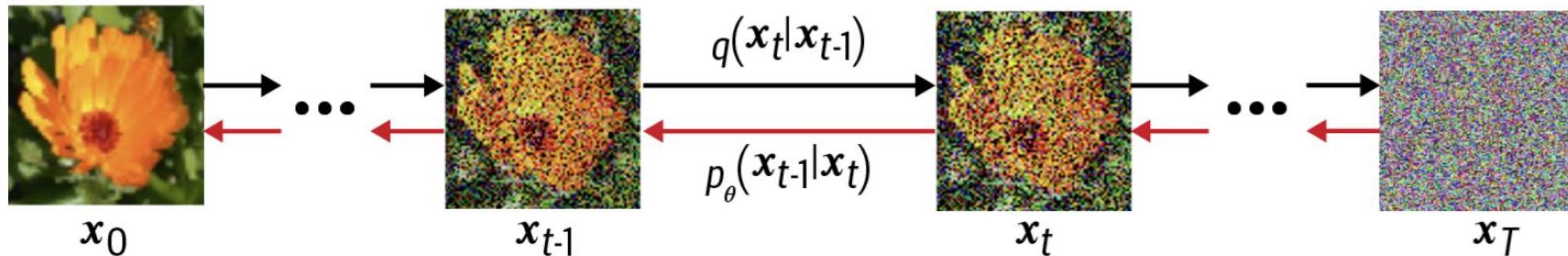
Modelos de Difusión con Eliminación de Ruido (DDM)

El proceso ocurre en **dos fases**:

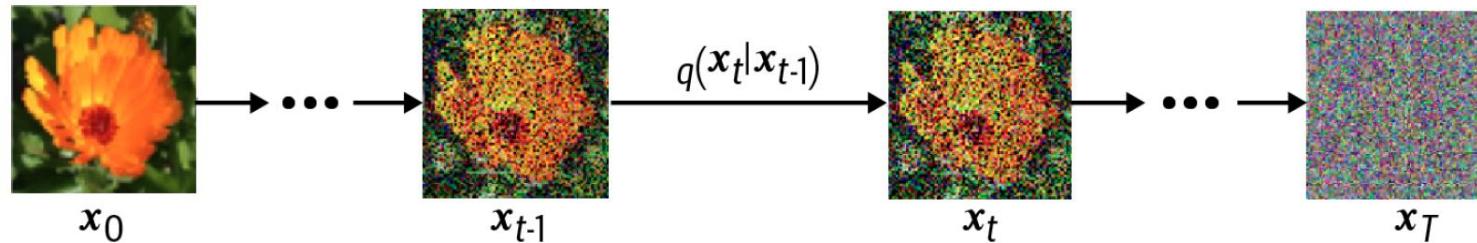
Difusión forward: se agrega ruido progresivamente.



Difusión backward: el modelo aprende a quitar el ruido.



El proceso de difusión hacia adelante



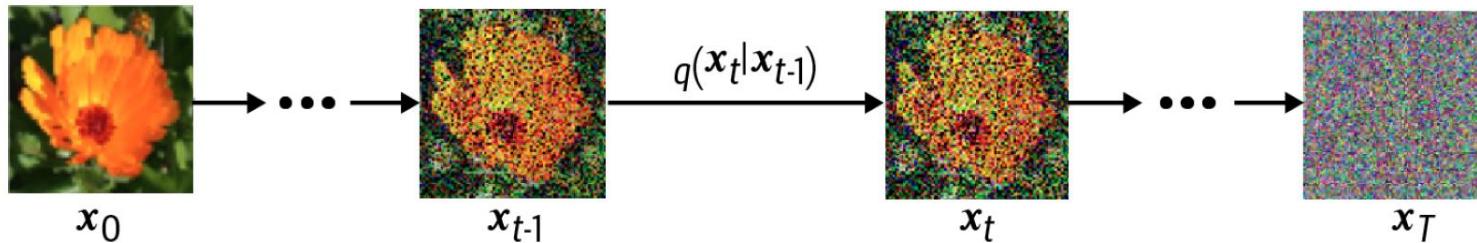
Idea básica:

Empezamos con una imagen real x_0 y le vamos agregando ruido poco a poco hasta que se convierte en puro ruido gaussiano.

Cada paso agrega una pequeña cantidad de ruido controlada por un parámetro β_t .

Cuanto más grande sea t , más ruido tiene la imagen.

El proceso de difusión hacia adelante



Idea básica:

Empezamos con una imagen real x_0 y le vamos **agregando ruido poco a poco** hasta que se convierte en puro ruido gaussiano.

Cada paso agrega una pequeña cantidad de ruido controlada por un parámetro β_t .

Cuanto más grande sea t , más ruido tiene la imagen.

Ecuación del proceso:

$$x_t = \underbrace{\sqrt{1 - \beta_t} x_{t-1}}_{\text{La imagen en el tiempo } t} + \underbrace{\sqrt{\beta_t} \epsilon_{t-1}}_{\text{Tomamos una fracción de la imagen anterior,}}$$

Cuanto más grande sea β_t , menor será $\sqrt{1 - \beta_t}$ es decir, **menos de la imagen original se mantiene y más ruido se añade.**

Asegura que la **varianza total permanezca constante** a lo largo de todos los pasos.

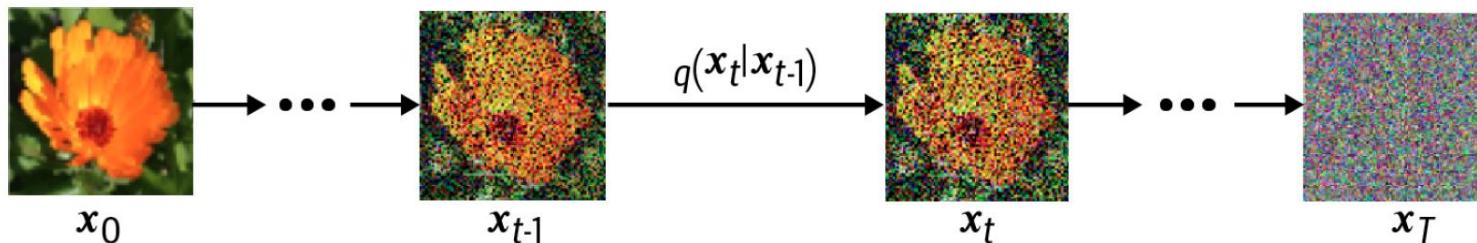
La escala del ruido que se añade al paso t .

Determina **cuánta perturbación aleatoria** introducimos.

Valores pequeños de β_t → ruido suave; valores grandes → ruido más fuerte.

Ruido Gaussiano con media 0 y varianza 1

El proceso de difusión hacia adelante



Idea básica:

Empezamos con una imagen real x_0 y le vamos **agregando ruido poco a poco** hasta que se convierte en puro ruido gaussiano.

Cada paso agrega una pequeña cantidad de ruido controlada por un parámetro β_t .

Cuanto más grande sea t , más ruido tiene la imagen.

Objetivo:

Lograr que después de muchos pasos ($T \approx 1000$), la imagen final x_T se parezca completamente a ruido puro $\mathcal{N}(0, I)$

Ecuación del proceso:

$$x_t = \underbrace{\sqrt{1 - \beta_t} x_{t-1}}_{\text{La imagen en el tiempo } t} + \underbrace{\sqrt{\beta_t} \epsilon_{t-1}}_{\text{Tomamos una fracción de la imagen anterior,}}$$

Cuanto más grande sea β_t , menor será $\sqrt{1 - \beta_t}$ es decir, **menos de la imagen original se mantiene y más ruido se añade**.

Asegura que la **varianza total permanezca constante** a lo largo de todos los pasos.

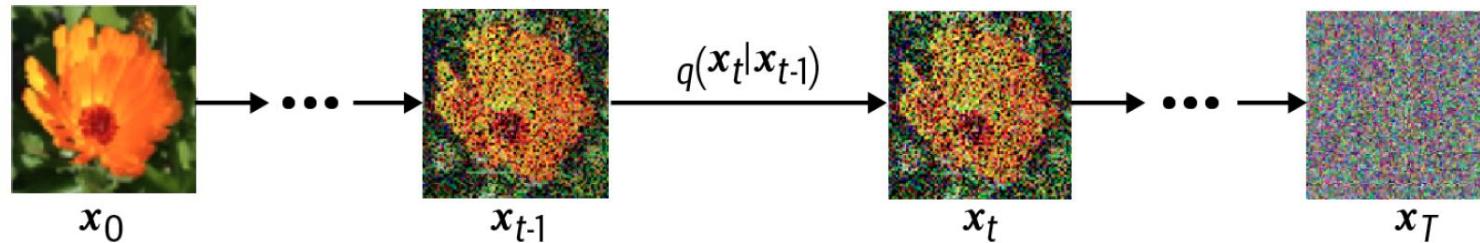
La escala del ruido que se añade al paso t .

Determina **cuánta perturbación aleatoria** introducimos.

Valores pequeños de $\beta_t \rightarrow$ ruido suave; valores grandes \rightarrow ruido más fuerte.

Ruido Gaussiano con media 0 y varianza 1

El proceso de difusión hacia adelante



Idea básica:

Empezamos con una imagen real x_0 y le vamos **agregando ruido poco a poco** hasta que se convierte en puro ruido gaussiano.

Cada paso agrega una pequeña cantidad de ruido controlada por el parámetro β_t .

Cuanto más grande sea t , más ruido tiene la imagen.

Objetivo:

Lograr que después de muchos pasos ($T \approx 1000$), la imagen final x_T se parezca completamente a ruido puro $\mathcal{N}(0, I)$

Observación:

El proceso de difusión directa **no aprende nada**; solo define cómo **corrompemos las imágenes** para que el modelo aprenda luego a revertir ese proceso.

Ecuación del proceso:

$$x_t = \underbrace{\sqrt{1 - \beta_t} x_{t-1}}_{\text{La imagen en el tiempo } t} + \underbrace{\sqrt{\beta_t} \epsilon_{t-1}}_{\text{Tomamos una fracción de la imagen anterior,}}$$

Cuanto más grande sea β_t , menor será $\sqrt{1 - \beta_t}$ es decir, **menos de la imagen original se mantiene y más ruido se añade**.

Asegura que la **varianza total permanezca constante** a lo largo de todos los pasos.

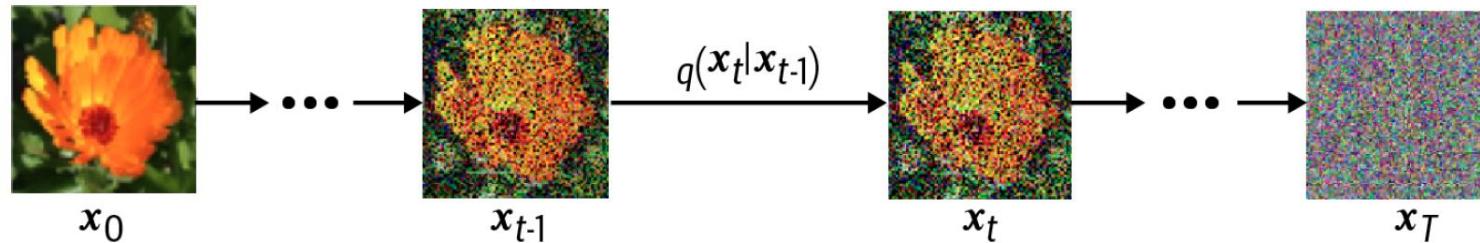
La escala del ruido que se añade al paso t .

Determina **cuánta perturbación aleatoria** introducimos.

Valores pequeños de $\beta_t \rightarrow$ ruido suave; valores grandes \rightarrow ruido más fuerte.

Ruido Gaussiano con media 0 y varianza 1,

El proceso de difusión hacia adelante



Ecuación del proceso:

¿cómo calcular x_t ?

$$x_t = \sqrt{1 - \beta_t} x_{t-1} + \sqrt{\beta_t} \epsilon_{t-1}$$

La podemos describir desde una perspectiva probabilística:

Forma probabilística (ecuación de distribución)

¿cómo está distribuido x_t ?

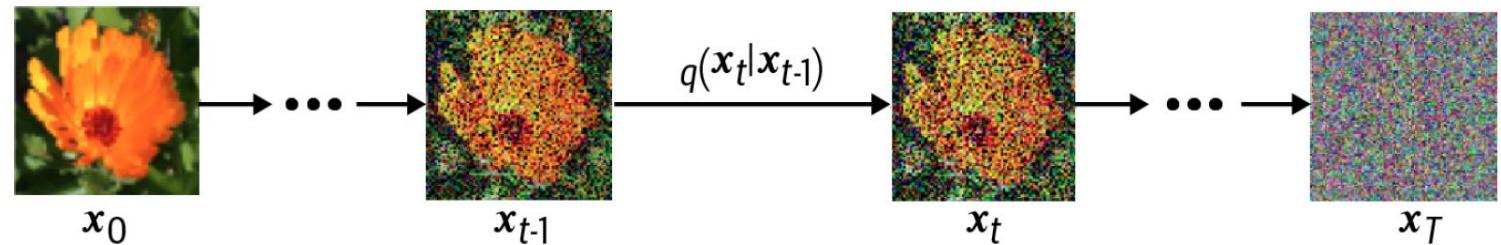
Dado x_{t-1} , x_t sigue una distribución normal cuya media es $\sqrt{1 - \beta_t} x_{t-1}$ y cuya varianza es β_t .

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1 - \beta_t} x_{t-1}, \beta_t I)$$

El truco de la reparametrización

¿Qué queremos?

Poder saltar directamente desde la imagen original x_0 a cualquier paso intermedio x_t , sin tener que aplicar el proceso de ruido t veces.



¿Cómo lo podemos hacer?

En lugar de añadir ruido paso a paso, podemos **reformular la ecuación** de forma que combine todos los pasos en uno solo.

$$x_t = \sqrt{1 - \beta_t} x_{t-1} + \sqrt{\beta_t} \epsilon_{t-1}$$

Definiciones útiles

$$\alpha_t = 1 - \beta_t$$

Fracción de la imagen que se conserva en cada paso.

$$\bar{\alpha}_t = \prod_{i=1}^t \alpha_i$$

Cantidad total de la imagen original que sobrevive tras t pasos.

Ecuación resultante

$$x_t = \underbrace{\sqrt{\bar{\alpha}_t} x_0}_{\text{Conserva información del señal original}} + \underbrace{\sqrt{1 - \bar{\alpha}_t} \epsilon}_{\text{Representa el ruido acumulado.}}$$

Desde el paso 0

- Podemos pensar que cada imagen x_t es una **mezcla lineal** entre:
 - La imagen original x_0 y ruido gaussiano ϵ .
 - El peso de cada uno depende de cuántos pasos de ruido hayamos simulado (t).

El Reparameterization Trick convierte el proceso iterativo en una **expresión cerrada**.

El truco de la reparametrización

El truco de reparametrización

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon, \quad \text{con } \epsilon \sim \mathcal{N}(0, I)$$

Nos dice **cómo obtener una muestra** de esa distribución usando una variable de ruido estándar.

Forma probabilística del proceso

$$q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t} x_0, (1 - \bar{\alpha}_t)I)$$

Esta es la **forma de distribución (teórica)**: describe la **probabilidad** de que x_t tome un valor dado, condicionado a la imagen original x_0 .

$$q(x_t|x_0)$$

Define **una distribución gaussiana** que describe todas las posibles imágenes ruidosas en el paso t .

Diffusion Schedules

Idea principal:

- En cada paso t , el modelo añade una cantidad de ruido controlada por β_t .
- Podemos elegir libremente cómo cambia β_t a lo largo del tiempo.
- A esta secuencia de valores $\{\beta_t\}$ se le llama **diffusion schedule**.

¿Por qué necesitamos una Schedule?

- Quisiéramos añadir una cantidad diferente de ruido en cada paso.
- Al principio, la imagen aún conserva muchos detalles → usamos **ruido suave**.
- Más adelante, ya es casi ruido puro → podemos añadir **ruido más fuerte**.
- El cambio de β_t con el tiempo determina **la estabilidad del entrenamiento y la calidad de las muestras generadas**.

Diffusion Schedules

1. Linear Schedule (Ho et al., 2020)

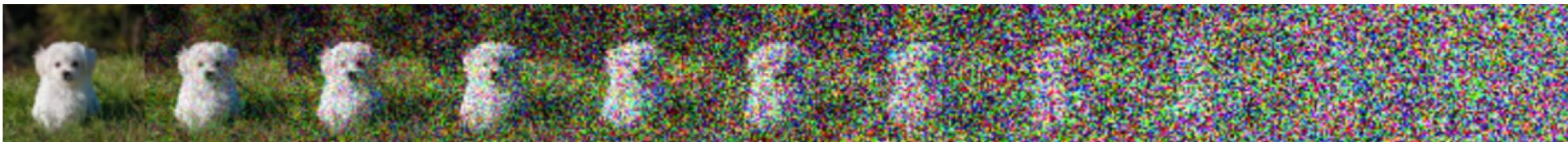
- β_t aumenta **linealmente** con t , típicamente desde 0.0001 hasta 0.02.
- Fue la configuración usada en el **paper original** de DDPM.
- **Ventaja:** simple e intuitiva.
- **Desventaja:** Tiende a degradar la señal demasiado pronto



2. Cosine Schedule (Nichol & Dhariwal, 2021)

- Posteriormente se descubrió que una programación **cosenoidal** mejora los resultados.
- El ruido aumenta **más lentamente** al inicio y **más suavemente** en general.
- Mejora la **eficiencia** del entrenamiento y la **calidad** de las imágenes generadas.

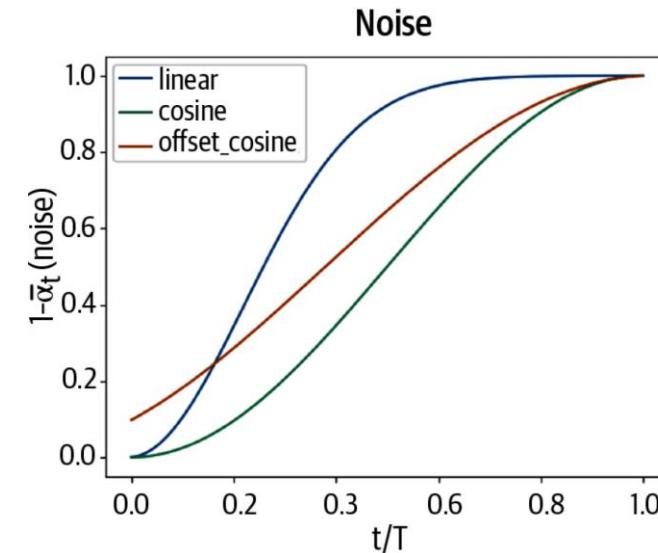
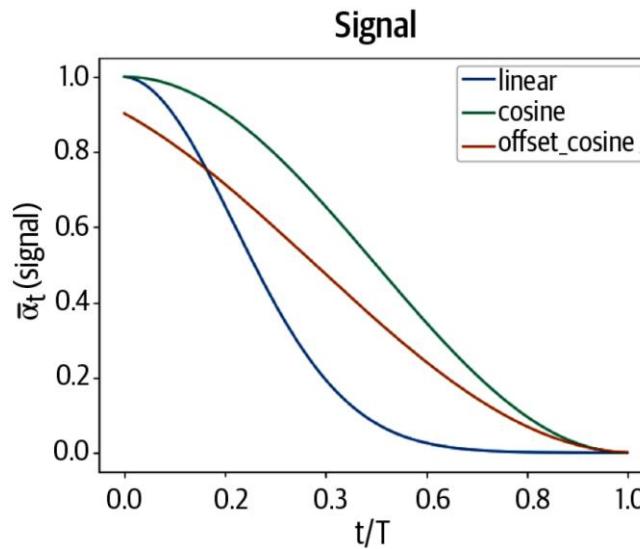
$$\alpha_t = \cos^2\left(\frac{t}{T} \frac{\pi}{2}\right)$$



Diffusion Schedules

3. Offset Cosine Schedule

- Variante ajustada del coseno: se añade un **desplazamiento (offset)** y un **factor de escala**.
- Evita que los primeros pasos sean demasiado pequeños.
- Ofrece un **equilibrio entre estabilidad y diversidad** en las muestras.



- El *diffusion schedule* controla el **ritmo de degradación de la imagen**.
- Elegir una buena schedule (lineal, cosenoidal u otras) es esencial para lograr:
 - **entrenamientos más estables,**
 - **mejor calidad de muestreo,**
 - **menor tiempo de convergencia.**

Diffusion Schedules

```
def linear_diffusion_schedule(diffusion_times):
    min_rate = 0.0001
    max_rate = 0.02
    betas = min_rate + tf.convert_to_tensor(diffusion_times) * (max_rate - min_rate)
    alphas = 1 - betas
    alpha_bars = tf.math.cumprod(alphas)
    signal_rates = alpha_bars
    noise_rates = 1 - alpha_bars
    return noise_rates, signal_rates

T = 1000
diffusion_times = [x/T for x in range(T)] ❶
linear_noise_rates, linear_signal_rates = linear_diffusion_schedule(
    diffusion_times
) ❷
```

- ❶ The diffusion times are equally spaced steps between 0 and 1.
- ❷ The linear diffusion schedule is applied to the diffusion times to produce the noise and signal rates.

Diffusion Schedules



Ejecutar celdas:

1.6 Diffusion schedules

```
def cosine_diffusion_schedule(diffusion_times): ❶
    signal_rates = tf.cos(diffusion_times * math.pi / 2)
    noise_rates = tf.sin(diffusion_times * math.pi / 2)
    return noise_rates, signal_rates

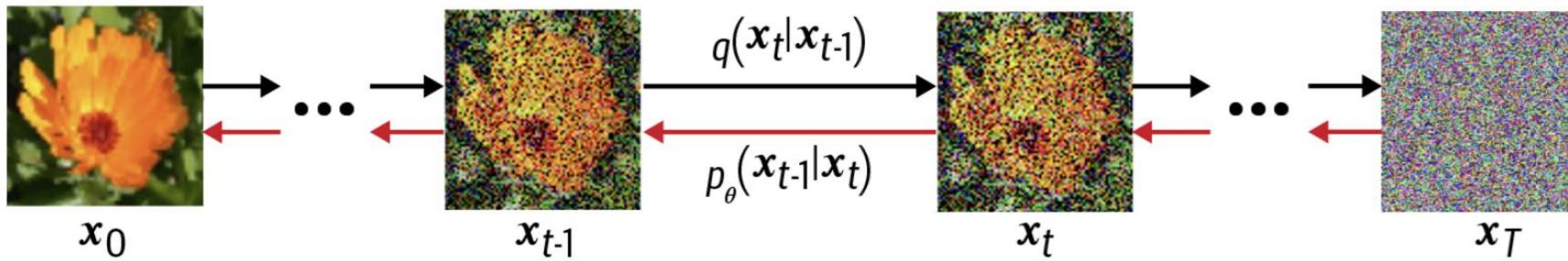
def offset_cosine_diffusion_schedule(diffusion_times): ❷
    min_signal_rate = 0.02
    max_signal_rate = 0.95
    start_angle = tf.acos(max_signal_rate)
    end_angle = tf.acos(min_signal_rate)

    diffusion_angles = start_angle + diffusion_times * (end_angle - start_angle)

    signal_rates = tf.cos(diffusion_angles)
    noise_rates = tf.sin(diffusion_angles)
    return noise_rates, signal_rates
```

- ❶ The pure cosine diffusion schedule (without offset or rescaling).
- ❷ The offset cosine diffusion schedule that we will be using, which adjusts the schedule to ensure the noising steps are not too small at the start of the noising process.

El proceso de difusión hacia atrás



Objetivo general:

Aprender un modelo neuronal para aproximar la distribución inversa $p_\theta(x_{t-1} | x_t)$ que **deshaga el proceso de difusión**, es decir, que reconstruya una imagen menos ruidosa a partir de una imagen más ruidosa.

El modelo neuronal p_θ

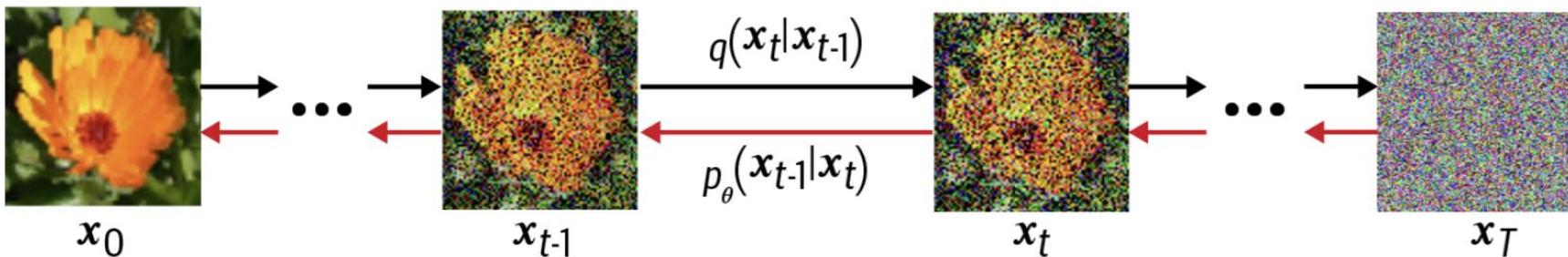
- El modelo $p_\theta(x_{t-1} | x_t)$ se entrena para predecir el ruido que fue añadido en el paso t.
- Esto permite reconstruir x_{t-1} como:

$$x_{t-1} \approx \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}} \epsilon_\theta(x_t, t) \right)$$

Factor de
normalización
para mantener
la varianza.

Escala la cantidad de
ruido predicho que se
debe eliminar.

El proceso de difusión hacia atrás



Objetivo general:

Aprender un modelo neuronal para aproximar la distribución inversa $p_\theta(x_{t-1} | x_t)$ que **deshaga el proceso de difusión**, es decir, que reconstruya una imagen menos ruidosa a partir de una imagen más ruidosa.

El modelo neuronal p_θ

- El modelo $p_\theta(x_{t-1} | x_t)$ se entrena para predecir el ruido que fue añadido en el paso t.
- Esto permite reconstruir x_{t-1} como:

$$x_{t-1} \approx \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_\theta(x_t, t) \right)$$

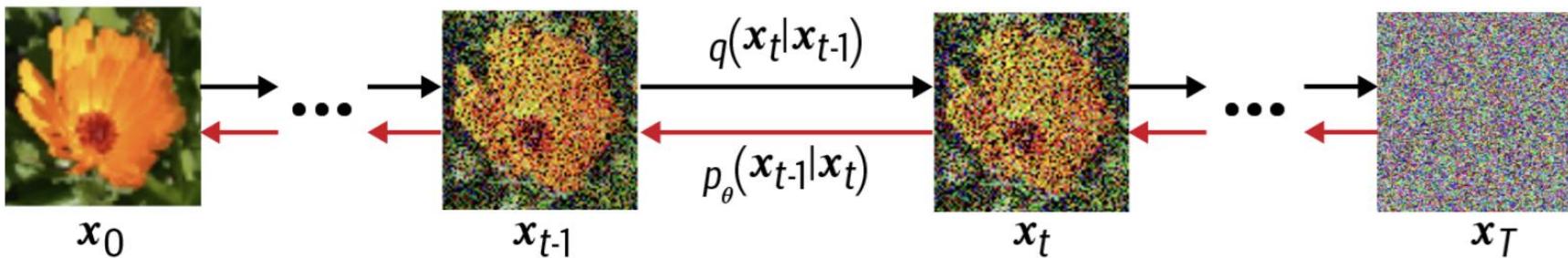
Factor de normalización para mantener la varianza.

Escala la cantidad de ruido predicho que se debe eliminar.

Función de pérdida (Loss)

- Se usa una pérdida de tipo **Mean Squared Error (MSE)**, igual que en un VAE.
- En cada iteración:
 1. Se toma una imagen real x_0 .
 2. Se le añade ruido t veces → obtenemos x_t .
 3. El modelo recibe x_t y t , y predice el ruido $\hat{\epsilon}$.
 4. Se calcula el error cuadrático entre ϵ y $\hat{\epsilon}$.
 5. Se actualizan los pesos de la red.

El proceso de difusión hacia atrás



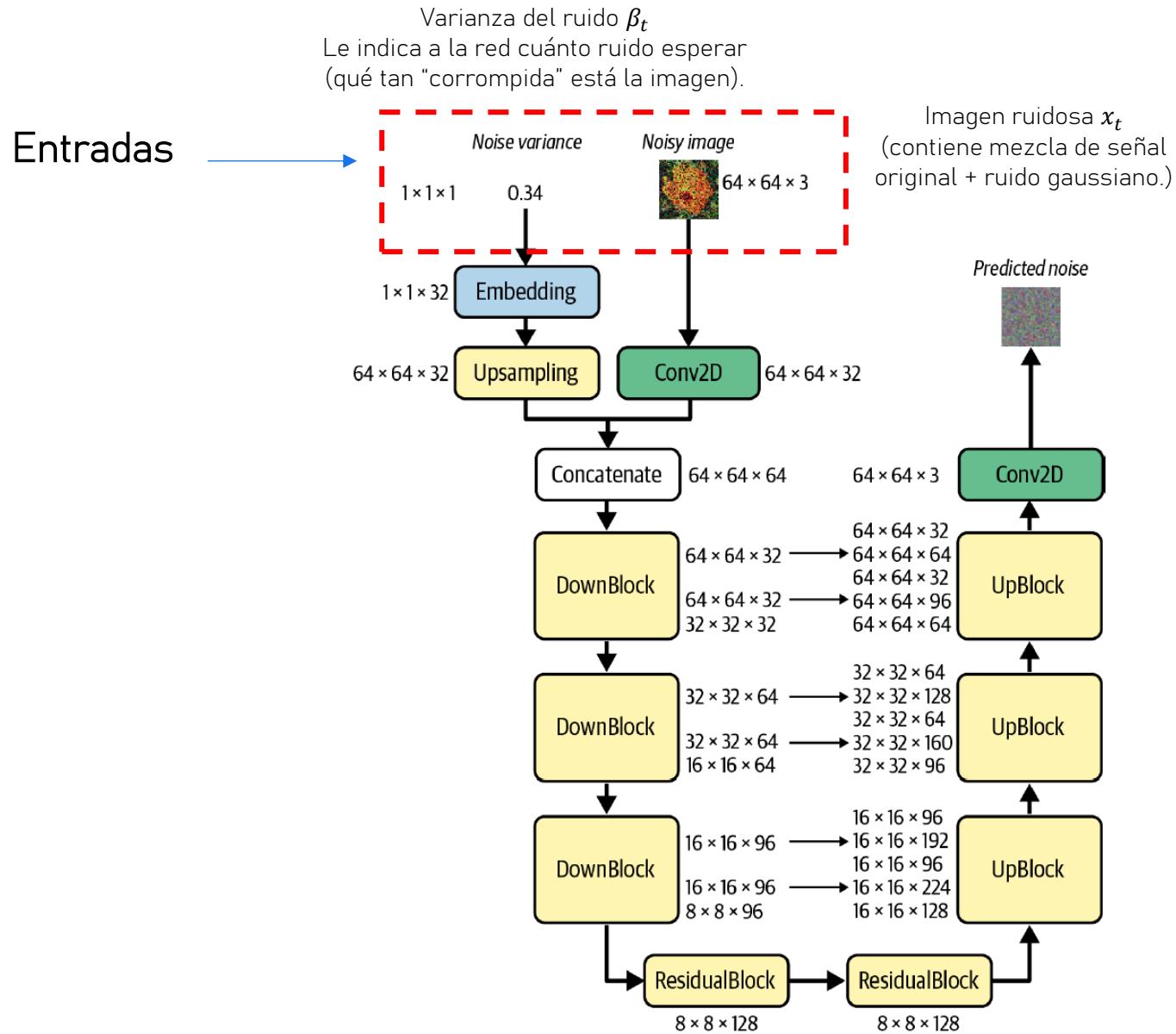
El uso de dos redes durante el entrenamiento

- Se mantienen **dos copias del modelo**:
 - **Red activa** (entrenada): actualizada mediante descenso de gradiente.
 - **Red EMA (Exponential Moving Average)**: promedio móvil exponencial de los pesos de la red activa.
- **Propósito**:
 - La EMA suaviza las fluctuaciones del entrenamiento.
 - Produce **imágenes más estables y coherentes** al generar.

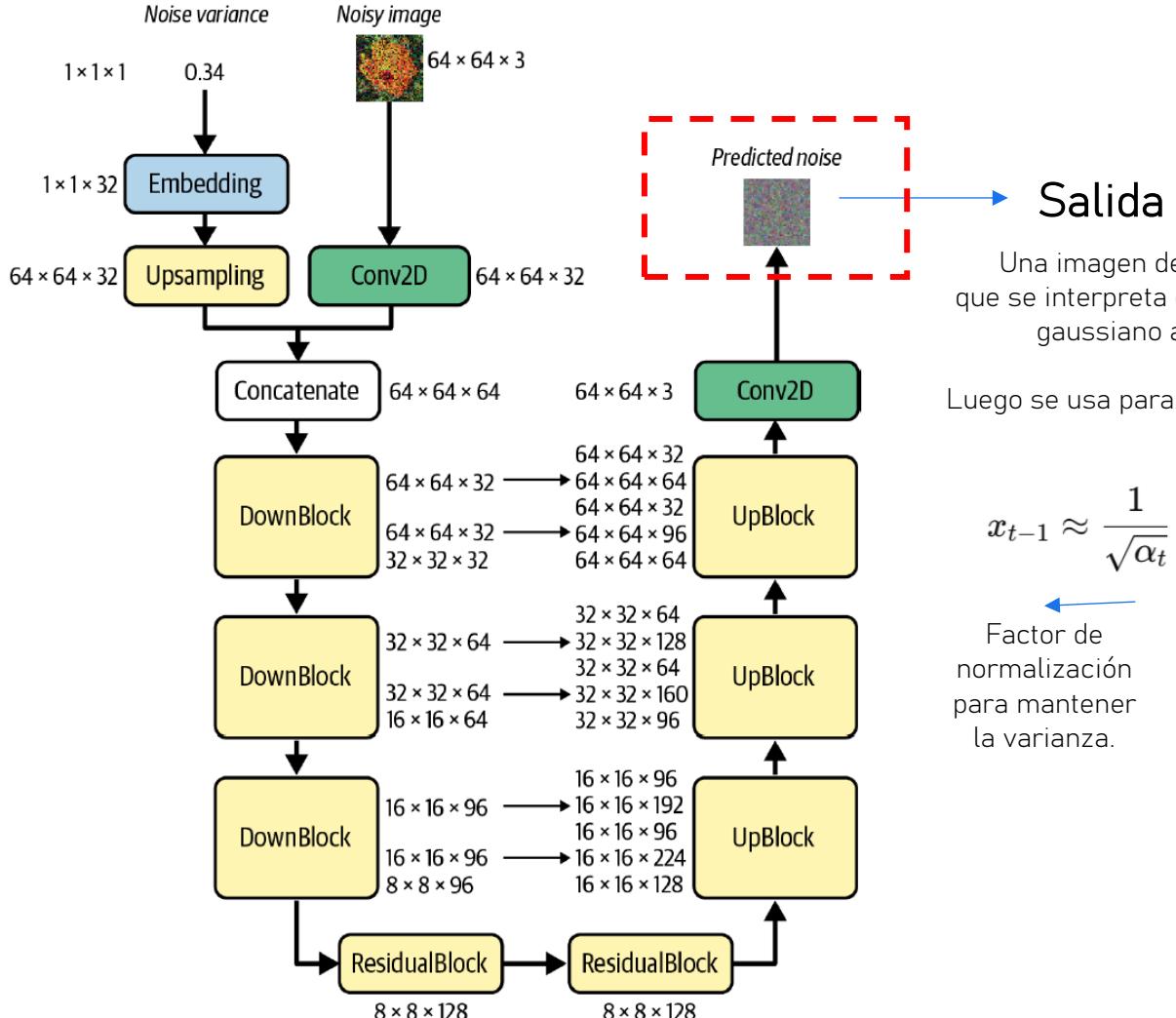
Durante la generación

- Partimos de ruido puro $x_T \sim \mathcal{N}(0, I)$.
- Aplicamos el modelo p_θ repetidamente (T veces):
 - $x_T \rightarrow x_{T-1} \rightarrow x_{T-2} \rightarrow \dots \rightarrow x_0$
- Al final obtenemos una **nueva imagen sintética**, coherente y realista.
- La versión EMA del modelo es la que se usa para **generar imágenes limpias y estables**.

Arquitectura del modelo neuronal p_θ : U-NET



Arquitectura del modelo neuronal p_θ : U-NET



Una imagen del mismo tamaño que x_t , que se interpreta como la predicción del ruido gaussiano añadido a esa imagen.

Luego se usa para reconstruir x_{t-1} mediante la ecuación:

$$x_{t-1} \approx \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_\theta(x_t, t) \right)$$

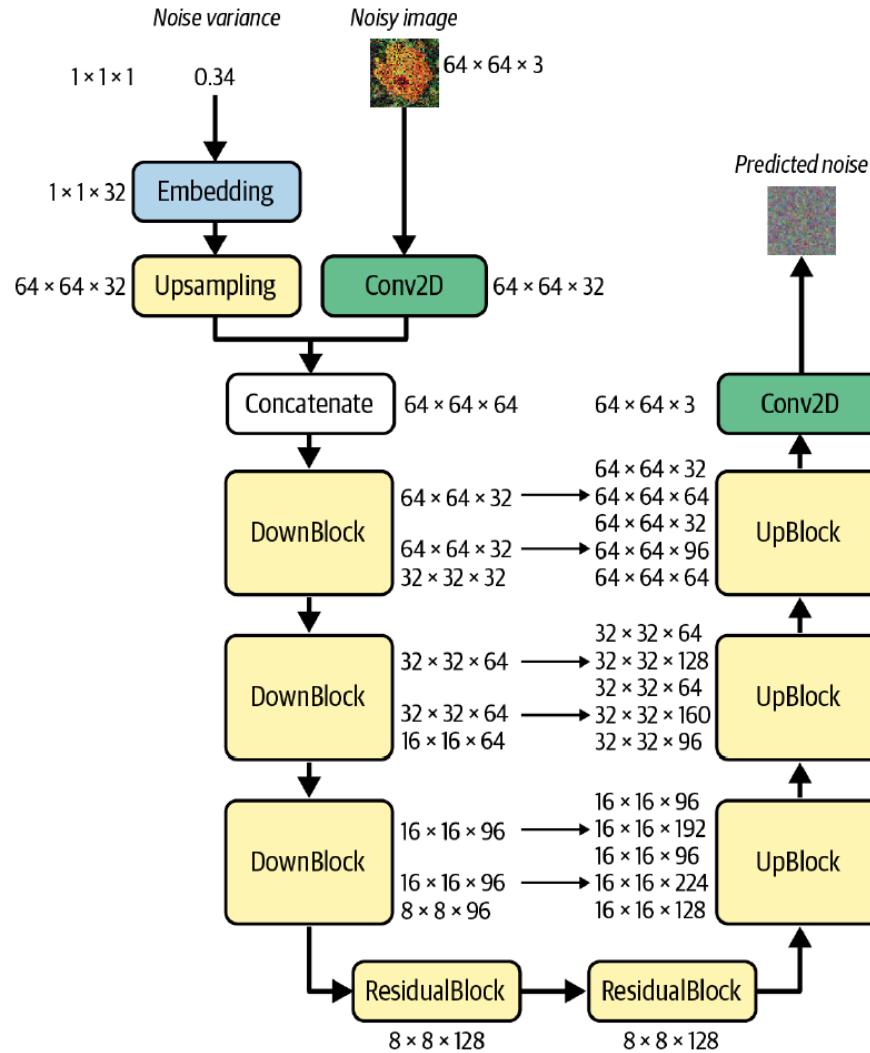
Factor de normalización para mantener la varianza.

Escala la cantidad de ruido predicho que se debe eliminar.

Arquitectura del modelo neuronal p_θ : U-NET

La U-Net es el núcleo arquitectónico del modelo de difusión:

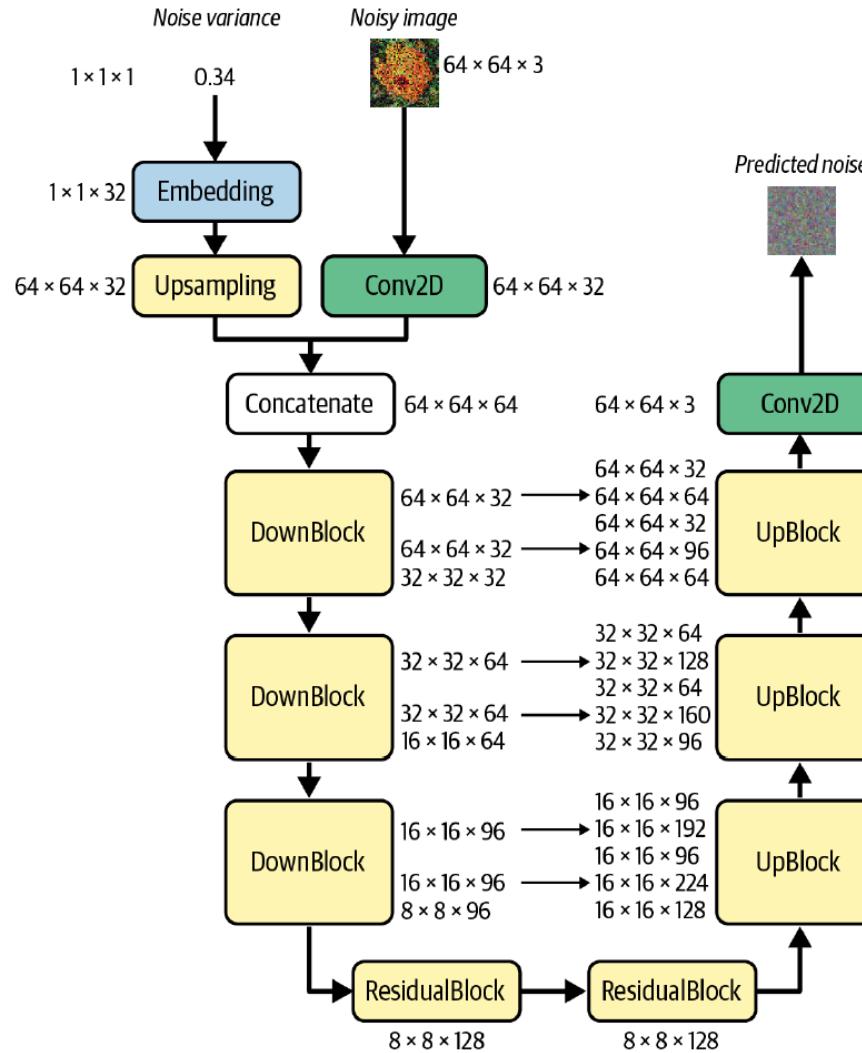
- Aprende a separa la estructura coherente (contenido de la imagen) del componente aleatorio (ruido).



Arquitectura del modelo neuronal p_θ : U-NET

Elección de arquitectura: U-Net

- El paper de DDPM (Ho et al., 2020) utiliza una arquitectura tipo **U-Net**, originalmente diseñada para segmentación de imágenes biomédicas.
- Su estructura simétrica la hace ideal cuando **la entrada y la salida tienen el mismo tamaño** (como en el caso de predecir ruido).



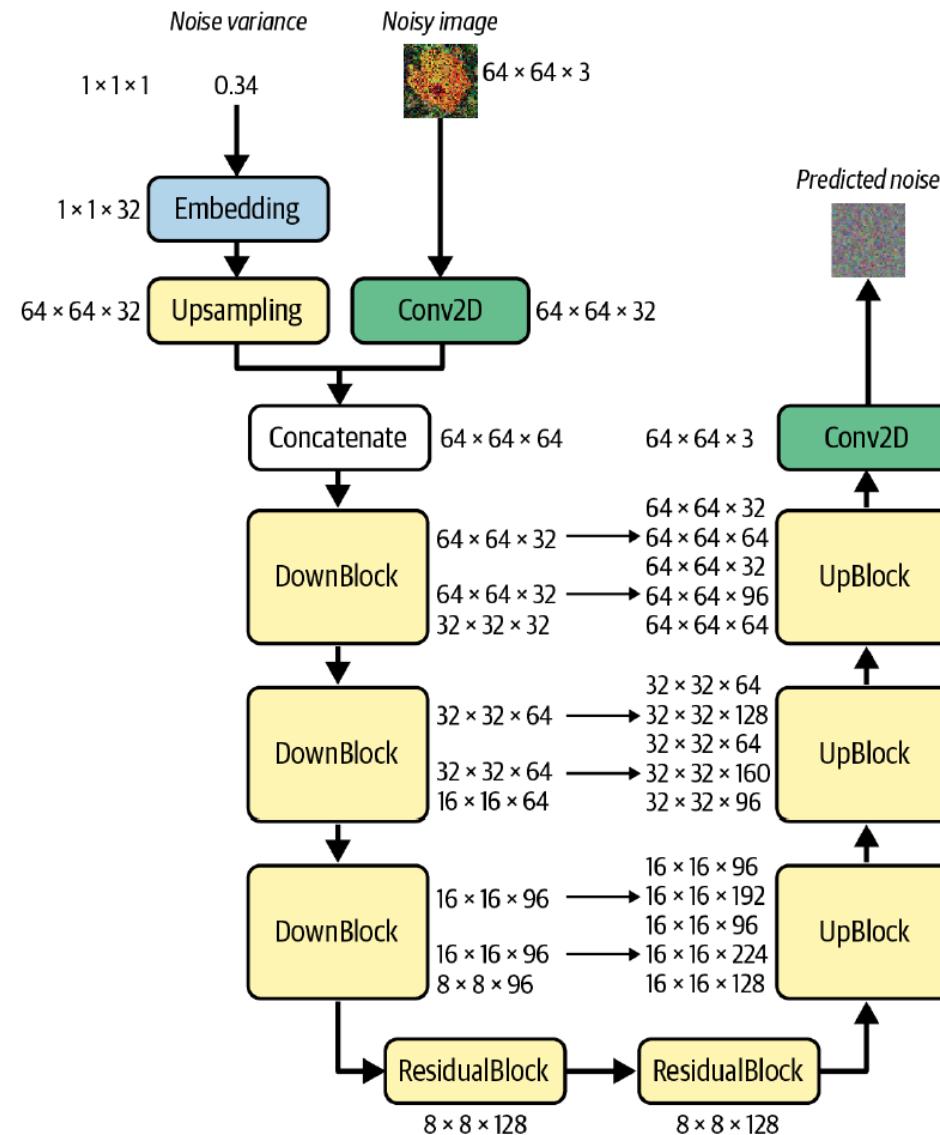
Arquitectura del modelo neuronal p_θ : U-NET

- Dos mitades principales:

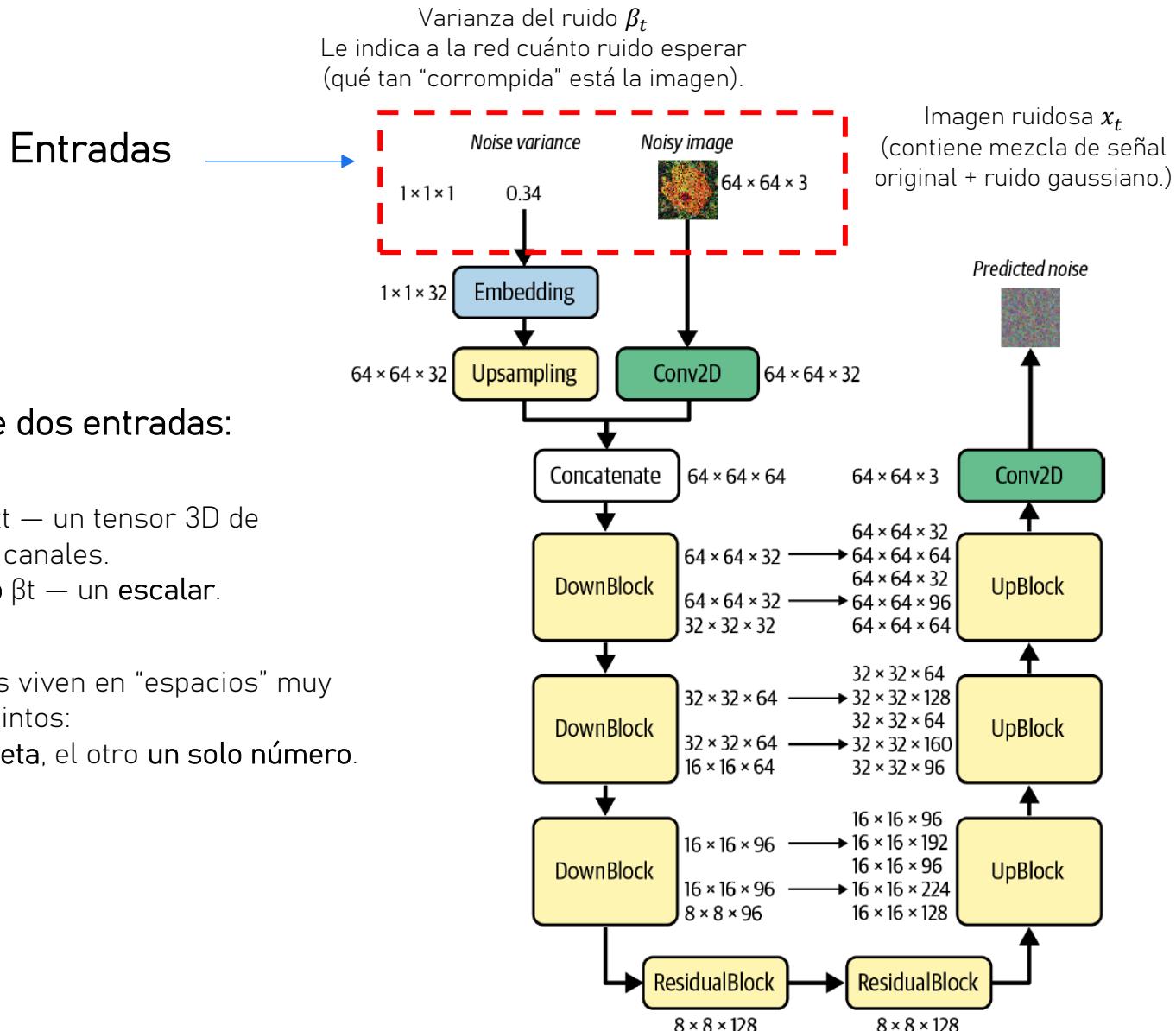
- **Downsampling path (codificador):**
Reduce la resolución espacial (altura y ancho),
pero **aumenta el número de canales** → obtiene representaciones más abstractas.
- **Upsampling path (decodificador):**
Recupera la resolución original,
mientras **reduce el número de canales** → reconstruye detalles espaciales.

- **Skip connections:**

- Conectan capas del *encoder* y *decoder* con igual resolución.
- Permiten que la información fluya directamente de las primeras capas a las últimas.
- Evitan pérdida de detalles finos (bordes, texturas).
- Esto diferencia a la U-Net de un VAE, que es **totalmente secuencial**.



Arquitectura del modelo neuronal p_θ : U-NET



Necesitamos una forma de combinar ambos en una representación rica y espacialmente significativa para la red.

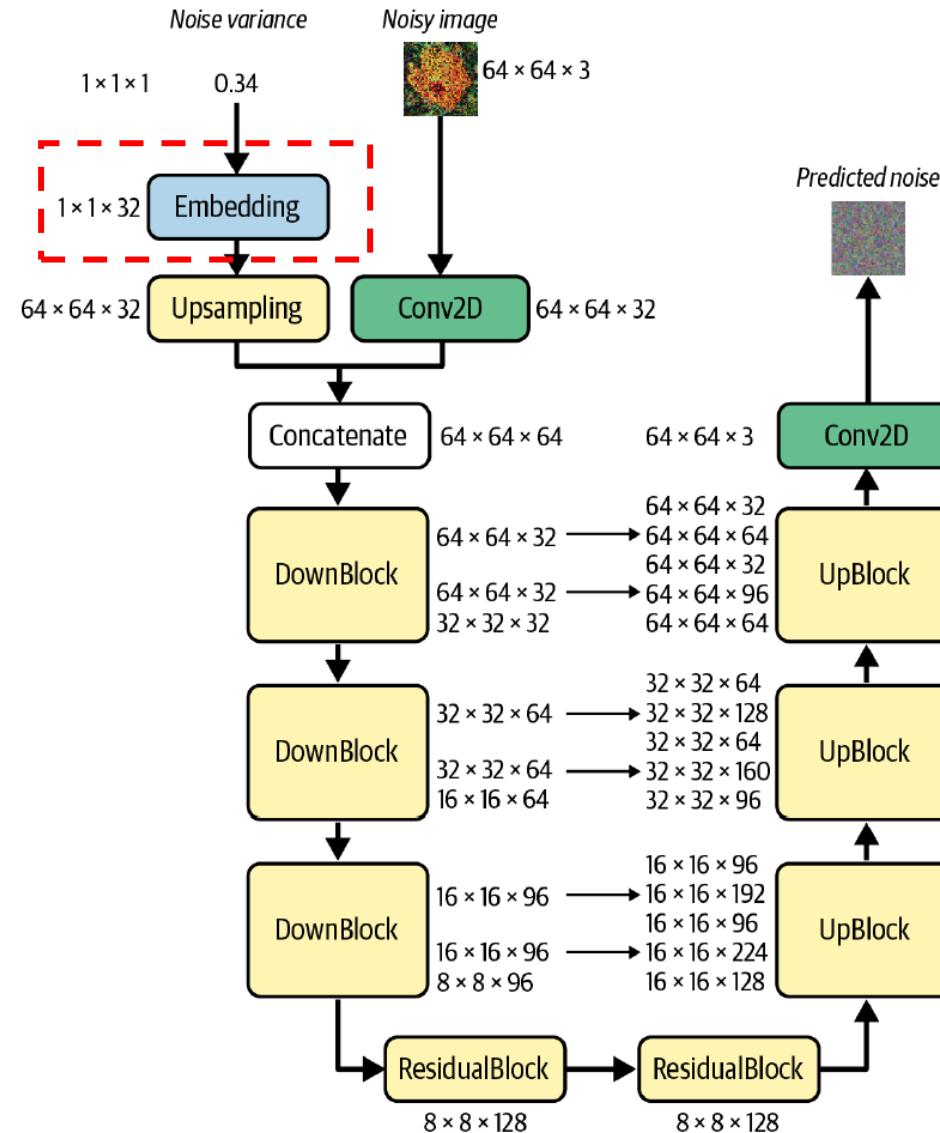
- Porque las **redes convolucionales** (CNNs) —como la U-Net— no están **diseñadas para procesar escalares globales** como parámetros de control
- Una CNN: opera **localmente** (sobre píxeles y vecindades espaciales).

Arquitectura del modelo neuronal p_θ : U-NET

Sinusoidal Embedding

Traduce ese número en una representación rica y espacialmente significativa para la red.

- Codifica la varianza del ruido β_t en un vector continuo de alta dimensión ($1 \times 1 \times 32$).
- Permite que la U-Net “sepa” cuánta corrupción tiene la imagen que está procesando.



Arquitectura del modelo neuronal p_θ : U-NET

¿Cómo se construye el embedding?

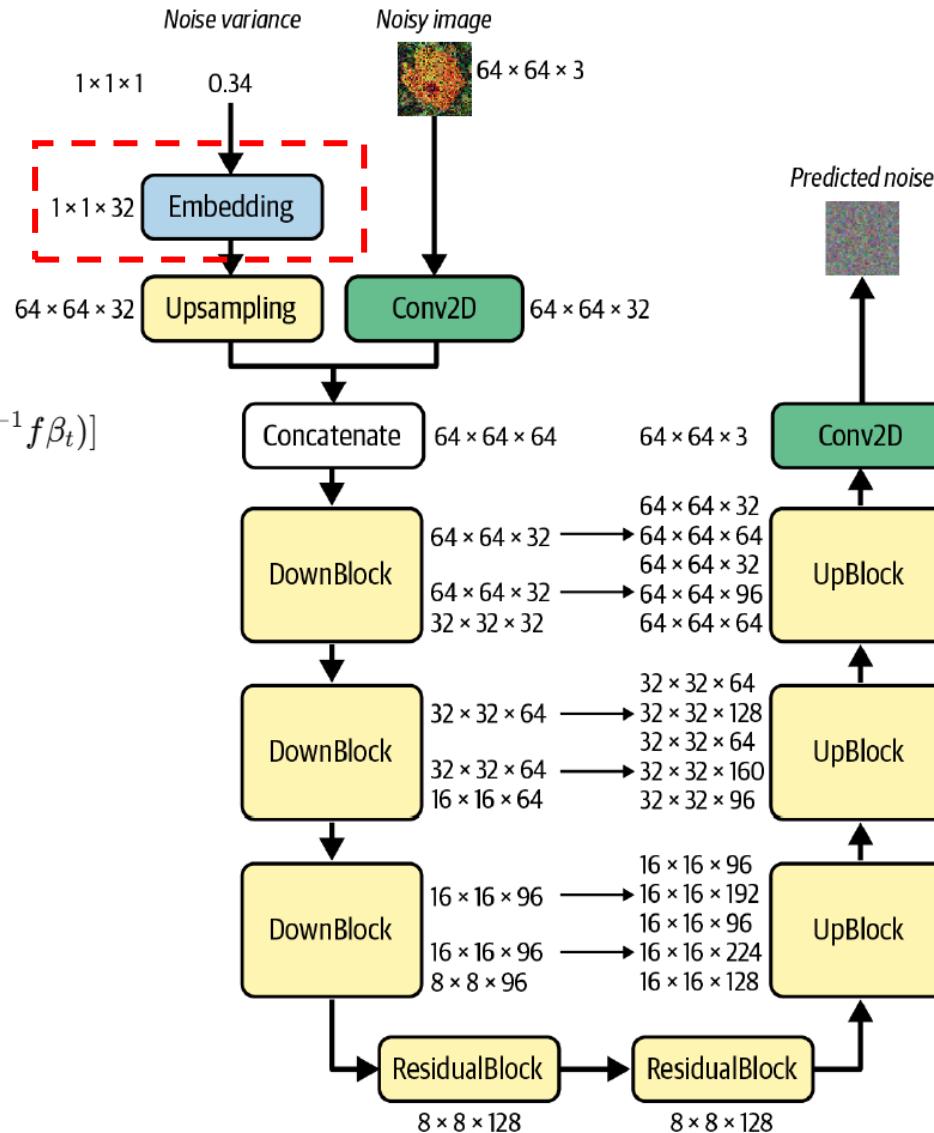
- Se calculan múltiples funciones seno y coseno a diferentes frecuencias sobre la varianza del ruido:

$$\gamma(\beta_t) = [\sin(2\pi e^0 f \beta_t), \dots, \sin(2\pi e^{L-1} f \beta_t), \cos(2\pi e^0 f \beta_t), \dots, \cos(2\pi e^{L-1} f \beta_t)]$$

L : número de frecuencias (por ejemplo, 16).

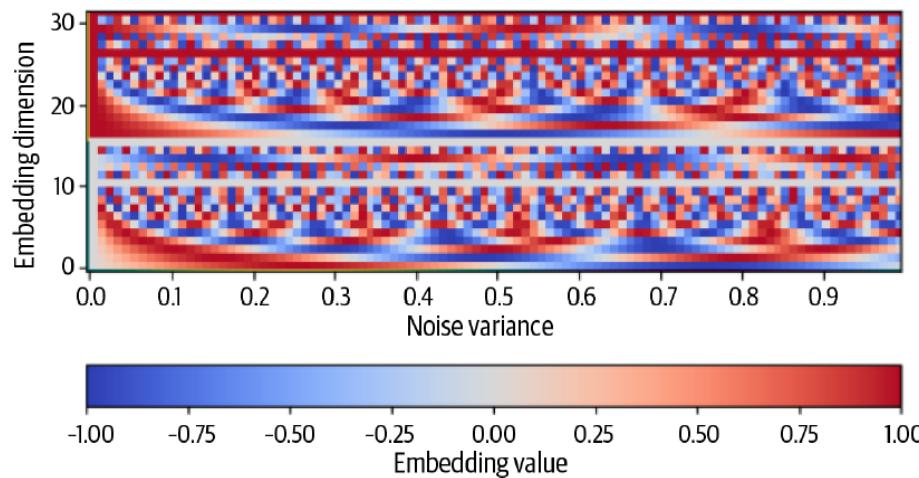
f : factor de escala que define la amplitud y la frecuencia de las oscilaciones.

Resultado: un vector de longitud $2L$ (por ejemplo, 32 dimensiones) que describe el nivel de ruido con alta fidelidad.



Arquitectura del modelo neuronal p_θ : U-NET

Intuición visual



Eje horizontal (x):

Representa la **varianza del ruido** —de 0 (sin ruido) a 1 (ruido máximo). A medida que avanza hacia la derecha, la imagen sería más ruidosa.

Eje vertical (y):

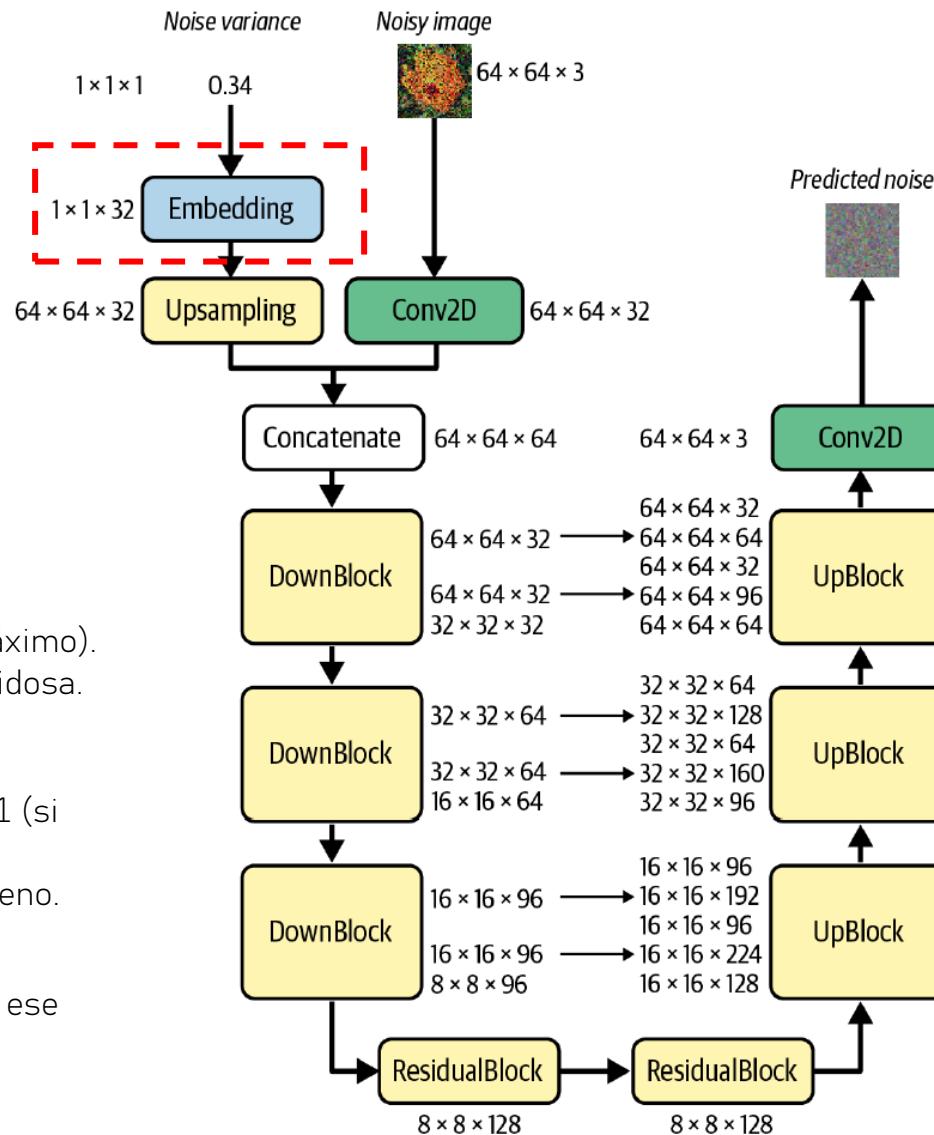
Representa las **dimensiones del embedding sinusoidal**, de 0 a 31 (si $L = 16 \rightarrow 32$ componentes).

Cada fila corresponde a **una frecuencia diferente** del seno o coseno.

Color:

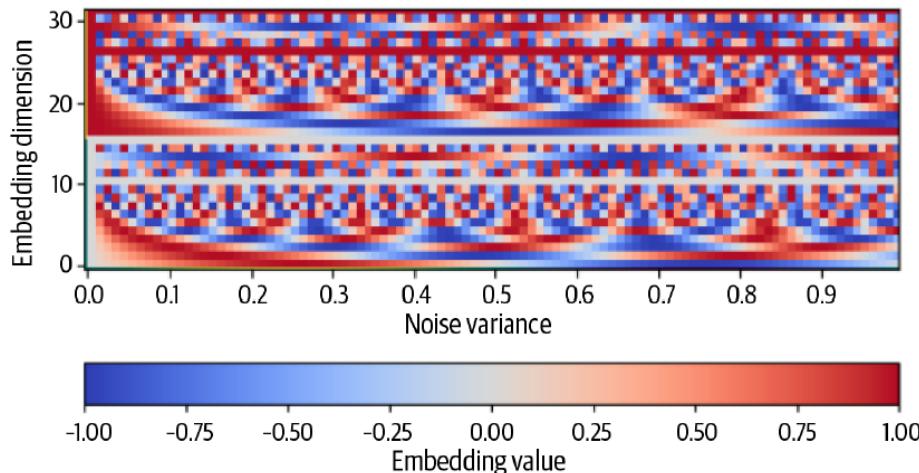
Muestra el **valor numérico** del embedding para esa frecuencia y ese nivel de ruido:

- Azul = valores negativos (-1)
- Blanco = cercanos a 0
- Rojo = valores positivos (+1)



Arquitectura del modelo neuronal p_θ : U-NET

Intuición visual

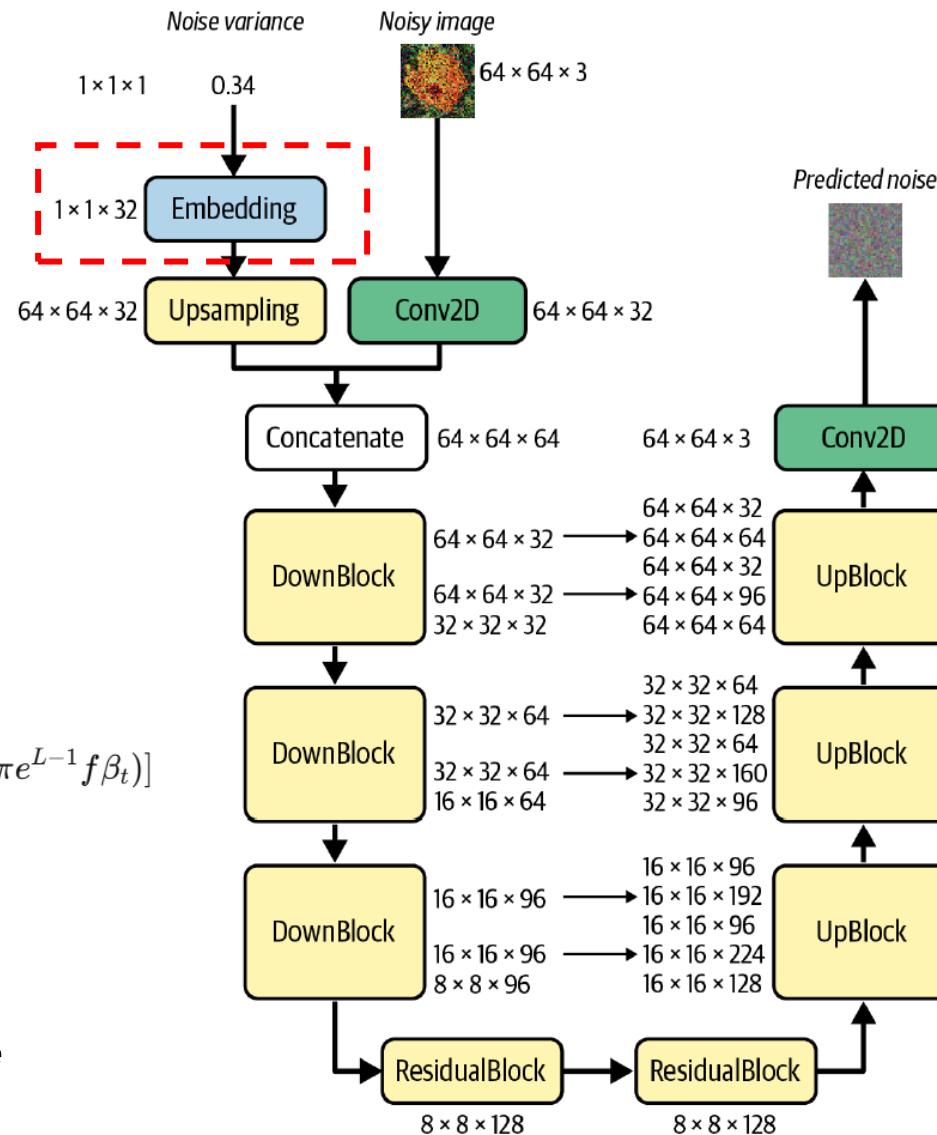


- Cada fila del mapa es una **onda sinusoidal** evaluada en función de la varianza del ruido.

$$\gamma(\beta_t) = [\sin(2\pi e^0 f \beta_t), \dots, \sin(2\pi e^{L-1} f \beta_t), \cos(2\pi e^0 f \beta_t), \dots, \cos(2\pi e^{L-1} f \beta_t)]$$

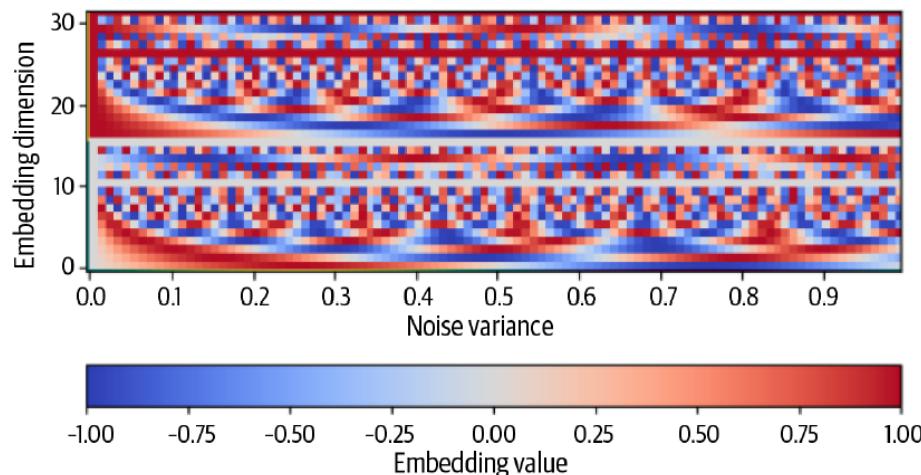
- Las filas inferiores oscilan **lentamente** → frecuencias bajas.
- Las filas superiores oscilan **rápidamente** → frecuencias altas.

Esto significa que el embedding contiene **múltiples escalas de variación** respecto al nivel de ruido.



Arquitectura del modelo neuronal p_θ : U-NET

Intuición visual

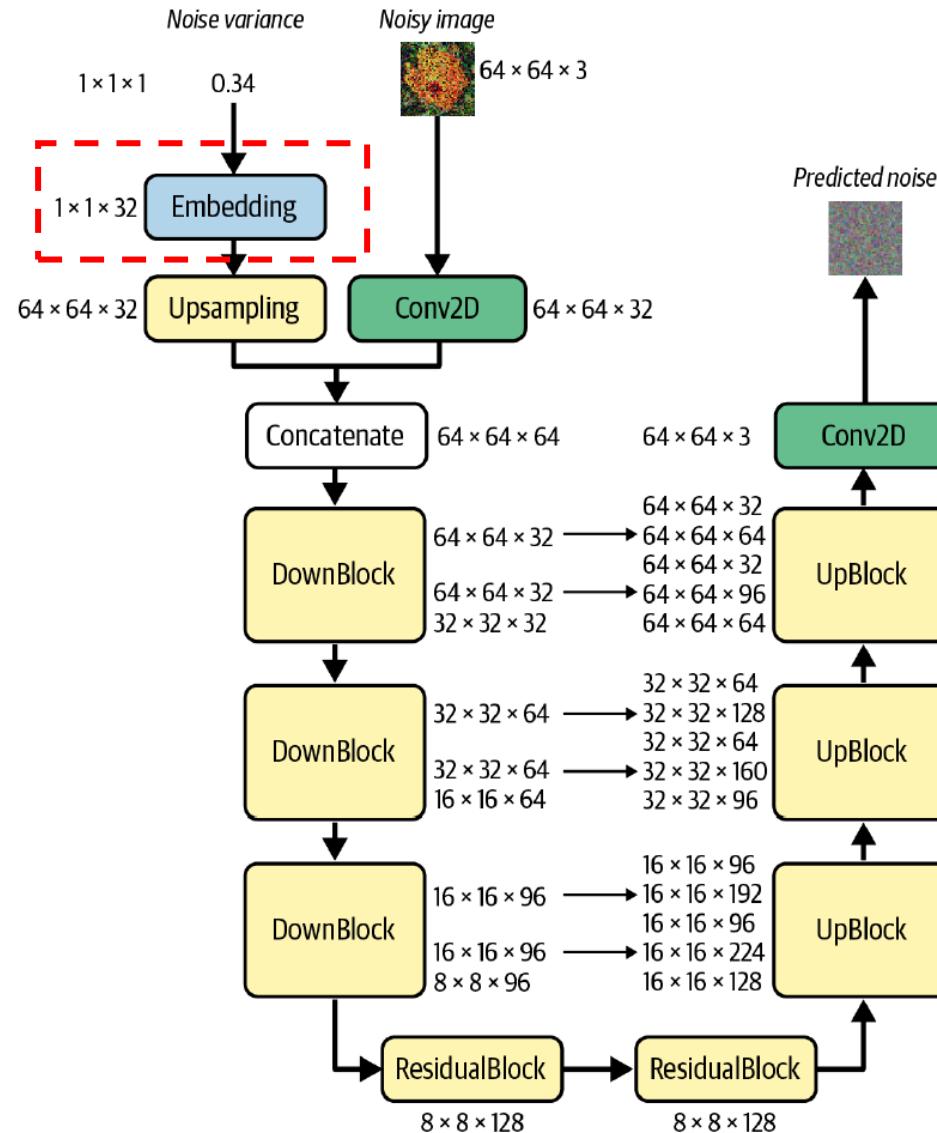


Cuando la red recibe la varianza del ruido, por ejemplo

$$\beta_t = 0.34$$

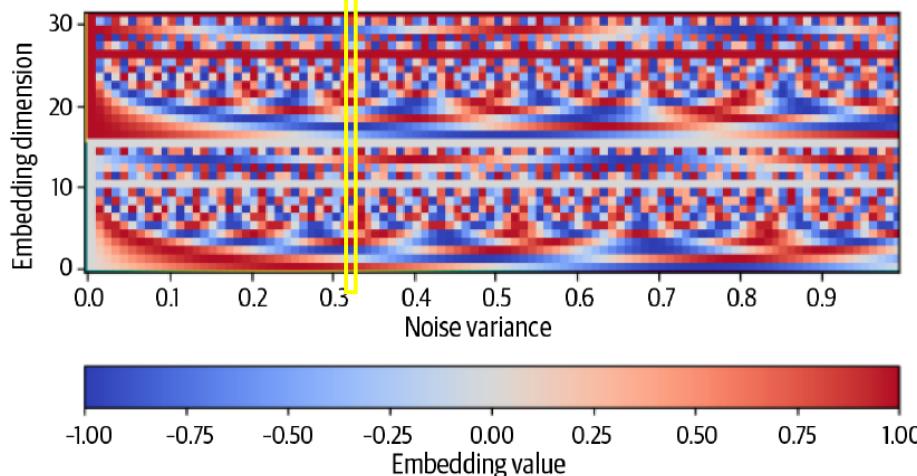


¿Qué hace?



Arquitectura del modelo neuronal p_θ : U-NET

Intuición visual



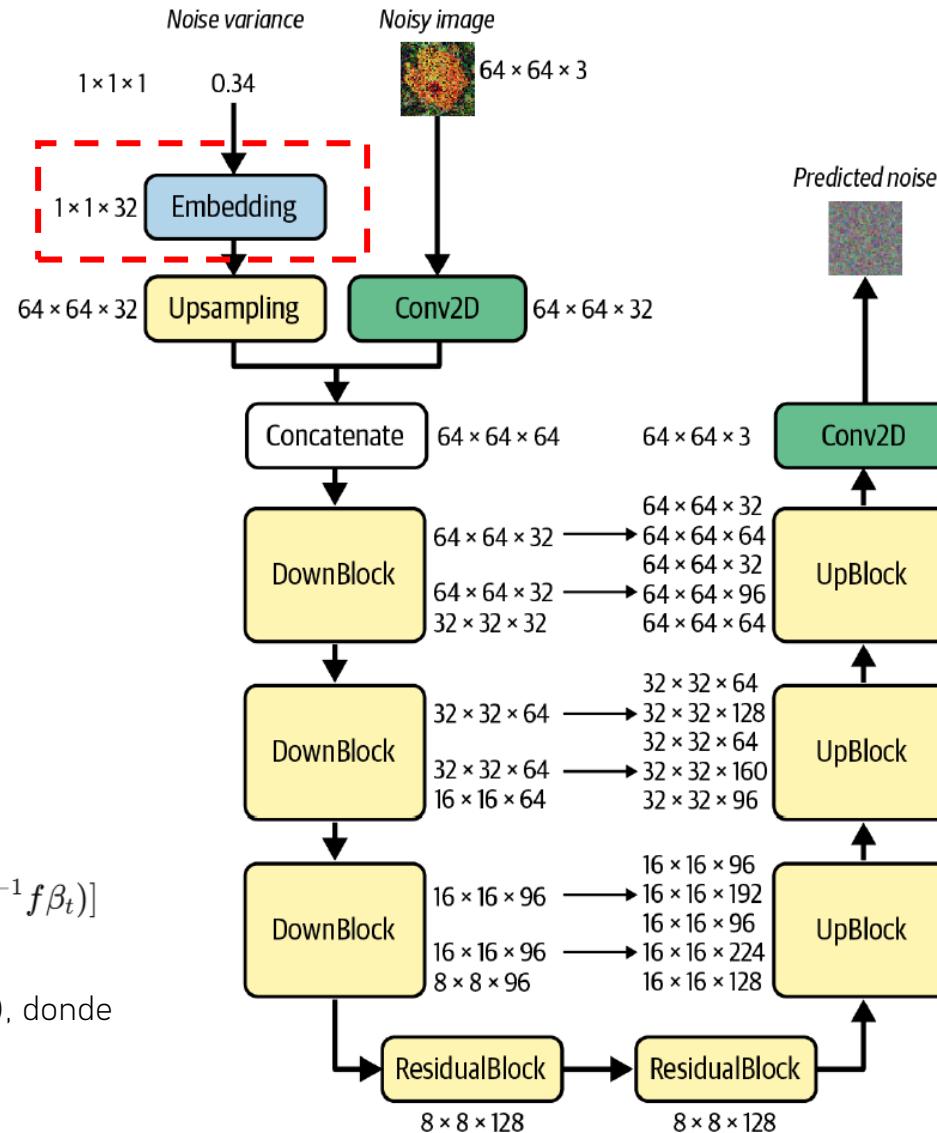
Cuando la red recibe la varianza del ruido, por ejemplo

$$\beta_t = 0.34$$

no trabaja con ese número directamente, sino que lo pasa por la función de codificación sinusoidal:

$$\gamma(\beta_t) = [\sin(2\pi e^0 f \beta_t), \dots, \sin(2\pi e^{L-1} f \beta_t), \cos(2\pi e^0 f \beta_t), \dots, \cos(2\pi e^{L-1} f \beta_t)]$$

El resultado es un vector de longitud $2L$ (por ejemplo, 32 valores), donde cada componente representa una frecuencia distinta.

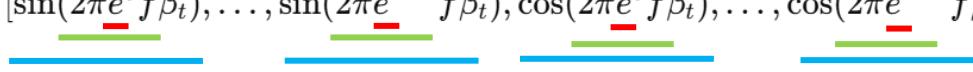


Arquitectura del modelo neuronal p_θ : U-NET

ddm.ipynb


Ejecutar celdas:

2.1 Codificación sinusoidal del paso de ruido

$$\gamma(\beta_t) = [\sin(2\pi e^0 f \beta_t), \dots, \sin(2\pi e^{L-1} f \beta_t), \cos(2\pi e^0 f \beta_t), \dots, \cos(2\pi e^{L-1} f \beta_t)]$$


```
def sinusoidal_embedding(x):
    frequencies = tf.exp(
        tf.linspace(
            tf.math.log(1.0),
            tf.math.log(1000.0),
            16,
        )
    )
    angular_speeds = 2.0 * math.pi * frequencies → Calcula las velocidades angulares  $2\pi e^i f$ 
    embeddings = tf.concat( → Concatenación de senos y cosenos
        [tf.sin(angular_speeds * x), tf.cos(angular_speeds * x)], axis=3 → Cálculo de componentes
    )
    return embeddings
```

Genera frecuencias crecientes exponencialmente

$$e^i$$

→ Concatenación de senos y cosenos

→ Cálculo de componentes sinusoidales.

$\sin(2\pi e^i f \beta_t), \cos(\dots)$

Arquitectura del modelo neuronal p_θ : U-NET

Ahora tenemos algo más rico, pero todavía es un **vector 1D**.

$$\gamma(\beta_t) \in \mathbb{R}^{32}$$

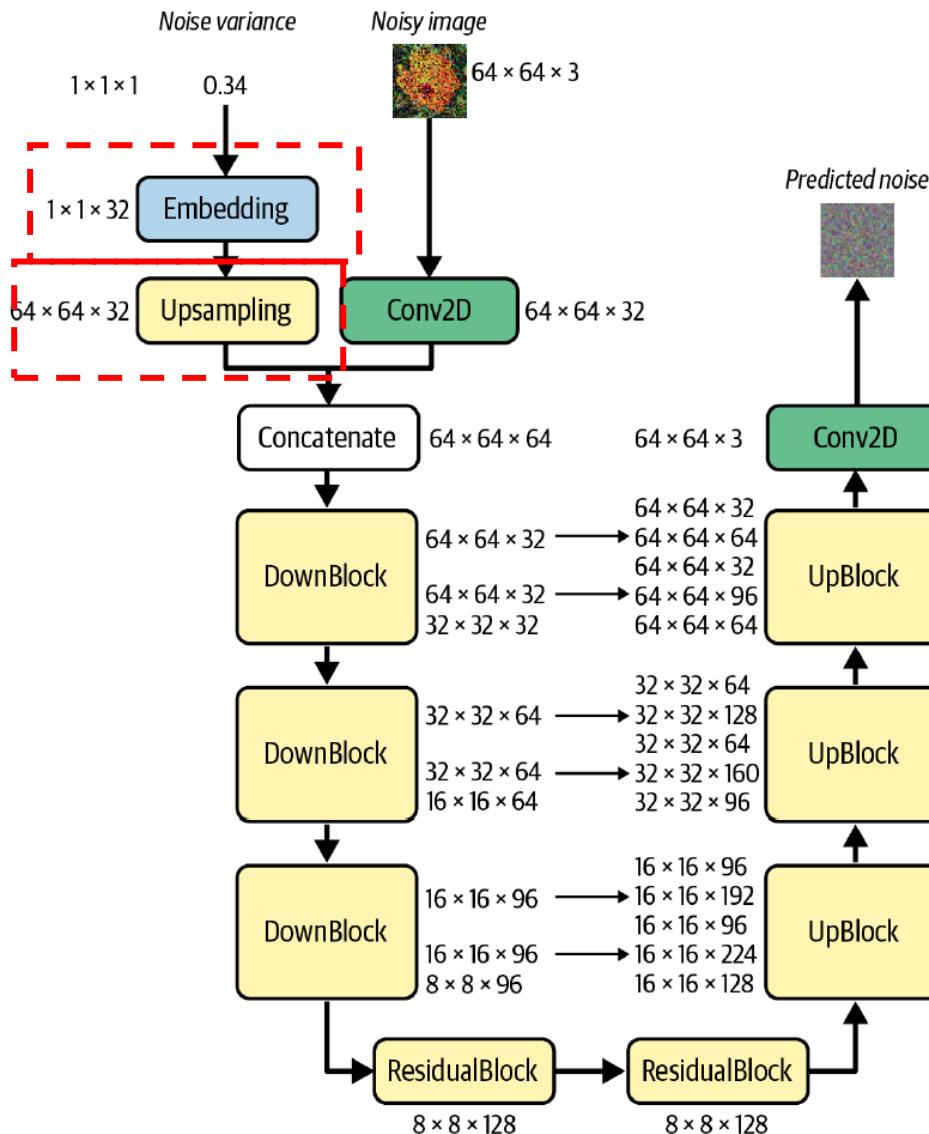
Las convoluciones operan sobre **mapas bidimensionales** (alto × ancho).

Así que, para que la información de la varianza esté disponible en **cada píxel de la imagen**, replicamos este vector en todas las posiciones espaciales:

En otras palabras:
cada píxel "recibe" el mismo vector de 32 valores,

$$(1, 1, 32) \Rightarrow (64, 64, 32)$$

Esto le permite a la red tratar la varianza como **otro conjunto de canales**,
como si fuera una capa extra con información global.



Arquitectura del modelo neuronal p_θ : U-NET

Ahora tenemos dos tensores con la misma forma espacial, pero diferente número de canales :

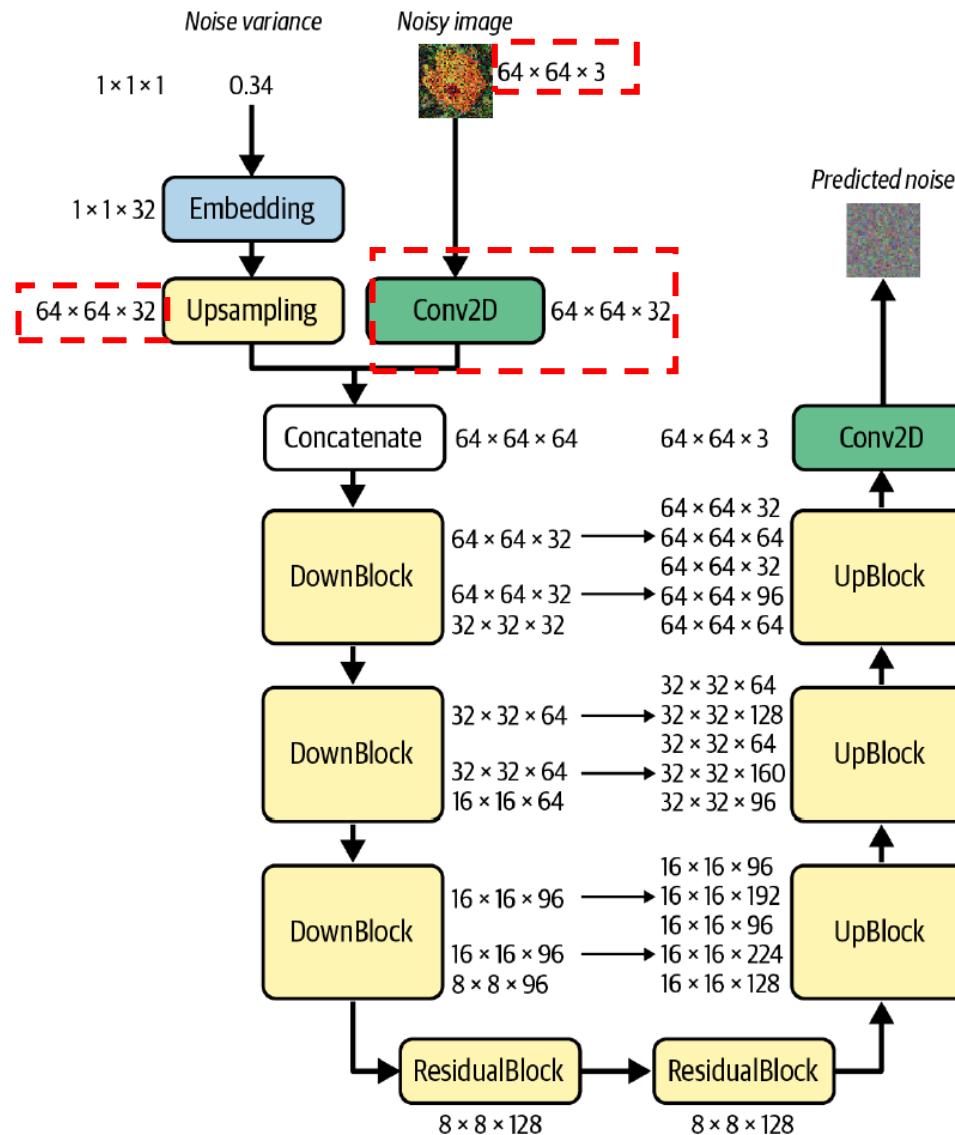
- La imagen ruidosa x_t ($64 \times 64 \times 3$)
- El embedding replicado: ($64 \times 64 \times 32$)

Para que ambos tensores también tengan el mismo número de canales y fusionarlos en una misma estructura:

- Se aplica una convolución con 32 filtros.
 $(64, 64, 3) \Rightarrow (64, 64, 32)$

Cada nuevo canal no representa un color, sino un **mapa de características** aprendido por la red.

- La red está "traduciendo" la imagen ruidosa a un **espacio de características** donde es más fácil aprender a separar el ruido del contenido visual.



Arquitectura del modelo neuronal p_θ : U-NET

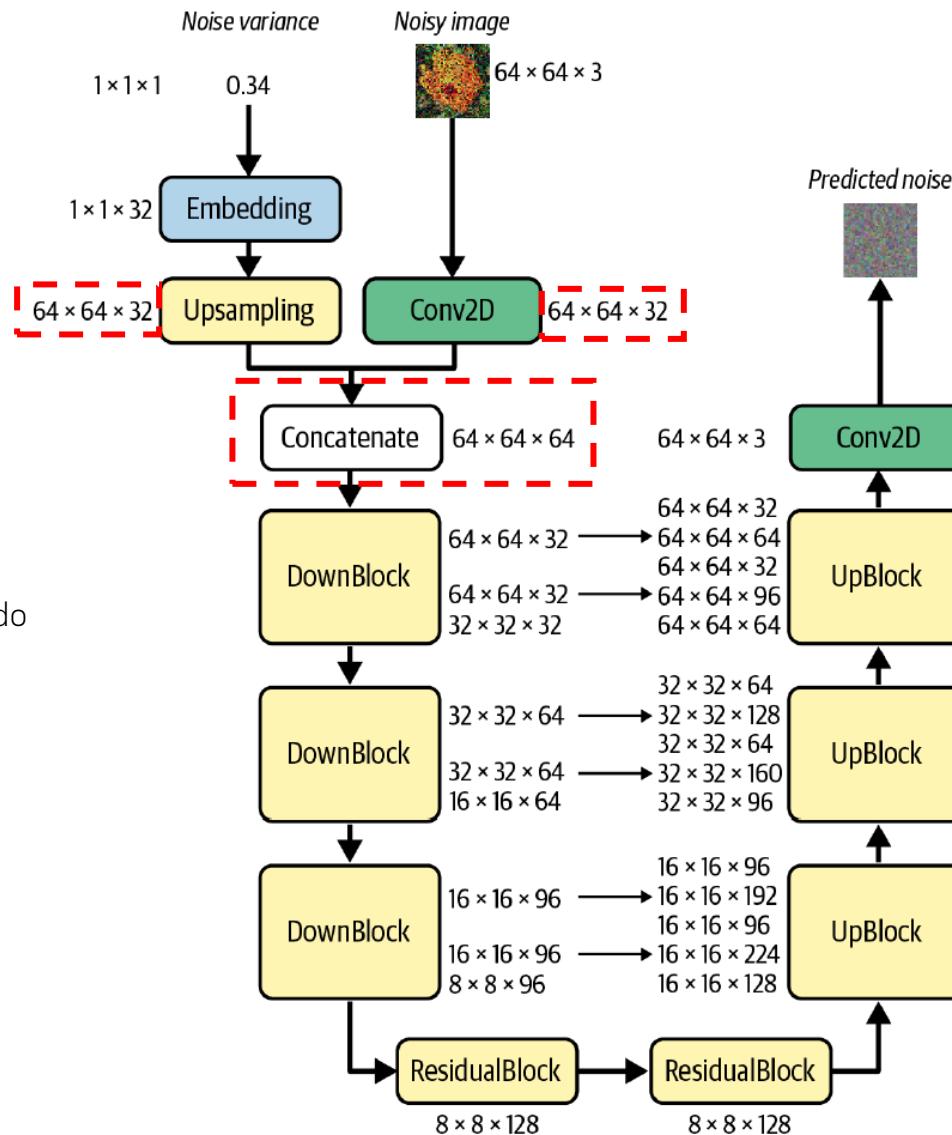
Finalmente, se concatenan ambos tensores:

$$(64, 64, 32)_{\text{imagen}} + (64, 64, 32)_{\text{embedding}} \Rightarrow (64, 64, 64)$$

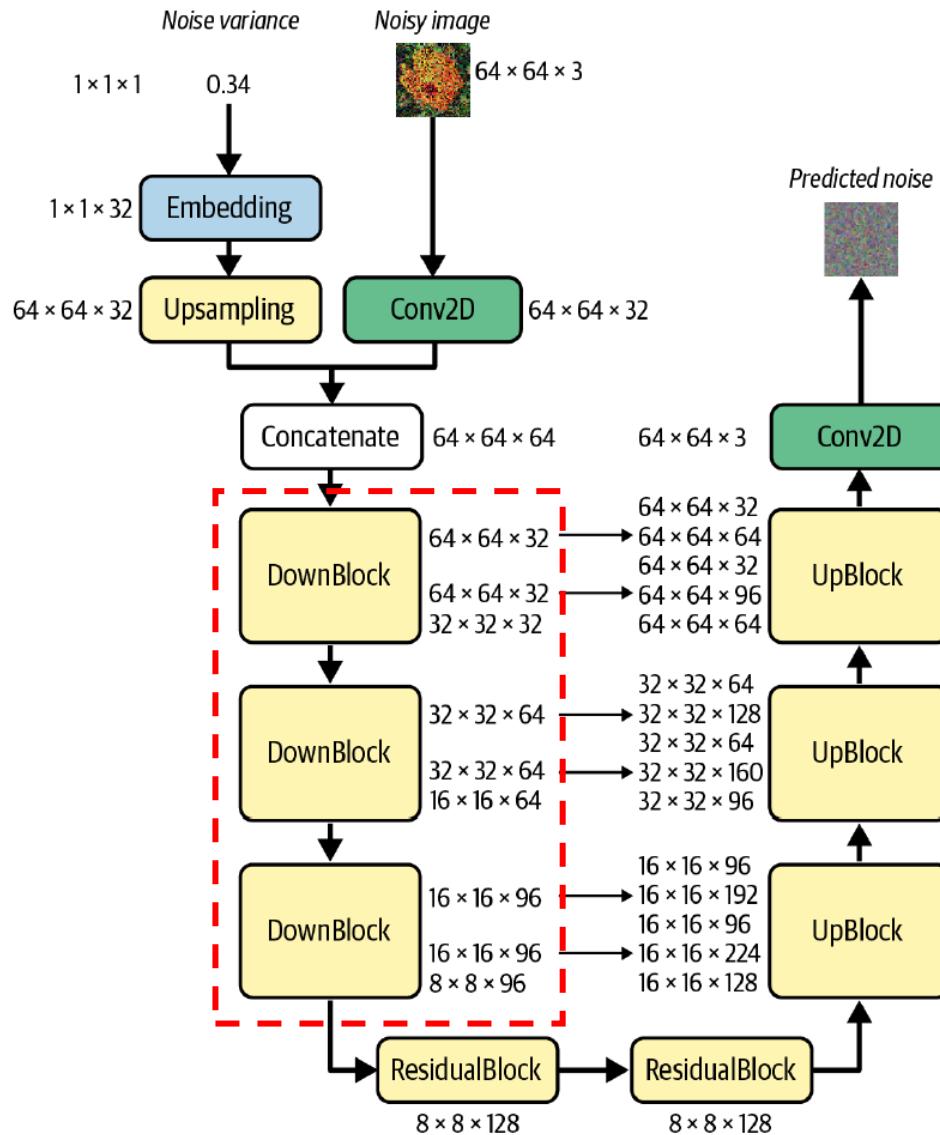
En este momento, la red ha unido **información local y global**:

- **Del lado de la imagen:**
se obtienen *características locales* —cómo es el ruido en cada zona, bordes, colores, texturas.
- **Del lado de la varianza del ruido:**
se inyecta *información global* —cuánto ruido hay en promedio, en qué etapa del proceso estamos.

Esta es toda la información que la U-Net necesita para empezar el denoising.



Arquitectura del modelo neuronal p_θ : U-NET



DownBlock: Codificación progresiva

Objetivo

- Reducir la resolución espacial de la imagen (de 64x64 → 32x32 → 16x16 → ...).
- Aumentar el número de canales para capturar representaciones más abstractas.
- Extraer progresivamente las características esenciales del ruido y del contenido visual.

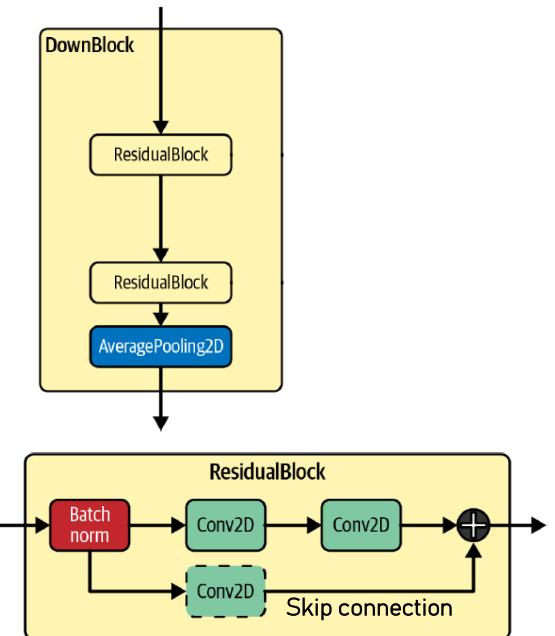
Estructura interna

Dos ResidualBlocks consecutivos

- Cada uno combina convoluciones y conexiones residuales (*skip connections*), lo que facilita que la red aprenda transformaciones más estables.

AveragePooling2D

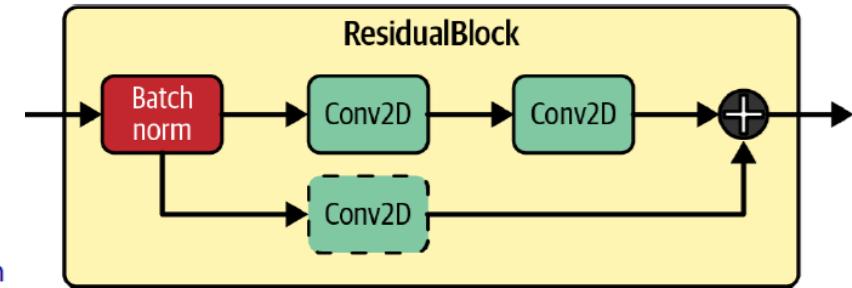
- Reduce a la mitad el tamaño espacial de la imagen.
- Permite que el siguiente nivel vea una versión "resumida" de la información.



Arquitectura del modelo neuronal p_θ : U-NET

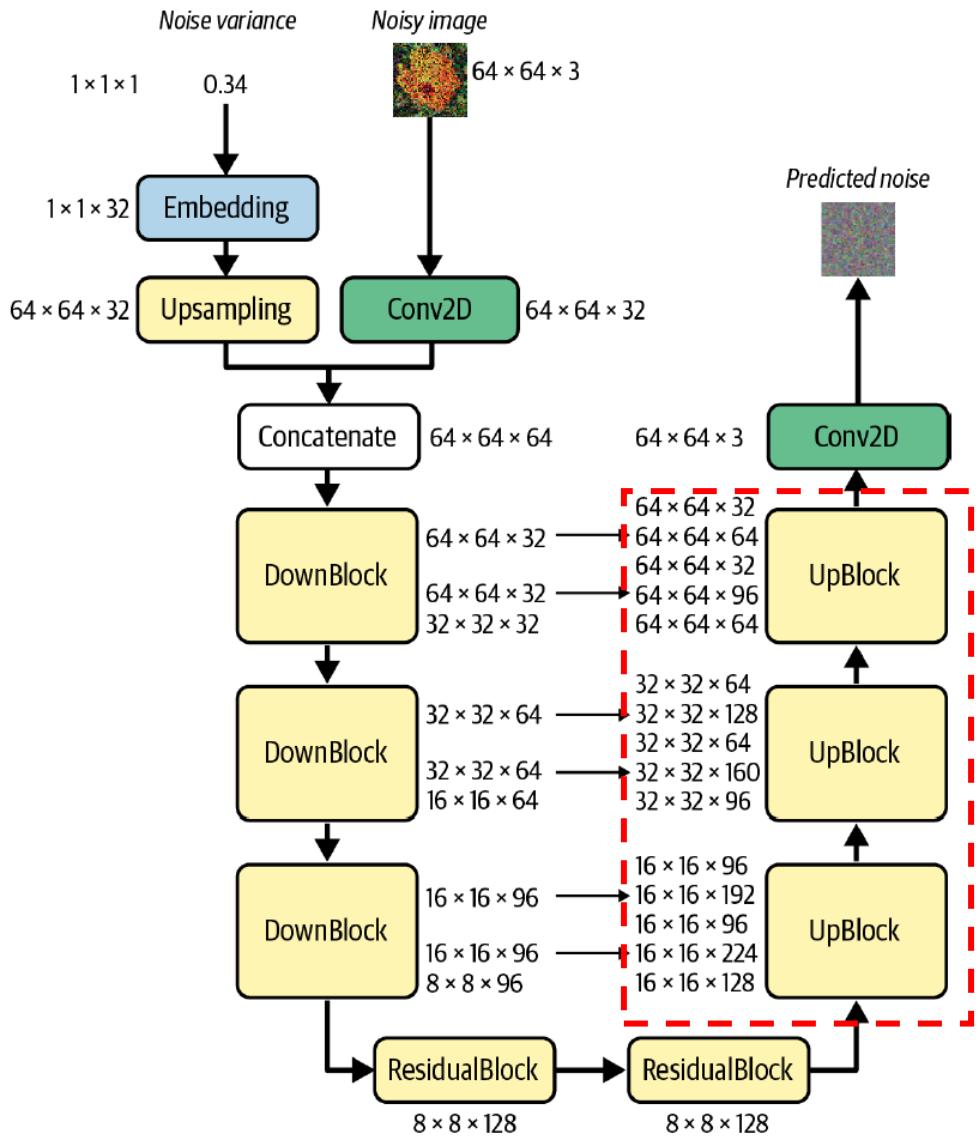
```
def ResidualBlock(width):
    def apply(x):
        input_width = x.shape[3]
        if input_width == width: ❶
            residual = x
        else:
            residual = layers.Conv2D(width, kernel_size=1)(x)
        x = layers.BatchNormalization(center=False, scale=False)(x) ❷
        x = layers.Conv2D(
            width, kernel_size=3, padding="same", activation=activations.swish
        )(x) ❸
        x = layers.Conv2D(width, kernel_size=3, padding="same")(x)
        x = layers.Add()([x, residual]) ❹
    return x

return apply
```



- ❶ Check if the number of channels in the input matches the number of channels that we would like the block to output. If not, include an extra Conv2D layer on the skip connection to bring the number of channels in line with the rest of the block.
- ❷ Apply a BatchNormalization layer.
- ❸ Apply two Conv2D layers.
- ❹ Add the original block input to the output to provide the final output from the block.

Arquitectura del modelo neuronal p_θ : U-NET

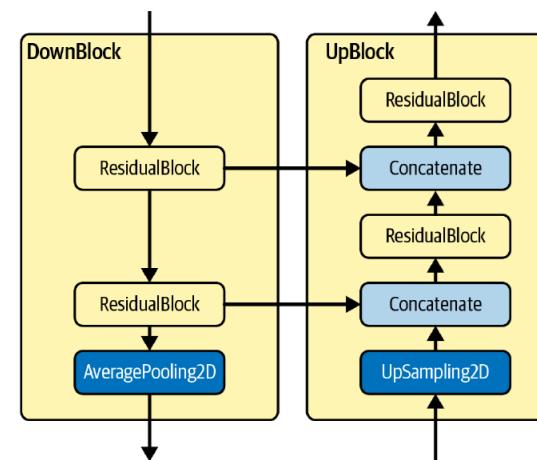


UpBlock: Decodificación progresiva

Realiza el proceso inverso:

- **Upsampling2D** duplica la resolución espacial (por ejemplo, $8 \times 8 \rightarrow 16 \times 16$).
- **Concatenate** une la salida del UpBlock con el *skip connection* del DownBlock correspondiente.
- **ResidualBlocks** afinan los detalles y reconstruyen las características locales.

Así, mientras el DownBlock “comprime” la información para entender el ruido global,
el UpBlock la “expande” para reconstruir una versión limpia de la imagen,
usando los detalles conservados por las conexiones residuales.



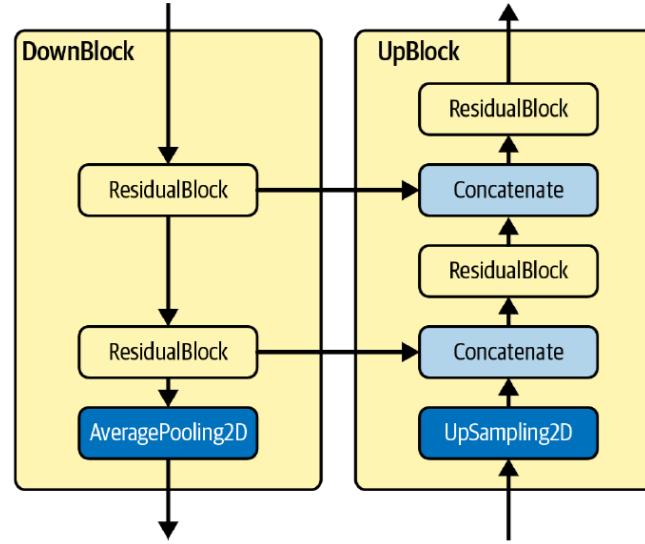
Arquitectura del modelo neuronal p_θ : U-NET

```
def DownBlock(width, block_depth):
    def apply(x):
        x, skips = x
        for _ in range(block_depth):
            x = ResidualBlock(width)(x) ①
            skips.append(x) ②
        x = layers.AveragePooling2D(pool_size=2)(x) ③
    return x

return apply

def UpBlock(width, block_depth):
    def apply(x):
        x, skips = x
        x = layers.UpSampling2D(size=2, interpolation="bilinear")(x) ④
        for _ in range(block_depth):
            x = layers.Concatenate()([x, skips.pop()]) ⑤
            x = ResidualBlock(width)(x) ⑥
    return x

return apply
```



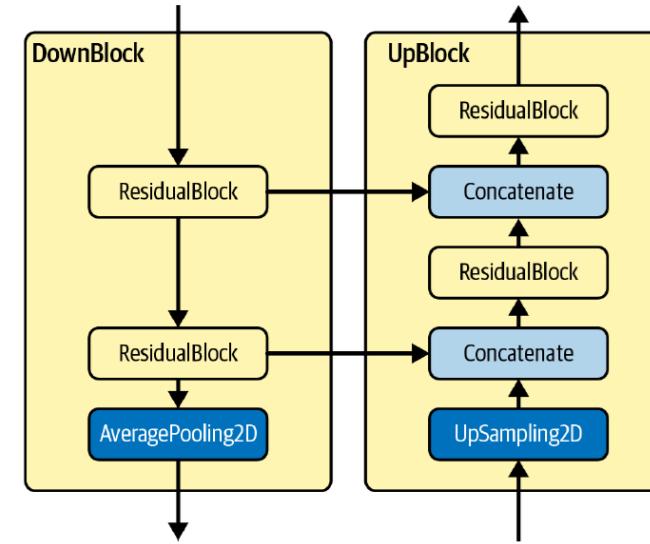
- ① The DownBlock increases the number of channels in the image using a Residual Block of a given width...
- ② ...each of which are saved to a list (skips) for use later by the UpBlocks.
- ③ A final AveragePooling2D layer reduces the dimensionality of the image by half.
- ④ The UpBlock begins with an UpSampling2D layer that doubles the size of the image.
- ⑤ The output from a DownBlock layer is glued to the current output using a Concatenate layer.
- ⑥ A ResidualBlock is used to reduce the number of channels in the image as it passes through the UpBlock.

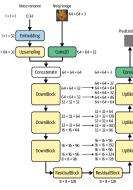
Arquitectura del modelo neuronal p_θ : U-NET

ddm.ipynb


Ejecutar celdas:

2.2 Definición de Bloque residual, DownBlock y UpBlock





Arquitectura del modelo neuronal p_θ : U-NET

```

noisy_images = layers.Input(shape=(64, 64, 3)) ❶ → La primera entrada es la imagen a la que le queremos eliminar el ruido.
x = layers.Conv2D(32, kernel_size=1)(noisy_images) ❷ → La imagen se pasa por una capa convolucional para incrementar el número de canales.

noise_variances = layers.Input(shape=(1, 1, 1)) ❸ → La segunda entrada de la U-Net es la varianza del ruido (un escalar).
noise_embedding = layers.Lambda(sinusoidal_embedding)(noise_variances) ❹ → Esto se codifica utilizando una incrustación sinusoidal
noise_embedding = layers.UpSampling2D(size=64, interpolation="nearest")(
    noise_embedding
) ❺ → Esta incrustación se replica a lo largo de las dimensiones espaciales para que coincida con el tamaño de la imagen de entrada.

x = layers.concatenate([x, noise_embedding]) ❻ → Los dos flujos de entrada se concatenan a lo largo de los canales.

skips = [] ❼ → La lista skips almacenará las salidas de las capas DownBlock que deseamos conectar con las capas UpBlock en las etapas posteriores.

x = DownBlock(32, block_depth = 2)([x, skips]) ❽ → El tensor se pasa por una serie de capas DownBlock que reducen el tamaño de la
x = DownBlock(64, block_depth = 2)([x, skips])   imagen, al mismo tiempo que aumentan el número de canales.
x = DownBlock(96, block_depth = 2)([x, skips])

x = ResidualBlock(128)(x) ❾ → El tensor se pasa luego por dos capas ResidualBlock, que mantienen constante el
x = ResidualBlock(128)(x)   tamaño de la imagen y el número de canales.

x = UpBlock(96, block_depth = 2)([x, skips]) ❿ → Luego, el tensor pasa por una serie de capas UpBlock que aumentan el tamaño de la imagen,
x = UpBlock(64, block_depth = 2)([x, skips])   mientras disminuyen el número de canales.
x = UpBlock(32, block_depth = 2)([x, skips])   Las conexiones de salto (skip connections) incorporan la salida de las capas DownBlock anteriores.

x = layers.Conv2D(3, kernel_size=1, kernel_initializer="zeros")(x) ❾ → La capa Conv2D final reduce el número de canales a tres (RGB).

unet = models.Model([noisy_images, noise_variances], x, name="unet") ❿ → La U-Net es un modelo de Keras que toma como entrada las imágenes
ruidosas y las varianzas del ruido, y produce como salida un mapa de
ruido predicho.

```

```

class DiffusionModel(models.Model):
    def __init__(self):
        super().__init__()
        self.normalizer = layers.Normalization()
        self.network = unet
        self.ema_network = models.clone_model(self.network)
        self.diffusion_schedule = cosine_diffusion_schedule
    ...
    def denoise(self, noisy_images, noise_rates, signal_rates, training):
        if training:
            network = self.network
        else:
            network = self.ema_network
        pred_noises = network(
            [noisy_images, noise_rates**2], training=training
        )
        pred_images = (noisy_images - noise_rates * pred_noises) / signal_rates
        return pred_noises, pred_images
    def train_step(self, images):
        images = self.normalizer(images, training=True) ①
        noises = tf.random.normal(shape=tf.shape(images)) ②
        batch_size = tf.shape(images)[0]
        diffusion_times = tf.random.uniform(
            shape=(batch_size, 1, 1, 1), minval=0.0, maxval=1.0
        ) ③
        noise_rates, signal_rates = self.cosine_diffusion_schedule(
            diffusion_times
        ) ④
        noisy_images = signal_rates * images + noise_rates * noises ⑤

```

```

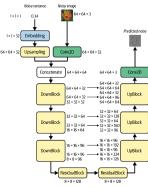
        with tf.GradientTape() as tape:
            pred_noises, pred_images = self.denoise(
                noisy_images, noise_rates, signal_rates, training=True
            ) ⑥
            noise_loss = self.loss(noises, pred_noises) ⑦
            gradients = tape.gradient(noise_loss, self.network.trainable_weights)
            self.optimizer.apply_gradients(
                zip(gradients, self.network.trainable_weights)
            ) ⑧
            self.noise_loss_tracker.update_state(noise_loss)

        for weight, ema_weight in zip(
            self.network.weights, self.ema_network.weights
        ):
            ema_weight.assign(0.999 * ema_weight + (1 - 0.999) * weight) ⑨

        return {m.name: m.result() for m in self.metrics}

```

- ① We first normalize the batch of images to have zero mean and unit variance.
- ② Next, we sample noise to match the shape of the input images.
- ③ We also sample random diffusion times...
- ④ ...and use these to generate the noise and signal rates according to the cosine diffusion schedule.
- ⑤ Then we apply the signal and noise weightings to the input images to generate the noisy images.
- ⑥ Next, we denoise the noisy images by asking the network to predict the noise and then undoing the noising operation, using the provided `noise_rates` and `signal_rates`.
- ⑦ We can then calculate the loss (mean absolute error) between the predicted noise and the true noise...
- ⑧ ...and take a gradient step against this loss function.
- ⑨ The EMA network weights are updated to a weighted average of the existing EMA weights and the trained network weights after the gradient step.



Arquitectura del modelo neuronal p_θ : U-NET

ddm.ipynb



Ejecutar celdas:

2.4 Clase DiffusionModel

Entrenamiento del modelo de difusión

```
model = DiffusionModel() ❶
model.compile(
    optimizer=optimizers.experimental.AdamW(learning_rate=1e-3, weight_decay=1e-4),
    loss=losses.mean_absolute_error,
) ❷

model.normalizer.adapt(train) ❸

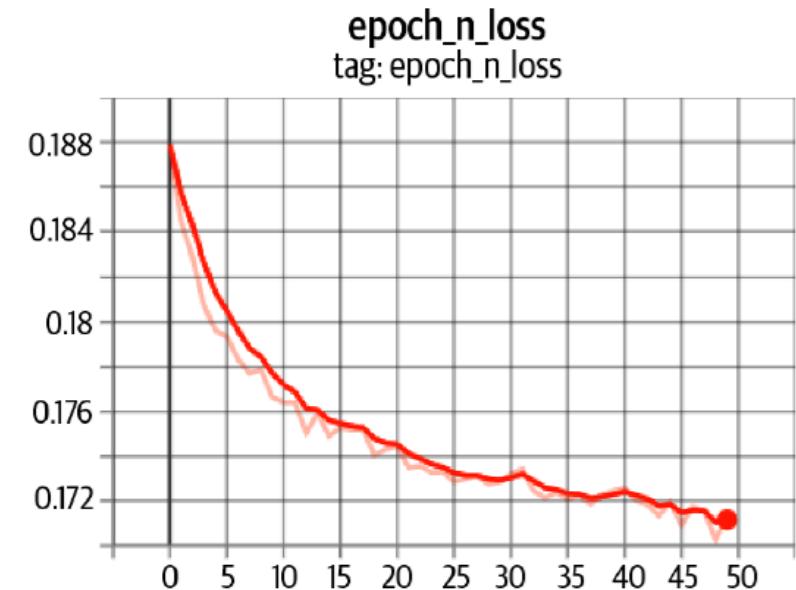
model.fit(
    train,
    epochs=50,
) ❹
```

- ❶ Instantiate the model.
- ❷ Compile the model, using the AdamW optimizer (similar to Adam but with weight decay, which helps stabilize the training process) and mean absolute error loss function.
- ❸ Calculate the normalization statistics using the training set.
- ❹ Fit the model over 50 epochs.



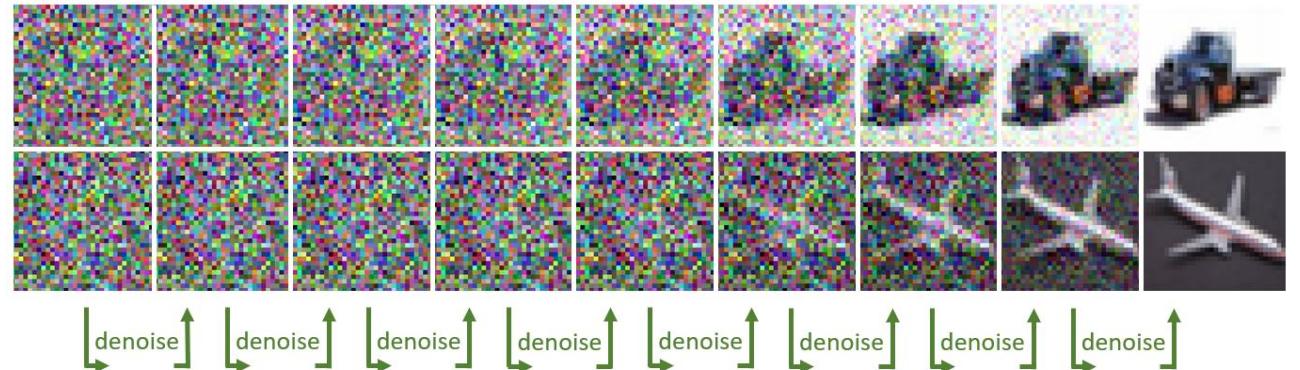
Ejecutar celdas:

3. Entrenar el modelo



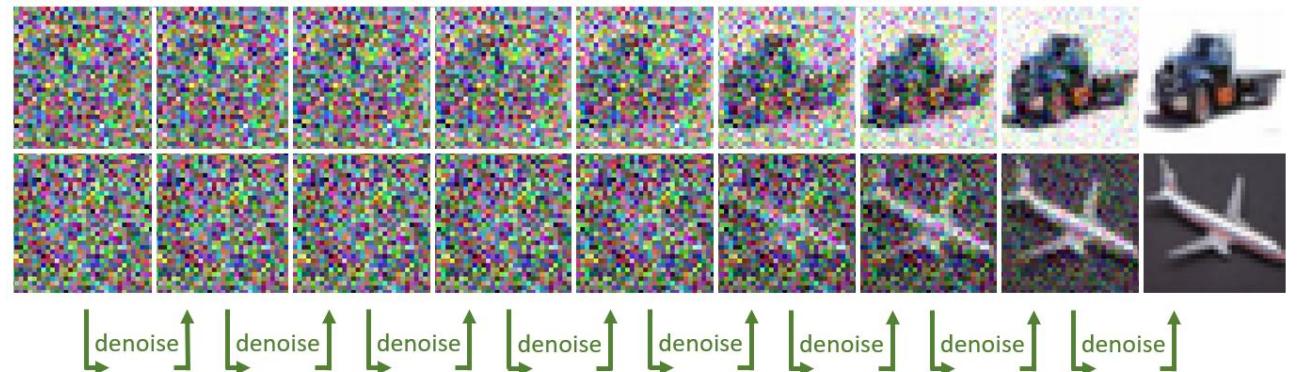
Muestreo a partir del DDM

- El proceso de **muestreo** consiste en **revertir la difusión**: partimos de ruido puro y aplicamos el modelo para ir eliminando el ruido paso a paso.



Muestreo a partir del DDM

- El proceso de **muestreo** consiste en **revertir la difusión**: partimos de ruido puro y aplicamos el modelo para ir eliminando el ruido paso a paso.



ddm.ipynb


4. Generación de imágenes

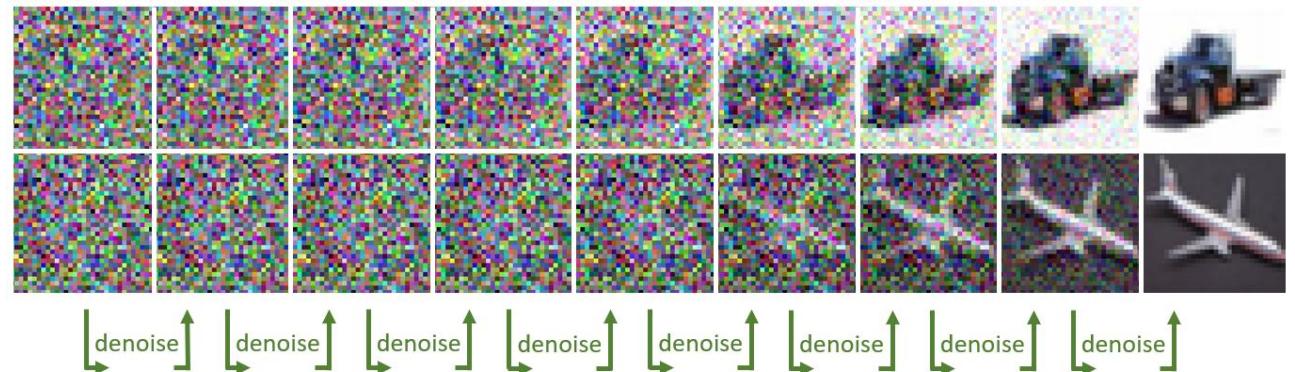
```
# Ruta al modelo completo guardado en Drive
model_path = "/content/drive/MyDrive/Colab Notebooks/CursoIAGenerativa/notebooks/Clase18_DM/models/ddm_full_model_epoch_008.keras"
```

```
# Cargar el modelo completo
loaded_ddm = tf.keras.models.load_model(model_path, compile=False)
```

- ddm_full_model_epoch_001.keras
- ddm_full_model_epoch_004.keras
- ddm_full_model_epoch_008.keras
- ddm_full_model_epoch_0011.keras
- ddm_full_model_epoch_0050.keras

Muestreo a partir del DDM

- El proceso de **muestreo** consiste en **revertir la difusión**: partimos de ruido puro y aplicamos el modelo para ir eliminando el ruido paso a paso.



ddm.ipynb
Google
colab

```
# Generación de imágenes aleatorias desde el modelo cargado
generated_images = loaded_ddm.generate(num_images=10, diffusion_steps=20).numpy()
display(generated_images)
```

```
def generate(self, num_images, diffusion_steps, initial_noise=None):
    if initial_noise is None:
        initial_noise = tf.random.normal(
            shape=(num_images, IMAGE_SIZE, IMAGE_SIZE, 3)
        )
    generated_images = self.reverse_diffusion(
        initial_noise, diffusion_steps
    )
    generated_images = self.denormalize(generated_images)
    return generated_images
```

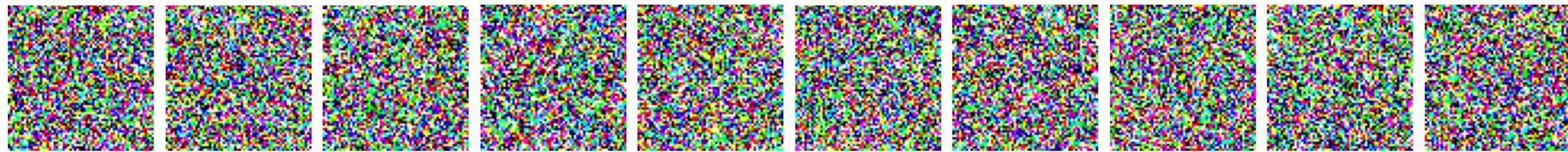


ddm_full_model_epoch_001.keras

¿Cómo se ven las imágenes generadas?

ddm_full_model_epoch_001.keras

¿Cómo se ven las imágenes generadas?



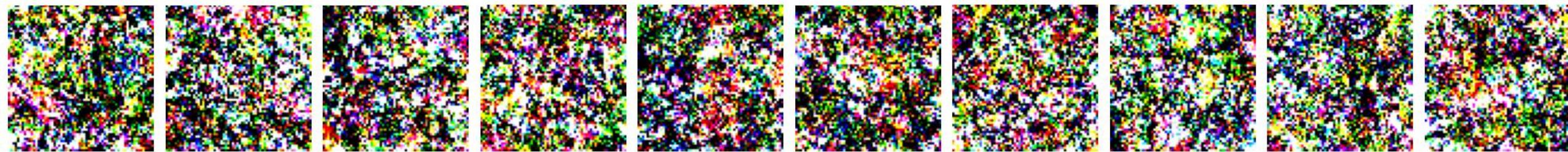


ddm_full_model_epoch_004.keras

¿Cómo se ven las imágenes generadas?

ddm_full_model_epoch_004.keras

¿Cómo se ven las imágenes generadas?



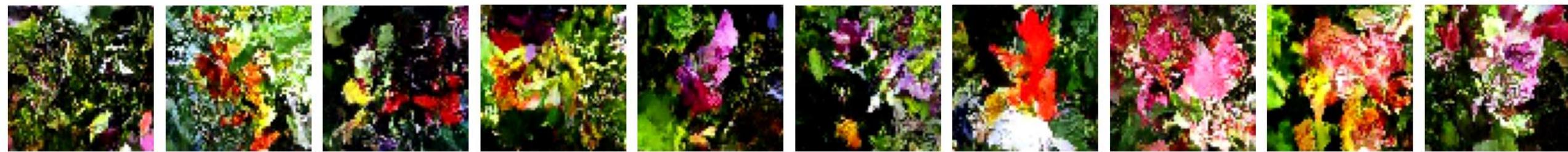


ddm_full_model_epoch_008.keras

¿Cómo se ven las imágenes generadas?

ddm_full_model_epoch_008.keras

¿Cómo se ven las imágenes generadas?





ddm_full_model_epoch_011.keras

¿Cómo se ven las imágenes generadas?

ddm_full_model_epoch_011.keras

¿Cómo se ven las imágenes generadas?



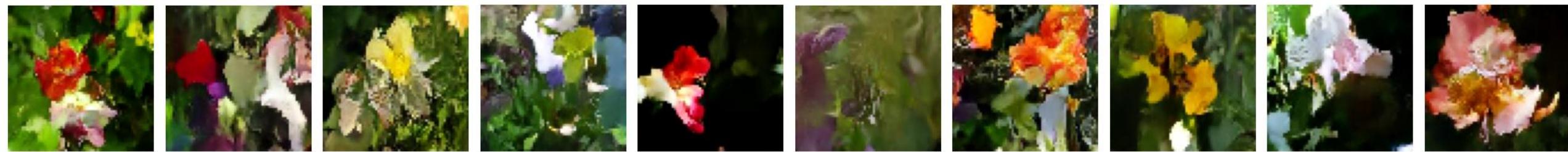


ddm_full_model_epoch_050.keras

¿Cómo se ven las imágenes generadas?

ddm_full_model_epoch_050.keras

¿Cómo se ven las imágenes generadas?

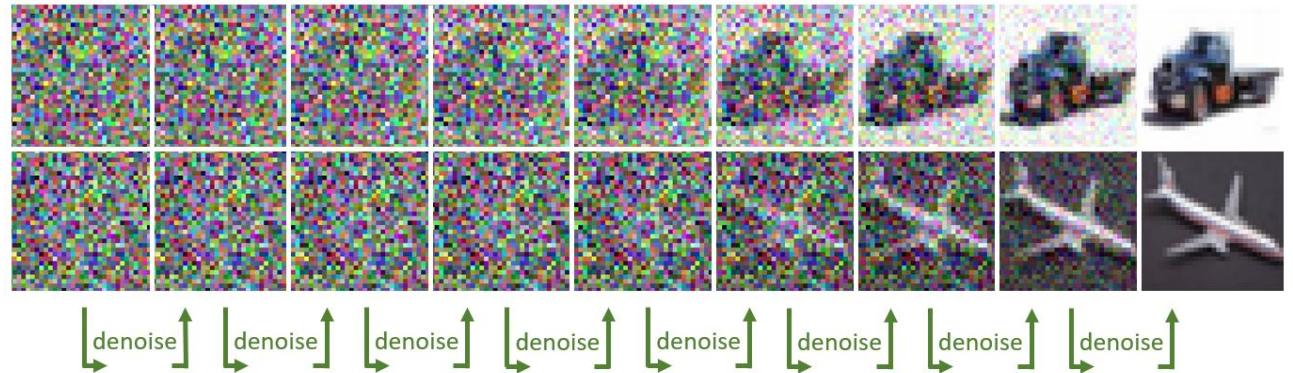


Muestreo a partir del DDM

- El proceso de **muestreo** consiste en **revertir la difusión**: partimos de ruido puro y aplicamos el modelo para ir eliminando el ruido paso a paso.

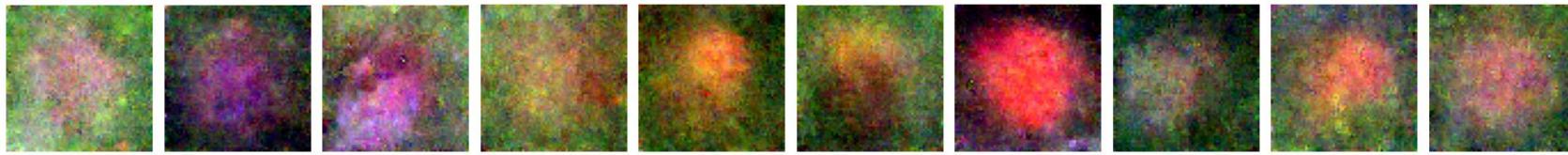
ddm.ipynb


```
# Observar la mejora usando diferentes valores de pasos de diffusion
for diffusion_steps in list(np.arange(1, 6, 1)) + [20] + [100]: → crea la lista de pasos [1, 2, 3, 4, 5, 20, 100].
    tf.random.set_seed(42)
    generated_images = ddm.generate(
        num_images=10,
        diffusion_steps=diffusion_steps,
    ).numpy()
    display(generated_images)
```



Pasos de difusión

1



2



3



4



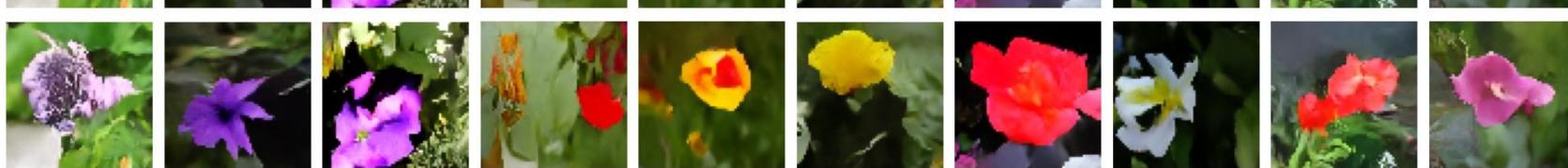
5



20



100



Muestreo a partir del DDM

- El proceso de **muestreo** consiste en **revertir la difusión**: partimos de ruido puro y aplicamos el modelo para ir eliminando el ruido paso a paso.

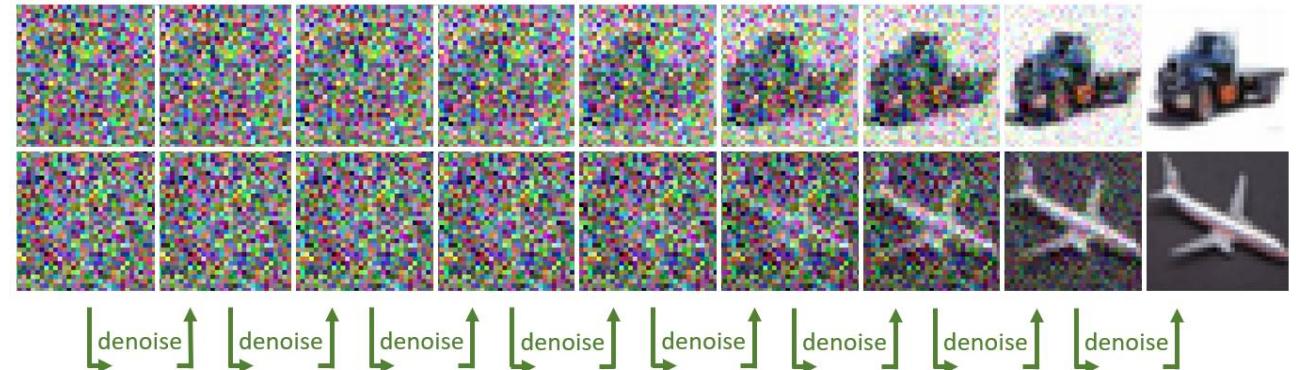
```
def spherical_interpolation_tf(a, b, t):
    """Interpolación esférica entre dos tensores en TensorFlow."""
    t = tf.cast(t, tf.float32)
    return tf.sin(t * math.pi / 2) * a + tf.cos(t * math.pi / 2) * b

# Generar interpolaciones en el espacio de ruido
tf.random.set_seed(100)

for i in range(5):
    a = tf.random.normal(shape=(IMAGE_SIZE, IMAGE_SIZE, 3))
    b = tf.random.normal(shape=(IMAGE_SIZE, IMAGE_SIZE, 3))
    t_values = tf.linspace(0.0, 1.0, 11) # 11 pasos (0 → 1)  t = [0.0, 0.1, 0.2, ..., 1.0]
    initial_noise = tf.stack(
        [spherical_interpolation_tf(a, b, t) for t in t_values], axis=0
    ) # (11, 64, 64, 3)

    generated_images = ddm.generate(
        num_images=initial_noise.shape[0],
        diffusion_steps=20,
        initial_noise=initial_noise,
    ).numpy()

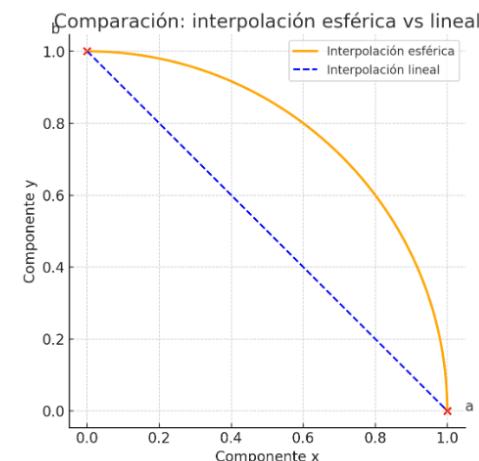
    display(generated_images, n=11)
```



1. Define una interpolación esférica

Combina dos tensores de ruido a y b mediante una trayectoria curva:

$$z(t) = \sin\left(\frac{\pi t}{2}\right) a + \cos\left(\frac{\pi t}{2}\right) b$$



- La trayectoria entre los puntos a y b sigue un **arco de circunferencia**, no una línea recta.
- En modelos de difusión, esta curva representa una transición **más natural** en el **espacio de ruido**, preservando la magnitud y evitando regiones no plausibles del espacio latente.

Muestreo a partir del DDM

- El proceso de **muestreo** consiste en **revertir la difusión**: partimos de ruido puro y aplicamos el modelo para ir eliminando el ruido paso a paso.

```

def spherical_interpolation_tf(a, b, t):
    """Interpolación esférica entre dos tensores en TensorFlow."""
    t = tf.cast(t, tf.float32)
    return tf.sin(t * math.pi / 2) * a + tf.cos(t * math.pi / 2) * b

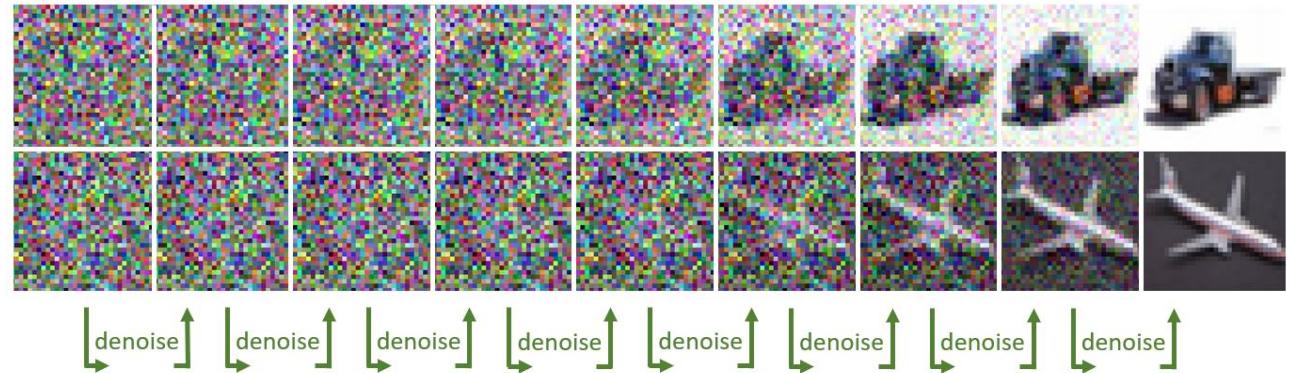
# Generar interpolaciones en el espacio de ruido
tf.random.set_seed(100)

for i in range(5):
    a = tf.random.normal(shape=(IMAGE_SIZE, IMAGE_SIZE, 3))
    b = tf.random.normal(shape=(IMAGE_SIZE, IMAGE_SIZE, 3))
    t_values = tf.linspace(0.0, 1.0, 11) # 11 pasos (0 → 1)  t = [0.0, 0.1, 0.2, ..., 1.0]
    initial_noise = tf.stack(
        [spherical_interpolation_tf(a, b, t) for t in t_values], axis=0
    ) # (11, 64, 64, 3)

    generated_images = ddm.generate(
        num_images=initial_noise.shape[0],
        diffusion_steps=20,
        initial_noise=initial_noise,
    ).numpy()

    display(generated_images, n=11)

```



2. Genera pares de ruidos aleatorios

Cada par (\mathbf{a}, \mathbf{b}) representa dos puntos distintos del espacio gaussiano \mathcal{X}_T .

Muestreo a partir del DDM

- El proceso de **muestreo** consiste en **revertir la difusión**: partimos de ruido puro y aplicamos el modelo para ir eliminando el ruido paso a paso.

```

def spherical_interpolation_tf(a, b, t):
    """Interpolación esférica entre dos tensores en TensorFlow."""
    t = tf.cast(t, tf.float32)
    return tf.sin(t * math.pi / 2) * a + tf.cos(t * math.pi / 2) * b

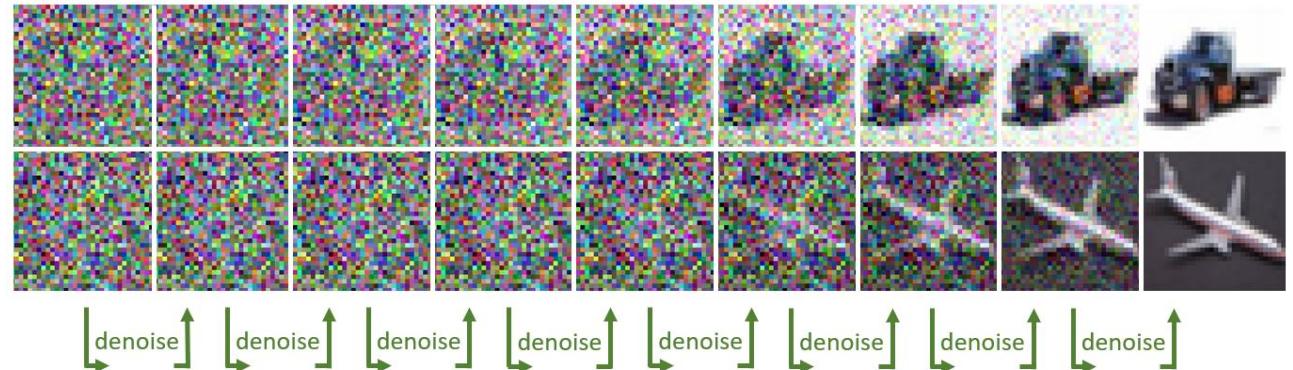
# Generar interpolaciones en el espacio de ruido
tf.random.set_seed(100)

for i in range(5):
    a = tf.random.normal(shape=(IMAGE_SIZE, IMAGE_SIZE, 3))
    b = tf.random.normal(shape=(IMAGE_SIZE, IMAGE_SIZE, 3))
    t_values = tf.linspace(0.0, 1.0, 11) # 11 pasos (0 → 1)  t = [0.0,0.1,0.2,...,1.0]
    initial_noise = tf.stack(
        [spherical_interpolation_tf(a, b, t) for t in t_values], axis=0
    ) # (11, 64, 64, 3)

    generated_images = ddm.generate(
        num_images=initial_noise.shape[0],
        diffusion_steps=20,
        initial_noise=initial_noise,
    ).numpy()

    display(generated_images, n=11)

```



3. Calcula 11 puntos intermedios

Con $t_values = [0, 0.1, \dots, 1]$, se generan 11 ruidos interpolados entre a y b .

Muestreo a partir del DDM

- El proceso de **muestreo** consiste en **revertir la difusión**: partimos de ruido puro y aplicamos el modelo para ir eliminando el ruido paso a paso.

```

def spherical_interpolation_tf(a, b, t):
    """Interpolación esférica entre dos tensores en TensorFlow."""
    t = tf.cast(t, tf.float32)
    return tf.sin(t * math.pi / 2) * a + tf.cos(t * math.pi / 2) * b

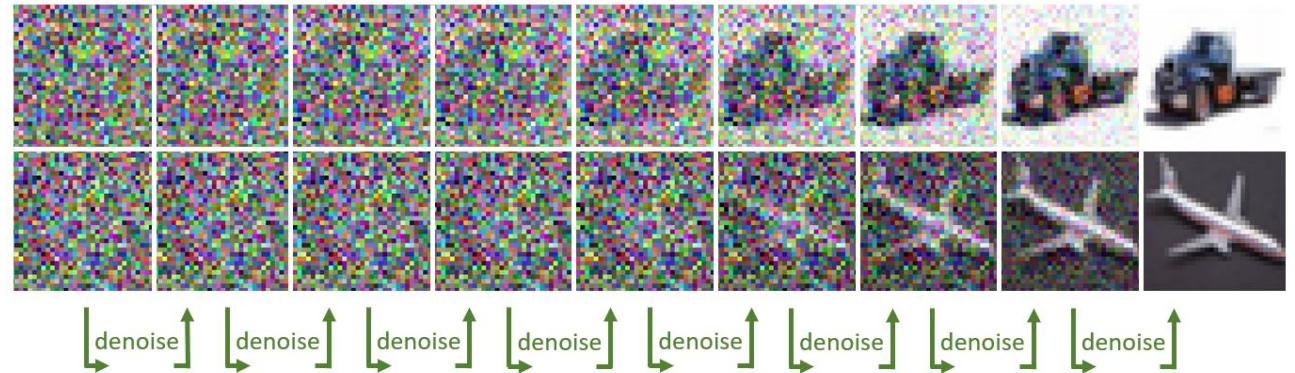
# Generar interpolaciones en el espacio de ruido
tf.random.set_seed(100)

for i in range(5):
    a = tf.random.normal(shape=(IMAGE_SIZE, IMAGE_SIZE, 3))
    b = tf.random.normal(shape=(IMAGE_SIZE, IMAGE_SIZE, 3))
    t_values = tf.linspace(0.0, 1.0, 11) # 11 pasos (0 → 1)  t = [0.0, 0.1, 0.2, ..., 1.0]
    initial_noise = tf.stack(
        [spherical_interpolation_tf(a, b, t) for t in t_values], axis=0
    ) # (11, 64, 64, 3)

generated_images = ddm.generate(
    num_images=initial_noise.shape[0],
    diffusion_steps=20,
    initial_noise=initial_noise,
).numpy()

display(generated_images, n=11)

```



4. Genera imágenes desde cada ruido

El modelo de difusión (ddm.generate) aplica 20 pasos de denoising a cada x_T , reconstruyendo 11 imágenes que muestran la transición visual completa.

Muestreo a partir del DDM

- El proceso de **muestreo** consiste en **revertir la difusión**: partimos de ruido puro y aplicamos el modelo para ir eliminando el ruido paso a paso.

```

def spherical_interpolation_tf(a, b, t):
    """Interpolación esférica entre dos tensores en TensorFlow."""
    t = tf.cast(t, tf.float32)
    return tf.sin(t * math.pi / 2) * a + tf.cos(t * math.pi / 2) * b

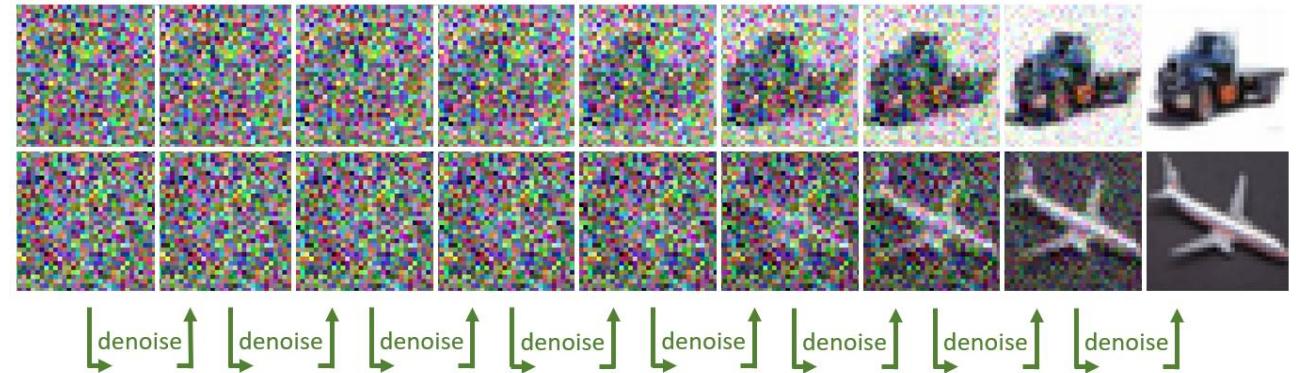
# Generar interpolaciones en el espacio de ruido
tf.random.set_seed(100)

for i in range(5):
    a = tf.random.normal(shape=(IMAGE_SIZE, IMAGE_SIZE, 3))
    b = tf.random.normal(shape=(IMAGE_SIZE, IMAGE_SIZE, 3))
    t_values = tf.linspace(0.0, 1.0, 11) # 11 pasos (0 → 1)  t = [0.0,0.1,0.2,...,1.0]
    initial_noise = tf.stack(
        [spherical_interpolation_tf(a, b, t) for t in t_values], axis=0
    ) # (11, 64, 64, 3)

generated_images = ddm.generate(
    num_images=initial_noise.shape[0],
    diffusion_steps=20,
    initial_noise=initial_noise,
).numpy()

display(generated_images, n=11)

```



5. Muestra las imágenes resultantes

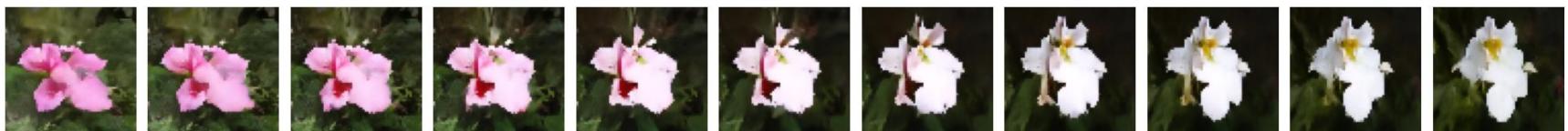
Con este
experimento
podemos ver que:

El espacio de ruido está
estructurado:

pequeñas variaciones en el ruido
producen cambios coherentes en la
imagen generada.

Las interpolaciones son
“suaves” y semánticamente
consistentes:

El modelo no memoriza imágenes,
sino que *aprende una distribución*
continua.





¿Hay un espacio latente en el modelo DDPM?

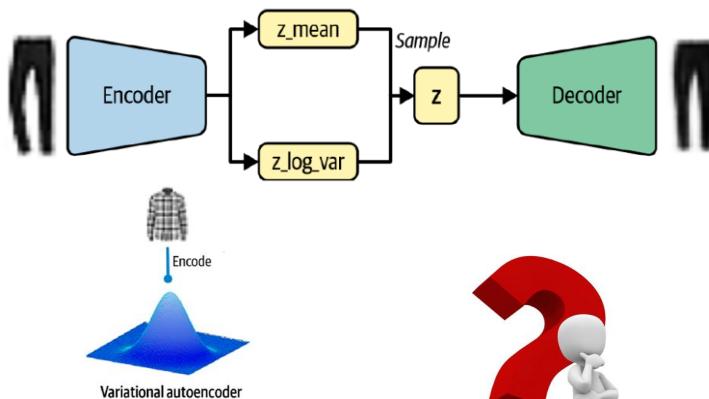
Espacio latente

En modelos generativos como los VAEs o las GANs, el proceso de generación parte de una variable latente z —generalmente distribuida como $z \sim \mathcal{N}(0, I)$ —y se transforma mediante una red neuronal:

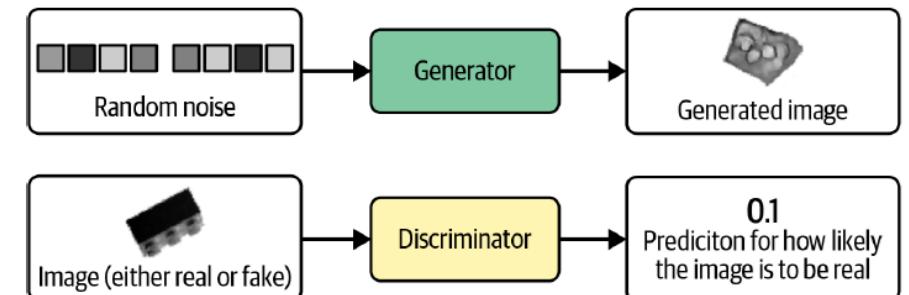
$$x_0 = G(z)$$

donde z es un **vector de baja dimensión** (p. ej. 128 o 512) que codifica **características semánticas** del dato (forma, color, estilo, identidad, etc.).

Este espacio Z es el **espacio latente explícito**, donde se pueden hacer operaciones algebraicas significativas (interpolaciones, direcciones, etc.).

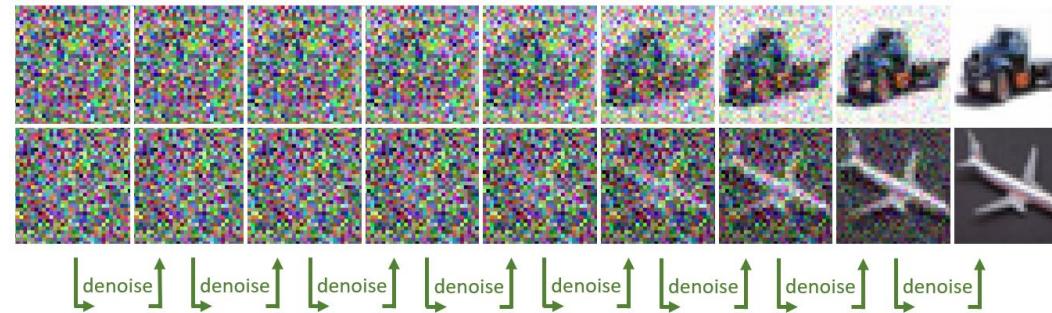


¿Hay un espacio latente en el modelo DDM?



En los **modelos de difusión denoising (DDPM)**, no hay una variable latente aprendida separada del espacio de datos.

El modelo trabaja directamente en el espacio de las imágenes $x \in \mathbb{R}^{H \times W \times C}$.



El modelo aprende una trayectoria probabilística $x_T \rightarrow x_{T-1} \rightarrow \dots \rightarrow x_0$
en el **espacio de los datos mismos**, no en un espacio latente separado.

No existe un “espacio latente” explícito en los DDPM, como en los VAEs o GANs.

Sin embargo, el espacio de ruido x_T puede considerarse un espacio latente implícito desde el cual se generan las muestras, ya que:

- Representa un punto de partida para la generación.
- Permite interpolaciones significativas.
- Muestra una estructura semántica emergente.